Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Vergleich von Ansätzen zur Speicherung unstrukturierter Daten in Lakehouses

Pascal Joos

Studiengang:	Software Engineering
Prüfer/in:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Betreuer/in:	Jan Schneider, M.Sc.
Beginn am:	20. April 2023
Beendet am:	20. Oktober 2023

Kurzfassung

Das Lakehouse ist eine neue Datenplattform für Unternehmen. Sie kam in den letzten Jahren auf und vereint die Vorteile von Data Lakes und Data Warehouses. Ähnlich wie Data Lakes können auch Lakehouses alle Arten von Daten speichern und dabei von offenen Dateiformaten Gebrauch machen. Lakehouses verfügen über eine Metadatenschicht, die wichtige Verwaltungs- und Optimierungsfunktionen ermöglicht, wie sie aus Data Warehouses bekannt sind. Durch diese Kombination aus Vorteilen beider Datenplattformen ist das Lakehouse für die effiziente Ausführung von traditionellen Analysemethoden, wie Reporting und OLAP, sowie Advanced-Analytics-Techniken, wie Machine Learning, gleichermaßen geeignet. Mit der zunehmenden Relevanz von Advanced-Analytics-Techniken, die auch unstrukturierte Daten verarbeiten können, stellt sich die Frage, wie unstrukturierte Daten am besten in einem Lakehouse abgespeichert werden sollten. Diese Arbeit untersucht, welcher Ansatz sich hinsichtlich quantitativer und qualitativer Aspekte am besten für die Speicherung unstrukturierter Daten eignet. Dafür werden zunächst verschiedene Ansätze identifiziert: Grundlegend ist es möglich, die unstrukturierten Daten mit in der erstellten Lakehouse-Tabelle abzuspeichern. Alternativ können die Daten auch separat abgelegt und im Lakehouse nur referenziert werden. Im nächsten Schritt wird betrachtet, wie sich die Ansätze auf konzeptioneller Ebene unterscheiden. Als zentraler Beitrag dieser Arbeit wird anschließend ein Framework entwickelt, mit dem verschiedene Benchmarks für unstrukturierte Daten ausgeführt werden können. Dieses Framework wird `LHBench-UnstructuredData` genannt und setzt auf einem bereits existierenden Benchmark-Framework für Lakehouses auf. `LHBench-UnstructuredData` erlaubt eine quantitative Gegenüberstellung der Ansätze. In einer Testumgebung wird die Funktionsweise exemplarisch an drei Lakehouse-Frameworks demonstriert. Mithilfe der dadurch erhaltenen Benchmarkergebnisse und der konzeptionellen Betrachtungen wird eine erste Einschätzung abgegeben, wie unstrukturierte Daten in Lakehouses abgespeichert werden sollten. Diese Analyse legt nahe, dass sich der beste Kompromiss zwischen Performanz und qualitativen Eigenschaften ergibt, wenn die unstrukturierten Daten direkt, zusammen mit den zugeordneten Metadaten, in einer Tabelle im Lakehouse abgelegt werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Abkürzungsverzeichnis

ACID Atomarität, Konsistenz, Isolation und Beständigkeit 19

BGR blau, grün und rot 47

CSV Comma-Separated Values 56

HDFS Hadoop Distributed File System 21

HiveQL Hive Query Language 31

ID Identifier 24

JDBC Java Database Connectivity 33

JSON Java Script Object Notation 24

KPI Key Performance Indicator 15

OLAP Online Analytical Processing 15

RDD Resilient Distributed Dataset 29

SQL Structured Query Language 15

SSH Secure Shell 63

YARN Yet Another Resource Negotiator 23

1 Einleitung

Mit der Etablierung von Artificial Intelligence wird für Unternehmen eine intelligente Auswertung von Geschäftsdaten immer wichtiger. Um fundierte Entscheidungen treffen zu können, ist es für Unternehmen unerlässlich Methoden wie Machine Learning oder Data Mining zu verwenden. Dennoch ist auch auf die altbekannte Bewertung mittels Key Performance Indicators (KPIs) weiterhin nicht zu verzichten, um eine Überwachung von Geschäftsprozessen über längere Zeiträume hinweg zu ermöglichen.

Bisherige Datenplattformen waren entweder für Reporting und Online Analytical Processing (OLAP) [CD97] oder für Advanced-Analytics-Methoden, wie Machine Learning, konzipiert, aber nicht für beide Anwendungsgebiete gleichzeitig geeignet. So ermöglicht die traditionellste Form von Datenplattformen, das Data Warehouse, die Generierung von Berichten mit KPIs, indem neue Daten in ein festes, vorgegebenes Datenschema transformiert werden und dann direkt SQL-Anfragen auf diesen Tabellen ausgeführt werden können. Im ab 2010 aufgekommenen Data Lake werden die Rohdaten größtenteils unverändert abgespeichert und zunächst nicht in ein normiertes Schema überführt. Dies ist vor allem für Machine Learning geeignet, da es dafür sinnvoll ist die Daten unverarbeitet vorliegen zu haben, um alle Verwendungsmöglichkeiten der Daten zu bewahren. Sind die Daten hingegen bereits in gewisser Form aggregiert oder transformiert, so gehen dadurch Informationen verloren, die für Machine-Learning-Anwendungen von Nutzen sein könnten. Jedoch sind Data Lakes weniger auf die Generierung von Berichten ausgelegt, da viele komfortable Verwaltungsfunktionen von Data Warehouses fehlen. Aufgrund des Bedarfs für beide Arten von Datenauswertungen haben daher Unternehmen begonnen, zweigleisig zu verfahren und verwenden dabei eine Kombination aus einem Data Lake und einem Data Warehouse. Dazu werden in einem zweiten Verarbeitungsschritt ausgewählte Daten aus dem Data Lake extrahiert und in transformierter Form in das Data Warehouse geladen. Durch diesen zweiten Schritt erhöht sich jedoch der Verwaltungsaufwand enorm und Teile der Daten müssen dupliziert in beiden Systemen abgespeichert werden [AGXZ21].

Die neue Datenplattform des Lakehouses zielt darauf ab, eine Lösung für dieses Problem zu bieten, indem es eine Mischform aus Data Lake und Data Warehouse darstellt. Damit ermöglicht es sowohl die Verwendung von Machine Learning und anderen Advanced-Analytics-Methoden, als auch Reporting und OLAP. Dabei versucht dieses Architekturmodell den Verwaltungsaufwand möglichst gering zu halten [SGL+23].

Unterschiedliche Lakehouse-Frameworks wie Delta Lake[1], Apache Iceberg[2] und Apache Hudi[3] sind bereits verfügbar. Auf den diesen Frameworks zugrundeliegenden Speichersystemen können mit Processing Engines, wie Apache Spark[4], Daten aller Art abgespeichert werden. Dies können sowohl strukturierte, semi-strukturierte als auch unstrukturierte Daten, wie Binärdateien oder Bilder sein. Bisher wurde noch nicht im Detail analysiert, wie unstrukturierte Daten bestmöglich in Lakehouses abgespeichert werden können. Für Apache Spark werden in der Dokumentation von Databricks zwar Empfehlungen zur Speicherung von unstrukturierten Daten abgegeben [Dat22; Dat23a; Dat23b]. Diese Vorschläge sind jedoch größtenteils weder begründet noch mit empirischen Daten untermauert.

Das Ziel dieser Arbeit besteht darin, unterschiedliche Ansätze zur Speicherung unstrukturierter Daten in Lakehouses zu betrachten und in Hinblick auf verschiedene Anwendungsszenarien sowohl qualitativ, als auch quantitativ zu vergleichen. Als zentrales Ergebnis dieser Arbeit wird dafür das Benchmark-Framework `LHBench-UnstructuredData` entwickelt, welches auf dem in der Literatur vorgeschlagenen Benchmark-Framework für Lakehouses `LHBench` aufsetzt. `LHBench-UnstructuredData` ermöglicht die Gegenüberstellung von verschiedenen Ansätzen zur Speicherung unstrukturierter Daten, sowohl hinsichtlich der Performanz, als auch des Speicherplatzbedarfs für verschiedene Aspekte der Verwendung der Testdaten. Das Framework unterstützt alle drei aktuell gängigen Lakehouse-Frameworks: Delta Lake, Apache Iceberg und Apache Hudi. Als Proof of Concept wird das Benchmark-Framework unter Verwendung von Bilderdatensätzen auf die drei Lakehouse-Frameworks angewendet. Basierend auf den erhaltenen Ergebnissen wird in der Folge eine Einschätzung und Empfehlung abgegeben, welcher Ansatz zur Speicherung unstrukturierter Daten in welchem Szenario verwendet werden sollte. Anwender können dann die für sie situativ passende Option auswählen. Zudem kann das entwickelte Benchmark-Framework in Zukunft für weitere Analysen bezüglich der Speicherung und Verwaltung unstrukturierter Daten verwendet werden.

Die Arbeit ist in folgender Weise gegliedert: In Kapitel 2 und Kapitel 3 werden die für diese Arbeit relevanten Grundlagen und Technologien beschrieben und eingeführt. Kapitel 4 geht auf verwandte Arbeiten ein, die Benchmark-Frameworks für Lakehouses sowie für Data Lakes vorstellen. In Kapitel 5 werden die Anforderungen spezifiziert, die von dem im Rahmen dieser Arbeit entwickelten Benchmark-Framework zu erfüllen sind. Kapitel 6 beschreibt verschiedene Ansätze zur Speicherung unstrukturierter Daten in Lakehouses und vergleicht diese konzeptionell. In Kapitel 7 wird die Methodik beschrieben, nach der in dieser Arbeit vorgegangen wird. Dazu wird eines der in Kapitel 4 vorgestellten Benchmark-Frameworks ausgewählt und erweitert. Kapitel 7 geht auch auf die Konfiguration der Testumgebung ein, die dann in der ausgeführten Evaluation verwendet wird. In Kapitel 8 werden schließlich die Ergebnisse der Anwendung des Benchmark-Frameworks präsentiert und Kapitel 9 diskutiert diese und gibt Empfehlungen ab, in welchen Anwendungsszenarien welche Ansätze zur Speicherung unstrukturierter Daten verwendet werden sollten. Die Ergebnisse der Arbeit werden in Kapitel 10 zusammengefasst und ein Ausblick auf weitere Aspekte gegeben, die in Zukunft noch betrachtet werden können.

[1] Siehe https://delta.io/
[2] Siehe https://iceberg.apache.org/
[3] Siehe https://hudi.apache.org/
[4] Siehe https://spark.apache.org/

2 Grundlagen

Dieses Kapitel führt in die drei Datenplattformen Data Warehouse, Data Lake und Lakehouse ein. Es beschreibt zentrale Eigenschaften und Funktionalitäten der verschiedenen Datenplattformen und grenzt diese voneinander ab. Das Kapitel beginnt mit der Einführung des Data Warehouses in Abschnitt 2.1. Daran schließt die Einführung des Data Lakes in Abschnitt 2.2 an. Zudem wird die Zwei-Schichten-Architektur, eine Mischform aus Data Warehouse und Data Lake, in Abschnitt 2.3 betrachtet. Schließlich wird in Abschnitt 2.1 das Lakehouse eingeführt.

2.1 Data Warehouse

Die Datenplattform des Data Warehouses kam Anfang der 90er Jahre auf und wurde vermehrt zur Datenanalyse eingesetzt. Unternehmen erkannten, dass es immer wichtiger wurde, Datenanalysen durchzuführen, um damit die unternehmenspolitische Entscheidungsfindung zu unterstützen. Durch die Anwendung von Reporting und OLAP können deskriptive Analysen angestellt werden, um KPIs wie zum Beispiel den Umsatz und die Verkaufszahlen einzelner Produkte zu erheben. Damit können sich Unternehmen Vorteile gegenüber ihrer Konkurrenz sichern. Da gerade in den 90er Jahren der wirtschaftliche Wettbewerb stark angestiegen ist, entwickelten Unternehmen ein Bedürfnis für ein System zur Datenanalyse, um diese Konkurrenzvorteile auszuschöpfen [VZ22, Seite 4].

Die damals bereits vorhandenen operationalen Datenbanken sind für die Datenanalyse nicht geeignet. Operationale Datenbanken wurden für die Datenhaltung in Anwendungen entworfen. Daher liegt bei diesen Datenbanken das Augenmerk auf der Unterstützung von hochgradig nebenläufigen Zugriffen mehrerer Nutzer, sowie auf der Sicherstellung der Datenkonsistenz durch transaktionale Operationen (ACID) [VZ22, Seite 4]. Operationale Datenbanken speichern jedoch typischerweise keine historischen Daten. Auch leiden diese unter schlechter Performanz bei der Ausführung von komplexen analytischen Anfragen oder beim Aggregieren großer Datenmengen [VZ22, Seite 4]. Gerade bei der Datenanalyse ist es jedoch wichtig, dass historische Daten betrachtet und hinsichtlich verschiedener Merkmale aggregiert werden können, um KPIs zu berechnen und Berichte zu generieren. Diese werden dann verwendet, um das Treffen von Geschäftsentscheidungen zu unterstützen. Unternehmen verfügen zudem über eine Vielzahl von verschiedenen operationalen IT-Systemen für diverse Teilunternehmensprozesse, häufig sogar von mehreren Anbietern mit Unterschieden in der Datenbankarchitektur. Für eine Datenanalyse müssen die Daten dieser Datenbanken, welche über unterschiedliche Tabellenschemas und Inhalte verfügen, in ein System zusammengeführt werden. Für diese Aufgabe ist ein einzelnes operationales Datenbanksystem ungeeignet. Stattdessen bedarf es eines größeren Systems, das diese Funktionalität unterstützt. Deshalb wurde das Data Warehouse eingeführt, das über die Funktionen einer gewöhnlichen Datenbank hinausgeht und zum Ziel hat, gerade diese Aspekte umzusetzen.

Inmon [Inm05, Seite 31] definiert das Data Warehouse als eine `subject-oriented`, `integrated`, `nonvolatile` und `time-variante` Sammlung von Daten um Entscheidungen des Managements zu unterstützen.

Daten in einem Data Warehouse sind also erstens themenbezogen (`subject-oriented`). Je nach Art des Unternehmens sind andere Daten für die Datenanalyse relevant. Zudem speichert ein Data Warehouse nur Daten zu bestimmten Themen, die von den Betreibern des Systems ausgewählt werden, je nachdem, welche Informationen als relevant betrachtet werden. Dadurch stellen die Daten, die in einem Data Warehouse gespeichert sind, nur einen Ausschnitt der gesamten existierenden Unternehmensinformationen dar. Außerdem werden die Daten meist in vorverarbeiteter und aggregierter Form abgespeichert und nicht in ihrer Rohform. Die Daten in einem Data Warehouse müssen aus mehreren Datenquellen zusammengeführt werden (`integrated`). Unternehmen verfügen meist über eine Vielzahl von operationalen IT-Systemen. Beispiele dafür sind Enterprise-Resource-Planning-, Manufacturing-Execution- und Supply-Chain-Management-Systeme [BP09; GK06; SVD00]. Von diesen IT-Systemen müssen relevante Daten in das Datenschema des Data Warehouses transformiert und in das Data Warehouse geladen werden [VZ22]. Dieser Prozess für die Extraktion, Transformation und das Laden der Daten in das Data Warehouse, wird gemeinhin als ETL-Prozess bezeichnet und entweder periodisch, ereignisgesteuert oder bei jeder Änderung an einer operationalen Datenbank ausgeführt [BG13, Seite 94]. In einem Data Warehouse werden Daten lange aufbewahrt und das Verändern oder Löschen von Daten ist nicht vorgesehen (`nonvolatile`) [Inm05, Seite 34]. Daten werden als Snapshot aus den Datenquellen geladen, also als eine Momentaufnahme der Daten der Datenquelle. Dieser Snapshot darf nicht verändert werden. Indem Änderungen in der Datenlage der Datenquellen regelmäßig als neue Snapshots in das Data Warehouse geladen werden, stellt ein Data Warehouse die historische Entwicklung der Daten dar [Inm05, Seite 34]. Es können also Auswertungen von KPIs über längere Zeiträume ausgeführt werden, um die Entwicklung dieser im Laufe der Zeit zu betrachten. Insbesondere lassen sich so auch frühere Stände von Daten einsehen (`time-variant`).

Um Datenanalysen auf Data Warehouses auszuführen, können Anfragesprachen unter Verwendung des OLAP-Paradigmas genutzt werden. Meist bieten Hersteller von Data Warehouses aber nur eine begrenzte Auswahl an Anfragesprachen und Engines an, mit denen auf die Daten zugegriffen werden kann. Eine weitere Möglichkeit der Datenanalyse ist die Erstellung von Berichten oder interaktiven Dashboards [VZ22].

Data Warehouses sind auf die Speicherung von strukturierten Daten begrenzt und verwenden meist proprietäre Dateiformate. Dies führt dazu, dass insbesondere auf die Rohdaten im Data Warehouse nicht zugegriffen werden kann. Dadurch sind diese für neuere Methoden der Datenanalyse, wie Machine Learning und weitere verteilte und parallele Verarbeitungsmethoden, nicht geeignet. Der Grund dafür liegt darin, dass die Daten aller Art dafür in möglichst unveränderter Form benötigt werden und auf diese auch möglichst direkt zugegriffen werden muss [AGXZ21]. Bietet ein Data Warehouse beispielsweise nur eine SQL-Schnittstelle, so lässt sich nicht effizient auf die Daten in einer Form zugreifen, wie es für Machine-Learning-Anwendungen notwendig ist [AGXZ21].

2.2 Data Lake

In Unternehmen hat sich das neuere Konzept des Data Lake etabliert. Das Aufkommen dieser neueren Form des Speichersystems für die Datenanalyse wurde vor allem von Unternehmen vorangetrieben. In der Forschung kam das Thema des Data Lakes erst infolgedessen auf [KW18].

In einem Data Lake können, im Gegensatz zum Data Warehouse, auch semi-strukturierte und unstrukturierte Daten gespeichert werden. Es wird keine gezielte Vorauswahl von Daten getroffen und es werden nicht nur Daten abgespeichert, die einen offensichtlichen Nutzen bei der Datenanalyse besitzen. Stattdessen werden jegliche Daten, die in einem Unternehmensprozess aufkommen im Data Lake abgespeichert [KW18]. Dazu werden meist offene Dateiformate auf kostengünstigen Speichersystemen verwendet. Beispiele für relevante Dateiformate sind Apache Parquet[1] und Apache ORC[2] [AGXZ21]. Als Speichersysteme kommen Cloud Object Stores oder verteilte Dateisysteme in Frage. Die Daten werden beim Abspeichern in ihrer Rohform beibehalten oder nur geringfügig transformiert. Anders als im Data Warehouse gibt es kein festes Schema, in das die Daten überführt werden müssen. Dadurch wird die Transformation der Daten bei der Ausführung der ETL-Prozesse eingespart. Transformationen und Aggregationen sind bei stark unstrukturierten Daten meist auch gar nicht sinnvoll möglich. Daten verbleiben in ihrem Originalformat und werden erst bei der Datenanalyse je nach Bedarf und Anwendungsfall in ein passendes Schema transformiert. Die für einen Anwendungsfall transformierten Daten werden dann in dieser veränderten Form wiederum im Speichersystem abgelegt und können mit Datenanalyse-Anwendungen zugegriffen werden. Das Prozessieren von Daten erfolgt also statt durch einen ETL-Prozess vielmehr nach einem ELT-Prozess. Die Daten werden extrahiert und zunächst in ihrer Rohform in den Data Lake geladen und dann erst je nach Bedarf für verschiedene Anwendungen transformiert [KW18].

Dies ermöglicht insbesondere die Anwendung von Advanced-Analytics-Methoden, wie Machine Learning, bei denen ein direkter Zugriff auf die Rohdaten notwendig ist, um ein Modell zu trainieren oder ein bereits trainiertes Modell anzuwenden. Auch Methoden des Data Mining sind beim Data Lake sinnvoller anwendbar, als bei Data Warehouses, da der schnelle Zugriff auf die Gesamtheit der untransformierten Rohdaten ermöglicht, diverse Fragestellungen zu betrachten und zuvor ungeahnte Ergebnisse zu erhalten [SGL+23].

Andererseits ist ein Data Lake für traditionellere Datenanalysemethoden, wie Reporting und KPI-Erstellung, ungeeignet. Diese bleiben für Unternehmen jedoch weiterhin äußerst relevant. Data Lakes bieten viele der Verwaltungsfunktionen über die Data Warehouses verfügen nicht an [AGXZ21]. Zum Beispiel befinden sich Daten nicht unbedingt in einer festen Tabellenstruktur und müssen über eine Processing Engine zugegriffen werden. Aufgrund des Mangels an komfortablen Funktionen und der begrenzten Performanz, wenn große Datenmengen aggregiert werden müssen, werden Data Lakes daher meist nicht für traditionelle Datenanalysemethoden verwendet.

Hinzu kommt, dass Data Lakes im Gegensatz zu Data Warehouses keine Atomarität, Konsistenz, Isolation und Beständigkeit (ACID) sicherstellen [AGXZ21]. Daher kann es vorkommen, dass im Datenbestand des Data Lakes Inkonsistenzen auftreten. Insbesondere kann ein Snapshot des Data Lakes fehlerhafte Werte enthalten. Es gibt im Allgemeinen auch keine Sicherheitsmechanismen, die verhindern, dass falsche oder doppelte Daten in den Data Lake eingefügt werden [KW18].

[1] Siehe https://parquet.apache.org/
[2] Siehe https://orc.apache.org/

Eine weitere Schwierigkeit von Data Lakes besteht darin, dass diese keine direkte Evolution des Datenschemas ermöglichen, insofern ein Datensatz überhaupt in einem definierten Schema vorliegt. Ältere Daten im Data Lake können nicht einfach in ihrem Schema aktualisiert werden, wenn sich dieses verändert hat. Dies führt dazu, dass, wenn neue Daten in einem anderen Format eingefügt werden, aufwendig über Metadaten definiert werden muss, wie diese mit älteren Daten zusammengeführt werden können. Allgemein muss aufwendig über Metadaten festgelegt werden, wie Daten verarbeitet werden können. Es gibt dafür jedoch kein standardisiertes Vorgehen, wie diese Metadaten erstellt und verwaltet werden können. Dadurch kann es leicht zu Fehlern kommen und es besteht die Gefahr, dass der Data Lake in einen Data Swamp [HGQ16] ausartet, wenn ältere Daten nicht mehr mit neueren Daten in Einklang gebracht werden können, Inkonsistenzen in den Daten auftreten und nicht mehr bekannt ist, welche Daten es überhaupt im Data Lake gibt und wofür diese verwendet werden können [KW18].

2.3 Zwei-Schichten-Architektur

Um sowohl Advanced-Analytics-Methoden, wie Machine Learning, als auch Reporting und OLAP zu unterstützen, haben viele Unternehmen begonnen, zweigleisig zu verfahren. Dabei werden Daten zunächst in einem Data Lake abgespeichert, um Machine Learning und weitere Advanced-Analytics-Methoden verwenden zu können. Wenige ausgewählte Daten, die als relevant für Reporting und OLAP betrachtet werden, werden zusätzlich noch mithilfe eines zusätzlichen ETL-Prozesses in ein separates Data Warehouse geladen [AGXZ21].

Viele Unternehmen wenden diese Architektur an. Sie wird von Armbrust et al. [AGXZ21] als „two-tier data lake + warehouse architecture“ bezeichnet.

Durch dieses zweigleisige Vorgehen erhöht sich jedoch die Komplexität stark. Die Daten müssen zunächst in den Data Lake überführt werden und dann in einem weiteren ETL-Prozess in das Data Wahrehouse geladen werden. Dadurch kann es zu längeren Verzögerungen kommen, bis die Daten auch im Data Warehouse vorliegen, welche unter Umständen mehrere Stunden oder sogar mehrere Tage betragen können. Die zusätzlichen ETL-Schritte erhöhen auch die Gefahr, dass es zu Fehlern kommt und Inkonsistenzen in den Daten auftreten, wodurch sich die Datenqualität reduziert. Außerdem werden Teile der Daten mehrfach abgespeichert und es muss zusätzlich zum Data Lake ein kostenintensives Data Warehouse betrieben werden. Es können auch Inkonsistenzen zwischen den beiden Kopien der Daten im Data Lake und Data Warehouse auftreten. Da es keine „Single Source of Truth“ gibt ist dann unklar, welche Daten korrekt sind [AGXZ21].

2.4 Lakehouse

In den letzten Jahren ist mit sogenannten Lakehouses eine weitere Art von Datenplattform aufgekommen, deren Ziel darin besteht, die Vorteile von Data Warehouse und Data Lake zu vereinen und auf diese Weise auch die Nachteile der Zwei-Schichten-Architektur zu vermeiden. Ein Lakehouse nutzt den direkten, kostengünstigen Speicher eines Data Lakes mit offenen Dateiformaten. Als

Speichersysteme können verteilte Dateisysteme, wie das Hadoop Distributed File System (HDFS)[3], oder Cloud Object Stores, wie zum Beispiel Amazon S3[4], verwendet werden. Dabei bietet es auch Verwaltungs- und Optimierungsfunktionen, wie ACID-Eigenschaften oder Schema-Evolution, wie dies bei einem Data Warehouse typischerweise der Fall ist [AGXZ21].

Um dies zu ermöglichen, kommen Lakehouse-Frameworks zum Einsatz, welche eine Metadatenschicht verwenden, die definiert, welche Dateien des zugrundeliegenden Speichersystems Teil einer bestimmten Version einer Tabelle sind. Dadurch lassen sich mithilfe dieser Metadaten eine Versionierung der Tabellen, sowie ACID-Eigenschaften umsetzen. Hinzu kommen Maßnahmen zur Performanzoptimierung, wie Caching, die Berechnung von Statistiken und Datenlayout-Optimierungen, die es damit ermöglichen, auch Datenanalysen wie Reporting und OLAP effizient durchzuführen. Diese und weitere Funktionen, sowie Details zu deren Umsetzung, werden in Abschnitt 3.2 am Beispiel des Lakehouse-Frameworks Delta Lake genauer beschrieben [AGXZ21].

Dadurch, dass ein Lakehouse offene Dateiformate verwendet, wie ein Data Lake auch unstrukturierte Daten abspeichern kann und dies mit zusätzlichen Funktionen wie ACID-Eigenschaften verbindet, sind Lakehouses auch für Datenanalysemethoden wie Machine Learning und Data Mining besonders geeignet [AGXZ21].

Beim Lakehouse ist im Gegensatz zu der häufig in Unternehmen vorherrschenden Zwei-Schichten-Architektur kein weiterer ETL-Prozess nötig, um die Daten zwischen verschiedenen Datenplattformen zu transferieren. Transformationen der Daten innerhalb des Lakehouses werden jedoch, wie auch beim Data Lake, weiterhin ausgeführt, beispielsweise zur Vorverarbeitung von Daten für eine spezielle Anwendung.

Schneider et al. [SGL+23] versuchen eine klare Definition für Lakehouses aufzustellen und untersuchen eine Auswahl an Data-Management-Tools dahingehend, ob sie diese Definition erfüllen können. Sie definieren Lakehouses als eine integrierte Datenplattform, die dieselbe Art von Speicher und dasselbe Dateiformat für Reporting und OLAP, Data Mining und Machine Learning, als auch Streaming verwendet [SGL+23].

Diese Definition gibt insbesondere vor, welche Analysemethoden ein Lakehouse unterstützen muss. Es muss einen breiten Katalog an Analyseformen abdecken und sowohl Methoden unterstützen, für die das Data Warehouse genutzt wird, als auch die, bei denen Data Lakes sich als vorteilhaft herausgestellt haben.

Hinzu kommt die Unterstützung von Streaming [KKR17], womit Reporting in Nahe-Echtzeit ermöglicht wird. Dabei werden Ergebnisse nicht fortlaufend komplett neu berechnet, sondern bei neu hinzukommenden Daten entsprechend inkrementell aktualisiert. Ein zweiter Aspekt des Streamings sind Stream Analytics. Dies bezeichnet Analysemethoden auf einem kontinuierlichen Datenstrom, wie Mustererkennung und Datenfilterung [KKR17]. Unstrukturierte Daten spielen bei Reporting in Nahe-Echtzeit und Streaming Analytics weniger häufig eine Rolle [SGL+23]. Daher ist diese Analysemethode im Kontext dieser Arbeit weniger relevant.

Die Definition von Schneider et al. beinhaltet zudem, dass in einem Lakehouse für alle Daten dieselbe Art von Speicher und dasselbe Dateiformat verwendet werden müssen. Dies trifft nicht auf die Metadaten zu, welche auch in einem anderen Dateiformat vorliegen dürfen. Dies hat zur

[3] Siehe https://hadoop.apache.org/
[4] Siehe https://aws.amazon.com/s3/

Folge, dass insbesondere die Zwei-Schichten-Architektur von dieser Definition als Lakehouse ausgeschlossen ist, da hier im Data Warehouse andere Arten von Speicher und andere Dateiformate verwendet werden als im Data Lake.

Auf Grundlage der durchgeführten Evaluation kommen Schneider et al. zu dem Schluss, dass die Frameworks Delta Lake, Apache Hudi und Apache Iceberg verwendet werden können, um Lakehouses zu bauen, die diese Definition erfüllen [SGL+23].

3 Relevante Technologien

In diesem Kapitel werden verschiedene Technologien kurz eingeführt und beschrieben, die für das Verständnis dieser Arbeit relevant sind. Es beginnt mit der Einführung des offenen Dateiformates Apache Parquet in Abschnitt 3.1. Dann werden die drei Lakehouse-Frameworks Delta Lake, Apache Iceberg und Apache Hudi betrachtet, die den Bau eines Lakehouses ermöglichen. Anschließend wird die Processing Engine Apache Spark in Abschnitt 3.5 und dann das HDFS in Abschnitt 3.6 eingeführt. Schließlich wird auch noch die Funktion von Apache Hadoop YARN und dem Apache Hive Metastore in den Abschnitten 3.7 und 3.8 kurz erklärt.

3.1 Apache Parquet

Apache Parquet ist ein spaltenorientiertes Dateiformat, welches zur effizienten Repräsentation von Daten verwendet werden kann. Es ist ein offenes Format und ist daher vor allem für Machine-Learning-Anwendungen geeignet, da diese häufig einen direkten Datenzugriff benötigen [SGL+23]. Dies ist bei proprietären Dateiformaten nicht möglich [AGXZ21]. Apache Parquet bietet zudem effiziente Datenkompression, wodurch sich Speicherplatz einsparen lässt, sowie verschiedene Encoding-Schemas. Durch die spaltenorientierte Anordnung lassen sich effizient spalten-weise Aggregationen auf den Daten durchführen, da die benötigten Werte direkt nebeneinander stehen. Apache Parquet wurde für die Verwendung auf dem HDFS optimiert [Apa22].

Das Dateiformat enthält Metadaten, die unter anderem angeben, welche Teile von Spalten der gespeicherten Tabelle in der Datei enthalten sind, um ein leichteres Auffinden von Daten zu ermöglichen [Apa22]. Diese Metadaten beinhalten unter anderem Min-Max-Statistiken. Diese sind nützlich bei der Verwendung für Daten auf dem HDFS. Jedoch rentiert sich bei Cloud Object Stores die Nutzung dieser Statistiken aufgrund der deutlich höheren Latenz nicht [ADS+20].

Durch den Aufbau des Dateiformats in Spalten eignet sich dieses Format besonders gut für strukturierte und semi-strukturierte Daten, da diese sich leicht in eine solche Tabellenstruktur einordnen lassen. Zudem unterstützt Apache Parquet verschachtelte Datentypen und kann somit auch für semi-strukturierte Daten verwendet werden. Parquet wird von vielen Processing Engines unterstützt [ADS+20].

3.2 Delta Lake

Delta Lake ist eines der drei Lakehouse-Frameworks, die von Schneider et al. [SGL+23] als solche identifiziert wurden. Delta Lake wurde ursprünglich von Databricks entwickelt [Dat23c] und ist seit 2016 verfügbar [ADS+20]. Mittlerweile ist es ein Open-Source-Projekt und Teil der Linux Foundation Projects, wird aber weiterhin stark von Databricks weiterentwickelt [Dat23c].

Die nachfolgenden Beschreibungen in diesem Abschnitt beruhen auf den Schilderungen von Armbrust et al. [ADS+20].

Delta Lake verwendet als offenes Dateiformat Apache Parquet und speichert diese Dateien in einem verteilten Dateisystem oder Cloud Object Store ab. Dabei kann sich eine Delta-Lake-Tabelle aus mehreren Parquet-Dateien zusammensetzen. Bei jedem Schreiben von neuen oder aktualisierten Einträgen in eine Tabelle wird eine neue Datei mit diesen hinzugefügten oder geänderten Daten erstellt. Mithilfe eines Logs, das dokumentiert, welche Dateien zu welcher Version einer Delta-Lake-Tabelle gehören, ermöglicht Delta Lake ACID-Eigenschaften und viele weitere Funktionen. Dieses Log ist dabei zusammen mit den eigentlichen Dateien im Speichersystem abgespeichert. Sowohl die Logdateien, als auch die Datendateien sind unveränderlich. Daher kann ohne Probleme Caching auf schnellen Speichermedien verwendet werden, um die Abfrage von regelmäßig verwendeten Dateien zu beschleunigen.

Durch die Verwendung des offenen Dateiformats Apache Parquet ist ein stark parallelisierter, direkter Zugriff auf die Daten für Machine Learning und Data Mining möglich. Bereits existierende Tools können diese leicht lesen, da Apache Parquet auch bereits bei Data Lakes ein häufig verwendetes Dateiformat ist und somit bereits viele Lösungen zur Datenanalyse existieren.

Delta Lake bietet Integrationen für verschiedene Processing Engines, wie Apache Spark. Über diese Integrationen können Processing Engines die Funktionalitäten von Delta Lake nutzen [Del23].

Um die Daten einer Delta-Lake-Tabelle zu lesen, wird durch die Processing Engine im Log überprüft, welche Dateien zu der zu lesenden Version gehören. Entsprechend ist es dann ausschließlich nötig, diese Dateien aus dem Speichersystem zu lesen. Alle anderen Dateien können übersprungen werden. Durch Partitionierung einer Delta-Lake-Tabelle nach einer Spalte ist es möglich, dies noch weiter auszunutzen, da bei Anfragen, welche nach dieser Spalte filtern, dann viele Datendateien der Tabelle direkt ausgeschlossen werden können.

Das Log der Delta-Lake-Tabelle enthält zudem noch weitere Metadaten, wie Min-Max-Statistiken für jede Spalte einer Datendatei, die Teil der Tabelle ist. Dadurch wird die Suche weiter beschleunigt, da Dateien übersprungen werden können, deren Min-Max-Werte außerhalb des Suchbereichs liegen.

Der Aufbau des Logs soll nun genauer beschrieben werden. In Abbildung 3.1 ist der Aufbau eines solchen Logs mit den dazugehörigen Datendateien exemplarisch dargestellt. Das Log besteht aus einer Verzeichnisstruktur, in der für jede ausgeführte Aktion auf der Delta-Lake-Tabelle ein neuer Logeintrag als JSON-Datei hinzugefügt wird. Dabei wird dieser Logeintrag atomar erstellt, um die Konsistenz des Logs sicherzustellen. Jeder Logeintrag hat eine eindeutige, aufsteigende ID, die gleichzeitig den Versionsstand der Delta-Lake-Tabelle repräsentiert. Der Logeintrag enthält eine Liste an in der Vergangenheit ausgeführten Aktionen, wie das Hinzufügen oder Löschen einer Datendatei aus der Tabelle. Jeder Aktion ist eine Datei zugeordnet, auf die sich diese bezieht (z.B. `add /a2dc5244f7f7.parquet`), sowie weitere Metadaten.

Um nun den Zustand der Delta-Lake-Tabelle zu einem bestimmten Versionsstand zu rekonstruieren, werden alle Logeinträge bis zu dieser Version betrachtet. Alle Dateien, die durch eine Aktion zur Tabelle hinzugefügt und danach nicht wieder entfernt wurden, sind dann Teil des Tabelleninhalts dieser Version.

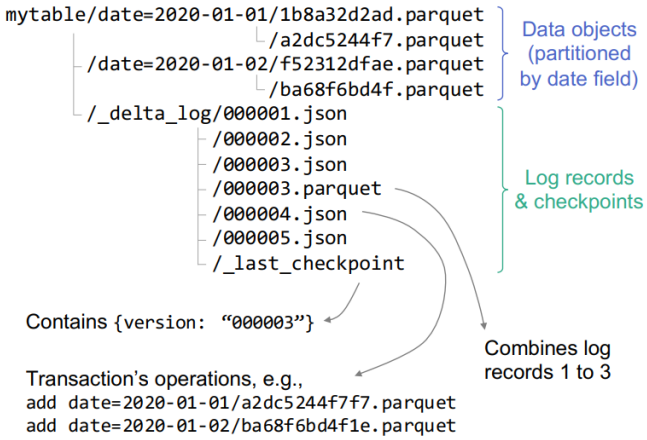


Abbildung 3.1: Aufbau einer Tabelle in Delta Lake mit Log.
Direkt übernommen von Armbrust et al. [ADS+20].

Um diesen Prozess zu beschleunigen, wird beim Erstellen von Logeinträgen in regelmäßigen Abständen ein zusätzlicher Checkpoint erstellt, der alle davorliegenden Logeinträge in einer Parquet-Datei zusammenfasst. Bei der Erstellung dieses Checkpoints können zusätzlich einige für den Log überflüssige Aktionen entfernt werden, zum Beispiel das Hinzufügen einer Datei, wenn diese bis zum Zeitpunkt des Checkpoints bereits wieder aus der Tabelle entfernt wurde. Damit lässt sich der Zugriff auf die Metadaten beschleunigen, da die dem Checkpoint vorausgehenden JSON-Dateien nicht mehr betrachtet werden müssen.

Durch die Versionierung des Logs unterstützt Delta Lake Time-Travel. Dies ist vor allem für Machine-Learning-Anwendungen sehr hilfreich, da der Zustand der Daten damit rekonstruierbar ist, auf denen ein Modell trainiert wurde. Dadurch kann auch zu einem späteren Zeitpunkt auf dem identischen Datensatz weitergearbeitet werden, auch wenn in der Zwischenzeit Änderungen an der Delta-Lake-Tabelle vorgenommen wurden.

Die ACID-Eigenschaften werden durch einen optimistischen Ansatz erzielt. Erst am Ende einer Transaktion wird überprüft, ob bereits eine konkurrierende Transaktion einen Logeintrag mit derselben ID geschrieben hat. In diesem Fall wird die Transaktion wiederholt. Dies kann im einfachsten Fall über eine Erhöhung der Log-ID erreicht werden, ohne dass erneut alle hinzuzufügenden Dateien geschrieben werden müssen. Damit wird die Serialisierbarkeit der durchgeführten Änderungen sichergestellt. Transaktionen werden von Delta Lake jedoch nur innerhalb einer Tabelle unterstützt. Über mehrere Delta-Lake-Tabellen hinweg sind Aufrufe nicht serialisierbar.

Weitere von Delta Lake unterstützte Funktionen sind Datenlayout-Optimierungen, bei welchen kleine Dateien in Hintergrundprozessen zu Dateien von normierter Größe zusammengefasst werden, sowie Audit Logging und Schema-Evolution.

3.3 Apache Iceberg

Apache Iceberg ist ein weiteres der drei aktuell etablierten Lakehouse-Frameworks. Es wurde initial von Netflix entwickelt und ist mittlerweile Open-Source und wird als Teil der Apache Software Foundation aktiv weiterentwickelt [Net18].

Im Gegensatz zu Delta Lake können für den zugrundeliegenden Data Lake bei Apache Iceberg außer Apache Parquet auch Apache ORC oder Apache Avro[1] als Dateiformat verwendet werden [Apa23h]. Dadurch besteht bei Apache Iceberg eine größere Flexibilität in der Auswahl des Dateiformats. Apache Iceberg unterstützt viele verschiedene Processing Engines, wie Apache Spark, Apache Flink[2] und Apache Hive[3] [Apa23g].

Der Aufbau einer Tabelle in Apache Iceberg ist in Abbildung 3.2 dargestellt. In der Metadatenschicht von Apache Iceberg, die über der Datenschicht liegt, wird für jede Änderung an einer Tabelle eine neue Metadatendatei erstellt. Dies geschieht optimistisch nebenläufig. Es können also Metadatendateien von mehreren Transaktionen nebenläufig erstellt werden. Um eine Transaktion abzuschließen wird dann durch einen atomaren Austausch die referenzierte Metadatendatei der Tabelle durch die neue

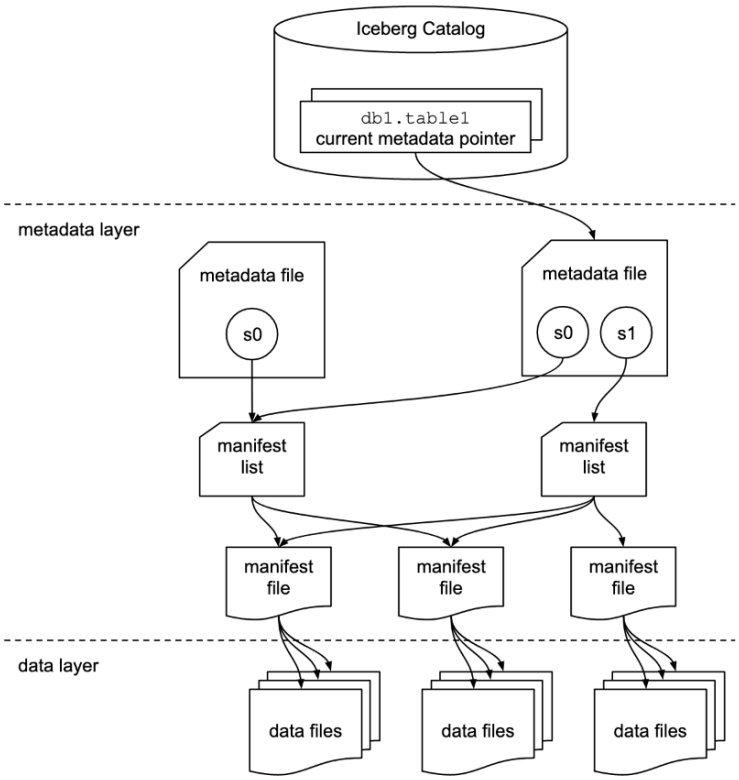


Abbildung 3.2: Aufbau einer Tabelle in Apache Iceberg mit Metadatenschicht.
Direkt übernommen aus der Dokumentation für Apache Iceberg [Apa23h].

[1] Siehe https://avro.apache.org/
[2] Siehe https://flink.apache.org/
[3] Siehe https://hive.apache.org/

ausgetauscht. Wenn die Metadatendatei der Tabelle seit dem Start der Transaktion bereits durch eine andere Transaktion ersetzt wurde, so muss die Transaktion wiederholt werden. Die Metadatendatei beschreibt alle bisherigen Snapshots, also Versionen der Tabelle, sowie weitere Metadaten zur Verwaltung der Tabelle.

Jeder Snapshot besteht aus einer Manifestliste, die definiert, welche Manifestdateien Teil des Inhalts der Tabelle des Snapshots sind. Dabei enthält jede Manifestdatei Referenzen auf eine oder mehrere Dateien aus der Datenschicht. Diese Dateien sind somit Teil der durch die Metadatendatei definierten Tabelle. Eine Manifestdatei kann von mehreren Snapshots referenziert werden. Daher müssen diese nur neu erstellt werden, wenn eine Änderung an den von ihnen referenzierten Dateien auftritt. Weitere Metadaten, in den Manifestdateien und der Manifestliste, werden verwendet, um das Lesen von Dateien zu vermeiden, die in der ausgeführten Transaktion nicht benötigt werden [Apa23h].

Durch den hierarchischen Aufbau der Metadaten können diese beim Ausführen einer Anfrage nur von einem einzelnen Rechenknoten der Processing Engine gelesen werden, der die für die Anfrage relevanten Dateien identifiziert. In Delta Lake hingegen kann das Lesen der Metadaten verteilt über mehrere Rechenknoten erfolgen, da die Metadaten in Tabellenform vorliegen. Das verteilte Vorgehen führt zu einem Overhead. Bei Anfragen, zu deren Beantwortung nur wenige Daten berücksichtigt werden müssen, ist die Performanz daher bei Apache Iceberg besser als bei Delta Lake. Dafür ist bei Anfragen, die mehr Daten berücksichtigen, dann jedoch die Parallelisierung beim Metadatenzugriff für Delta Lake von Vorteil [JKP+23a].

Apache Iceberg garantiert Snapshot-Isolation. Das bedeutet, dass eine Transaktion nur Daten lesen kann, die zum Zeitpunkt des Startes der Transaktion vorlagen. Zudem kann die Transaktion nur abgeschlossen werden, wenn zur Zeit des Commits keine andere Transaktion Daten geschrieben hat, die mit der ersten im Konflikt stehen [JKP+23a].

Optional kann in Apache Iceberg auch Serialisierbarkeit, wie bei Delta Lake, erzielt werden. Serialisierbarkeit stellt zusätzlich sicher, dass das Ergebnis von ausgeführten konkurrierenden Transaktionen immer durch eine serielle Abfolge dieser Transaktionen repräsentierbar ist [JKP+23a].

Apache Iceberg verfügt über viele der Funktionen, die auch Delta Lake anbietet. Dazu gehören Funktionen wie Schema-Evolution, Partitionierung, Time-Travel, Rollbacks und Datenlayout-Optimierungen [Apa23g].

3.4 Apache Hudi

Das dritte der verfügbaren Lakehouse-Frameworks ist Apache Hudi. Apache Hudi wurde 2016 von Uber entwickelt und ist seit 2017, wie auch Apache Iceberg, als Open-Source-Projekt Teil der Apache Software Foundation [Lub21].

Die Datendateien in Apache Hudi, welche die Zeilen der Lakehouse-Tabelle enthalten, können in Formaten wie Apache Parquet, Apache ORC, oder auch HFile[4] von Apache HBase abgespeichert werden [Apa23c]. Auch bei Apache Hudi ist der Zugriff mit vielen verschiedenen Processing Engines möglich, wobei sowohl Batch-, als auch Stream-Processing unterstützt wird [Apa23f].

[4] Siehe https://hbase.apache.org/devapidocs/org/apache/hadoop/hbase/io/hfile/HFile.html

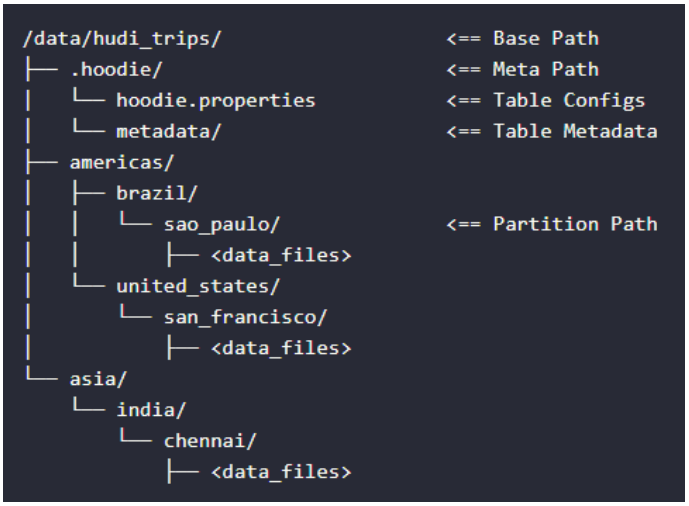


Abbildung 3.3: Aufbau einer Tabelle in Apache Hudi.
Direkt übernommen aus der Dokumentation für Apache Hudi [Apa23c].

Im Folgenden wird der Aufbau des Apache-Hudi-Storage-Formats beschrieben, welches anhand eines Beispiels in Abbildung 3.3 illustriert ist: Es bildet für jede Lakehouse-Tabelle eine Verzeichnisstruktur. Wie auch bei den anderen beiden Lakehouse-Frameworks können Tabellen partitioniert werden. Dabei werden diese in eine entsprechende Verzeichnisstruktur unterteilt, um die Performanz bei der Verarbeitung von Anfragen zu verbessern. Das Verzeichnis `.hoodie` enthält das Transaktionslog und im Unterverzeichnis `metadata` weitere Metadaten. Die Datei `hoodie.properties` speichert allgemeine Konfigurationseinstellungen der Lakehouse-Tabelle, die einmalig bei der Erstellung der Tabelle festgelegt werden. Dies sind Konfigurationen, wie die Namen der Tabelle und der verwendeten Database, nach welchen Spalten partitioniert wird und vieles weiteres [Apa23d]. Die `hoodie.properties`-Datei wird sowohl beim Lesen, als auch beim Schreiben verwendet, um diese Informationen auszulesen. Bei jeder Transaktion werden neue Metadaten-Dateien im `.hoodie`-Verzeichnis erstellt, die sich über einen aufsteigenden Wert im Dateinamen in den Transaktionslog einsortieren. Die Metadaten-Dateien kodieren durch ihren Dateinamen zudem den Fortschritt der Transaktion, also ob diese angefragt, gestartet oder abgeschlossen wurde. Diese Metadaten-Dateien werden atomar erstellt, womit auch Apache Hudi eine optimistische Nebenläufigkeit von Transaktionen ermöglicht [Apa23c].

Im Gegensatz zu Delta Lake und Apache Iceberg bietet Apache Hudi jedoch ausschließlich Snapshot-Isolation und keine volle Serialisierbarkeit. Wie auch bei Delta Lake können die Metadaten bei Apache Hudi verteilt, also über mehrere Knoten hinweg zugegriffen werden [JKP+23a].

Wie auch die anderen beiden Lakehouse-Frameworks bietet auch Apache Hudi viele zusätzliche Funktionen an, wie beispielsweise Time-Travel, Rollbacks, Datenlayout-Optimierungen, Indexing und Schema-Evolution [Apa23e].

3.5 Apache Spark

Apache Spark ist ein Cluster-Computing-Framework zur Verarbeitung großer Datenmengen. Es wurde initial an der Universität Berkeley im Jahre 2009 entwickelt und 2010 als Open-Source-Projekt veröffentlicht. Schließlich wurde es 2013 Teil der Apache Software Foundation [SDC+16].

Apache Spark prozessiert, im Gegensatz zu anderen Frameworks, wie Apache Hadoop, die Daten im Arbeitsspeicher. Dazu verwendet Apache Spark sogenannte Resilient Distributed Datasets (RDDs), welche unveränderliche Sammlungen von Daten darstellen, die auf dem Cluster verteilt sind [VZ22].

Alle drei Lakehouse-Frameworks können mit Apache Spark integriert werden. Die Integration mit Apache Spark ist bei allen drei die am stärksten ausgereifte Integration mit einer Processing Engine [CAG+23]. Über Apache Spark können Lakehouse-Tabellen in diesen Frameworks erstellt, abgerufen und manipuliert werden. Der gesamte Funktionsumfang, einschließlich Time-Travel, Schema-Evolution und vieles weiteres ist anwendbar.

Apache Spark unterstützt sowohl Batch-, als auch Stream-Processing von Daten. Auch können interaktive Anfragen und Machine-Learning-Pipelines verwendet werden. So können zum Beispiel SQL-Anfragen über Apache Spark SQL an das Lakehouse gestellt werden [SDC+16].

Eine Anwendung in Apache Spark kann auf einem Cluster ausgeführt werden, welches von einem Clustermanager wie zum Beispiel Apache Hadoop YARN verwaltet wird [Apa23j].

Abbildung 3.4 veranschaulicht den Aufbau eines solchen Clusters. Der Aufbau setzt sich wie folgt zusammen: Das sogenannte Driver Program, das auf dem Masterknoten des Clusters läuft, enthält den `SparkContext`, welcher Tasks an Executors sendet. Tasks sind Arbeitseinheit, die der Executor dann ausführt. Diese Executors können über mehrere Rechnerknoten verteilt sein. Bei der Ausführung von Aufrufen an Apache Spark, zum Beispiel beim Erstellen einer Lakehouse-Tabelle, wird ein Job erstellt. Der Driver-Prozess erstellt dann einen Ausführungsplan für diesen Job, der aus mehreren Stages besteht. Diese sind wiederum in einzelne, von Executors ausführbare Tasks

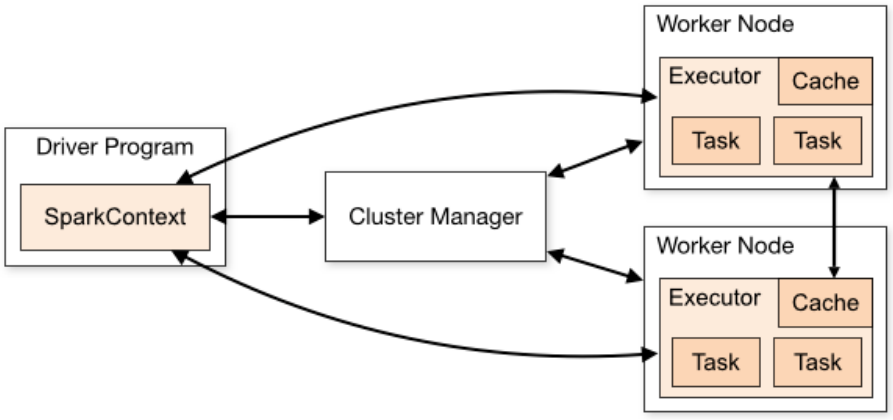


Abbildung 3.4: Architektur von Apache Spark bei Ausführung auf einem Cluster.
Direkt übernommen aus der Dokumentation für Apache Spark [Apa23j].

aufgeteilt. Der Driver-Prozess verteilt dann diese Tasks auf die verschiedenen Executors, welche diese nebenläufig ausführen. Dabei verteilt der Clustermanager die in dem Cluster zur Verfügung stehenden Ressourcen auf die Rechnerknoten [VZ22].

Als Einstiegspunkt verwendet Apache Spark eine Spark Session [VZ22]. Diese kann entsprechend konfiguriert werden, sodass in dieser das gewünschte Lakehouse-Framework verwendet werden kann. Über diese Spark Session kann ein Nutzer dann Tabellen erstellen, lesen und verwalten.

Apache Spark arbeitet mit sogenannten `DataFrames` als Abstraktion einer Tabelle. Wie eine Tabelle ist ein `DataFrame` in Spalten aufgeteilt, die über einen Namen identifiziert werden. Der `DataFrame` enthält in dieser Spaltenstruktur eine über Rechnerknoten verteilte Sammlung an Daten. Über diese Abstraktion des `DataFrames` können mit Apache Spark Tabellen aus verschiedenen Quellen geladen, manipuliert und auf ein Speichersystem geschrieben werden [Apa23l].

3.6 Apache Hadoop Distributed File System

Apache Hadoop ist eine Plattform, die sich aus mehreren Komponenten zusammensetzt. Sie wird für die verteilte Speicherung von Daten und verteilte Datenverarbeitung verwendet [VZ22]. Apache Hadoop wurde im Jahr 2006 ein eigenständiges Projekt, nachdem es zunächst als ein Teil von Apache Nutch[5] entwickelt worden war. Seit 2008 ist Apache Hadoop Teil der Apache Software Foundation [Whi12].

Ein integraler Bestandteil von Apache Hadoop ist das Hadoop Distributed File System (HDFS). Dabei handelt es sich um ein Dateisystem zur Speicherung von Daten mit automatischer Replikation, um ein hohes Maß an Verfügbarkeit zu erreichen [VZ22]. Das Dateisystem kann über mehrere physische Knoten in einem Cluster verteilt sein. Alternativ kann es aber auch in einem pseudo-verteilten Cluster ausgeführt werden. Das heißt das YARN-Cluster verfügt in diesem Fall über einen einzelnen Rechnerknoten, auf dem die verschiedenen Apache-Hadoop-Komponenten des Clusters in separaten Java-Prozessen ausgeführt werden [Apa23a].

Das HDFS verwendet dabei unterschiedliche Arten von Knoten: Mehrere `DataNodes` und mindestens einen `NameNode`. Die `DataNodes` enthalten dabei die tatsächlichen Daten. Der `NameNode` ist für die Verwaltung der Metadaten zuständig und behält den Überblick über die von den `DataNodes` gespeicherten Daten. Clients interagieren ausschließlich mit dem `NameNode`, der dann entsprechend Aufgaben an die `DataNodes` delegiert. Das HDFS teilt die zu speichernden Daten in Blöcke auf. Diese Blöcke sind dabei unabhängig von den Blöcken des zugrundeliegenden Betriebssystems. Blöcke können über mehrere `DataNodes` hinweg repliziert werden, um eine Datenredundanz zu erreichen. Für den Fall, dass der `NameNode` abstürzt, gibt es zusätzlich noch einen sekundären `NameNode`. Dieser wird regelmäßig über den Systemzustand informiert, um gegebenenfalls die Aufgabe des primären `NameNodes` zu übernehmen [VZ22].

Anstelle des HDFSs oder anderer verteilter Dateisysteme kommen häufig auch Cloud Object Stores, wie Amazon S3 oder Google Cloud Storage[6], als Speichersysteme für Data Lakes oder Lakehouses zum Einsatz [AGXZ21]. Diese bieten gegenüber dem HDFS viele Vorteile. So ermöglichen sie

[5] Siehe https://nutch.apache.org/
[6] Siehe https://cloud.google.com/storage

beispielsweise Geo-Replikation, um schnellere ortsunabhängige Antwortzeiten zu erreichen, eine sehr hohe Datenhaltbarkeit, also dass Daten nicht verloren gehen oder unlesbar werden, sowie sehr preiswerte Speicherkosten [ADS+20]. Dafür ist jedoch die Latenz beim Lesen einer Datei in einem Cloud Object Store deutlich höher [AGXZ21].

3.7 Apache Hadoop YARN

Ein weiterer Bestandteil von Apache Hadoop ist Apache Hadoop Yet Another Resource Negotiator (YARN). Dabei handelt es sich um ein Framework für die Verwaltung und Überwachung von Cluster-Ressourcen und für das Planen von Jobs. Diese unterschiedlichen Funktionalitäten sollen dabei voneinander separiert sein, damit diese unabhängig voneinander ablaufen können [VZ22].

Die zentrale Komponente von YARN stellt der `ResourceManager` dar. Dieser verteilt Ressourcen an alle Anwendungen, die im System aktiv sind und erhält neue Anwendungen von Clients. Die erste Teilkomponente des `ResourceManagers`, welche `ApplicationManager` genannt wird, nimmt Aufträge zur Ausführung von Anwendungen entgegen und ordnet diesen erste Ressourcen zu. Für jede der neuen Anwendungen wird zunächst ein `ApplicationMaster`-Prozess gestartet, welcher Ressourcen für die entsprechende Anwendung anfordert. Die zweite Teilkomponente des `ResourceManagers`, der `Scheduler`, verteilt dann diese angeforderten Ressourcen an die Anwendungen, insofern die Ressourcen verfügbar sind. Schließlich befindet sich noch auf jedem physischen Knoten des Clusters jeweils ein `NodeManager`. Dieser beobachtet die aktuelle Nutzung der Ressourcen auf dem entsprechenden Knoten und teilt diese Informationen dem `ResourceManager` mit. Dieser entscheidet dann, anhand dieser Informationen, welche Anwendungen auf welchen Knoten gestartet werden, bzw. welche der bereits laufenden Anwendungen weitere angefragte Ressourcen zugeteilt bekommen [VZ22].

3.8 Apache Hive Metastore

Apache Hive ist eine weitere Processing Engine und kann zur Ausführung von Batch-Anfragen an Daten in Data Lakes und mittlerweile auch in Lakehouses über die Hive Query Language (HiveQL) verwendet werden [VZ22].

Darüber hinaus beinhaltet Apache Hive auch den Hive Metastore. In diesem werden zentralisiert Metadaten zu Tabellen gespeichert, deren Daten sich zum Beispiel auf einem HDFS, Amazon S3 oder Google Cloud Storage befinden. Über die Metastore-Service-API können dann andere Processing Engines, wie Apache Spark und Presto[7], auf die Tabellenmetadaten zugreifen [Apa23b].

Im Hive Metastore werden Metadaten zu den existierenden Datenbanken, Tabellen und Partitionen abgespeichert. Diese enthalten unter anderem eine Liste der in der Tabelle enthaltenen Spalten und geben an, wo die Daten der Tabelle im Speichersystem abgespeichert sind.

[7] Siehe https://prestodb.io/

Mithilfe dieser Informationen ist es nach der initialen Registrierung im Hive Metastore nicht mehr nötig, den Aufbau der Tabelle bei anschließenden Anfragen mitanzugeben [Apa15]. Da im Metastore hinterlegt ist, wo sich die Daten der Tabellen befinden, ist es auch möglich, Tabellen direkt über ihren Namen zu referenzieren. Ohne die Verwendung des Hive Metastores wäre es hingegen notwendig, bei jedem Zugriff auf eine Tabelle den genauen Pfad anzugeben, unter dem ihre Daten gespeichert sind.

4 Verwandte Arbeiten

In diesem Kapitel werden in den Abschnitten 4.1 und 4.2 zwei Artikel betrachtet, in denen Benchmark-Frameworks eingeführt werden, mit denen sich Lakehouse-Frameworks hinsichtlich Performanz gegenüberstellen lassen. Abschnitt 4.3 grenzt anschließend kurz ab, warum diese Benchmark-Frameworks in ihrer gegebenen Form nicht genügen, um verschiedene Ansätze zur Speicherung unstrukturierter Daten in Lakehouses zu vergleichen. Schließlich wird in Abschnitt 4.4 noch ein weiterer Artikel aufgegriffen, welcher ein Benchmark-Framework für Data Lakes vorstellt, in dem unter anderem unstrukturierte Daten verwendet werden.

4.1 LST-Bench

Camacho-Rodríguez et al. [CAG+23] stellen das Framework LST-BENCH[1] vor, das für das Benchmarking unterschiedlicher Lakehouse-Frameworks verwendet werden kann. Dieses wird angewendet, um drei der aktuell relevanten Lakehouse-Frameworks, Delta Lake, Apache Iceberg und Apache Hudi zu vergleichen. Es basiert dabei auf dem für Data Warehouses etablierten TPC-DS[2]-Benchmark. Dieser wird jedoch um einige relevante Aspekte erweitert, die im Kontext von Lakehouses auf Cloud Object Stores wichtig sind. So wird zum Beispiel die Performanz bei nebenläufigen Zugriffen, die Time-Travel-Performanz und die Performanzeinbußen über längere Zeit betrachtet. Dazu werden spezielle Workloads erstellt, die diese Aspekte abdecken sollen. Zusätzlich werden noch weitere Metriken berechnet, wie beispielsweise die Stabilität des Lakehouses über Zeit, um Aspekte wie die Performanzeinbußen zwischen Lakehouses zu vergleichen [CAG+23].

Das Framework LST-BENCH wurde von Microsoft als Open-Source-Projekt veröffentlicht und wird aktiv weiterentwickelt [Mic23]. Das Framework wird als Anwendung direkt auf dem vom Anwender verwendeten Rechner ausgeführt. Die eigentliche Interaktion mit dem Speichersystem findet dann über die verwendete Processing Engine statt. Die Engine und das verwendete Cluster müssen von dem Betreiber des Clusters nach den jeweiligen Bedürfnissen vorkonfiguriert werden. Als Processing Engine können beliebige, von den Lakehouse-Frameworks unterstützte Engines verwendet werden, die sowohl Tabellen lesen, als auch schreiben können. In den in dem Artikel beschriebenen Experimenten wurde Apache Spark verwendet, allerdings können prinzipiell auch andere Engines verwendet werden, wie beispielsweise Trino[3]. Eine Voraussetzung ist jedoch, dass auf die Processing Engine über eine Java Database Connectivity (JDBC)-Verbindung zugegriffen werden können muss. Das Framework führt standardmäßig SQL-Anfragen über diese JDBC-Verbindung auf dem Speichersystem aus. Mittlerweile unterstützt LST-BENCH aber auch die direkte Verwendung einer

[1] Siehe https://github.com/microsoft/lst-bench
[2] Siehe https://www.tpc.org/tpcds/
[3] Siehe https://trino.io/

Spark Session. Ebenso ist es möglich, das Framework um weitere Verbindungsmöglichkeiten zu erweitern, über die die SQL-Anfragen an die Processing Engine weitergeleitet werden können [Cam23]. Dadurch ist das Framework nicht mehr auf JDBC-Verbindungen limitiert.

Die auszuführenden SQL-Anfragen liegen in einer Verzeichnisstruktur als SQL-Dateien vor und werden vom Nutzer beim Starten eines Benchmarks durch übergebene Konfigurationsdateien referenziert. In diesen Konfigurationsdateien werden sowohl Informationen zur Ausführungsumgebung, als auch die genauen Abläufe der auszuführenden Workloads definiert. Mithilfe dieser Konfigurationen werden die unterschiedlichen SQL-Anfragen dann in der definierten Reihenfolge ausgeführt [CAG+23].

Unter Verwendung von LST-BENCH werden die drei relevanten Lakehouse-Frameworks verglichen. In den durchgeführten Experimenten wird ein Spark Cluster mit 16 Worker Nodes verwendet, bereitgestellt von Azure Virtual Machine Scale Sets[4]. Die verwendeten TPC-DS-Datensätze sind auf einem Azure Data Lake Storage[5] gespeichert. Die Lakehouse-Frameworks wurden alle in ihren Standardeinstellungen verwendet [CAG+23].

Camacho-Rodríguez et al. kommen unter anderem zu den folgenden Ergebnissen: Wenn über einen längeren Zeitraum immer mehr Daten und Metadaten zu einer Lakehouse-Tabelle hinzukommen, so fällt die Performanz stark ab, vor allem wenn viele kleine Änderungen durchgeführt werden. Besonders Delta Lake und Apache Iceberg sind davon stark betroffen. Mithilfe von Datenlayout-Optimierungen, die von allen drei Lakehouse-Frameworks angeboten werden, lässt sich dieser Effekt jedoch abschwächen. Apache Hudi führt beim Schreiben von Daten zusätzliche Vorverarbeitungen durch. Dadurch ist zwar die initiale Latenz bei Schreibvorgängen bei Apache Hudi besonders hoch, aber dafür bleibt die Performanz im Anschluss bei Änderungen an der Tabelle stabil. Ein weiteres Ergebnis ist, dass Anpassungen an der Konfiguration von Lakehouses große Trade-offs mit sich bringen können. Es muss also je nach Verwendungszweck entschieden werden, welche speziellen Konfigurationen sinnvoll sind, um das Lakehouse für ein bestimmtes Szenario zu optimieren [CAG+23].

Allgemein weisen Camacho-Rodríguez et al. darauf hin, dass die beobachteten Ergebnisse stark von der verwendeten Processing Engine abhängig sind. Je nachdem, wie gut diese von den unterschiedlichen Lakehouse-Frameworks unterstützt wird, können sich große Unterschiede ergeben. Auch die verwendeten Versionen und Konfigurationen der Lakehouse-Frameworks sind von großem Einfluss auf die Ergebnisse. Durch neuere Versionen können sich die Performanzeigenschaften verändern, wenn die Frameworks weiter verbessert und optimiert werden. Daher sollen die präsentierten Ergebnisse vor allem aufzeigen, dass das LST-BENCH-Framework für den Vergleich von Vor- und Nachteilen verschiedener Konfigurationen geeignet ist, aber keine Bewertung abgeben, welches der Lakehouse-Frameworks sich grundsätzlich am besten eignet [CAG+23].

[4] Siehe https://azure.microsoft.com/en-us/products/virtual-machine-scale-sets/
[5] Siehe https://azure.microsoft.com/en-us/products/storage/data-lake-storage

4.2 LHBench

Jain et al. [JKP+23a] stellen ebenfalls ein Benchmark-Framework vor, genannt LHBench[6]. Mit diesem werden auch die drei Lakehouse-Frameworks Delta Lake, Apache Iceberg und Apache Hudi verglichen, wobei Unterschiede im Design der Lakehouse-Frameworks betrachtet und in Hinsicht auf Performanz und Funktionalität gegenübergestellt werden [JKP+23a].

Das LHBench-Framework ist als Open-Source-Projekt öffentlich auf GitHub zugänglich. Das Framework verwendet Apache Spark als Processing Engine. Dies ist fest im Aufbau des Benchmarks verankert [JKP+23b]. Als Cloud Provider kann standardmäßig Amazon Web Services[7] mit Amazon S3 als Speichersystem und einem Amazon EMR[8] Cluster zur Ausführung der Benchmarks verwendet werden. Alternativ ist es auch möglich, die Google Cloud Plattform mit Google Cloud Storage als Speichersystem und einem Dataproc[9] Cluster für die Ausführung zu nutzen. Eine Erweiterung auf andere Cloud Provider ist ebenso möglich [JKP+23d].

Gemäß dem Artikel von Jain et al. wurde für die Experimente ein EMR Cluster mit 16 Worker Nodes verwendet. Bei den Lakehouse-Frameworks und dem EMR-Cluster wurden keine Änderungen an den Standardeinstellungen vorgenommen. Dies begründen die Autoren damit, dass ein Lakehouse grundsätzlich sehr diverse Aufgaben unterstützen können muss und eine Optimierung hinsichtlich eines bestimmten Anwendungsfalls für das Experiment daher nicht sinnvoll ist. Außerdem vereinfacht ein System, das ohne Optimierungsaufwand für alle Aufgabenbereiche eingesetzt werden kann, die Anwendung durch den Nutzer [JKP+23a].

Das LHBench-Framework bietet unterschiedliche Benchmarks an. Unter anderem kann der TPC-DS-Benchmark verwendet werden, um die Performanz der unterschiedlichen Lakehouse-Frameworks beim Schreiben und Lesen von Daten zu vergleichen. Dabei kommen Jain et al. zu demselben Ergebnis wie auch Camacho-Rodríguez et al. mit LST-BENCH: Apache Hudi ist beim Schreiben von großen Datenmengen in Lakehouse-Tabellen deutlich langsamer als Apache Iceberg und Delta Lake. Dies wird von Jain et al. ebenfalls damit begründet, dass Apache Hudi für das Aktualisieren einzelner Einträge optimiert ist und daher umfangreiche Vorverarbeitungen durchführt, damit alle Einträge über einen eindeutigen Schlüssel identifiziert werden können [JKP+23a].

Im Bezug auf das Lesen von großen Tabellen ist Apache Hudi ebenfalls langsamer als Delta Lake. Dies liegt daran, dass Apache Hudi im Vergleich zu Delta Lake und Apache Iceberg standardmäßig deutlich kleinere Dateien verwendet, um die Tabellendaten abzuspeichern. Dadurch erhöht sich der Overhead, wenn beim Lesen viele kleine Dateien abgerufen werden müssen. Da die Kompression von Tabellenspalten bei kleineren Dateien weniger effektiv ist, benötigt Apache Hudi deshalb auch mehr Speicherplatz für erstellte Tabellen. Apache Iceberg ist beim Lesen von Tabellen auch langsamer als Delta Lake. Dies liegt daran, dass Apache Iceberg einen deutlich langsameren, speziellen, selbst entwickelten Reader für Apache Parquet verwendet und nicht den von Apache Spark bereitgestellten [JKP+23a].

[6] Siehe https://github.com/lhbench/lhbench
[7] Siehe https://aws.amazon.com/
[8] Siehe https://aws.amazon.com/emr/
[9] Siehe https://cloud.google.com/dataproc

Das LHBench-Framework bietet zudem einen TPC-DS-Refresh-Benchmark und einen sogenannten Merge-Micro-Benchmark an. Diese können verwendet werden, um verschiedene Updatestrategien zu vergleichen. Als Updatestrategien können Merge-on-Read (MoR) und Copy-on-Write (CoW) unterschieden werden, die beide von den meisten Lakehouse-Frameworks unterstützt werden. Jain et al. kommen zu dem Ergebnis, dass Merge-on-Read beim Ausführen von großen Merges schneller ist als Copy-on-Write. Im Gegensatz dazu ist Copy-on-Write bei anschließend ausgeführten Anfragen auf der veränderten Tabelle deutlich schneller, da Datendateien mit Änderungen an Einträgen schon beim Schreibvorgang in neue Dateien kopiert werden. Bei Merge-on-Read findet dies hingegen erst beim Lesen der Einträge statt. Da die meisten Anwendungen für die Lakehouses verwendet werden eine hohe Performanz bei Lesezugriffen benötigen, um möglichst schnelle Analysen von Daten zu ermöglichen, unterstützen alle Lakehouse-Frameworks Copy-on-Write. Auf Kosten von höherer Latenz beim Schreiben wird so eine geringere Latenz beim Lesen ermöglicht [JKP+23a].

Das GitHub Repository von LHBench[6] enthält noch weitere Benchmarks, die im Artikel jedoch nicht vorgestellt werden. Außerdem lassen sich auch weitere Benchmarks hinzufügen [JKP+23b].

LHBench unterstützt sowohl die Ausführung von einzelnen SQL-Statements, als auch die Verwendung von ganzen Methoden in denen auf die Spark-API zugegriffen werden kann und misst deren Performanz. Dies ist möglich, da das Framework eine Spark Session zum Zugriff auf das Lakehouse verwendet und diese Session zur Verwendung in den Benchmarks zugreifbar macht [JKP+23b].

4.3 Unterstützung von unstrukturierten Daten

Sowohl LST-BENCH [CAG+23], als auch LHBench [JKP+23a] bauen grundsätzlich auf dem TPC-DS-Benchmark auf. Dieser wurde initial für Data Warehouses entwickelt und verwendet ausschließlich strukturierte Daten. Auch die erweiterten spezielleren Benchmarks, die die beiden Benchmark-Frameworks zu der Grundfunktionalität des TPC-DS-Benchmarks hinzufügen, verwenden weiterhin den TPC-DS-Datensatz und damit keine unstrukturierten Daten. Daher ermöglichen also beide Frameworks in ihrem Grundzustand nicht den Vergleich von verschiedenen Ansätzen, wie unstrukturierte Daten in Lakehouses gespeichert werden können. In den Artikeln von Camacho-Rodríguez et al. und Jain et al. wurde auch über die vorgestellten Benchmarks hinaus nicht betrachtet, wie unstrukturierte Daten in Lakehouses abgespeichert werden können und sollten. Daher besteht die Notwendigkeit, dass diese Arbeit zunächst verschiedene Ansätze zur Speicherung unstrukturierter Daten in Lakehouses identifiziert. Diese Arbeit stellt zudem ein neues Benchmark-Framework, basierend auf einem der beiden bereits existierenden Benchmark-Frameworks für Lakehouses, vor, welches den Vergleich dieser Ansätze ermöglicht. Da die beiden in diesem Kapitel eingeführten Benchmark-Frameworks keine unterschiedlichen Ansätze zur Speicherung unstrukturierter Daten unterstützen, ist dieses neue Benchmark-Framework ebenfalls ein relevanter und neuer Beitrag und ermöglicht dann die Evaluation unterschiedlicher Ansätze zur Speicherung unstrukturierter Daten. Wie die beiden bereits existierenden Benchmark-Frameworks erweitert werden müssen, um solch ein Framework zu erhalten, das den Vergleich der Ansätze zur Speicherung unstrukturierter Daten ermöglicht, wird detailliert in Kapitel 7 ausgeführt.

4.4 DLBench

Sawadogo und Darmont [SD21] stellen ein Benchmark-Framework für Data Lakes vor, genannt DLBench, das sowohl strukturierte, als auch unstrukturierte Daten verwenden kann. Als unstrukturierte Daten werden von DLBench Textdokumente unterschiedlicher Seitenzahl genutzt. Diese Textdokumente werden dabei in einem vorgegebenen Schema im Data Lake abgelegt und mit Metadaten aufbereitet. Das Benchmark-Framework ermöglicht insbesondere nicht den Vergleich von unterschiedlichen Ansätzen, wie die Dokumente im Data Lake gespeichert werden können. Es betrachtet auch nicht die Performanz zum Schreiben von neuen Daten in Data Lakes, sondern nur die Performanz beim Datenabruf, sowie spezielle Methoden der Textanalyse, wie Text-Mining. Damit liegt der Fokus bei diesem Benchmark-Framework nicht auf der Performanz von unterschiedlichen Ansätzen zur Speicherung unstrukturierter Daten, sondern vielmehr, auf der Performanz von Methoden zur Analyse von unstrukturierten Daten, die sich bereits in einem Data Lake befinden. Eine Abwandlung dieses Benchmark-Frameworks zur Verwendung mit Lakehouse-Frameworks ist nicht sinnvoll möglich, zumal das DLBench-Repository nicht frei öffentlich zugänglich ist.

5 Anforderungen an das Benchmark-Framework

Um unterschiedliche Ansätze zur Speicherung unstrukturierter Daten in Lakehouses auf Performanz untersuchen zu können, wird im Rahmen dieser Arbeit ein entsprechendes Framework für Benchmarks entwickelt. Mithilfe dieses Benchmark-Frameworks können dann die verschiedenen Ansätze quantitativ verglichen und anhand von unterschiedlichen Anwendungsszenarien bewertet werden.

Um die Ansätze umfassend quantitativ vergleichen zu können, muss das in dieser Arbeit vorgestellte Benchmark-Framework einige grundlegende Anforderungen erfüllen. Zur Bestimmung der Anforderungen wurden zu Beginn der Arbeit die verschiedenen relevanten Technologien für die Umsetzung und Verwendung eines Lakehouses testweise verwendet, um einfache Performanztests mit unstrukturierten Daten auszuführen, ohne dabei auf ein existierendes Benchmark-Framework aufzubauen. Auf Grundlage dessen und der Betrachtung und Verwendung der bereits existierenden Benchmark-Frameworks für Lakehouses wurde identifiziert, welche Aspekte für ein Benchmark-Framework für unstrukturierte Daten essentiell notwendig sind. Mithilfe dieser Betrachtungen wurden dann die zentralen Anforderungen formuliert. Eine Übersicht über diese ist in Tabelle 5.1 gegeben. Die identifizierten Anforderungen werden im Folgenden im Detail ausgeführt.

A1	Erstellung von Benchmarks, mit: **A1.1**: Unterstützung aller drei Lakehouse-Frameworks **A1.2**: Unterstützung aller Ansätze zur Speicherung unstrukturierter Daten **A1.3**: Unterstützung unterschiedlicher Speichersysteme; vor allem HDFS
A2	Bereitstellung detaillierter Benchmarkergebnisse, mit: **A2.1**: Erfassung von Informationen zur Benchmark- und Umgebungskonfiguration **A2.2**: Erfassung von Ausführungszeiten von Teilaufgaben **A2.3**: Erfassung des Speicherplatzbedarfs der Lakehouse-Tabelle
A3	Visualisierung und Gegenüberstellung der Ergebnisse mehrerer Benchmarks
A4	Sequenzielle Ausführung mehrerer Benchmarks ohne Nebeneffekte
A5	Generierung von Bilderdatensätzen mit unterschiedlichen Bildgrößen

Tabelle 5.1: Anforderungen an das Benchmark-Framework.

Die erste Anforderung besteht darin, dass das Framework die Erstellung von Benchmarks ermöglichen muss (A1).

Dabei sollen diese Benchmarks für unterschiedliche Lakehouse-Frameworks ausführbar sein (A1.1). Nach den Ergebnissen der Evaluation von Schneider et al. [SGL+23] erlauben mit Delta Lake, Apache Iceberg und Apache Hudi mindestens drei Frameworks die Konstruktion von vollwertigen Lakehouses. Diese drei Lakehouse-Frameworks müssen von dem Benchmark-Framework unterstützt werden.

Zudem müssen mithilfe der Benchmarks alle identifizierten Ansätze zur Speicherung unstrukturierter Daten testbar sein (A1.2 vgl. Kapitel 6). Die Ausführung des Benchmarks muss für alle Ansätze gleich ablaufen, um eine Vergleichbarkeit sicherzustellen. Dabei ist sowohl die Performanz von Schreiboperationen, wie auch Leseoperationen zu betrachten, um die Ansätze feingranular vergleichen zu können. Daher sollte sowohl das Schreiben einer Lakehouse-Tabelle mit dem entsprechenden Ansatz, als auch das Lesen von dieser Lakehouse-Tabelle betrachtet werden. Die verschiedenen Ansätze zur Speicherung werden in Kapitel 6 eingeführt.

Das Benchmark-Framework sollte auch unterschiedliche Speichersysteme unterstützen (A1.3). Als Speichersysteme für ein Lakehouse kommen sowohl Cloud Object Stores, wie Amazon S3 oder Google Cloud Storage, als auch verteilte Dateisysteme, wie HDFS, in Frage. Im Rahmen dieser Arbeit werden die Benchmarks auf einer einzelnen virtuellen Maschine ausgeführt. Dabei wird HDFS als Speichersystem verwendet, weshalb das Framework HDFS unterstützen muss. Die Beweggründe für diese Wahl werden in Abschnitt 7.3 ausgeführt. Darüber hinaus sollten auch Cloud Object Stores verwendbar sein. Dies ist sinnvoll, um das Framework dann in Zukunft auch für Benchmarks von größerem Umfang auf solchen Cloud Object Stores verwenden zu können, um eine realistischere Produktivumgebung abbilden zu können.

Die zweite Hauptanforderung besteht darin, dass die Benchmarks nach der Ausführung detaillierte Ergebnisse bereitstellen sollten (A2). Die Ergebnisse müssen dabei genug Metadaten mitliefern, um eine Reproduzierbarkeit und Validierbarkeit dieser Ergebnisse zu ermöglichen. Daher müssen diese ausführliche Informationen zur Benchmark- und Umgebungskonfiguration enthalten (A2.1). Die Umgebungskonfiguration beinhaltet, welche Technologien im Benchmark verwendet wurden und deren exakte Versionen, sowie welche Konfigurationsparameter bei diesen Technologien im Detail gesetzt wurden. Die Informationen zur Benchmarkkonfiguration müssen Aspekte beinhalten, wie welcher Benchmark ausgeführt wurde, welcher Datensatz zur Erstellung der Lakehouse-Tabelle verwendet wurde, wo die Lakehouse-Tabellen abgespeichert wurden und welche Kompressionsmethode verwendet wurde. Alle diese Metadaten sind essentiell, damit die eigentlichen Ergebnisse nachvollziehbar und interpretierbar werden.

Darüber hinaus müssen die eigentlichen Performanz-Ergebnisse zurückgegeben werden (A2.2). Für jede der Teilaufgaben, die im Benchmark ausgeführt werden, muss separat die Ausführungszeit in feiner Granularität aufgezeichnet werden. Teilaufgaben sind Schritte, wie das Erstellen einer Lakehouse-Tabelle aus einem Testdatensatz oder das Anfragen von Daten aus einer Lakehouse-Tabelle. Für alle diese Teilaufgaben sollten separat Ergebnisse erfasst und zurückgegeben werden. In diesem Kontext ist die Aufzeichnung in Millisekunden sinnvoll. Eine höhere Auflösung würde die Aussagekraft der Daten nicht weiter erhöhen.

Schließlich sollen die Benchmarks auch den Speicherplatzbedarf aufzeichnen, den die im Benchmark erstellte Lakehouse-Tabelle auf dem Speichersystem benötigt (A2.3). Anhand dessen können dann die unterschiedlichen Ansätze zur Speicherung unstrukturierter Daten auch dahingehend verglichen werden, wie viel Speicherplatz die zu speichernden Daten bei der Verwendung entsprechender Ansätze benötigen.

Eine weitere Anforderung ist die automatische Generierung von Visualisierungen, die die Ergebnisse mehrerer ausgeführter Benchmarks vergleichen (A3). In einem automatisiert erstellten, zusammenfassenden Report soll aufgelistet sein, welche Benchmarks ausgeführt wurden und wie die Umgebung bei der Ausführung dieser Benchmarks konfiguriert war. Dieser Report soll zudem graphische Darstellungen enthalten, die die Ergebnisse der Benchmarks gegenüberstellen. Die Visualisierungen sollten die Ausführungszeiten der verschiedenen Teilaufgaben, die in den Benchmarks ausgeführt wurden, zeigen und diese Ausführungszeiten zwischen den Benchmarks vergleichen. Darüber hinaus soll eine weitere Visualisierung auch die Unterschiede im Speicherplatzbedarf darstellen. Dieser Report ermöglicht dann eine einfache und direkte Interpretation der Ergebnisse. Es erübrigt sich dann, die benötigten Benchmarkergebnisse zusammenzusuchen und manuell graphische Darstellungen für diese zu erstellen. Stattdessen lässt sich nach der Ausführung einer Reihe von Benchmarks direkt die Analyse der Ergebnisse durchführen.

Um sicherzustellen, dass es bei der Ausführung der Benchmarks zu keinen unerwünschten Interaktionen kommt, welche die Messwerte verfälschen könnten, kommt noch die Anforderung A4 hinzu: Mehrere Benchmarks müssen sequenziell ausführbar sein, ohne dass es dabei zu Nebeneffekten kommt. Es muss möglich sein, eine Reihe an Benchmarks auszuwählen und mit einem Befehl zu starten, sodass diese dann nacheinander ausgeführt werden, ohne die Ausführung jedes Benchmarks separat initiieren zu müssen. Dadurch wird erreicht, dass nach der Ausführung für alle diese gewählten Benchmarks eine gemeinsame graphische Auswertung erstellt werden kann. Dies vereinfacht die Verwendung dahingehend, dass der Nutzer nur einmal zu Beginn auswählen muss, welche Benchmarks er mit welcher Konfiguration ausführen möchte. Im Anschluss liegt dann direkt die Visualisierung zu diesen Benchmarks vor und kann verwendet werden. Bei der sequenziellen Ausführung mehrerer Benchmarks ist es besonders wichtig, dass ein zuvor ausgeführter Benchmark den nachfolgenden Benchmark nicht in seiner Performanz beeinflussen kann. Daher ist es essentiell, dass die Ausführungsumgebung vor der Ausführung des nächsten Benchmarks wieder in den Ausgangszustand zurückversetzt wird, bevor der nächste Benchmark ausgeführt wird, damit jeder Benchmark dieselben Startbedingungen hat.

Eine letzte Anforderung besteht darin, dass Bilderdatensätze mit unterschiedlicher Bildgröße automatisiert generiert werden können sollen (A5). Diese Anforderung ist unabhängig von den vorherigen zu sehen und fügt eine Funktionalität hinzu, die nicht mit der grundsätzlichen Funktionsweise des Benchmark-Frameworks zusammenhängt. Es ist jedoch für die Evaluation sinnvoll, die Performanz der Ansätze zur Speicherung unstrukturierter Daten bei der Verwendung von unterschiedlichen Testdatensätzen zu vergleichen. Deshalb soll betrachtet werden, ob es Unterschiede zwischen den Ansätzen bei der Verwendung von Datensätzen mit unterschiedlichen Bildgrößen gibt. Zum Beispiel ist es vorstellbar, dass sich einer der Ansätze bei Datensätzen mit besonders großen Einzelbildern gegenüber den anderen in der Performanz absetzt. Daher ist die Erstellung von Datensätzen mit Bildern unterschiedlicher Größe sinnvoll, um diese dann in den Benchmarks verwenden zu können. Diese Datensätze sollen mit einem separaten Skript automatisiert generiert werden können.

Wenn ein Framework für Benchmarks all diese Anforderungen erfüllt, kann dieses sinnvoll und einfach verwendet werden, um verschiedene Ansätze zur Speicherung unstrukturierter Daten in Lakehouses quantitativ zu vergleichen. Durch die Verwendung dieses Frameworks lässt sich dann für unterschiedliche Anwendungsszenarien eine begründete Auswahl eines dieser Ansätze treffen.

6 Ansätze zur Speicherung unstrukturierter Daten

Dieses Kapitel grenzt in Abschnitt 6.1 zunächst ab, worum es sich bei unstrukturierten Daten im Detail handelt und weshalb diese für Unternehmen von Relevanz sind. In Abschnitt 6.2 werden dann vier Ansätze zur Speicherung unstrukturierter Daten in Lakehouses eingeführt und beschrieben. Die Ansätze wurden mittels einer Internetrecherche und anschließender testweiser Verwendung mit Apache Spark identifiziert. Schließlich stellt Abschnitt 6.3 die verschiedenen Ansätze auf konzeptioneller Ebene gegenüber und vergleicht diese hinsichtlich qualitativer Aspekte.

6.1 Unstrukturierte Daten

Unstrukturierte Daten sind Daten, die nicht in ein festes Datenschema einsortiert werden können. Diese Daten lassen sich nicht in klare Spalten aufteilen [Ebe+16]. Damit unterscheiden sie sich von strukturierten Daten, die nach einem gewissen, bekannten Schema vorliegen und entsprechend in ein solches spaltenorientiertes Schema eingeordnet werden können [BH06].

Semi-strukturierte Daten wiederum lassen sich, wie unstrukturierte Daten, nicht in ein solches Schema einsortieren. Jedoch enthalten diese Daten Informationen, wie Tags oder Marker, die ihnen eine gewisse Struktur verleihen. Damit lassen sich solche Daten gruppieren oder in Hierarchien einsortieren. Durch die Verwendung dieser weiteren Informationen vereinfacht sich die Analyse von semi-strukturierten Daten [Abi97].

Unstrukturierte Daten hingegen enthalten keine strukturgebenden Informationen. Alternativ kann es sein, dass die Daten zwar gewisse Metadaten enthalten, die Existenz dieser dem Datenanalysten aber nicht bekannt ist [Abi97]. Beispiele für unstrukturierte Daten sind Bilder, Videos, Binärdateien, Textdateien und noch viele weitere. Die größten Teile der von Unternehmen generierten Daten sind unstrukturiert. Solche unstrukturierten Daten können viele Informationen enthalten, die einen Mehrwert für Unternehmen bieten [Ebe+16]. In der Vergangenheit konnten diese aber nur beschränkt von den unstrukturierten Daten profitieren, da es schwierig war, Informationen aus unstrukturierten Daten zu extrahieren. Mit dem Aufkommen von Machine Learning und anderen Advanced-Analytics-Methoden ist es mittlerweile jedoch möglich geworden, auch unstrukturierte Daten gewinnbringend zu analysieren [Mon23].

Unstrukturierte Daten lassen sich nicht direkt in ein relationales Modell überführen, wie es von relationalen Datenbanken oder Data Warehouses verwendet wird [Ebe+16]. In Lakehouses bzw. Data Lakes lassen sich diese Daten jedoch ablegen [AGXZ21]. Im Folgenden werden nun vier Ansätze zur Speicherung unstrukturierter Daten in Lakehouses eingeführt.

6.2 Speicherung von unstrukturierten Daten in Lakehouses

6.2.1 BinaryFileFormat

Um unstrukturierte Daten in einem Lakehouse abzuspeichern, besteht zunächst die Möglichkeit, diese als Binärdatei direkt in einer Lakehouse-Tabelle abzulegen. Die zu speichernde Datei wird also im verwendeten Dateiformat, wie zum Beispiel Apache Parquet, auf dem Speichersystem abgelegt und über die Metadatenschicht des Lakehouses in die Tabelle eingebunden. Die unstrukturierten Daten befinden sich bei diesem Ansatz also direkt mit in der erstellten Tabelle. So können auf diese beim Lesen der Tabelle ohne Umwege direkt zugegriffen werden. Beim Speichern werden dabei Dateien zunächst in Bytearrays konvertiert, falls es sich nicht bereits um Binärdateien handelt. So ist die Verwendung nicht auf Binärdateien beschränkt [Apa23k].

Apache Spark bietet bereits eine Datenquelle an, mit welcher Dateien in dieser Form eingelesen und gespeichert werden können. Diese Datenquelle wird von Apache Spark `Binary File` genannt [Apa23i]. Daher wird dieser Ansatz zur Speicherung unstrukturierter Daten im Folgenden als `BinaryFileFormat` bezeichnet.

Bei der `Binary-File-Datenquelle` werden Dateien als Binärdateien eingelesen und in einen `DataFrame` konvertiert. In diesem `DataFrame` wird jede eingelesene Datei in eine eigene Zeile eingefügt. Der `DataFrame` enthält für jeden Eintrag eine Spalte mit den Rohdaten der Binärdatei, sowie mehrere Spalten mit Metadaten [Apa23i].

Im Detail ist das Datenschema wie folgt aufgebaut (vgl. [Apa23i]):

- `path`: Pfad zur ursprünglichen Datei als String
- `modificationTime`: Zeitpunkt der letzten Änderung der Ursprungsdatei (als Timestamp)
- `length`: Länge bzw. Größe der Ursprungsdatei in Bytes (als Long)
- `content`: Die eingelesene Sequenz von Bytes (die eigentliche Datei)

Listing 6.1 enthält ein Beispiel für die Verwendung des `BinaryFileFormat` mit Apache Spark und der Sprache Scala. Dazu wird beim Einlesen `binaryFile` als Format angegeben. Dadurch werden alle `.jpg`-, `.jpeg`-, `.png`- und `.bin`-Dateien, die im `sourceDataPath` rekursiv gefunden werden, in den zuvor beschriebenen `DataFrame` konvertiert.

Listing 6.1 Einlesen von Dateien im `BinaryFileFormat` mit Apache Spark.

```
val dataFrame = spark.read
      .format("binaryFile")
      .option("recursiveFileLookup", "true")
      .option("pathGlobFilter", "{*.jpg,*.jpeg,*.png,*.bin}")
      // find .jpg, .jpeg, .png and .bin files
      .load(sourceDataPath)
```

Listing 6.2 Abspeichern eines `DataFrames` als Tabelle in einem Delta Lake mit Apache Spark.

```
dataFrame.write
      .format("delta")
      .mode("overwrite")
      .option("path", tableLocation)
      .saveAsTable(tableName)
```

Der erstellte `DataFrame` kann dann über einen `write`-Befehl als eine Lakehouse-Tabelle in einem angegebenen Lakehouse-Framework-Format abgespeichert werden. Listing 6.2 zeigt, wie solch ein `write`-Befehl verwendet werden kann, um den zuvor erstellten `DataFrame` in einem Delta Lake abzulegen.

6.2.2 OnlyMetadataFormat

Ein weiterer Ansatz zur Speicherung unstrukturierter Daten in Lakehouses besteht darin, die eigentlichen Dateien nicht innerhalb einer Lakehouse-Tabelle abzuspeichern. Die Tabelle enthält also, im Gegensatz zum `BinaryFileFormat`, die Dateien nicht direkt als Rohdaten. Stattdessen werden die zu speichernden Dateien separat auf dem Speichersystem abgelegt und von der Lakehouse-Tabelle lediglich über Pfadangaben referenziert. Dazu wird in der Tabelle in einer zusätzlichen Spalte ein Pfad angeben, der auf den tatsächlichen Speicherort der Datei verweist. Beim Zugriff auf eine Datei über eine Lakehouse-Tabelle mit diesem Ansatz muss zunächst der Pfad aus der Lakehouse-Tabelle gelesen und dann über diesen Pfad auf die eigentliche Datei zugegriffen werden. Darüber hinaus können bei diesem Ansatz weitere Metadaten, welche die unstrukturierten Daten näher beschreiben, zur Tabelle hinzugefügt werden.

Dieser Ansatz zur Speicherung unstrukturierter Daten wird als `OnlyMetadataFormat` bezeichnet, da die Lakehouse-Tabelle ausschließlich Metadaten zu den eigentlichen Dateien enthält und nicht die Dateien selbst.

In Apache Spark existiert jedoch keine direkte Unterstützung dieses Ansatzes, wie es beim `BinaryFileFormat` der Fall ist. Daten können so nicht direkt über die Angabe einer speziellen Art von Datenquelle in einen `DataFrame` mit entsprechendem Aufbau geladen werden.

Daher muss dieser Ansatz zur Speicherung unstrukturierter Daten manuell umgesetzt werden. Zunächst wird dazu das gewünschte Tabellenschema festgelegt. In einem zweiten Schritt wird dann eine Tabelle auf Grundlage dieses Schemas erstellt und mit den entsprechenden Informationen zu den abzuspeichernden Dateien befüllt. Dafür kann in Scala die Methode `spark.createDataFrame()` verwendet werden. Dieser Methode werden eine `ArrayList[Row]` mit den Daten zur Befüllung der Tabelle und ein Tabellenschema vom Typ `org.apache.spark.sql.types.StructType` übergeben. Ein Pseudocode der dieses Vorgehen illustriert ist in Listing 6.3 gegeben.

Das Tabellenschema kann beispielsweise wie folgt aufgebaut werden:

- `path`: Pfad zu der eigentlichen Datei auf dem Speichersystem als String
- `modificationTime`: Zeitpunkt der letzten Änderung der Ursprungsdatei zum Zeitpunkt der Erstellung der Lakehouse-Tabelle

Listing 6.3 Einlesen von Dateien im `OnlyMetadataFormat` mit Apache Spark als Pseudocode.

```
val metadataList = retrieveMetadataFromStorageSystem(sourceDataPath)
val dataFrameSchema = StructType(
      Seq(
        StructField("path", StringType),
        StructField("modificationTime", TimestampType),
        StructField("length", LongType),
        StructField("accessTime", TimestampType),
        StructField("owner", StringType)
      )
)

val dataFrame = spark.createDataFrame(metadataList, dataFrameSchema)
```

- `length`: Länge bzw. Größe der Ursprungsdatei in Bytes (als Long)
- `accessTime`: Letzter Zeitpunkt des Zugriffs auf die Ursprungsdatei zum Zeitpunkt der Erstellung der Lakehouse-Tabelle
- `owner`: Der Benutzername des Erstellers der Ursprungsdatei

In dieser Arbeit wurde dieses Schema für das `OnlyMetadataFormat` ausgewählt, da die ersten drei Spalten mit dem `BinaryFileFormat` übereinstimmen. Dadurch lassen sich bei beiden Ansätzen dieselben SQL-Anfragen verwenden, um auf diese Spalten zuzugreifen. Eine andere Benennung der Spalten wäre zwar auch möglich, allerdings könnten dann keine SQL-Anfragen übergreifend bei beiden Ansätzen angewendet werden. Die letzten beiden Spalten repräsentieren exemplarische, zusätzliche Metadaten, die im HDFS zu gespeicherten Dateien abgerufen werden können. Hier wären beliebig weitere Metadaten über die zu speichernden Dateien denkbar.

6.2.3 OnlyMetadataFormatAndFetch

Eine Variation des `OnlyMetadataFormat` ist das `OnlyMetadataFormatAndFetch`. In Bezug auf den Aufbau des Tabellenschemas und die Art, wie die Daten vorliegen, ist dieser Ansatz identisch zum `OnlyMetadataFormat`. Der einzige Unterschied zwischen beiden Ansätzen besteht im Vorgehen beim Lesen von Einträgen aus einer bereits erstellten Lakehouse-Tabelle.

Bei dem zuvor vorgestellten Ansatz werden bei Anfragen nur die Pfade zurückgegeben, die dann das Ergebnis der Anfrage darstellen, und nicht die unstrukturierten Daten selbst. Das ist hier nun anders. Dazu werden beim `OnlyMetadataFormatAndFetch` die Dateien, die von den abgefragten Einträgen referenziert werden, zusätzlich noch in den Arbeitsspeicher geladen. Dadurch simuliert diese Variation, dass die Dateien im Anschluss direkt verwendet werden, wie dies auch beim `BinaryFileFormat` möglich ist. Denn indem bei dem `BinaryFileFormat` die Dateien nach dem Laden aus der Lakehouse-Tabelle direkt als Binärdatei vorliegen, sind diese ebenfalls unmittelbar verwendbar.

Beim `OnlyMetadataFormat` wird dieses Laden der Dateien in den Arbeitsspeicher nicht durchgeführt. Dieser Ansatz simuliert also eher eine Anwendung, bei der die eigentlichen Dateien nicht benötigt werden und ein Lesen der Metadaten bereits ausreichend ist.

In dem Benchmark-Framework, das in dieser Arbeit vorgestellt wird, wird für die Berechnung des Speicherplatzes, den eine erstellte Tabelle benötigt, beim `OnlyMetadataFormatAndFetch` die Größe der Ursprungsdateien mit einberechnet. Beim `OnlyMetadataFormat` wird hingegen nur die tatsächlich erstellte Lakehouse-Tabelle ohne Berücksichtigung der Ursprungsdateien in die Berechnung einbezogen.

6.2.4 ImageFormat

Zusätzlich zu den zwei konzeptionell verschiedenen Ansätzen `BinaryFileFormat` und `OnlyMetadataFormat`, bietet Apache Spark noch eine spezielle Möglichkeit an, um Bilddateien abzurufen und in einen `DataFrame` zu konvertieren. Dies ist die `Image-Datenquelle`. Diese wird durch die Übergabe von `image` als Format bei der Ausführung von Listing 6.4 verwendet. Dabei können ausschließlich Bilder eingelesen werden [Apa23m]. Im Kontext dieser Arbeit wird dieser Ansatz zur Speicherung unstrukturierter Daten als `ImageFormat` bezeichnet.

Wie beim `BinaryFileFormat` werden die Bilder beim `ImageFormat` direkt innerhalb der Lakehouse-Tabelle abgespeichert. Zusätzlich werden jedoch komprimierte Bilder, wie zum Beispiel PNG- oder JPG-Dateien, zuvor in ihre Rohdarstellung dekomprimiert. Die dekomprimierten Bilder werden dann in einer OpenCV[1]-kompatiblen Bytedarstellung in der Tabelle abgespeichert. Dadurch lassen sich die Bilder in Folge direkt mit Modulen wie OpenCV verarbeiten [Apa23m].

Das von Apache Spark verwendete Datenschema für den von der `Image-Datenquelle` erstellten `DataFrame` enthält alle relevanten Metadaten, um das Einlesen der Bilder mit OpenCV zu ermöglichen. Es ist aus einer einzelnen, zusammengesetzten Spalte `image` aufgebaut. Diese hat folgende Unterspalten (vgl. [Dat23a]):

- `image.origin`: Pfad zur ursprünglichen Datei als String
- `image.height`: Höhe des Bildes in Pixel
- `image.width`: Breite des Bildes in Pixel
- `image.nChannels`: Anzahl der Farbkanäle des Bildes
- `image.mode`: Integer, der angibt, wie die Spalte `data` verwendet werden muss
- `image.data`: Die eigentlichen Bilddaten im Binärformat. Enthält einen dreidimensionalen Array mit den Dimensionen `height`, `width` und `nChannels` mit den Farbwerten für blau, grün und rot (BGR), wenn es sich um ein Bild mit drei Farbkanälen handelt. Der Array ist in zeilenweiser Anordnung gespeichert. In Kombination mit der Spalte `mode` von OpenCV interpretierbar.

[1] Siehe https://opencv.org/

Listing 6.4 Einlesen von Dateien im `ImageFormat` mit Apache Spark.

```
val dataFrame = spark.read
.format("image")
.option("recursiveFileLookup", "true")
.option("pathGlobFilter", "{*.jpg,*.jpeg,*.png,*.bin}")
// find .jpg, .jpeg, .png and .bin files
.load(sourceDataPath)
```

Dieser Aufbau unterscheidet sich stark von den anderen Ansätzen zur Speicherung unstrukturierter Daten. Da es für die Ausführung der Benchmarks möglich sein soll, dass SQL-Anfragen, die auf einzelne Spalten der Tabellen zugreifen, für alle Ansätze gleichermaßen ausgeführt werden können, wird daher im Rahmen der Benchmarks in dieser Arbeit das Datenschema angepasst. Dazu wird das zusammengesetzte Feld `image` aufgelöst und dessen untergeordnete Felder als eigenständige Spalten in das Schema aufgenommen. Zudem wird die Spalte `origin` in `path` umbenannt. Damit ergibt sich für das `ImageFormat` der folgende Aufbau:

- `path`
- `height`
- `width`
- `nChannels`
- `mode`
- `data`

Der Inhalt und die Semantik der Spalten ändert sich dadurch nicht. Durch diesen Aufbau verfügen nun alle Ansätze zur Speicherung unstrukturierter Daten über eine Spalte `path`. Daher können SQL-Anfragen bei allen Ansätzen auf diese Spalte zugreifen. Die Spalten `modificationTime` und `length` können beim `ImageFormat` nicht auf einfache Weise hinzugefügt werden, da die `Image-Datenquelle` diese Informationen für die Bilder nicht ausliest. Diese Spalten im Nachhinein zu erzeugen würde einen zeitlichen Mehraufwand bei der Erstellung des `DataFrames` bedeuten und die Ergebnisse der Benchmarks verzerren. Daher lassen sich SQL-Anfragen, die auf diese Spalten zugreifen, nicht auf dem `ImageFormat` durchführen.

6.3 Konzeptionelle Unterschiede der Ansätze

In diesem Abschnitt werden die zuvor eingeführten Ansätze zur Speicherung unstrukturierter Daten auf konzeptioneller Ebene bewertet und gegenübergestellt. Einen Überblick über die im Laufe dieses Abschnitts vorgestellten Unterschiede der Ansätze hinsichtlich qualitativer Aspekte bietet die Tabelle 6.1.

Qualitative Aspekte	BinaryFile-Format	OnlyMetadata-Format	OnlyMetadata-FormatAnd-Fetch	ImageFormat
Diverse Analysemöglichkeiten	●	◔	●	◕
Einfachheit der Verwendung	●	◔	◔	●
Flexibilität des Schemas	◑	●	●	◑
Tracking im Lakehouse	●	○	○	●

● voll unterstützt ◕ überwiegend unterstützt ◑ teilweise unterstützt
◔ wenig unterstützt ○ nicht unterstützt

Tabelle 6.1: Qualitative Bewertung der Ansätze zur Speicherung unstrukturierter Daten.

Beim `OnlyMetadataFormat` werden Metadaten direkt in der Lakehouse-Tabelle abgespeichert. Dadurch ist dieser Ansatz vor allem für Anwendungsfälle geeignet, bei denen nur die Metadaten benötigt werden, nicht aber die eigentlichen Dateien, auf die sich die Metadaten beziehen. Dies ist zum Beispiel bei Anfragen der Fall, die nur die Anzahl oder die Größe der referenzierten Dateien abfragen.

Wenn hingegen auch die eigentlichen Dateien benötigt werden, müssen diese beim `OnlyMetadataFormat` in einem zusätzlichen Schritt, über die Referenzen in der Lakehouse-Tabelle abgerufen werden. Dieses Szenario stellt der Ansatz `OnlyMetadataFormatAndFetch` dar. Beim `BinaryFileFormat` und beim `ImageFormat` liegen die Dateien hingegen direkt nach dem Lesen der Tabelle zur Verwendung vor. Wie sich dieser zusätzliche Verarbeitungsschritt von `OnlyMetadataFormatAndFetch` auf die Performanz beim Lesen von Daten aus einer Lakehouse-Tabelle auswirkt ist durch die Gegenüberstellung mit dem `BinaryFileFormat` und dem `ImageFormat` im Laufe dieser Arbeit zu betrachten.

Je nach Bedarf kann das `OnlyMetadataFormat` auch mit speziellen Metadaten, die für ein bestimmtes Anwendungsszenario relevant sind, ergänzt werden. Dies ist vor allem sinnvoll, wenn die Berechnung dieser Metadaten komplex ist. Beim Anfragen dieser Informationen kann dadurch eine höhere Performanz erreicht werden, als wenn diese Metadaten erst neu berechnet werden müssen. Andererseits wird dadurch das Schreiben von Einträgen in die Tabelle deutlich aufwendiger. Daher lohnt sich dieser Trade-off nur, wenn die Metadaten mehrfach benötigt werden.

Das `ImageFormat` hat den Vorteil, dass die Dateien dekodiert und in ein Schema konvertiert werden, das die direkte Verarbeitung der Dateien mit OpenCV ermöglicht [Dat23a]. Beim Einlesen der Dateien mit OpenCV müssen diese dann nicht mehr dekodiert werden, da diese schon im entsprechenden Format vorliegen. Mit OpenCV können dann effizient Machine-Learning- und Deep-Learning-Modelle auf den Dateien ausgeführt oder trainiert werden. Dafür müssen die Dateien jedoch bereits beim Schreiben in die Lakehouse-Tabelle dekodiert werden. Dies erhöht den Schreibaufwand. Dieser Trade-off ist daher nur dann sinnvoll, wenn schon bei der Erstellung der Tabelle klar ist, dass die Dateien für OpenCV-Anwendungen verwendet werden sollen. Soll auf die Daten mit Hilfe einer anderen Bibliothek als OpenCV zugegriffen werden, welche ein anderes

Format für die Datenverarbeitung verwendet, dann müssen die Daten zunächst in einem weiteren Schritt in das passende Format konvertiert werden. Eine weitere Beschränkung besteht zudem darin, dass das `ImageFormat` nur für Bilder verwendet werden kann [Dat23a]. Andere unstrukturierte Daten können nicht sinnvoll in das OpenCV-kompatible Format konvertiert werden.

In der Folge soll nun die Einfachheit der Verwendung der verschiedenen Ansätze zur Speicherung unstrukturierter Daten verglichen werden. Das `BinaryFileFormat` und das `ImageFormat` werden von Apache Spark direkt unterstützt und können mit einem einzelnen Befehl, wie in Listing 6.1 und Listing 6.4 dargestellt, eingelesen und mit einem weiteren Befehl in eine Lakehouse-Tabelle geschrieben werden. Dadurch sind diese Ansätze sehr leicht zu verwenden.

Das `OnlyMetadataFormat` und damit auch das `OnlyMetadataFormatAndFetch` werden von Apache Spark nicht auf dieselbe Art direkt unterstützt, wie dies bei den anderen Ansätzen der Fall ist. Es bedarf einer manuellen Erstellung. Dazu müssen die zu speichernden Dateien identifiziert, die Metadaten für diese Dateien abgerufen und in ein individuell definiertes Tabellenschema abgespeichert werden. Auch beim Lesen muss zusätzlich noch die eigentliche referenzierte Datei abgerufen werden. Daher ist die Verwendung des `OnlyMetadataFormat` für den Nutzer aufwendiger. Zudem ist dieser bei der Erstellung dazu gezwungen, Entwurfsentscheidungen für das Tabellenschema zu treffen. Auch gibt es durch die Verwendung von individuellen Schemas keinen standardisierten Aufbau der Lakehouse-Tabellen, wie dies beim `BinaryFileFormat` und beim `ImageFormat` der Fall ist. Dadurch können Tabellen unterschiedlich aufgebaut und Spalten inkonsistent benannt sein. Dies erhöht auch hier die Komplexität für Data Scientists bei der Analyse der Daten, wenn der Aufbau der Tabellen nicht ausreichend dokumentiert oder einheitlich gehandhabt wird. Darüber hinaus muss beim Hinzufügen neuer Einträge das festgelegte Schema genau beachtet werden.

Schließlich gilt es noch, die Datenqualität bei unterschiedlichen Aktualisierungsoperationen zu betrachten. Wenn Daten mithilfe des `BinaryFileFormat` oder des `ImageFormat` in einer Tabelle in einem Lakehouse abgespeichert werden, dann können jegliche Updates von Einträgen dieser Lakehouse-Tabelle ohne Probleme ausgeführt werden. Dies ist möglich, da das verwendete Lakehouse-Framework entsprechende Verwaltungsfunktionen unterstützt. Wenn Einträge gelöscht oder aktualisiert werden, so bleiben die Datendateien, in denen die Einträge gespeichert waren, erhalten. Über Time-Travel kann dann ein früherer Zustand der Tabelle einschließlich der früheren Inhalte abgefragt werden. Da sich in diesen Einträgen direkt die eigentlichen unstrukturierten Daten befinden, kann auf diese also auch nach Updates noch zugegriffen werden. Auf diese Weise kann zum Beispiel für Machine-Learning-Anwendungen, der immer gleiche Versionsstand der Daten verwendet werden und so eine Reproduzierbarkeit von zuvor trainierten Modellen ermöglicht werden.

Beim `OnlyMetadataFormat` kann es durch Updates jedoch zu Inkonsistenzen kommen, zum Beispiel wenn die Ursprungsdateien, die von Einträgen in der Lakehouse-Tabelle referenziert werden, verschoben oder gelöscht werden. In diesem Fall ist die in der Tabelle gespeicherte Referenz nicht mehr aktuell. Bei der Verwendung der Tabelle ist dann das Zugreifen auf die Dateien nicht mehr möglich, wodurch die Tabelle unbrauchbar wird. Bei Aktualisierungen ist also zu beachten, dass auch die Referenzen in den Einträgen der Lakehouse-Tabelle korrekt mit aktualisiert werden. Dies erhöht den Aufwand und die Fehleranfälligkeit deutlich. Jedoch ist festzuhalten, dass selbst bei korrekter Aktualisierung der korrespondierenden Einträge das Rekonstruieren von älteren Tabellenzuständen nicht mehr zuverlässig möglich ist. Wenn eine Originaldatei gelöscht, verschoben oder ersetzt wurde, so lässt sich auf diese über Time-Travel nicht mehr zugreifen, da diese nicht mehr dort abliegt, wo der alte Stand des Eintrags dies verortet. Dadurch können beim `OnlyMetadataFormat`

die Time-Travel-Funktionen des Lakehouse-Ansatzes nicht mehr sinnvoll im Zusammenhang mit den unstrukturierten Daten verwendet werden, wenn solche Updates ausgeführt wurden. Es müsste viel manueller, fehleranfälliger Verwaltungsaufwand betrieben werden, um zu erreichen, dass die unstrukturierten Daten mit den Metadaten synchronisiert bleiben. Nur dann könnte eine korrekte Time-Travel-Funktionalität ermöglicht werden.

Da sich die Originaldateien außerhalb des Lakehouses befinden können die Lakehouse-Frameworks diese Daten nicht überwachen. Dementsprechend können für diese keine erweiterten Funktionen durch die Lakehouse-Frameworks bereitgestellt werden. Auch die ACID-Eigenschaften, die Lakehouse-Frameworks ermöglichen, gelten nicht für diese außerhalb des Lakehouses befindlichen Dateien. So laufen Änderungen an den Originaldateien unter anderem nicht atomar ab, sodass inkonsistente Zustände der Daten auftreten können.

7 Methodik

In diesem Kapitel wird die Entwicklung des Benchmark-Frameworks beschrieben, das zur quantitativen Bewertung der unterschiedlichen Ansätze zur Speicherung unstrukturierter Daten verwendet werden kann. Grundsätzlich wurde entschieden, dass das erstellte Benchmark-Framework auf einem bereits existierenden Benchmark-Framework für Lakehouses aufbauen und dieses um die benötigte Funktionalität erweitern soll. Dies ist sinnvoll, da dadurch auf einem etablierten Framework aufgebaut werden kann, mit dem bereits verschiedene Lakehouse-Frameworks auf Performanz hin verglichen werden konnten. Mit der Erweiterung eines bereits vorhandenen Frameworks orientiert sich diese Arbeit hinsichtlich der Methodik an diesem Framework. Auch erhöht die Erweiterung eines existierenden Frameworks die Vergleichbarkeit der erhaltenen Ergebnisse.

Dieses Kapitel beginnt daher in Abschnitt 7.1 mit einer Betrachtung, welches der in Kapitel 4 vorgestellten Benchmark-Frameworks für Lakehouses am besten für eine Erweiterung auf unstrukturierte Daten geeignet ist. Anschließend wird beschrieben, welche Erweiterungen vorgenommen wurden, um das gewählte Benchmark-Framework derartig auszubauen, dass die in Kapitel 5 definierten Anforderungen erfüllt sind. Auch beschreibt der Abschnitt den Aufbau und die Funktionsweise des erstellten Benchmark-Frameworks. Schließlich wird in Abschnitt 7.3 ausgeführt, wie die Testumgebung konfiguriert wurde, die dann für die mit dem Benchmark-Framework durchgeführten Evaluationen genutzt wurde.

Tabelle 7.1 zeigt einen vereinfachten Überblick über das verfolgte Vorgehen zur Entwicklung und Verwendung des Benchmark-Frameworks in der vorgelegten Arbeit.

7.1 Auswahl eines Benchmark-Frameworks

In Kapitel 4 wurden zwei Benchmark-Frameworks für Lakehouses eingeführt: LST-BENCH von Camacho-Rodríguez et al. [CAG+23] und LHBench von Jain et al. [JKP+23a]. Im Folgenden wird eines der beiden Frameworks ausgewählt und dann erweitert, um damit die Performanz der unterschiedlichen Ansätze zur Speicherung unstrukturierter Daten zu vergleichen. Das auszuwählende Framework sollte die zuvor formulierten Anforderungen (siehe Kapitel 5) bestmöglich unterstützen. Die Aspekte der Anforderungen, die das gewählte Framework nicht abdeckt, müssen über eine Erweiterung des Frameworks erfüllt werden können.

Beide Frameworks basieren grundsätzlich auf dem TPC-DS-Benchmark. Dieser ist ein Benchmark mit ausschließlich strukturierten Daten. Um unstrukturierte Daten zu unterstützen, müssten also beide Frameworks in jedem Fall erweitert werden. Dazu ist es notwendig, dass sie einen anderen Datensatz und andere Anfragen verwenden als vom TPC-DS-Benchmark vorgegebenen.

Im ersten Schritt wird zunächst LST-BENCH von Camacho-Rodríguez et al. [CAG+23] hinsichtlich seiner Eignung für die Erweiterung betrachtet. LST-BENCH bietet viele neue Funktionen gegenüber dem TPC-DS-Benchmark. So kann es einen Pool von Verbindungs-Sessions gleichzeitig verwenden,

1.	Identifizierung von Anforderungen an ein Benchmark-Framework für unstrukturierte Daten (Siehe Kapitel 5)
2.	Auswahl eines Benchmark-Frameworks für Lakehouses (Siehe Abschnitt 7.1) **2.1.** Recherche nach Benchmark-Frameworks für Lakehouses **2.2.** Erprobung von zwei Benchmark-Frameworks für Lakehouses **2.3.** Bewertung der Erweiterbarkeit der zwei Benchmark-Frameworks **2.4.** Auswahl eines der zwei Benchmark-Frameworks anhand der Bewertung
3.	Entwicklung des Benchmark-Frameworks basierend auf dem gewählten Framework (Siehe Abschnitt 7.2)
4.	Aufsetzen und Konfiguration einer Testumgebung (Siehe Abschnitt 7.3)
5.	Wahl von auszuführenden Benchmarks (Siehe Kapitel 8)
6.	Ausführen der Benchmarks mit automatischer Erstellung der Ergebnisse (Siehe Kapitel 8)

Tabelle 7.1: Vorgehen zur Entwicklung und Anwendung des Benchmark-Frameworks.

um zu untersuchen, wie konkurrierende Zugriffe die Performanz beeinflussen. Außerdem kann der Performanzeinbußen über Zeit evaluiert werden, wenn Tabellen wiederholt aktualisiert werden [CAG+23]. Diese und weitere Aspekte, die LST-BENCH einführt, sind allerdings für den Vergleich der Ansätze zur Speicherung unstrukturierter Daten weniger relevant. Hier sollten sich sowohl die Performanzeinbußen von Tabellen, als auch die konkurrierenden Zugriffe, unabhängig von dem verwendeten Ansatz, ähnlich verhalten. Diese Metriken sind beim quantitativen Vergleich der unterschiedlichen Lakehouse-Frameworks sinnvoll, nicht aber in Bezug auf die verschiedenen Ansätze.

LST-BENCH kann mit vielen unterschiedlichen Processing Engines verwendet werden, darunter auch Apache Spark. Es kann zudem mit Speichersystemen von unterschiedlichen Providern für den Speicher des Testdatensatzes und der erstellten Lakehouse-Tabellen ausgeführt werden. Die Verwendung von HDFS als Speichersystem ist also leicht möglich. Mit dem Spark Thrift Server[1] kann dann über JDBC auf Apache Spark, das auf einem YARN-Cluster ausgeführt wird, zugegriffen werden. Alternativ unterstützt LST-BENCH mittlerweile auch die direkte Verwendung einer Spark Session. Damit kann direkt über diese Spark Session auf die auf dem YARN-Cluster ausgeführte Instanz von Apache Spark zugegriffen werden.

[1] Siehe https://spark.apache.org/docs/3.3.3/sql-distributed-sql-engine.html

Jedoch können sowohl über die JDBC-Verbindung, als auch über die Spark Session nur SQL-Anfragen ausgeführt werden. Diese Limitierung ist tief in den Aufbau von LST-BENCH eingewoben. In den Konfigurationsdateien, die definieren wie LST-BENCH ausgeführt wird, müssen die auszuführenden SQL-Anfragen in Form von SQL-Dateien festgelegt werden. Diese Dateien werden dann von dem Framework entsprechend des Ablaufs der Workloads in der korrekten Reihenfolge geparst und ausgeführt. Daher lassen sich insbesondere Befehle, welche die Spark-API verwenden, wie zum Beispiel die in Listing 6.1, nicht über dieses Framework ausführen. Ein Umbau, der auch die Ausführung von solchen Befehlen ermöglicht, wäre sehr aufwendig und müsste verändern, wie die auszuführenden Workloads definiert werden. Die Verwendung der Spark-API ist jedoch zwingend notwendig, um die verschiedenen Ansätze zur Speicherung unstrukturierter Daten anwenden zu können. Analog zu Listing 6.1 müssen Daten mit den von Apache Spark bereitgestellten Datenquellen geladen werden. Im Falle des `OnlyMetadataFormat` muss manuell auf das Speichersystem zugegriffen werden, um die notwendigen Metadaten abzufragen. Die Ausführung von solchen Befehlen ist mit Spark SQL nicht möglich. Dies widerspricht der Anforderung A1.2 und daher eignet sich LST-BENCH nicht als Grundlage für das zu entwickelnde Benchmark-Framework.

In einem zweiten Schritt wird nun betrachtet, ob das Framework LHBench von Jain et al. besser geeignet ist. LHBench unterstützt die Lakehouse-Frameworks Delta Lake, Apache Iceberg und Apache Hudi und erfüllt damit A1.1 [JKP+23a].

Die von LHBench unterstützen Benchmarks bauen, ebenso wie LST-BENCH, auf dem TPC-DS-Benchmark auf. Dieser unterstützt keine unstrukturierten Daten [JKP+23a]. LHBench ermöglicht es jedoch, neue Benchmarks zu erstellen, die dann für die verschiedenen Lakehouse-Frameworks ausgeführt werden können. Im Gegensatz zu LST-BENCH wird der Ablauf eines Benchmarks bzw. eines Workloads bei LHBench nicht über Konfigurationsdateien definiert. Für jeden Benchmark wird vielmehr eine separate Scala-Datei verwendet. Diese definiert, welche Aktionen nacheinander ausgeführt werden sollen. Über die Scala-Methode `runQuery` können SQL-Anfragen ausgeführt und deren Ausführungsdauer aufgezeichnet werden. Dabei können die SQL-Anfragen als String übergeben werden. Für ausgeführte Anfragen kann zusätzlich die Dauer des sogenannten Query Plannings gemessen werden. Das Query Planning ist dabei als der Zeitraum zwischen dem Absenden der SQL-Anfrage und dem Start des ersten dazu korrespondierenden Apache-Spark-Jobs, der an der Verarbeitung der SQL-Anfrage beteiligt ist, definiert [JKP+23a]. Das Query Planning ist also die Phase, in der die Processing Engine den Ablauf der Ausführung einer Anfrage plant.

Darüber hinaus können auch ganze Scala-Methoden ausgeführt und deren Ausführungsdauer gemessen werden. Dazu wird eine Higher-Order-Function `runFunc` verwendet, der die auszuführende Scala-Methode übergeben werden kann. LHBench verwendet als Processing Engine Apache Spark und erstellt dazu eine Spark Session. Diese Spark Session ist dann innerhalb der Scala-Dateien des Benchmarks verfügbar. Daher kann in den an `runFunc` übergebenen Scala-Methoden diese Spark Session verwendet werden, um auf die gesamte Spark-API zuzugreifen [JKP+23b]. Innerhalb des Aufbaus des LHBench-Frameworks ist es so insbesondere möglich, Benchmarks zu erstellen, welche die im Rahmen dieser Arbeit zu untersuchenden Ansätze zur Speicherung unstrukturierter Daten implementieren und evaluieren. Demnach kann LHBench derart erweitert werden, dass die Anforderung A1.2 erfüllt wird.

Aktuell unterstützt LHBench die Speichersysteme Amazon S3 und Google Cloud Storage [JKP+23d]. Eine Erweiterung auf weitere Speichersysteme wie HDFS ist jedoch möglich.

Bei der Ausführung eines Benchmarks mit LHBench werden eine Datei mit dem Logoutput der ausgeführten Benchmark-Datei, eine CSV-Datei mit den Performanzergebnissen, sowie ein ausführlicher Report in Form einer JSON-Datei erstellt. Diese Dateien werden auf der ausführenden Maschine lokal abgelegt. Die letzten beiden Dateien werden zusätzlich noch auf dem verwendeten Speichersystem abgespeichert. Der JSON-Report enthält sehr ausführliche Informationen zur Benchmark- und Umgebungskonfiguration, wie auch die detaillierten Ausführungszeiten der Teilaufgaben in Millisekunden [JKP+23b; JKP+23e]. Die von Anforderung A2.3 geforderte Aufzeichnung des Speicherplatzbedarfs der erstellten Lakehouse-Tabellen ist jedoch standardmäßig nicht im Report enthalten. Die Funktionalität zur Berechnung und Ausgabe dieser Information muss also noch implementiert werden.

LHBench ermöglicht bereits, dass mehrere Benchmarks nacheinander ausgeführt werden können [JKP+23c]. Es muss aber noch im Detail beleuchtet werden, welche Maßnahmen umgesetzt werden müssen, um sicherzustellen, dass sich nacheinander ausgeführte Benchmarks nicht gegenseitig beeinflussen können, um damit Anforderung A4 zu erfüllen. Eine automatische Visualisierung und Gegenüberstellung von ausgeführten Benchmarks, wie in der Anforderung A3 gefordert, muss ebenfalls noch erstellt werden. Auch ist entsprechend der Anforderung A5 ein einfaches Skript zur Erstellung von Bilderdatensätzen mit Bildern unterschiedlicher Größe zu implementieren.

Die nötigen Erweiterungen sind mit LHBench, im Gegensatz zu LST-BENCH, allesamt umsetzbar. Daher wurde LHBench ausgewählt, um auf dessen Grundlage ein Benchmark-Framework für den Vergleich von verschiedenen Methoden zur Speicherung unstrukturierter Daten umzusetzen. Im nächsten Abschnitt wird im Detail beschrieben, wie die notwendigen Erweiterungen umgesetzt wurden, um alle Anforderungen an das Framework zu erfüllen.

7.2 Erweiterung von LHBench

Für die Erweiterung von LHBench wurde ein Fork des Repositorys auf GitHub erstellt. Auf diesem Fork *LHBench-UnstructuredData* wurden dann die im Folgenden beschriebenen Erweiterungen umgesetzt.

In einem ersten Schritt wurde die Unterstützung von HDFS als Speichersystem hinzugefügt. Dazu wird durch ein übergebenes Argument bei der Ausführung der Benchmarks YARN als `spark.master` gesetzt, wodurch Apache Spark auf einem YARN-Cluster ausgeführt wird. Dafür muss ein entsprechendes YARN-Cluster bereits eingerichtet sein.

Im zweiten Schritt wurde die Anforderung A1.2 umgesetzt, indem Benchmarks für die verschiedenen Ansätze zur Speicherung unstrukturierter Daten auf Lakehouses implementiert wurden. `LHBench-UnstructuredData` unterstützt weiterhin die bereits in LHBench implementierten Benchmarks, fügt jedoch weitere hinzu. Dazu wurden neue Benchmark-Dateien in Scala erstellt. Diese werden dann dediziert ausgeführt. Einen einfachen Überblick über diese bietet das in Abbildung 7.1 dargestellte Klassendiagramm. Jede der Benchmark-Dateien implementiert einen separaten Benchmark. Im Rahmen dieser Arbeit entspricht dies je einem der Ansätze zur Speicherung unstrukturierter Daten.

Eine zusätzliche Datei `FormatBenchmark.scala` wird als Abstraktion verwendet. Diese enthält eine abstrakte Klasse, die von den speziellen Ansätzen zur Speicherung unstrukturierter Daten erweitert wird. Die Klasse definiert den allgemeinen Ablauf der Benchmarks für unstrukturierte Daten. Dafür

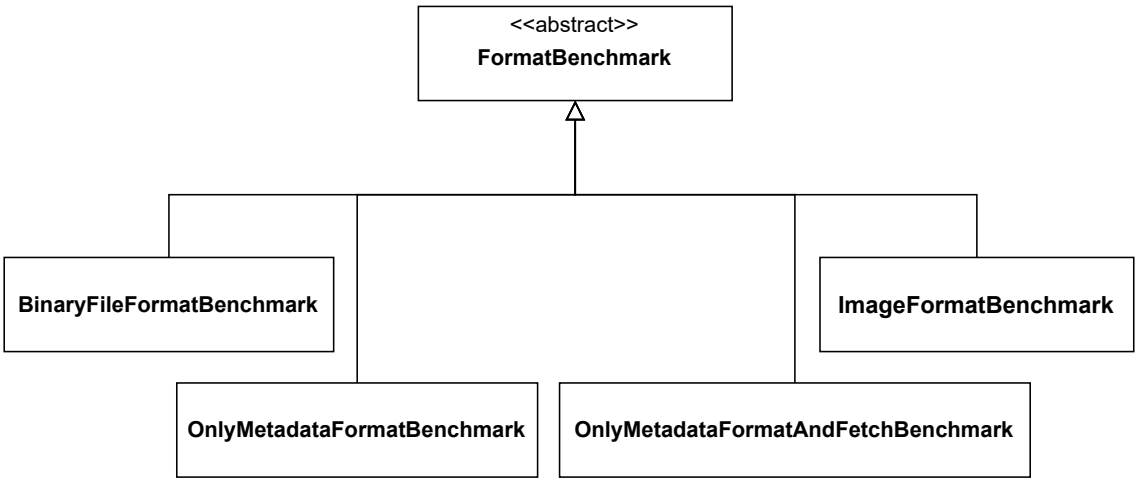


Abbildung 7.1: Überblick über die erstellten Benchmarks.

gibt die Methode `runInternal` an, welche Schritte nacheinander ausgeführt und aufgezeichnet werden. Jeder zu messende Ausführungsschritt wird dabei von einer separaten Methode umgesetzt, die entweder an die Methode `runQuery` oder `runFunc` übergeben wird, die dann die übergebene Methode aufruft und deren Ausführungsdauer aufzeichnet. Die speziellen Methoden unterscheiden sich für die zu untersuchenden Ansätze und werden daher in der Basisklasse nur als abstrakte Methoden definiert, um dann von den erbenden Klassen unter Verwendung des entsprechenden Ansatzes implementiert zu werden. Durch diesen Aufbau lässt sich eine feste Abfolge an Ausführungsschritten mit allen verschiedenen Ansätzen auf dieselbe Weise ausführen. Zudem muss dieser Ablauf somit nur einmal definiert werden. In den einzelnen Benchmark-Dateien müssen dann nur noch die Methoden für die Ausführungsschritte individuell implementiert werden. Darüber hinaus ist auch das Parsen von übergebenen Argumenten in die `FormatBenchmark`-Datei ausgelagert. Dadurch sind diese für alle Benchmarks, die auf unstrukturierten Daten operieren, generalisiert. Dieser Aufbau ermöglicht das einfache Hinzufügen weiterer Benchmarks, die Variationen der eingeführten Ansätze zur Speicherung unstrukturierter Daten oder neue Ansätze implementieren.

7.2.1 Teilschritte der `runInternal`-Methode

Nun wird ausgeführt, welche Schritte die Methode `runInternal` der abstrakten Klasse `FormatBenchmark` nacheinander ausführt. Ein Überblick über die `runInternal`-Methode ist als vereinfachter Kontrollflussgraph in Abbildung 7.2 ersichtlich.

Zunächst werden alle Lakehouse-Tabellen, welche sich noch aus vorhergehenden Ausführungen auf dem Speichersystem befinden, gelöscht. (`createSparkDB`). Dadurch beginnt jeder Benchmark auf einem leeren, noch ungebrauchten Speichersystem.

Als nächstes wird ein Warmup für das verwendete Cluster ausgeführt (`runClusterWarmup`). Dabei wird ein `DataFrame` mit einer Spalte erstellt, der 100 Millionen Einträge enthält und als Lakehouse-Tabelle abgespeichert. Damit ist sichergestellt, dass etwaige Initialisierungsprozesse für das Cluster oder die

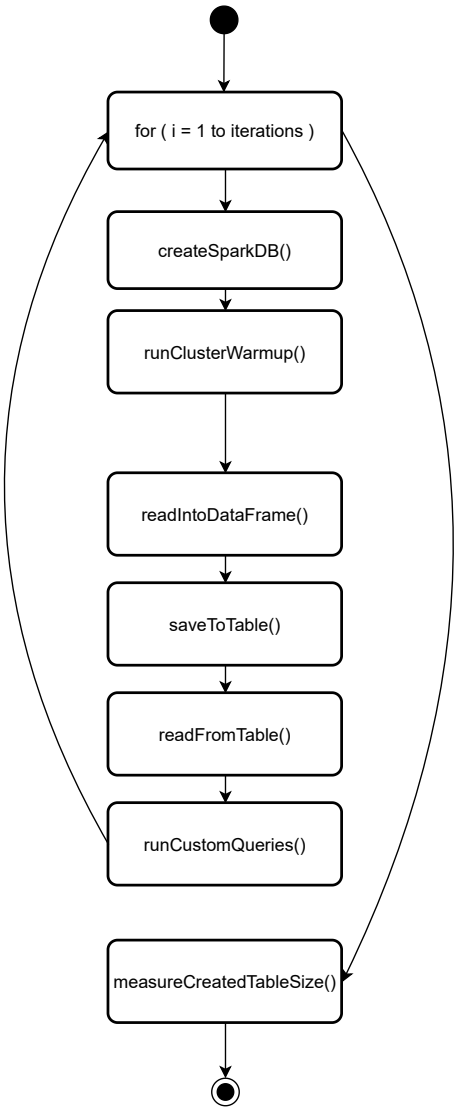


Abbildung 7.2: Vereinfachter Ablauf der Methode `runInternal` in `FormatBenchmark` als Kontrollflussdiagramm.

erstellte Spark Session bereits vor der Ausführung der Benchmarks abgeschlossen werden. Auch dies trägt dazu bei, dass jeder ausgeführte Benchmark im selben Ausgangszustand gestartet wird. Damit wird die Vergleichbarkeit und Reproduzierbarkeit der ausgeführten Benchmarks erhöht.

Nach diesen Initialisierungsschritten werden dann die eigentlichen, Benchmarkschritte ausgeführt. Als erstes wird eine Methode `readIntoDataFrame` ausgeführt, wozu sie an die Higher-Order-Function `runFunc` übergeben wird. In dieser Methode wird der Testdatensatz entsprechend dem verwendeten Ansatz zur Speicherung unstrukturierter Daten in einen `DataFrame` geladen. Wie dies für die unterschiedlichen Ansätze ausgeführt werden kann, wurde bereits in Abschnitt 6.2 aufgezeigt.

Die nächste Methode `saveToTable` verwendet diesen erstellten `DataFrame` und speichert ihn als Lakehouse-Tabelle auf dem Speichersystem ab. Dabei wird für die Lakehouse-Tabelle das Format des im Benchmark verwendeten Lakehouse-Frameworks verwendet, also zum Beispiel Delta Lake.

Die Kombination dieser beiden Methoden entspricht somit dem Schreiben eines Testdatensatzes in eine Lakehouse-Tabelle und kann somit die Dauer des Schreibvorgangs der verwendeten Ansätze zur Speicherung unstrukturierter Daten messen.

Als nächstes wird von `readFromTable` eine SQL-Anfrage ausgeführt. Dabei wird die gesamte zuvor erstellte Lakehouse-Tabelle in einen `DataFrame` eingelesen und durch Ausführung des `collect`-Befehls als Array in den Spark-Driver-Prozess geladen. Dieser Teilschritt kann somit die Dauer des Lesevorgangs des verwendeten Ansatzes zur Speicherung unstrukturierter Daten messen.

Darüber hinaus besteht auch die Möglichkeit, manuell beim Start des Benchmarks speziellere SQL-Anfragen zu übergeben, die im Ablauf von `runInternal` dann als nächstes abgearbeitet werden (`runCustomQueries`). Details zur Implementierung und Verwendung dieser manuellen Anfragen folgen später in diesem Abschnitt.

Nachdem alle Messungen in `runInternal` durchgeführt wurden, wird schließlich noch der Speicherplatz, den die zuvor erstellte Lakehouse-Tabelle auf dem Speichersystem einnimmt, mit der Methode `measureCreatedTableSize` gemessen und dem Report hinzugefügt. Dazu wird der verwendete Speicherplatz des Verzeichnisses, das die Daten der Lakehouse-Tabelle enthält, auf dem HDFS abgefragt. Mit dieser Methode wird die Anforderung 2.3, welche besagt, dass die Benchmarkergebnisse den Speicherplatzbedarf der erstellten Lakehouse-Tabelle beinhalten müssen, umgesetzt.

Bei der Ausführung eines Benchmarks kann zudem übergeben werden, wie oft dieser wiederholt werden soll. Die ersten sechs Schritte von `runInternal` werden dann entsprechend dem übergebenen Wert wiederholt ausgeführt und im Report aufgeführt. Bei der Visualisierung der Ergebnisse kann dann für jeden Benchmark der Mittelwert der ausgeführten Wiederholungen dargestellt werden. Dadurch lässt sich die Zuverlässigkeit der Ergebnisse erhöhen. So kann es zum Beispiel vorkommen, dass die Auslastung des verwendeten Clusters schwankt, gerade wenn das Cluster aus einer einzelnen virtuellen Maschine besteht und die zugrundeliegenden physischen Maschinen auch noch für weitere Anwendungen genutzt werden. Außerdem können Schwankungen auftreten, wenn in einem Cluster, das aus mehreren Knoten aufgebaut ist, die Verbindungsqualität des Netzwerks zwischen den Knoten fluktuiert.

Die neu erstellten Benchmarks müssen zusätzlich noch in den Dateien `run-benchmark.py` und `scripts/benchmarks.py` definiert werden.

7.2.2 Übergebene Argumente an die `FormatBenchmark`-Datei

Der Start des Benchmark-Frameworks erfolgt durch die Ausführung des Skripts `run-benchmark.py`. Diesem können mehrere Argumente zur Konfiguration der einzelnen Benchmarks übergeben werden. Manche dieser Argumente werden direkt von `run-benchmark.py` bzw. `benchmarks.py` verwendet, um die allgemeinen Umgebungskonfigurationen sowie die Auswahl der auszuführenden Benchmarks festzulegen. Weitere Argumente werden weitergegeben und dann in den eigentlichen Scala-Dateien des Benchmarks verwendet.

Die `FormatBenchmark`-Datei erwartet die folgenden Argumente:

- `source-data-path`: Gibt den Pfad auf dem Speichersystem an, unter dem der Testdatensatz abgespeichert ist, der für die Benchmarks verwendet werden soll.

- `benchmark-path`: Der Pfad auf dem Speichersystem, unter dem dann während der Ausführung der Benchmarks die Lakehouse-Tabellen erstellt werden.
- `format`: Gibt an, welches Lakehouse-Framework für die Lakehouse-Tabellen verwendet werden soll. Dieses Argument wird durch die Benchmarks in `benchmarks.py` definiert und muss nicht vom Nutzer direkt übergeben werden.
- `iterations`: Optionales Argument, das angibt, wie oft die Benchmarks wiederholt werden sollen. Ist standardmäßig auf eins gesetzt.
- `debug`: Kann genutzt werden, um das Spark-Log-Level von `Warn` auf `Info` zu setzen, um detailliertere Ausgaben in den Logs zu erhalten.
- `compression`: Wird verwendet, um festzulegen, welche Kompressionsmethode für Parquet-Dateien verwendet werden soll. Entsprechend diesem übergebenen Wert wird dann der Kompressionscodec in der Konfiguration von Apache Spark gesetzt. Der Standardwert hierfür ist `Snappy`[2]. Weitere Optionen sind zum Beispiel `Gzip`, `LZO` oder auch die Nutzung keines Kompressionsverfahrens. Mithilfe dieses Arguments können also auch unterschiedliche Kompressionsmethoden verglichen werden.
- `custom-queries`: Kann genutzt werden, um die manuell übergebenen SQL-Anfragen einzulesen. Diese Anfragen können bei der Übergabe mit einem Namen bezeichnet werden. Die ausgeführten Anfragen werden dann auch mit diesem Namen in den später generierten Visualisierungen aufgeführt. In den übergebenen SQL-Statements können bestimmte Platzhalter durch String-Interpolation zur Laufzeit durch korrespondierende Variablen ersetzt werden. Alle Felder, die in der `FormatBenchmark`-Datei definiert werden, können in den SQL-Statements verwendet werden, indem diese von `${}` umschlossen werden. So kann zum Beispiel über die Verwendung von `${tableName}` die vom Benchmark erstellte Lakehouse-Tabelle im SQL-Statement referenziert werden. Durch Verwendung dieses `custom-queries` Arguments können somit speziellere Anfragen erstellt werden, die über das Auslesen der gesamten Tabelle hinausgehen. So können beispielsweise nur solche Einträge abgefragt werden, deren enthaltene bzw. referenzierte Datei eine bestimmte Größe hat oder zu einem bestimmten Zeitpunkt erstellt oder modifiziert wurde.
- `number-partitions`: Legt fest, in wie viele Partitionen die in den Lakehouse-Tabellen gespeicherten Daten aufgeteilt werden sollen.

7.2.3 Visualisierungen

Im nächsten Schritt wird auf die Umsetzung der automatisierten Visualisierung und Gegenüberstellung von Ergebnissen mehrerer Benchmarks eingegangen (Anforderung A3). LHBench erlaubt es, mehrere Benchmarks an das `run-benchmark.py`-Skript zu übergeben, die dann nacheinander ausgeführt werden. Einige relevante Maßnahmen zur Sicherstellung, dass diese Ausführungen sich nicht gegenseitig beeinflussen, wurden in diesem Abschnitt schon beschrieben.

[2] Siehe https://github.com/google/snappy

Die Funktionalität von LHBench wurde um die automatische Generierung von Visualisierungen ergänzt. Dazu wird, nachdem alle Benchmarks abgeschlossen sind, ein neues Pythonskript `scripts/graphicalReport.py` aufgerufen. Dieses Skript erstellt einen zusammenfassenden Report über alle ausgeführten Format-Benchmarks hinweg. Für die Erstellung des zusammenfassenden Reports werden die JSON-Dateien abgefragt, die zuvor für jeden der ausgeführten Benchmarks erstellt wurden. Dazu werden diese über die IDs der Benchmarks identifiziert. Der zusammenfassende Report wird dabei in Form einer Markdown-Datei erstellt. Dieser Markdown-Datei werden zunächst allgemeine Informationen hinzugefügt, indem diese den JSON-Dateien entnommen werden. Diese Informationen sind die Liste der ausgeführten Benchmarks, wie diese konfiguriert wurden und die wichtigsten Umgebungsinformationen.

Im nächsten Schritt werden unterschiedliche Visualisierungen, basierend auf den Daten in den JSON-Dateien, mit Matplotlib[3] erstellt und der Markdown-Datei hinzugefügt. Beispiele für diese Visualisierungen können in Kapitel 8 eingesehen werden. Es werden vier Arten von Visualisierungen generiert:

Bei der ersten Art von Visualisierung handelt es sich um ein Säulendiagramm, das die Ausführungszeiten für alle ausgeführten gemessenen Teilschritte der Benchmarks zeigt. Die Teilschritte sind auf der X-Achse aufgetragen. Für jeden Teilschritt werden die verschiedenen verwendeten Ansätze zur Speicherung unstrukturierter Daten gegenübergestellt. Diese Art von Visualisierung enthält immer nur die Benchmarkergebnisse für eines der Lakehouse-Frameworks, wie zum Beispiel die Werte von allen Benchmarks, die mit Delta Lake ausgeführt wurden. Für jedes verwendete Lakehouse-Framework wird also jeweils eine separate Visualisierung generiert und dem Report hinzugefügt. Aufgrund dieses Aufbaus eignet sich diese Visualisierung, um die Performanz unterschiedlicher Ansätze für jeden der einzelnen Teilschritte untereinander zu vergleichen. Dies ermöglicht Aussagen über das Abschneiden der Ansätze in unterschiedlichen Aufgabenfeldern, wie Lese- und Schreibperformanz.

In einer Variante dieser Art von Visualisierung wird identisch vorgegangen, doch die Ausführungszeiten für Query-Planning-Teilschritte werden aus der Darstellung ausgelassen. Damit können bei dieser Variante, wie bei der vorherigen Visualisierung, die unterschiedlichen Ansätze hinsichtlich der einzelnen Teilschritte untereinander verglichen werden, außer dass in dieser Variante die Query-Planning-Teilschritte nicht abgebildet sind. Dadurch erhöht sich die Übersichtlichkeit der Visualisierung und es können prägnant die für die Betrachtung wichtigen Teilschritte abgebildet werden.

Im Unterschied zu den ersten beiden Arten von Visualisierungen werden bei einer weiteren Art für jeden der Teilschritte die verschiedenen Lakehouse-Frameworks gegenübergestellt. Jede dieser Visualisierungen enthält nur die Benchmarks für einen einzelnen Ansatz zur Speicherung unstrukturierter Daten. Damit gibt es von dieser Art von Visualisierung im Report je eine Visualisierung pro unterschiedlichem Ansatz. Diese Art von Visualisierung erlaubt somit die Gegenüberstellung der Performanz-Metriken der unterschiedlichen Lakehouse-Frameworks in den Teilschritten. Damit kann mit dieser Art von Visualisierung die Performanz der unterschiedlichen Lakehouse-Frameworks im Bezug auf den ausgeführten Workload verglichen werden. Auch bei diesen Visualisierungen kann das Query Planning ausgelassen werden.

[3] Siehe https://matplotlib.org/

Als viertes wird auch noch separat für jedes der Lakehouse-Frameworks eine Gegenüberstellung des Speicherplatzbedarfs der unterschiedlichen Ansätze erstellt. Auch hierbei handelt es sich um ein Säulendiagramm. Auf der Y-Achse ist der Speicherplatzbedarf der im Benchmark erstellten Lakehouse-Tabelle in Megabytes aufgetragen. Mit diesen Visualisierungen kann verglichen werden, wie viel Speicherplatz die unterschiedlichen Ansätze benötigen.

Die Erstellung dieser graphischen Darstellungen läuft komplett automatisiert ab. Sie bedarf keiner vor- oder nachträglichen Konfiguration. Es können dabei jegliche der Benchmarks für die unterschiedlichen Ansätze ausgeführt werden, ebenso kann auch jedes der drei unterstützen Lakehouse-Frameworks verwendet werden. Diese werden automatisch zu dem Report hinzugefügt. Um beispielsweise die Ansätze `OnlyMetadataFormat`, `BinaryFileFormat` und `ImageFormat` mit den Lakehouse-Frameworks Delta Lake und Apache Iceberg zu testen, kann diese Konfiguration dem Ausführungsbefehl `run-benchmark.py` übergeben werden. Dann werden all diese Benchmarks in den Report miteinbezogen. Die Visualisierung, die nur die Ergebnisse für Apache Hudi enthält, wird dann zum Beispiel ausgelassen, da kein Benchmark mit Apache Hudi ausgeführt wurde. Ohne Probleme können auch Benchmarks mit neuen Ansätzen zur Speicherung unstrukturierter Daten, sowie für gegebenenfalls neu aufkommende Lakehouse-Frameworks hinzugefügt werden. Für diese werden dann ohne jegliche Umstellung die Visualisierungen automatisch miterstellt. Zudem ist es unerheblich, welche Teilschritte innerhalb der Benchmarks ausgeführt und gemessen werden. Das `graphicalReport.py`-Skript fügt automatisch alle Teilschritte auf der X-Achse einer Visualisierung hinzu, die in mindestens einer der JSON-Dateien enthalten sind und für die entsprechende Visualisierung benötigt werden. Es ist also insbesondere auch möglich, dass manche Benchmarks die Performanz für Teilschritte aufzeichnen und diese dann in den Visualisierungen vorkommen, während andere Benchmarks diese Teilschritte nicht enthalten. Alle Visualisierungen werden automatisch passend skaliert, um alle Werte sinnvoll anzeigen zu können.

7.2.4 Skript zur Generierung von Bilderdatensätzen

Als letzter Schritt wurde noch ein einfaches Skript zur Generierung von Bilderdatensätzen erstellt. Dieses Skript erlaubt bei der Ausführung die Erstellung von Bilderdatensätzen mit Bildern unterschiedlicher Bildgrößen, mit denen dann die Performanz der Ansätze zur Speicherung unstrukturierter Daten bei Verwendung dieser Datensätze verglichen werden kann. Es können entweder 100 Bilder mit je ungefähr 10 Megabyte Größe oder zehn Bilder mit je 100 Megabyte Größe erstellt werden. Dadurch ergeben sich zwei unterschiedliche Datensätze, die jeweils eine Gesamtgröße von einem Gigabyte aufweisen. In den erzeugten Bildern wird dabei der Wert für jeden Pixel zufällig gesetzt. Ein Beispiel ist in Abbildung 7.3 abgebildet. Dieses Skript `utilities/datasetGenerator/datasetGenerator.py` wurde dem Benchmark-Framework `LHBench-UnstructuredData` hinzugefügt.

7.3 Anwendung des Benchmarks

In diesem Abschnitt der Arbeit wird der Aufbau der Ausführungsumgebung, die im Kontext dieser Arbeit für die Evaluierung der Ansätze zur Speicherung unstrukturierter Daten verwendet wird, sowie der Ablauf der Ausführungen von `LHBench-UnstructuredData` in dieser Umgebung beschrieben. Ein Überblick darüber ist in Abbildung 7.4 gegeben.

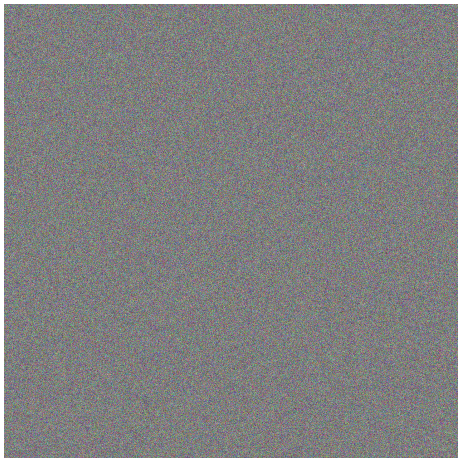


Abbildung 7.3: Von `datasetGenerator.py` zufällig generiertes Bild.

LHBench verwendet einen Aufbau, bei dem über das eigentliche Cluster hinaus noch eine weitere physische oder virtuelle Maschine notwendig ist, von der aus die Benchmarks gestartet werden. Auf dieser Maschine, in der Abbildung als `Lokale VM` bezeichnet, führt der Anwender das Skript `run-benchmark.py` aus, um potentiell mehrere Benchmarks zu starten (Schritt 1 in Abbildung 7.4). Dadurch werden nacheinander die auszuführenden Benchmarks konfiguriert und die zum jeweiligen Benchmark gehörende Scala-Datei kompiliert. Diese kompilierte Datei wird via Secure Shell (SSH) auf den Masterknoten des eigentlichen Clusters kopiert (Schritt 2). Auf diesem Cluster wird über einen `spark-submit`-Befehl die Datei ausgeführt, die den eigentlichen Benchmark enthält. Bei dem Cluster kann es sich bei LHBench entweder um ein Cluster von Amazon EMR oder Dataproc handeln. Wie in Abschnitt 7.2 beschrieben, erlaubt `LHBench-UnstructuredData` allerdings auch die Verwendung eines YARN-Clusters. Für die in dieser Arbeit durchgeführte Evaluation wird ein pseudo-verteiltes YARN-Cluster verwendet. Dieses YARN-Cluster besteht also aus einem einzelnen Rechnerknoten. Die verschiedenen Apache-Hadoop-Komponenten des Clusters werden in separaten Java-Prozessen ausgeführt [Apa23a]. Dieser Rechnerknoten wird in Abbildung 7.4 als

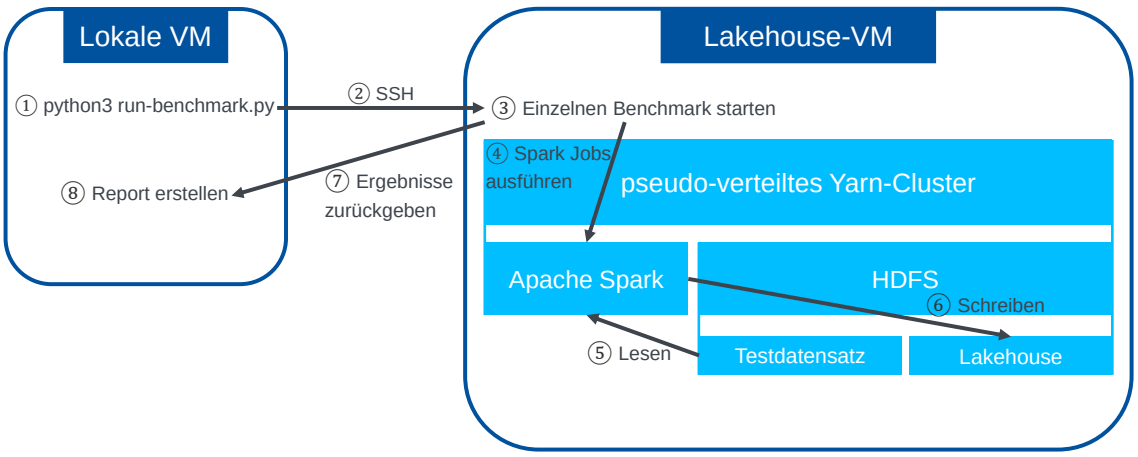


Abbildung 7.4: Überblick über den Aufbau der Ausführungsumgebung und den Ablauf einer Ausführung.

Listing 7.1 Beispiel für die Ausführung von `run-benchmark.py` mit verschiedenen Argumenten.

```
python3 run-benchmark.py
  --cloud-provider yarn
  --cluster-hostname 192.168.123.456 --ssh-user ubuntu --ssh-id-file ./../private_key.txt
  --spark-conf spark.driver.memory=10000m --spark-conf spark.executor.memory=10000m
  --spark-conf spark.executor.cores=4 --spark-conf spark.kryoserializer.buffer.max=128m
  --benchmark-path 'hdfs://lakehouseevaluation:9000/user/ubuntu/lakehouse/benchmarks'
  --source-data-path 'hdfs://lakehouseevaluation:9000/user/ubuntu/sourceDatasets
                                                              /flowersDataset'
  --iterations 5
  --number-partitions 300
  --compression snappy
  --custom-queries 'Select half=SELECT * FROM ${tableName} LIMIT 1835;'
  --benchmark binary-file-format-delta,only-metadata-format-delta
```

`Lakehouse-VM` bezeichnet. Das YARN-Cluster verwendet HDFS als Speichersystem und führt die Apache Spark Session aus (Schritt 4). Die einzelnen Benchmarks erstellen eine solche Session und verwenden diese dann, um die Lese- und Schreiboperationen auf dem HDFS auszuführen (Schritte 5 und 6). Auf dem HDFS befindet sich der Testdatensatz, der für den Benchmark verwendet werden soll und dort werden dann auch die Lakehouse-Tabellen erstellt. Wenn ein einzelner Benchmark beendet wurde, werden die von diesem Benchmark erstellten Log-, JSON- und CSV-Dateien via SSH wieder in die `Lokale VM` kopiert (Schritt 7). Sobald alle Benchmarks auf die `Lakehouse-VM` geladen und ausgeführt wurden, wird mithilfe der erhaltenen JSON-Dateien der zusammenfassende Report erstellt und die Ergebnisse gegenübergestellt (Schritt 8).

Dieser beschriebene Aufbau, den LHBench und damit auch `LHBench-UnstructuredData` verwendet, erlaubt es, dass kein dauerhafter Zugriff auf das Cluster, auf dem das Lakehouse läuft, notwendig ist. Stattdessen wird nur temporär via SSH auf das Cluster zugegriffen.

Durch die Ausführung von `run-benchmark.py` wird die übergebene Liste an Benchmarks gestartet und nacheinander ausgeführt. Zur Konfiguration der Benchmarks können weitere Argumente an `run-benchmark.py` übergeben werden. Ein Beispiel für die Verwendung von `run-benchmark.py` ist in Listing 7.1 dargestellt. Manche der übergebenen Argumente werden, wie in Abschnitt 7.2 beschrieben, an die auszuführende Scala-Datei weitergegeben und von dieser verarbeitet. Dies sind die Argumente `benchmark-path`, `source-data-path`, `iterations`, `number-partitions`, `compression` und `custom-queries`. Deren Funktion wurde bereits in Abschnitt 7.2 ausgeführt. Der `cloud-provider` gibt an, was für ein Cluster für die Ausführung der Benchmarks verwendet wird. Die Argumente `cluster-hostname`, `ssh-user` und `ssh-id-file` werden verwendet, um die SSH-Verbindungen zur `Lakehouse-VM` herzustellen. Mit `spark-conf` können spezielle Konfigurationsparameter für Apache Spark festgelegt werden. Diese Konfigurationsanpassungen werden bei der Ausführung von `spark-submit` dann als Argumente an Apache Spark übergeben. Dieses `spark-conf` Argument kann mehrfach verwendet werden, um mehrere Konfigurationsparameter festzulegen. Für die Vergleichbarkeit der unterschiedlichen Lakehouse-Frameworks ist es sinnvoll, möglichst die Standardkonfiguration von Apache Spark zu verwenden, um die Ergebnisse nicht zum Vorteil eines der Lakehouse-Frameworks zu verzerren. Im Rahmen des in dieser Arbeit durchgeführten Experiments wurden daher nur die Anzahl der zu nutzenden Kerne und die Menge des verfügbaren Arbeitsspeichers beim Spark Executor und Spark Driver angepasst. Dies ist not-

wendig, um den Arbeitsspeicher auf der verwendeten virtuellen Maschine möglichst effektiv zu nutzen. Darüber hinaus wurde die Buffergröße des Spark-Kryo-Serialisierers erhöht, um einen Buffer Overflow beim Serialisieren von Daten zu verhindern. Schließlich werden mit dem Argument `benchmark` alle auszuführenden Benchmarks in einer kommagetrennten Liste spezifiziert. Für die verschiedenen Ansätze zur Speicherung unstrukturierter Daten sind dabei folgende Benchmarks verfügbar: `binary-file-format-<Lakehouse>`, `only-metadata-format-<Lakehouse>`, `only-metadata-format-and-fetch-<Lakehouse>` und `image-format-<Lakehouse>`, wobei `<Lakehouse>` entweder durch `delta`, `iceberg` oder `hudi` ersetzt werden kann. Damit steht eine Gesamtzahl von zwölf Benchmarks zur Verfügung.

Im Kontext dieser Arbeit wurde ein YARN-Cluster verwendet, da sich dies leicht auf einer einzelnen virtuellen Maschine aufsetzen und verwenden lässt. Es handelt sich dabei um eine kostengünstigere Lösung als wenn eine Public Cloud zur Anwendung kommen würde. Das lokale YARN-Cluster konnte verwendet werden, um während der Implementierung von `LHBench-UnstructuredData` regelmäßige Tests und schließlich auch die finale Evaluation durchzuführen. Bei einem „Pay-As-You-Go“-Bezahlmodell eines Cloud Providers wären hingegen in dieser Testphase Kosten für jede Nutzung angefallen. Diese wären umso höher, wenn ein größeres Cluster verwendet worden wäre. Allerdings ist das eingesetzte Cluster recht klein, da es auf die Größe der virtuellen Maschine begrenzt und auch nicht über mehrere physische Knoten verteilt ist. Die verwendete virtuelle Maschine verfügt über 32 Gigabyte Arbeitsspeicher und einen AMD EPYC-Milan Prozessor mit $2,45$ GHz, von dem sechs vCPUs der virtuellen Maschine zugeordnet sind.

Als Dateiformat, in dem die in einer Lakehouse-Tabelle enthaltenen Daten abgespeichert werden, wird in dieser Evaluation Apache Parquet mit Snappy als Kompressionsmethode verwendet. Diese Wahl hatte bereits auch das LHBench-Framework getroffen [JKP+23a]. Dies ist sinnvoll, da Delta Lake aktuell ausschließlich Apache Parquet unterstützt. Indem alle Lakehouse-Frameworks dasselbe Dateiformat verwenden, wird eine höhere Vergleichbarkeit erreicht, die nicht vom verwendeten Dateiformat abhängt.

Die verwendete Processing Engine ist bereits von LHBench vorgegeben, da die Verwendung von Apache Spark fest in den Aufbau des Frameworks integriert ist [JKP+23a]. Die Wahl von Apache Spark erweist sich als sinnvoll, da alle drei Lakehouse-Frameworks dafür eine gute Integration anbieten [CAG+23].

Als Testdatensätze werden im Rahmen dieser Evaluation Datensätze mit Bildern verwendet. Es wurden als Stellvertreter für unstrukturierte Daten ausschließlich Bilder ausgewählt, da das `ImageFormat` auf das Einlesen von Bildern beschränkt ist. Andere unstrukturierte Daten können bei diesem Ansatz nicht verwendet werden. Die Begrenzung auf Bilder ermöglicht den Vergleich aller definierter Ansätze zur Speicherung unstrukturierter Daten. Die Verwendung von unterschiedlichen Testdatensätzen für verschiedene Ansätze hätte hingegen die Aussagekraft der Ergebnisse reduziert, da eine Vergleichbarkeit zwischen Datensätzen, die verschiedene Dateitypen enthalten, schwer zu erreichen ist. Ein Vergleich der Ansätze mit anderen Typen von unstrukturierten Daten unter Ausschluss des `ImageFormat` wäre in Zukunft möglich.

Bei ausgeführten Testläufen mit verschiedenen Datensätzen hat sich gezeigt, dass das `ImageFormat` sehr arbeitsspeicherintensiv ist: Beim Abspeichern von Bildern im `ImageFormat` werden diese zunächst dekomprimiert. Dadurch kommt es zu einer großen Erhöhung des Datenvolumens, wie in Kapitel 8 noch im Detail gezeigt wird. Dies hat zur Folge, dass beim Lesen der erstellten Lakehouse-Tabelle eine hohe RAM-Auslastung auftritt. Bei der Ausführung von `collect()` mit

Technologie	Verwendete Version
Apache Spark	3.3.3
Delta Lake	2.3.0
Apache Iceberg	1.3.1
Apache Hudi	0.13.1
Apache Hadoop	3.3.5
Apache Hive Metastore	3.1.3
Scala	2.12.15
Java	17.0.8

Tabelle 7.2: Versionen der Technologien, welche in den Benchmarks eingesetzt wurden.

Apache Spark muss der gesamte `DataFrame` in den Speicher des Spark-Driver-Prozesses geladen werden. Bei großen Datensätzen reicht damit der Arbeitsspeicher der `Lakehouse-VM` nicht aus. Dies ist insbesondere bei der Verwendung von Apache Hudi als Lakehouse-Framework der Fall. Es kommt beim Lesen der Tabelle zu Out-Of-Memory-Fehlern und zu mehrfach abgebrochenen Apache-Spark-Tasks. Dies führt am Ende dazu, dass die Ausführung des Apache-Spark-Jobs final abgebrochen wird. Aus diesem Grund wurde die Anzahl an Partitionen, in die der verwendete Datensatz beim Schreiben in eine Lakehouse-Tabelle aufgeteilt wird, auf 300 festgelegt. Dies hilft beim Lesen dieser Tabelle Out-Of-Memory-Fehler zu verhindern, da die gelesenen Daten in kleinere Dateien aufgeteilt sind. Diese Einstellung wird auf alle Ansätze zur Speicherung unstrukturierter Daten und alle ausgeführten Benchmarks gleichermaßen angewendet. So wird die Vergleichbarkeit nicht beeinträchtigt.

Im Rahmen dieser Evaluation wird die Ausführung jedes Benchmarks, bestehend aus je einer Kombination aus einem Ansatz zur Speicherung unstrukturierter Daten und einem Lakehouse-Framework, fünf mal wiederholt. Bei der Verwendung des `run-benchmark.py`-Befehls wird also `iterations` auf fünf gesetzt. Im Report wird automatisch der Mittelwert dieser fünf Ausführungen berechnet und in den Visualisierungen angezeigt. Dadurch können kleine Schwankungen in den Ausführungszeiten, die zum Beispiel durch veränderte Gesamtlasten der zugrundeliegenden Recheninfrastruktur entstehen können, abgefedert werden.

Tabelle 7.2 listet für alle relevanten Technologien, die in den ausgeführten Benchmarks zum Einsatz kommen, die genutzten Versionen auf, um eine Reproduktion der in Kapitel 8 präsentierten Ergebnisse zu ermöglichen.

8 Ergebnisse

In diesem Kapitel der Arbeit werden die Ergebnisse der verschiedenen ausgeführten Benchmarks präsentiert.

8.1 Ergebnisse der Benchmarks mit einem Blumendatensatz

Als erstes wurde eine Reihe an Benchmarks durchgeführt, um alle Ansätze zur Speicherung unstrukturierter Daten und alle Lakehouse-Frameworks zu vergleichen. Dazu wurden die im Kontext dieser Arbeit vorgestellten Benchmarktypen ausgeführt und automatisiert ein zusammenfassender Report für diese Ausführung erstellt. Die Benchmarks verwenden den Blumendatensatz `tf_flowers`[1] von TensorFlow. Dieser enthält insgesamt 3670 Blumenbilder unterschiedlicher Größe mit einer Gesamtgröße von 233 Megabyte.

Als, über `custom-queries` manuell übergebene, zusätzliche Anfragen werden folgende SQL-Statements in den Benchmarks ausgeführt:

- `Select by daisy`: SELECT * FROM ${tableName} WHERE path LIKE "hdfs://.../daisy%";
- `Select half`: SELECT * FROM ${tableName} LIMIT 1835;

Der `tf_flowers` Datensatz besteht aus mehreren Unterverzeichnissen, die nach Art der abgebildeten Blume sortiert sind. Mit dem `Selct by daisy` SQL-Statement werden nur die Einträge aus der Tabelle zurückgegeben, bei deren Bild es sich um ein Gänseblümchen handelt, also die Spalte `path` einen Pfad zum Unterverzeichnis `daisy` enthält. Die zweite Anfrage limitiert die Anzahl zurückgegebener Einträge auf die Hälfte der Gesamtzahl an Einträgen in der Tabelle, also in diesem Fall auf 1835.

Abbildung 8.1 zeigt die Performanz der verschiedenen Ansätze zur Speicherung unstrukturierter Daten unter der Verwendung des Lakehouse-Frameworks Delta Lake. Auf der Horizontalen sind immer die verschiedenen Teilschritte der Benchmarks abgebildet. Zudem sind für die ausgeführten SQL-Anfragen jeweils die Messwerte der Query-Planning-Phase mit abgebildet. In den folgenden Grafiken sind diese ausgelassen. Das Query Planning wird im Kontext dieser Benchmarks nicht im Detail betrachtet, da dieses bei den ausgeführten Anfragen nur einen kleinen Anteil der Gesamtausführungszeit darstellt und sich zwischen den verschiedenen Ansätzen zur Speicherung unstrukturierter Daten nur sehr geringfügig unterscheidet.

Eine Übersicht ohne die Query-Planning-Phasen, ist in Abbildung 8.2 abgebildet. Der erste von den Benchmarks ausgeführte Teilschritt `readIntoDataFrame` hat eine sehr geringfügige Ausführungsdauer, welche bei Verwendung des Blumendatensatzes maximal 0,3 Sekunden beträgt. In diesem

[1] Siehe https://www.tensorflow.org/datasets/catalog/tf_flowers

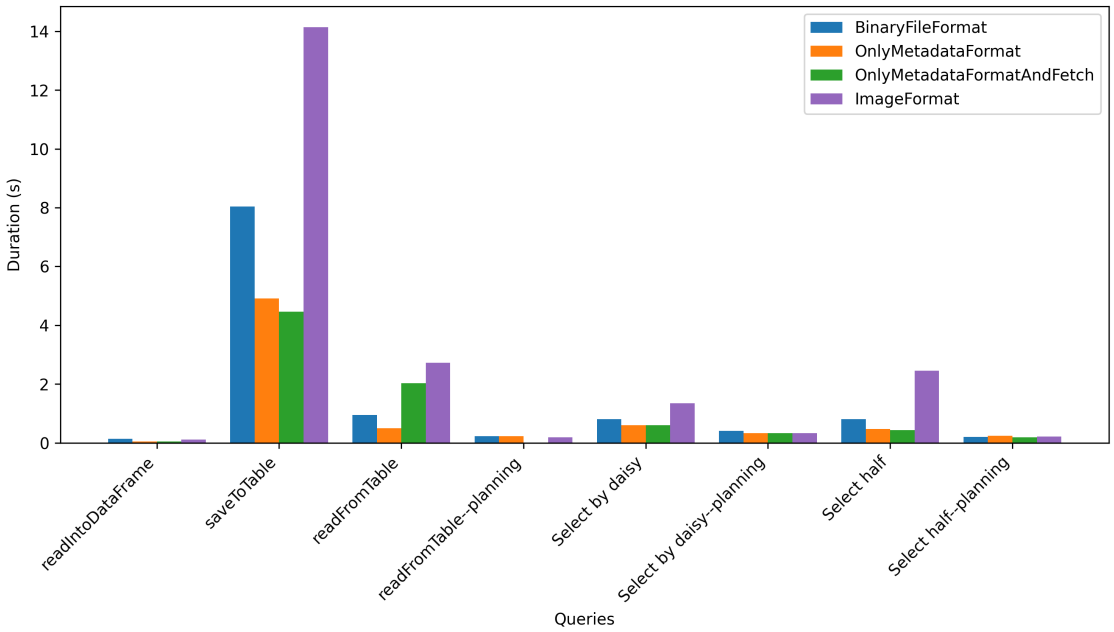


Abbildung 8.1: Vergleich der Performanz der Ansätze zur Speicherung unstrukturierter Daten mit Delta Lake und dem Blumendatensatz (mit Query Planning).

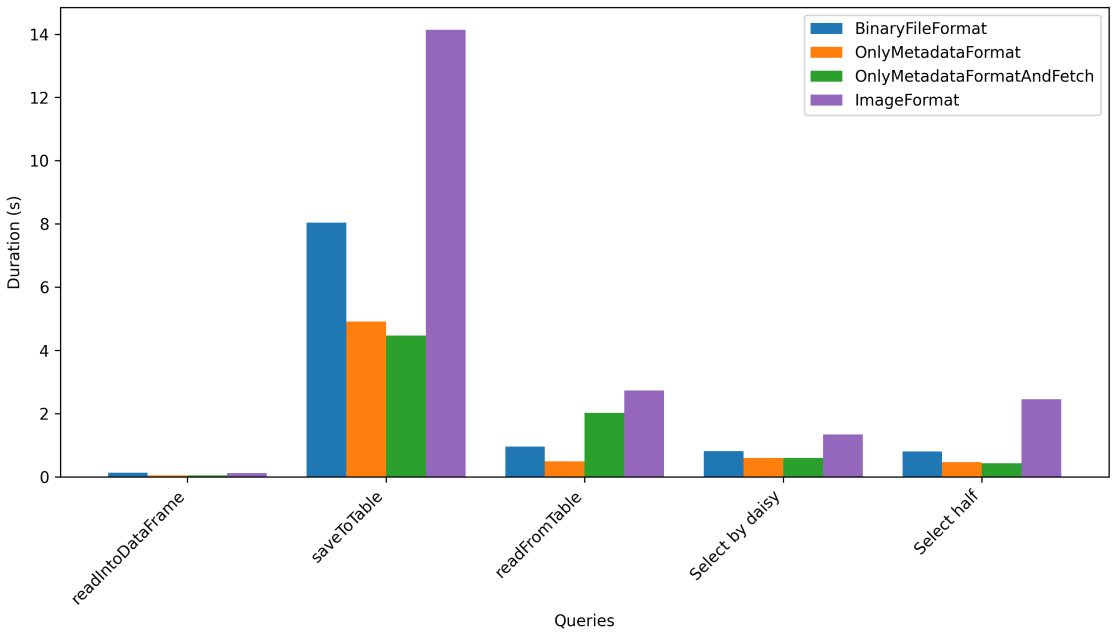


Abbildung 8.2: Vergleich der Performanz der Ansätze zur Speicherung unstrukturierter Daten mit Delta Lake und dem Blumendatensatz (ohne Query Planning).

Ausführungsschritt wird für den jeweiligen verwendeten Ansatz zur Speicherung unstrukturierter Daten ein `DataFrame` auf Grundlage des Blumendatensatzes erstellt. Wie die Ergebnisse zeigen, wird dieser Schritt aufgrund der `Lazy Evaluation` von `DataFrames` nur implizit ausgeführt [Apa23o]. Dies ist an den geringen Ausführungszeiten für diesen Teilschritt erkennbar. Die eigentlichen Daten werden durch die Erstellung eines `DataFrames` nicht tatsächlich vom HDFS in den ausführenden Apache-Spark-Driver-Prozess geladen. Solche Operationen werden von Apache Spark erst dann ausgeführt, wenn die Daten tatsächlich benötigt werden. Stattdessen liegen die Daten weiterhin verteilt auf dem verwendeten HDFS vor. Dadurch kann Apache Spark Ausführungszeiten optimieren, indem nicht benötigte Teile von Datensätzen oder Lakehouse-Tabellen nicht geladen werden [Apa23o]. Auch die beim `OnlyMetadataFormat` ausgeführten manuellen Abfragen von Metadaten erzeugen hier keinen relevanten Mehraufwand gegenüber den anderen Ansätzen.

Bei dem nächsten Teilschritt `saveToTable` zeigt sich ein klarer Unterschied zwischen den Ansätzen zur Speicherung unstrukturierter Daten. Hier werden nun tatsächlich die Bilder aus dem Testdatensatz geladen, in das entsprechende Datenschema konvertiert und als Delta-Lake-Tabelle wieder auf dem HDFS abgespeichert. Dabei ist zwischen den Ansätzen `OnlyMetadataFormat` und `OnlyMetadataFormatAndFetch` nur ein geringer Unterschied ersichtlich. Dieser ist durch kleine Varianzen zwischen einzelnen Ausführungen zu erklären, da beide Ansätze bis zu diesem Ausführungsschritt identisch vorgehen. Mit einer durchschnittlichen Ausführungszeit von 14,4 Sekunden benötigt das `ImageFormat` in diesem Schritt etwa 67 Prozent länger als das `BinaryFileFormat` mit 8,6 Sekunden. Das `OnlyMetadataFormat` benötigt mit 5,2 Sekunden noch einmal ungefähr ein Drittel weniger Zeit als das `BinaryFileFormat`.

Das `OnlyMetadataFormat` muss ausschließlich die Metadaten in der Lakehouse-Tabelle ablegen, ruft daher eine deutlich geringere Datenmenge vom HDFS ab und ist deshalb in diesem Teilschritt am schnellsten. Beim `BinaryFileFormat` müssen alle Bilder abgerufen, in ein Bytearray konvertiert und als Lakehouse-Tabelle abgespeichert werden. Für das `ImageFormat` werden die PNG-Bilder zusätzlich noch dekomprimiert. Dadurch erhöht sich das Datenvolumen deutlich. Dieses erhöhte Datenvolumen, aber auch der zusätzliche Verarbeitungsschritt führen dazu, dass dieser Ansatz für das Schreiben in eine Lakehouse-Tabelle deutlich länger als die anderen Ansätze benötigt.

Der nächste Teilschritt `readFromTable` liest die zuvor geschriebene Lakehouse-Tabelle wieder ein und lädt diese in ihrer Gesamtheit in den Arbeitsspeicher des Spark-Driver-Prozesses. Auch hier ist zu sehen, dass das `ImageFormat` deutlich länger als das `BinaryFileFormat` und `OnlyMetadataFormat` benötigt. Dies liegt wieder daran, dass das `ImageFormat` eine größere Datenmenge erzeugt, die in diesem Teilschritt wieder in den Arbeitsspeicher geladen werden muss. Aus demselben Grund ist das `BinaryFileFormat` auch hier etwas langsamer als das `OnlyMetadataFormat`. Bei dem `OnlyMetadataFormatAndFetch` zeigt sich eine Veränderung: In diesem Teilschritt benötigt dieser Ansatz länger als das `BinaryFileFormat`. Das `OnlyMetadataFormatAndFetch` ruft nach dem Lesen der Lakehouse-Tabelle die in dieser referenzierten Bilder vom HDFS ab und lädt diese als Bytearrays in den Arbeitsspeicher. Dieser Schritt kommt zusätzlich zu dem eigentlichen Abrufen der Lakehouse-Tabelle noch hinzu und führt zu der längeren Ausführungsdauer in diesem Teilschritt.

Auch bei den manuellen Anfragen zeigt sich, dass das `ImageFormat` am längsten braucht: Es benötigt für die Anfrage `Select half` noch einmal mehr Zeit als bei der Anfrage `Select by daisy`. Dies liegt vermutlich daran, dass mehr Bilder gelesen werden müssen. Die Anzahl an Bildern im Unterverzeichnis `daisy` stellt nämlich weniger als die Hälfte der Gesamtzahl der Bilder dar. Da bei den manuellen Anfragen beim Ansatz `OnlyMetadataFormatAndFetch` der zusätzliche Schritt zur Abfrage der Bilder vom HDFS nicht ausgeführt wird, ist bei diesem Ansatz die Ausführungszeit

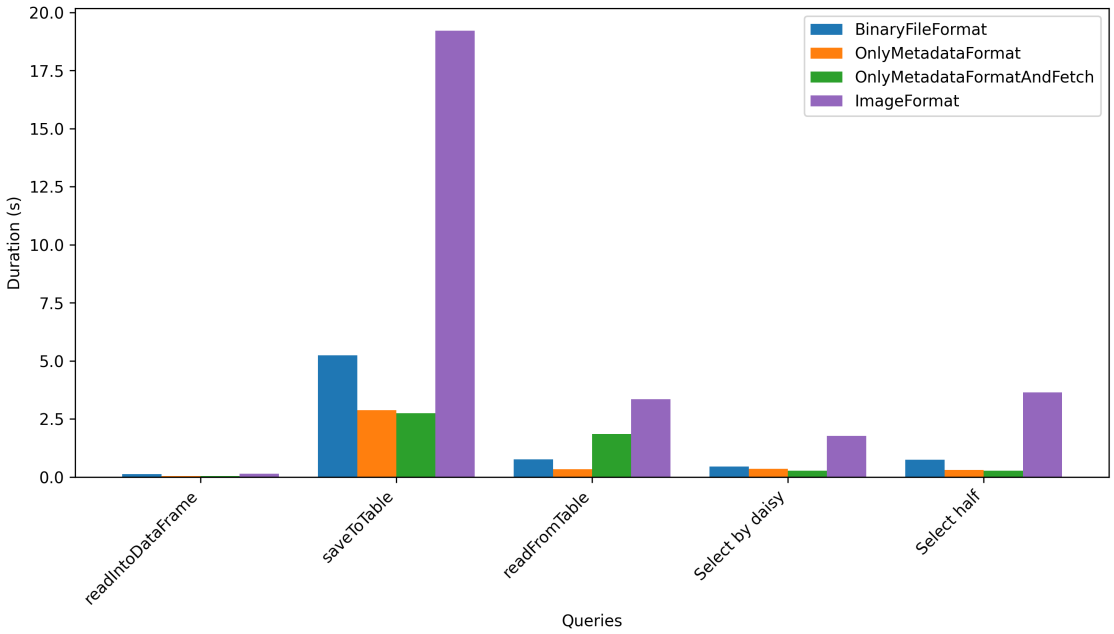


Abbildung 8.3: Vergleich der Performanz der Ansätze zur Speicherung unstrukturierter Daten mit Apache Iceberg und dem Blumendatensatz.

ungefähr identisch zum `OnlyMetadataFormat`. Es zeigt sich im Allgemeinen, dass die manuell übergebenen Anfragen, durch die nur Teile der Lakehouse-Tabelle zurückgegeben werden, nur geringfügig schneller oder teilweise sogar langsamer ablaufen als der `readFromTable` Teilschritt, der die gesamte Tabelle abfragt. Diese Reduzierung der Gesamtmenge von zu lesenden Daten bringt in diesem Fall also nur wenig Performanzvorteile. Vor allem bei insgesamt wenig zu lesenden Daten, wie beim `OnlyMetadataFormat`, ist der Aufwand für die Evaluation, welche Einträge überhaupt gelesen werden sollen zusammen mit dem anschließenden Abrufen der identifizierten Einträge, bei `Select by daisy` größer als der Aufwand für das Laden aller Daten. Es könnte noch betrachtet werden, wie sich die Dauer des Lesens verhält, wenn die Lakehouse-Tabelle nach den Arten von abgebildeten Blumen partitioniert vorliegt. Dann sollte die Anfrage `Select by daisy` deutlich schneller ablaufen, da die größten Teile der Daten direkt ausgeschlossen werden können.

In Abbildung 8.3 und Abbildung 8.4 ist die gemessene Performanz der verschiedenen Ansätze zur Speicherung unstrukturierter Daten unter Verwendung der anderen beiden Lakehouse-Frameworks Apache Iceberg und Apache Hudi, abgebildet. Bei dem Vergleich der Ansätze lassen sich bei beiden sehr ähnliche Beobachtungen machen wie zuvor mit Delta Lake. Einzig bei dem Teilschritt `saveToTable` weicht die relative Ausführungsdauer zwischen den Ansätzen stark ab. So ist bei Delta Lake das `ImageFormat` etwa um 67 Prozent langsamer als das `BinaryFileFormat`. Bei Apache Hudi sind es etwa 133 Prozent und bei Apache Iceberg sogar 249 Prozent. Bei Apache Hudi ist das `OnlyMetadataFormat` nur geringfügig schneller als das `BinaryFileFormat`. Die zuvor beschriebene Tatsache bleibt bestehen, dass das `OnlyMetadataFormat` schneller als das `BinaryFileFormat` und dieses wiederum schneller als das `ImageFormat` ist. Es zeigt sich also, dass die für Delta Lake beschriebenen Effekte auch auf Apache Iceberg und Apache Hudi zutreffen. In der verwendeten Ausführungsumgebung lassen sich keine größeren Unterschiede im Verhalten der Lakehouse-

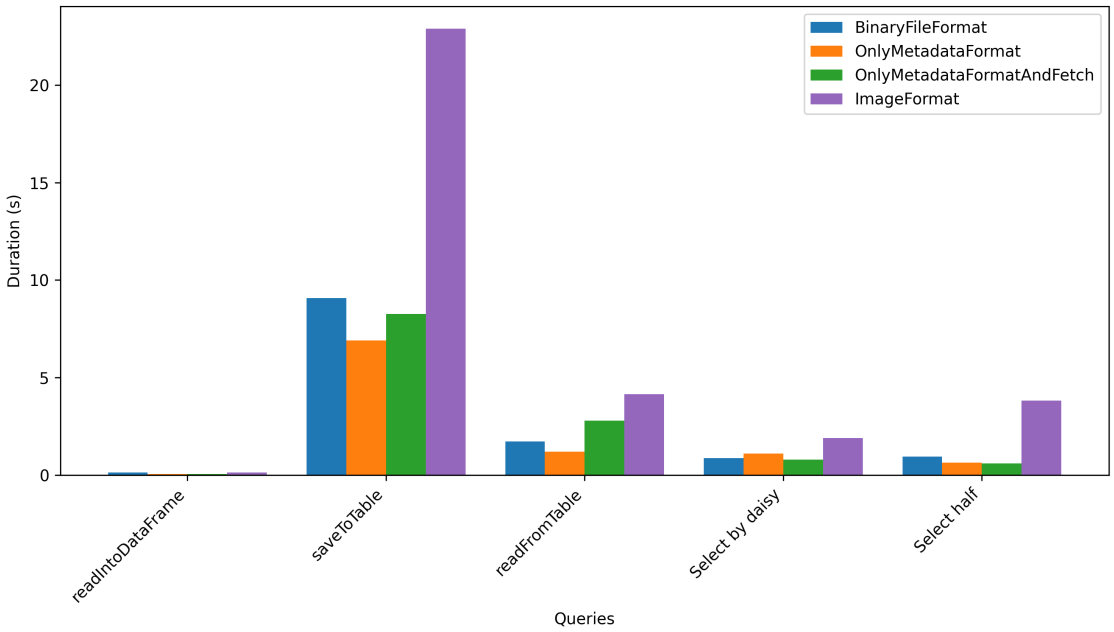
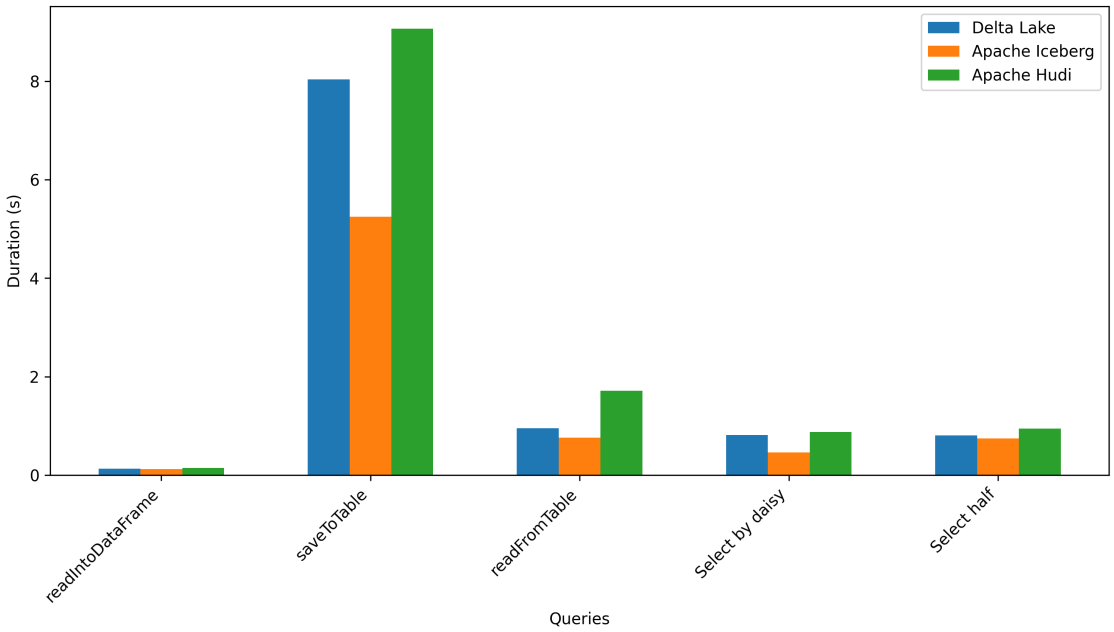


Abbildung 8.4: Vergleich der Performanz der Ansätze zur Speicherung unstrukturierter Daten mit Apache Hudi und dem Blumendatensatz.

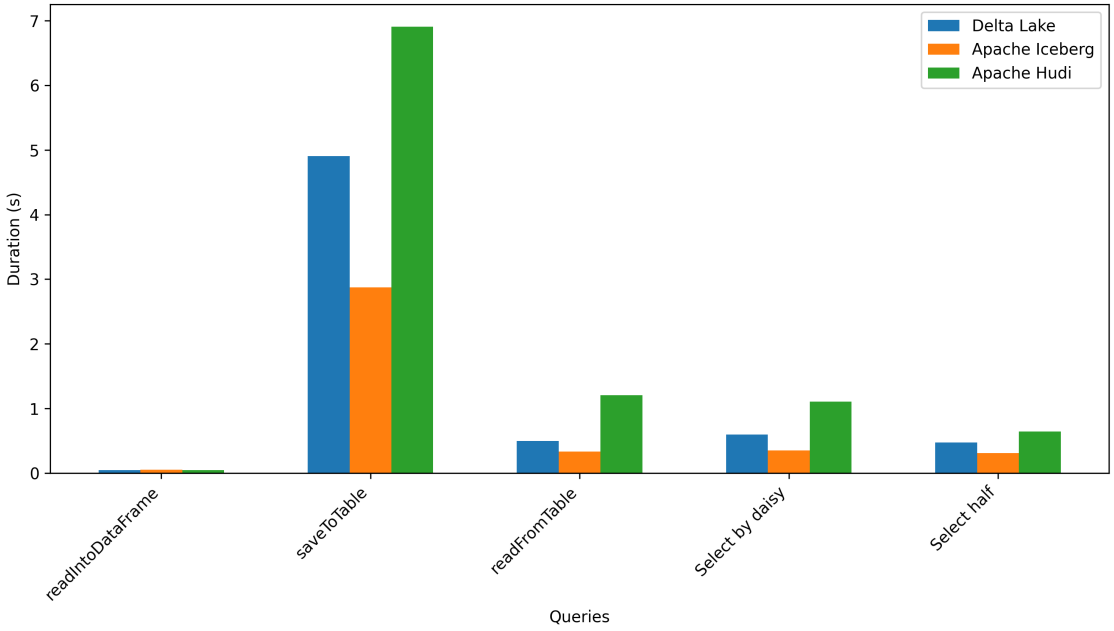
Frameworks hinsichtlich der Ansätze zur Speicherung unstrukturierter Daten beobachten. Eine vorläufige Empfehlung, welcher Ansatz in einer bestimmten Situation verwendet werden sollte, kann somit im Rahmen dieser Arbeit weitestgehend unabhängig von der Wahl des Lakehouse-Frameworks abgegeben werden.

Die Abbildungen 8.5 und 8.6 zeigen insgesamt vier Visualisierungen, in denen jeweils die Performanz zwischen den verschiedenen Lakehouse-Frameworks für einen der Ansätze zur Speicherung unstrukturierter Daten gegenübergestellt wird. Hier zeigt sich, dass bei den meisten Ansätzen Apache Iceberg die kürzeste Ausführungsdauer beim Schreib- und Lesevorgang bietet. Nur beim `ImageFormat` ist Delta Lake performanter. Apache Hudi ist bei allen Ansätzen am langsamsten. Dies stimmt mit den Ergebnissen überein, die von Jain et al. [JKP+23a] vorgestellt wurden, welche ihre Beobachtung damit begründen, dass Apache Hudi insbesondere beim Speichern von Daten aufwendige Vorverarbeitungen durchführt.

Abbildung 8.7 vergleicht den Speicherplatzbedarf der erstellten Lakehouse-Tabellen für die unterschiedlichen Ansätze zur Speicherung unstrukturierter Daten unter Verwendung von Apache Iceberg. Der benötigte Speicherplatz ist für die Tabelle, die das `OnlyMetadataFormat` verwendet, mit nur 768 Kilobyte deutlich am geringsten. Der Grund dafür liegt in der Tatsache begründet, dass in dieser Tabelle ausschließlich die Metadaten zu den Bildern abgespeichert werden, die ein sehr geringes Datenvolumen ausmachen. Der verwendete Speicherplatz ist im Vergleich zu den anderen Ansätzen so gering, dass er in der Visualisierung nicht darstellbar ist. Bei dem `BinaryFileFormat` werden die Bilder als Bytearray mit in der Tabelle abgelegt. Dies führt dazu, dass bei diesem Ansatz der Speicherplatzbedarf mit 229 Megabyte deutlich höher ist.

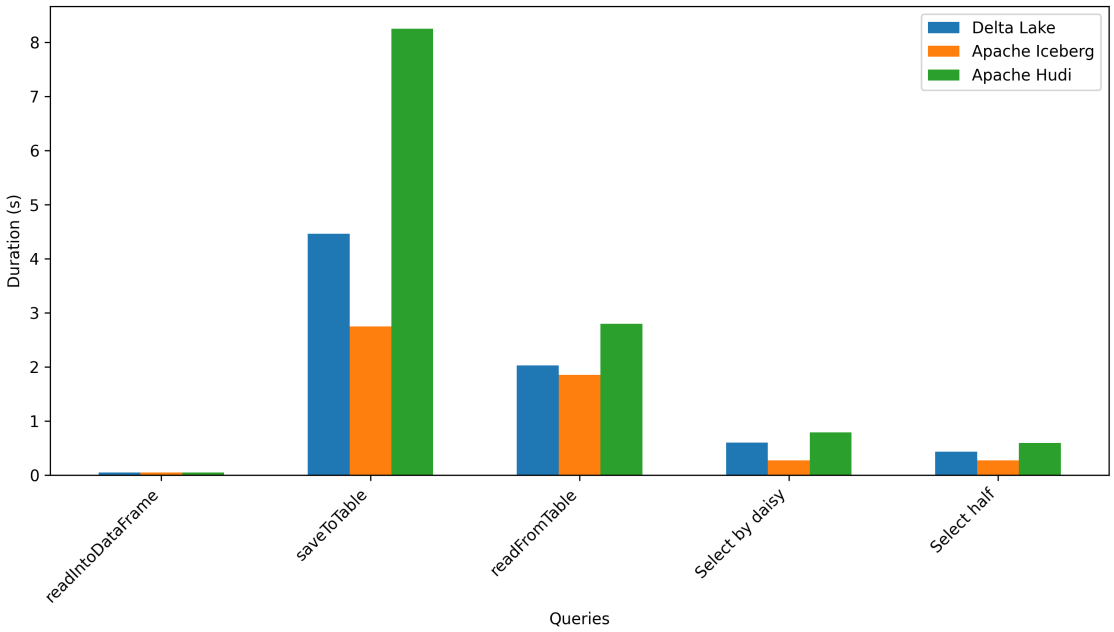


(a) BinaryFileFormat

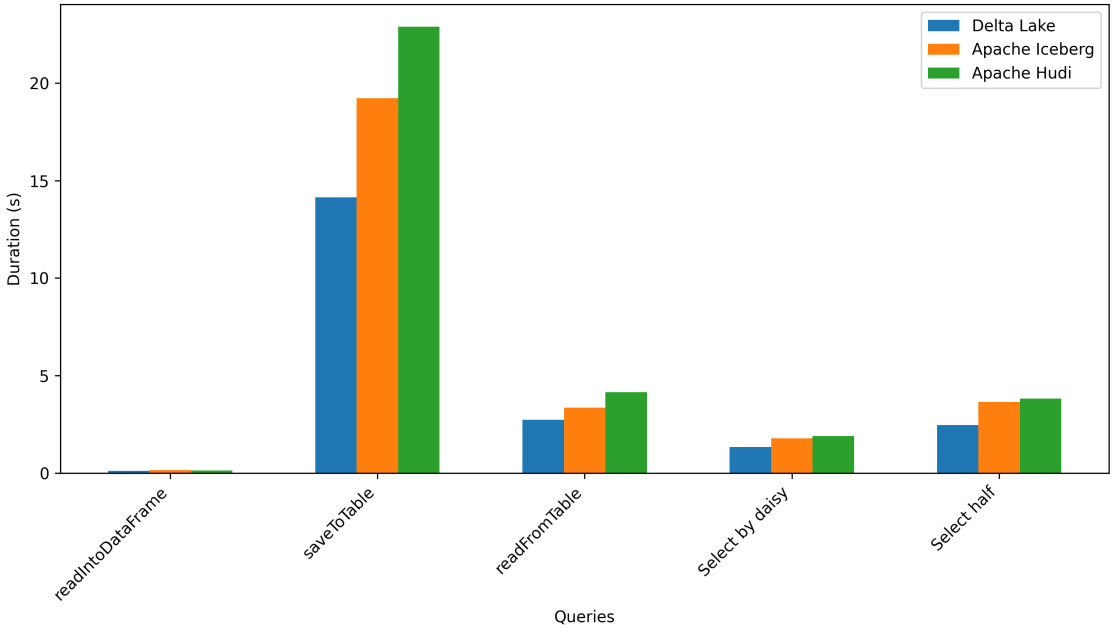


(b) OnlyMetadataFormat

Abbildung 8.5: Vergleich der Performanz zwischen den Lakehouse-Frameworks mit dem Blumendatensatz (Teil 1).



(c) OnlyMetadataFormatAndFetch



(d) ImageFormat

Abbildung 8.6: Vergleich der Performanz zwischen den Lakehouse-Frameworks mit dem Blumendatensatz (Teil 2).

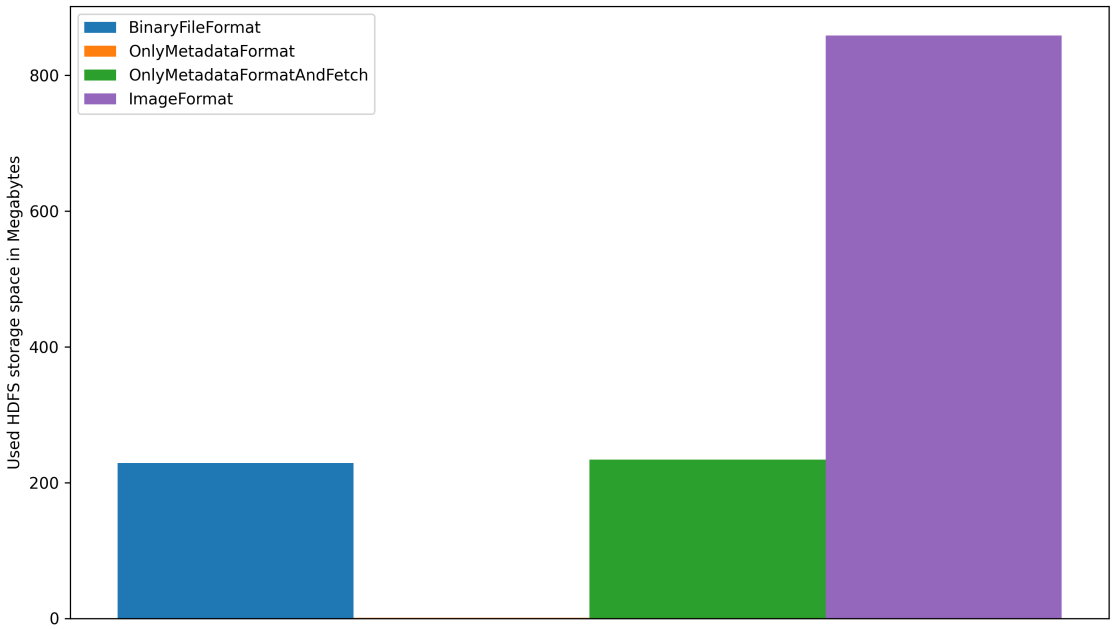


Abbildung 8.7: Vergleich des Speicherplatzbedarfs der erstellten Lakehouse-Tabelle mit Apache Iceberg und dem Blumendatensatz.

Mit 233 Megabyte benötigt der Ansatz `OnlyMetadataFormatAndFetch` nur unwesentlich mehr Speicher als das `BinaryFileFormat`. Bei diesem Ansatz wird im Unterschied zum `OnlyMetadataFormat` zusätzlich zu der erstellten Tabelle noch der Testdatensatz mitgerechnet. Dieser verbleibt bei diesem Ansatz nach der Erstellung einer Tabelle im HDFS, damit auf die Bilder zugegriffen werden kann, wenn diese benötigt werden. Daher wird in diesem Fall der Speicherplatz des Testdatensatzes miteinberechnet. Bei den Ansätzen `BinaryFileFormat` und `ImageFormat` werden die Bilder mit in der Lakehouse-Tabelle abgespeichert, sodass anschließend der Testdatensatz nicht mehr benötigt wird. Deshalb fließt der Testdatensatz bei diesen Ansätzen nicht in die Berechnung des Speicherplatzbedarfs ein. Das `ImageFormat` benötigt deutlich mehr Speicherplatz als die anderen Ansätze. Unter Verwendung des Blumendatensatzes sind es 859 Megabyte, die von der erstellten Tabelle eingenommen werden. Dieser hohe Wert kommt dadurch zustande, dass beim `ImageFormat` die Bilder vor dem Abspeichern dekomprimiert werden, wodurch sich das Datenvolumen stark erhöht. Bei den anderen Lakehouse-Frameworks zeigen sich dieselben Effekte. Aus diesem Grund werden deren Ergebnisse hier bezüglich des Speicherplatzbedarfs nicht noch einmal gesondert aufgeführt.

8.2 Ergebnisse der Benchmarks mit den generierten Bilderdatensätzen

Abschließend wird untersucht, wie sich die Größe der Bilder im Datensatz auf die Performanz auswirkt. Dazu wurde, wie in Abschnitt 7.2 beschrieben, ein Skript zur automatischen Generierung von Bilderdatensätzen erstellt. Unter Verwendung dieses Skripts wurden zwei Datensätze erstellt:

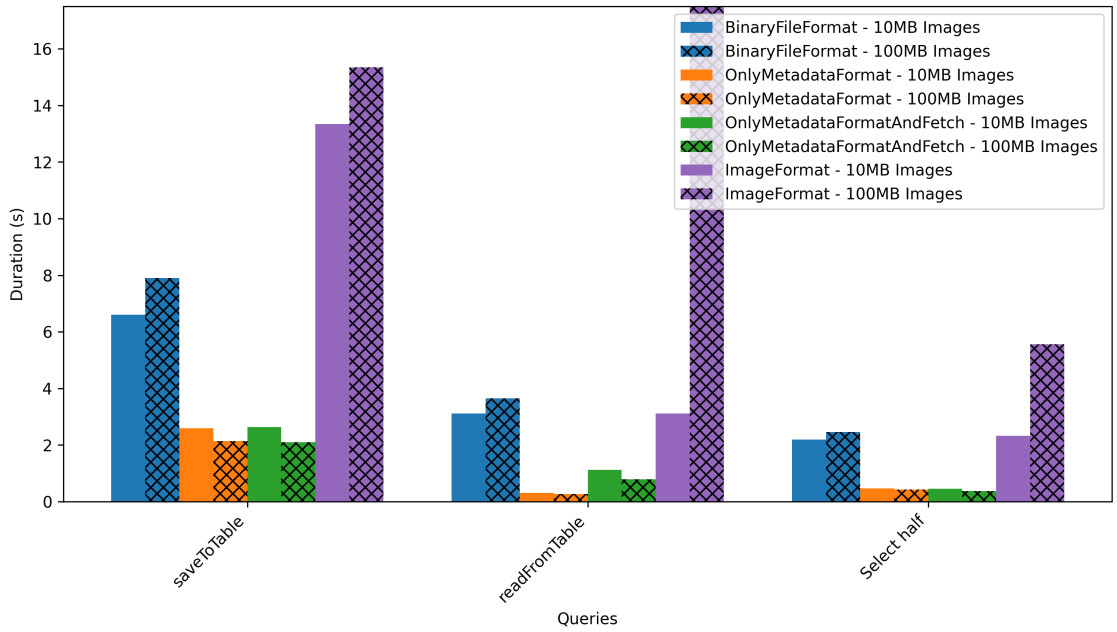


Abbildung 8.8: Vergleich der Performanz der Ansätze zur Speicherung unstrukturierter Daten zwischen den Datensätzen mit 100 Bildern von je 10MB Größe und 10 Bildern mit je 100MB Größe.

Der erste Datensatz enthält 100 Bilder mit je 10 Megabyte Größe. Der zweite zehn Bilder mit je 100 Megabyte Größe. Dadurch weisen beide Datensätze eine Gesamtgröße von einem Gigabyte auf. Auf diese Weise kann untersucht werden, wie sich die Anzahl und Größe der einzelnen in dem Datensatz enthaltenen Dateien auf die Performanz auswirkt, bei gleichbleibender Gesamtgröße des Datensatzes.

Abbildung 8.8 zeigt den Vergleich der Performanz der einzelnen Teilschritte zwischen diesen zwei Datensätzen. Diese Visualisierung wurde speziell für die betrachtete Fragestellung erstellt. Sie ist nicht in dem Report enthalten, der automatisiert durch `LHBench-UnstructuredData` generiert wird. Der Teilschritt `readIntoDataFrame` wurde in der Visualisierung ausgelassen, da dieser, wie schon bei dem Blumendatensatz, sehr schnell abläuft und wenig aussagekräftig ist. Die Säulen der Visualisierung beziehen sich immer paarweise auf einen der Ansätze bei Verwendung der beiden verschiedenen Datensätze. Wie zu erwarten, zeigt sich beim `OnlyMetadataFormat` und beim `OnlyMetadataFormatAndFetch` kein deutlicher Unterschied bei der Ausführungsdauer für die beiden Datensätze. Bei diesen Ansätzen müssen die Bilder nicht direkt gelesen werden. So hat die Größe der Bilder keinen Einfluss auf die Ergebnisse. Bei dem Bilderdatensatz mit den Bildern von 100 Megabyte Größe ist die Ausführungszeit sogar ein wenig geringer. Dies liegt daran, dass dieser Datensatz nur zehn Bilder enthält, im Gegensatz zum zweiten mit 100 Bildern. Dadurch sind bei diesem Datensatz weniger Einträge in der Tabelle enthalten, die sowohl geschrieben, als auch dann in den SQL-Anfragen wieder gelesen werden müssen.

Bei den andern Ansätzen zeigt sich hingegen, dass sowohl die Schreib- als auch die Leseperformanz bei dem Datensatz mit den Bildern von 100 Megabyte Größe geringer ist. Dieser Unterschied fällt jedoch in Anbetracht der Verzehnfachung der Bildgröße, vor allem beim `BinaryFileFormat`

recht gering aus. Beim `ImageFormat` zeigt sich eine sehr lange Dauer des Lesevorgangs: Die Ausführungsdauer für `readFromTable` ist in der Visualisierung in Abbildung 8.8 aufgrund des hohen Wertes abgeschnitten. Sie beträgt 72,7 Sekunden. Diese lange Dauer lässt sich damit erklären, dass das Lesen des Datensatzes im `ImageFormat` nahezu den gesamten Arbeitsspeicher der virtuellen Maschine beansprucht. Apache Spark muss deshalb aufwendig Daten auf der Festplatte zwischenspeichern. Dies reduziert die Performanz drastisch [Apa23n].

9 Diskussion

In diesem Kapitel werden in Abschnitt 9.1 verschiedene Anwendungsszenarien unterschieden, in denen unstrukturierte Daten im Kontext von Datenplattformen verwendet werden können. Abschnitt 9.2 gibt dann vorläufige Empfehlungen ab, welche der Ansätze zur Speicherung unstrukturierter Daten in den verschiedenen Anwendungsszenarien verwendet werden sollten. Dabei basieren diese Empfehlungen sowohl auf den erhaltenen Resultaten mithilfe des erstellten Benchmark-Frameworks, als auch auf der konzeptionellen Gegenüberstellung der Ansätze. Schließlich wird in Abschnitt 9.3 der Beitrag dieser Arbeit eingeordnet und auch auf Limitierungen hingewiesen, denen die in dieser Arbeit vorgestellten Ergebnisse unterliegen. Auf Grundlage dieser Limitierungen wird beschrieben, welche Aspekte in Zukunft noch in der weiteren Forschung betrachtet werden sollten.

9.1 Anwendungsszenarien

Im Folgenden werden verschiedene Anwendungsmöglichkeiten von unstrukturierten Daten eingeführt.

Auf Daten in einem Lakehouse können einfache analytische Methoden angewendet werden, wie zum Beispiel Reporting und OLAP [AGXZ21]. Dieser Anwendungsfall ist jedoch bei unstrukturierten Daten nur von geringem Nutzen: So können Metadaten, wie die Größe oder die Anzahl von Dateien genutzt werden, um einfache Übersichten zu erstellen. Wenn aber über die unstrukturierten Daten keine weiteren spezielleren Metadaten bekannt und in der Lakehouse-Tabelle hinterlegt sind, so können darüber hinaus keine komplexeren Fragestellungen mit Reporting und OLAP betrachtet werden. Für solch einfache Analysemethoden genügt es, auf die Metadaten der gespeicherten Daten zuzugreifen. Die unstrukturierten Daten an sich können mit SQL-Anfragen nicht sinnvoll für Reporting und OLAP verwendet werden.

Ein weiteres Anwendungsszenario ist die Anwendung von Advanced-Analytics-Methoden auf dem Lakehouse. Dabei kann es sich um Methoden wie Machine Learning und Data Mining handeln [AGXZ21]. So kann zum Beispiel Image Recognition auf den in einem Lakehouse gespeicherten Bildern durchgeführt werden. Außerdem kann Data Mining verwendet werden, um auf einer Menge von unstrukturierten Daten neue Muster zu erkennen. Unstrukturierte Daten wie Bilder, Videos und Textdateien sind für solche Anwendungen besonders geeignet. Damit lassen sich Prognosen erstellen und Geschäftsprozesse besser verstehen.

Diese Advanced-Analytics-Methoden können alternativ auch direkt auf den im Speichersystem abgelegten unstrukturierten Daten ausgeführt werden. Anstatt die Dateien aus einem Lakehouse zu laden, um dann darauf ein Modell zu trainieren oder anzuwenden, können diese stattdessen auch einfach ohne eine spezielle Metadatenschicht in einer Verzeichnisstruktur vorliegen und von dort geladen werden. Dieses Abrufen kann mit Apache Spark erfolgen und neu hinzukommende Daten

können optional durch Streaming automatisiert abgerufen werden, wenn die Advanced-Analytics-Methoden dies unterstützen. Streaming ist sowohl bei Verwendung eines Lakehouses, als auch bei direktem Zugriff auf die Dateien möglich. Der direkte Zugriff auf die Dateien im Speichersystem erspart es, die unstrukturierten Dateien in einem Lakehouse abzuspeichern. Theoretisch kann so die Zeit für das Abspeichern eingespart werden. Jedoch gehen durch den direkten Zugriff viele Vorteile verloren, die ein Lakehouse bei der Datenverwaltung bietet: Daten werden in einem Lakehouse zentralisiert abgespeichert und können mit diversen Processing Engines standardisiert abgerufen werden [ADS+20]. Die manuelle Erstellung einer Verzeichnisstruktur mit unstrukturierten Dateien ist hingegen viel fehleranfälliger und bietet keine standardisierte Zugriffsmöglichkeit auf die Dateien. Daten können beispielsweise verloren gehen oder unerwartet verschoben werden. Ein Lakehouse stellt hingegen durch ACID-Eigenschaften eine möglichst hohe Datenqualität sicher und bietet zudem wertvolle Verwaltungsfunktionen wie Time-Travel. Dadurch kann ein älterer Zustand einer Tabelle wiederhergestellt werden und Analysen auf einem festen, reproduzierbaren Stand der Daten durchgeführt werden. Weiter bietet ein Lakehouse unterschiedliche Optimierungen zur Beschleunigung der Abfragegeschwindigkeit der Daten [ADS+20]. Aus diesen Gründen ist es für Unternehmen sinnvoll, unstrukturierte Daten zentralisiert in einem Lakehouse abzuspeichern und für Analyseanwendungen auf die Daten im Lakehouse zuzugreifen.

9.2 Anwendungsempfehlungen

Dieser Abschnitt gibt eine Empfehlung ab, welche der Ansätze zur Speicherung unstrukturierter Daten in Lakehouses in verschiedenen Szenarien verwendet werden sollten.

Wenn ausschließlich einfache analytische Anwendungen auf den Daten ausgeführt werden sollen, werden im Lakehouse nur die Metadaten zu gespeicherten unstrukturierten Daten benötigt. Die eigentlichen, unstrukturierten Daten haben bei diesen Anwendungen keinen Nutzen. In diesem speziellen Fall ist es daher am effizientesten, das `OnlyMetadataFormat` zur Speicherung unstrukturierter Daten zu verwenden. Das heißt, nur Metadaten im Lakehouse abzulegen. Wie in Kapitel 8 gezeigt, ist dieser Ansatz, sowohl bei Schreib- als auch Leseoperationen schneller, wenn nur auf die Metadaten zugegriffen werden muss. Auch wird Speicherplatz im Lakehouse eingespart, der ansonsten von den unstrukturierten Daten eingenommen werden würde. Wenn sichergestellt ist, dass die tatsächlichen Dateien nie benötigt werden, wäre es sogar denkbar, diese dauerhaft zu löschen und nur die Metadaten aufzubewahren. Es muss dabei jedoch darauf geachtet werden, dass Änderungen an den Originaldaten auch im Lakehouse abgebildet werden. Dieses Szenario stellt jedoch einen extremen Randfall dar, der nur äußert selten auftreten sollte. Bei Daten ist häufig im Vorhinein nicht im Detail bekannt, für welche Analysen diese in Zukunft eventuell noch benötigt werden könnten. Lakehouses bieten, ähnlich wie Data Lakes, den Vorteil, dass Daten abgespeichert werden können, ohne dass beim Abspeichern bereits die exakte zukünftige Nutzung dieser Daten bekannt sein muss.

In dem zweiten Szenario, bei welchem verschiedene Advanced-Analytics-Methoden mit den Daten im Lakehouse durchgeführt werden, werden hingegen die eigentlichen, unstrukturierten Daten benötigt. Methoden wie Machine Learning müssen die Daten verwenden, etwa um zum Beispiel ein Modell zu trainieren. Hier kann es also sinnvoll sein, die unstrukturierten Daten mit im Lakehouse abzulegen. In Kapitel 8 hat sich das `OnlyMetadataFormat` beim Abspeichern der Daten als am schnellsten erwiesen. Auch beim Lesen der Tabelle ist das `OnlyMetadataFormat`

schneller als die anderen Ansätze zur Speicherung unstrukturierter Daten. Im Gegensatz zu den anderen Ansätzen müssen die unstrukturierten Dateien anschließend noch vom Speichersystem geladen werden, bevor diese verwendet werden können. Bei kleinen Dateien kann es sein, dass das `OnlyMetadataFormatAndFetch`, welches dieses Abrufen der Dateien mit einbezieht, etwas schlechter abschneidet, als das `BinaryFileFormat`. Sobald die Dateien aber etwas größer werden, kann sich dieser Mehraufwand des Abrufens wieder ausgleichen. Das `OnlyMetadataFormat`, bzw. das `OnlyMetadataFormatAndFetch`, ist hinsichtlich der Performanz für diesen Anwendungsfall am besten geeignet.

Was die Datenkonsistenz anbelangt hat das `OnlyMetadataFormat` jedoch große Nachteile. Änderungen an den Originaldateien werden nicht automatisch in die Lakehouse-Tabelle übernommen. So kann es leicht passieren, dass die Referenzen auf die Originaldateien bei Änderungen nicht oder nicht korrekt in der Lakehouse-Tabelle aktualisiert werden. In der Folge sind die Dateien, auf die sich die Lakehouse-Tabelle bezieht, gar nicht mehr verfügbar. In einer großen Produktivumgebung ist eine Wiederherstellung des korrekten Zustands dann nicht mehr realistisch. Ebenso sind weitere Verwaltungsfunktionen von Lakehouses, wie Time-Travel und korrekte Versionierung, nicht mehr unbedingt anwendbar. Der manuelle Aufwand für die Erstellung einer Tabelle im `OnlyMetadataFormat` sowie die Gefahr einer mangelnden Standardisierung des Aufbaus dieser kommt hinzu. Dies erhöht die Komplexität beim Zugriff auf die Lakehouse-Tabelle. Daher ist es vor allem bei kleineren Dateien weniger sinnvoll, diesen Ansatz in diesem Anwendungsfall zu verwenden. Der bei kleinen Dateien eher geringfügige Performanzvorteil rentiert sich in Anbetracht des hinzukommenden manuellen Verwaltungsaufwands und der Fehleranfälligkeit, die dieser Ansatz mit sich bringt, nicht.

In diesem Anwendungsfall sollte stattdessen das `BinaryFileFormat` verwendet werden. Dieses ist einfach zu verwenden, da es zum Beispiel von Apache Spark direkt unterstützt wird. Das `BinaryFileFormat` ermöglicht auch die Verwendung aller Verwaltungsfunktionen eines Lakehouses und garantiert somit eine hohe Datenqualität und einen geringen Verwaltungsaufwand. Erst wenn die einzelnen zu speichernden Dateien größer werden, kann sich die Abwägung lohnen, ob das `OnlyMetadataFormat` verwendet werden sollte, da dann der dadurch erreichte Performanzvorteil größer ist. Entscheidend wichtig ist dann aber, dass Data Scientists sich der Anfälligkeiten dieses Ansatzes bewusst sind. Es ist sicherzustellen, dass bei allen Veränderungen am Datenbestand die Datenintegrität erhalten bleibt, indem die Referenzen auf die Originaldateien korrekt verwaltet werden. Wenn das `OnlyMetadataFormat` verwendet werden soll, bedarf es daher einer sehr guten Dokumentation der erstellten Lakehouse-Tabellen. Dies gilt insbesondere dann, wenn viele Personen auf die Daten zugreifen sollen. In großen Unternehmen, in denen viele Personen aus unterschiedlichen Disziplinen und mit variierendem Know-how auf ein Lakehouse zugreifen müssen, ist es fragwürdig, ob es der Performanzvorteil gegenüber dem Mehraufwand durch die Verwendung spezieller Sonderlösungen und der erhöhten Gefahr von Datenkorruption tatsächlich wert ist. Gerade bei großen Datenbeständen ist die Sicherstellung einer hohen Datenqualität äußerst wichtig. Dies sollte in den meisten Fällen einer begrenzten Performanzverbesserung vorgezogen werden.

Das `ImageFormat` ist in diesem Anwendungsfall im Allgemeinen nicht zu empfehlen. Dieses ist sowohl beim Schreiben als auch beim Lesen von Tabellen deutlich langsamer. Außerdem benötigt es zusätzlichen Speicherplatz, da die abgespeicherten Bilder zuvor dekomprimiert werden. Die Verwendung des `ImageFormats` kann sich nur lohnen, wenn vor der Speicherung sichergestellt ist, dass die Bilder später mehrfach aus der Lakehouse-Tabelle abgerufen und nach jedem Abrufen zwangsläufig mit OpenCV oder einer vergleichbaren Bibliothek bearbeitet werden müssen. In

diesem Fall müssen die Bilder nach dem Lesen aus der Lakehouse-Tabelle nicht mehr dekomprimiert werden. Dadurch reduziert sich die benötigte Zeit für die Vorverarbeitung der Daten für Machine-Learning-Anwendungen oder ähnliche Analysemethoden. Doch selbst in diesem Szenario sind die Kosten für den zusätzlich benötigten Festplattenspeicher weiterhin ein relevanter Faktor. Zudem ist es unwahrscheinlich, dass der Verwendungszweck eines Datensatzes zu einem solch hohen Grad bereits im Vorhinein bekannt ist. Stattdessen kann es vorkommen, dass zu einem späteren Zeitpunkt neue Einsatzmöglichkeiten der Daten identifiziert werden, die zuvor noch nicht bedacht wurden. Daher ist es in vielen Fällen sinnvoller, die Daten in ihrer komprimierten Form im Lakehouse abzuspeichern und die gelesenen Bilder nur nach Bedarf zu dekomprimieren. Die Dokumentation von Databricks rät ebenfalls von der Verwendung der `Image-Datenquelle` ab [Dat23a].

Bei der dritten Anwendungsmöglichkeit, bei der direkt auf im Speichersystem abgelegte Dateien zugegriffen wird, kann das `OnlyMetadataFormat` verwendet werden, um über die in der Lakehouse-Tabelle aufgeführten Referenzen die zu lesenden Originaldateien zu identifizieren. Dadurch lässt sich das Problem einer mangelnden Standardisierung des Zugriffes auf die Dateien in diesem Szenario adressieren. Weiterhin bleibt jedoch auch in diesem Szenario der Verlust der Verwaltungsfunktionen und der Maßnahmen zur Sicherstellung der Datenqualität bestehen. Es ist daher nicht empfehlenswert, für Analysen die unstrukturierten Daten direkt aus dem Speichersystem abzurufen.

Zusammenfassend lässt sich festhalten, dass das `BinaryFileFormat` den besten Kompromiss aus Performanz, Speicherplatzbedarf, Sicherstellung von hoher Datenqualität, Einfachheit der Benutzung und Flexibilität in der Verwendung der gespeicherten unstrukturierten Daten, bietet.

9.3 Beitrag der Arbeit und Limitierungen

Der Hauptbeitrag dieser Arbeit besteht in der Entwicklung des neuen Benchmark-Frameworks für Lakehouses, genannt `LHBench-UnstructuredData`. Es erweitert das bestehende Benchmark-Framework LHBench, um verschiedene Ansätze zur Speicherung unstrukturierter Daten in Lakehouses vergleichen zu können. `LHBench-UnstructuredData` ermöglicht dabei die Evaluation der verschiedenen Ansätze und Lakehouse-Frameworks hinsichtlich Performanz und Speicherplatzbedarf und erstellt automatisiert einen Report mit Visualisierungen, der die ausgeführten Benchmarks gegenüberstellt. Dieser Report ermöglicht sowohl die einfache Reproduktion als auch eine unmittelbare Auswertung der Ergebnisse. Mithilfe von `LHBench-UnstructuredData` und der mit diesem Framework gewonnenen Resultate lassen sich auf diese Weise die verschiedenen Ansätze quantitativ bewerten. Das erstellte Benchmark-Framework wurde im Rahmen dieser Arbeit verwendet, um erste Benchmarks zur Gegenüberstellung der verschiedenen Ansätze auszuführen und auszuwerten. Die in dieser Arbeit präsentierten und eingeordneten Ergebnisse demonstrieren, wie `LHBench-UnstructuredData` verwendet werden kann und erlauben eine Einschätzung der quantitativen Vor- und Nachteile der verschiedenen Ansätze. Die Nutzung von `LHBench-UnstructuredData` dient insbesondere als Proof of Concept um zu zeigen, dass dieses Framework effektiv für die Betrachtung der Ansätze zur Speicherung unstrukturierter Daten eingesetzt werden kann. Die Auswertungen in dieser Arbeit haben gezeigt, dass die implementierte Erweiterung von LHBench die definierten Anforderungen an ein Benchmark-Framework zur Bewertung der unterschiedlichen Ansätze erfüllt. Es ermöglicht damit die objektive Betrachtung der Ansätze hinsichtlich Performanz und Speicherplatzbedarf auf eine einfache Art und Weise.

Durch die Ausführung eines einzelnen Befehls wird eine Reihe an Benchmarks mit verschiedenen Ansätzen zur Speicherung unstrukturierter Daten und verschiedenen Lakehouse-Frameworks nacheinander automatisiert ausgeführt. Die Ergebnisse werden unmittelbar nach der Ausführung in einem zusammenfassenden Report dargestellt. Der Ablauf erweist sich als einfach und ermöglicht einen Vergleich der Ansätze. `LHBench-UnstructuredData` kann leicht um weitere Benchmarks erweitert werden und so andere Ansätze zur Speicherung unstrukturierter Daten betrachten, die über die in dieser Arbeit beleuchteten Ansätze hinausgehen. Die Erstellung der Reports passt sich automatisch an die Benchmarks an, die ausgeführt werden sollen. Dies ermöglicht eine problemlose Erweiterung sowohl auf neue Ansätze zur Speicherung unstrukturierter Daten als auch neue, zukünftig aufkommende, Lakehouse-Frameworks.

Die in dieser Arbeit präsentierten Ergebnisse unterliegen jedoch einigen Limitierungen: Für die Auswertung wurde in dieser Arbeit eine virtuelle Maschine verwendet, auf der ein YARN-Cluster ausgeführt wurde. Das Speichersystem, auf dem die Lakehouse-Tabellen abliegen, ist in diesem Fall also ein HDFS. In vielen Unternehmen werden mittlerweile jedoch meist Cloud Object Stores, wie Amazon S3 oder Google Cloud Storage als Speichersysteme für Datenplattformen eingesetzt. Das verwendete Cluster besteht zudem nur aus einem einzelnen physischen Knoten mit begrenzten verfügbaren Ressourcen. In realistischen Produktivumgebung bestehen die Cluster meist aus einer Vielzahl an Knoten mit mehr zugeordneten Ressourcen. Dadurch ergeben sich weitere Effekte wie etwa netzwerkbedingte Latenzen und die Möglichkeit des parallelisierten Zugriffs, die ebenfalls berücksichtigt werden müssen. Daher wäre es zu prüfen, inwiefern sich die im Rahmen dieser Arbeit erzielten Ergebnisse auf solche Produktivumgebung übertragen lassen.

Eine weitere Limitierung der Ausführungsumgebung besteht darin, dass sich die verwendete virtuelle Maschine auf einer Recheninfrastruktur befindet, die auch noch für weitere virtuelle Maschinen und Anwendungen verwendet wird. Dadurch kann die Performanz der virtuellen Maschine gewissen Schwankungen unterliegen, wenn die Infrastruktur zu verschiedenen Zeitpunkten unterschiedlich stark ausgelastet wird. Indem die Benchmarks mit je fünf Wiederholungen ausgeführt und die Ergebnisse dann gemittelt wurden, lässt sich diese Varianz zumindest teilweise abfedern. Über einen längeren Zeitraum anhaltende Schwankungen in der Auslastung lassen sich mit dieser Maßnahme jedoch nicht adressieren. Deshalb wäre es in Zukunft interessant zu betrachten, wie sich die verschiedenen Ansätze zur Speicherung unstrukturierter Daten verhalten, wenn `LHBench-UnstructuredData` auf einem größeren Cluster in der Cloud verwendet wird, das eventuell weniger Schwankungen unterliegt. Mit einer solchen Untersuchung könnte betrachtet werden, ob sich die erhaltenen Ergebnisse auch auf andere Umgebungskonfigurationen generalisieren lassen.

Des weiteren waren die ausgeführten Benchmarks, insbesondere mit dem `ImageFormat`, stark durch den verfügbaren Arbeitsspeicher in der virtuellen Maschine begrenzt, was teilweise zu Out-Of-Memory-Fehlern geführt hat. Es ist daher offen, ob das `ImageFormat` in manchen Testfällen etwas besser abgeschnitten hätte, wenn den Benchmarks mehr Arbeitsspeicher zur Verfügung gestanden hätte. Aufgrund des limitierten Arbeitsspeichers konnten auch nur Datensätze von eher kleinerem Umfang, bis zu einer Größe von einem Gigabyte, betrachtet werden. Auch hier wäre es sinnvoll, auf einem Cluster in der Cloud größere Datensätze zu testen. Zudem könnten auch noch Datensätze mit sehr großen Dateien mit mehreren Gigabyte Größe betrachtet werden.

Abschließend wurden die verschiedenen Ansätze in dieser Arbeit hinsichtlich qualitativer Unterschiede bewertet und basierend auf den konzeptionellen Aspekten und den Messergebnissen eine Empfehlung abgegeben, welche der Ansätze zur Speicherung unstrukturierter Daten sich für bestimmte Anwendungsszenarien am besten eignen.

10 Zusammenfassung und Ausblick

Lakehouses vereinen die Vorteile von Data Warehouses und Data Lakes, indem sie die Speicherung von strukturierten, semi-strukturierten und unstrukturierten Daten in offenen Dateiformaten auf Cloud Object Stores oder verteilten Dateisystemen erlauben. Eine Metadatenschicht ermöglicht zudem wichtige Verwaltungs- und Optimierungsfunktionen, wie sie aus Data Warehouses bekannt sind, wie beispielsweise ACID-Eigenschaften, Versionierung und Schema-Evolution. Diese Kombination ermöglicht sowohl die effiziente Ausführung von traditionellen Analysemethoden, wie Reporting und OLAP, als auch Advanced-Analytics-Techniken, wie Machine Learning.

Auf die Daten in Lakehouses kann mit verschiedenen Processing Engines, wie Apache Spark, zugegriffen werden. Aktuelle Frameworks, die für die Konstruktion solcher Lakehouses verwendet werden können, sind Delta Lake, Apache Iceberg und Apache Hudi.

In dieser Arbeit wurden verschiedene Ansätze betrachtet, wie unstrukturierte Daten in einem Lakehouse abgespeichert werden können. Dazu wurden vier Ansätze identifiziert. Grundsätzlich können die unstrukturierten Dateien entweder als Bytearray mit in der erstellten Lakehouse-Tabelle gespeichert werden (`BinaryFileFormat`). Alternativ kann in der Tabelle nur eine Referenz auf die eigentliche Datei abgelegt werden, verbunden mit weiteren Metadaten, über die dann die Originaldatei vom Speichersystem abgerufen werden kann (`OnlyMetadataFormat`). Für Bilddateien bietet Apache Spark einen weiteren Ansatz an, um diese abzuspeichern: Dabei werden diese zunächst dekomprimiert und dann in einem OpenCV-kompatiblen Format im Lakehouse abgelegt (`ImageFormat`). Diese verschiedenen Ansätze wurden zunächst auf konzeptioneller Ebene verglichen. Dabei konnte beispielsweise festgestellt werden, dass das ausschließliche Speichern einer Referenz in der Tabelle manche der von Lakehouses angebotenen Verwaltungsfunktionen einschränkt.

Darüber hinaus wurden die verschiedenen Ansätze auch hinsichtlich ihrer Performanz und dem Speicherplatzbedarf betrachtet. Dazu wurde ein bereits existierendes Benchmark-Framework für den Vergleich von verschiedenen Lakehouse-Frameworks verwendet und um entsprechende Funktionen und Benchmarks für den Vergleich der verschiedenen Ansätze erweitert. Dieses Framework `LHBench-UnstructuredData` ermöglicht die Durchführung von Benchmarks zur Evaluation unterschiedlicher Ansätze zur Speicherung unstrukturierter Daten mit verschiedenen Lakehouse-Frameworks. Im Anschluss an die Ausführung wird automatisiert ein Report mit Visualisierungen erstellt, der die Ergebnisse der Benchmarks gegenüberstellt. Das Framework lässt sich leicht auf weitere Ansätze zur Speicherung unstrukturierter Daten erweitern.

In der Folge wurde `LHBench-UnstructuredData` eingesetzt, um im Rahmen dieser Arbeit eine erste Evaluation der definierten Ansätze durchzuführen. Dazu wurde eine Testumgebung aufgebaut und konfiguriert, auf der die Benchmarks anschließend ausgeführt wurden. Die erhaltenen Ergebnisse der ausgeführten Benchmarks mit den verschiedenen Testdatensätzen wurden in Kapitel 8 vorgestellt und bewertet.

Basierend auf der konzeptionellen und der quantitativen Betrachtung der verschiedenen Ansätze, wurde schließlich eine Einschätzung abgegeben, welche der Ansätze in welchen Anwendungsszenarien eingesetzt werden sollten. In den allermeisten Fällen erscheint es dabei am sinnvollsten, die unstrukturierten Daten als Bytearray mit in der Lakehouse-Tabelle abzulegen (`BinaryFileFormat`). Aufgrund der Einschränkungen der verwendeten Testumgebung handelt es sich allerdings nur um vorläufige Ergebnisse mit begrenzter Aussagekraft, die vorrangig als Proof of Concept für das erstellte Benchmark-Framework zu erachten sind.

Ausblick

Für unstrukturierte Daten gibt es bereits viele Möglichkeiten und Ansätze, um diese zu analysieren und auf diese Weise Erkenntnisse für Unternehmen zu gewinnen. Durch den anhaltenden Aufschwung der Entwicklungen im Bereich der künstlichen Intelligenz werden in Zukunft voraussichtlich noch weitere Advanced-Analytics-Methoden hinzukommen, die unstrukturierte Daten verarbeiten können. Ebenso können bereits existierende Methoden weiter an Relevanz und Qualität hinzugewinnen. Dadurch wird sich für Unternehmen die Nutzbarkeit von unstrukturierten Daten in den nächsten Jahren weiter erhöhen. Deshalb ist es wichtig, dass auch Lakehouses einfache und effektive Methoden anbieten, um unstrukturierte Daten verwalten zu können.

Diese Arbeit trägt dazu bei, Ansätze zur Speicherung unstrukturierter Daten zu identifizieren und zu vergleichen. Die erhaltenen Ergebnisse unterliegen dabei jedoch Limitierungen, da die verwendete Testumgebung nur wenig repräsentativ für Produktivumgebungen ist, wie sie in Unternehmen vorliegen. Daher sollte in zukünftigen Arbeiten `LHBench-UnstructuredData` auch in einer umfangreicheren, realistischeren Umgebung durchgeführt werden. Auch könnte untersucht werden, welche weiteren Ansätze es noch gibt, um unstrukturierte Daten in Lakehouses abzulegen und wie diese dann im Vergleich zu den bereits eingeführten abschneiden. Es könnte insbesondere dabei betrachtet werden, welche speziellen Metadaten zu unstrukturierten Daten in Lakehouse-Tabellen noch mit abgespeichert werden sollten und welche dieser Informationen einen Mehrwert in verschiedenen Anwendungsfällen bieten. Darüber hinaus könnte nachgeforscht werden, ob andere Processing Engines als Apache Spark weitere interessante individualisierte Ansätze anbieten, um unstrukturierte Daten in Lakehouses abzulegen. Diese könnten dann mit den in dieser Arbeit eingeführten Ansätzen verglichen werden.

Literaturverzeichnis

[Abi97] S. Abiteboul. „Querying semi-structured data“. In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1997, S. 1–18. DOI: 10.1007/3-540-62222-5_33 (zitiert auf S. 43).

[ADS+20] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. van Hovell, A. Ionescu, A. Łuszczak, M. Świtakowski, M. Szafrański, X. Li, T. Ueshin, M. Mokhtar, P. Boncz, A. Ghodsi, S. Paranjpye, P. Senster, R. Xin, M. Zaharia. „Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores“. In: *Proceedings of the VLDB Endowment* 13.12 (Aug. 2020), S. 3411–3424. DOI: 10.14778/3415478.3415560 (zitiert auf S. 23–25, 31, 78).

[AGXZ21] M. Armbrust, A. Ghodsi, R. Xin, M. Zaharia. „Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics“. In: *Proceedings of CIDR*. Bd. 8. 2021 (zitiert auf S. 15, 18–21, 23, 30, 31, 43, 77).

[Apa15] Apache Hive. *Design*. Aug. 2015. URL: https://cwiki.apache.org/confluence/display/hive/design (zitiert auf S. 32).

[Apa22] Apache Parquet. *Documentation*. März 2022. URL: https://parquet.apache.org/docs/ (zitiert auf S. 23).

[Apa23a] Apache Hadoop. *Hadoop: Setting up a Single Node Cluster.* März 2023. URL: https://hadoop.apache.org/docs/r3.3.5/hadoop-project-dist/hadoop-common/SingleCluster.html (zitiert auf S. 30, 63).

[Apa23b] Apache Hive. *Hive Metastore Server (HMS)*. 2023. URL: https://hive.apache.org/ (zitiert auf S. 31).

[Apa23c] Apache Hudi. *Apache Hudi Technical Specification*. Okt. 2023. URL: https://hudi.apache.org/tech-specs (zitiert auf S. 27, 28).

[Apa23d] Apache Hudi. *HoodieTableConfig*. Okt. 2023. URL: https://github.com/apache/hudi/blob/master/hudi-common/src/main/java/org/apache/hudi/common/table/HoodieTableConfig.java (zitiert auf S. 28).

[Apa23e] Apache Hudi. *Overview*. Okt. 2023. URL: https://hudi.apache.org/docs/overview (zitiert auf S. 28).

[Apa23f] Apache Hudi. *What is Hudi*. Okt. 2023. URL: https://hudi.apache.org/ (zitiert auf S. 27).

[Apa23g] Apache Iceberg. *Documentation*. Juli 2023. URL: https://iceberg.apache.org/docs/1.3.1/ (zitiert auf S. 26, 27).

[Apa23h] Apache Iceberg. *Iceberg Table Spec*. Okt. 2023. URL: https://iceberg.apache.org/spec/ (zitiert auf S. 26, 27).

[Apa23i] Apache Spark. *Binary File Data Source*. Aug. 2023. URL: https://spark.apache.org/docs/3.3.3/sql-data-sources-binaryFile.html (zitiert auf S. 44).

[Apa23j] Apache Spark. *Cluster Mode Overview*. Aug. 2023. URL: https://spark.apache.org/docs/3.3.3/cluster-overview.html (zitiert auf S. 29).

[Apa23k] Apache Spark. *Data Types*. Aug. 2023. URL: https://spark.apache.org/docs/3.3.3/sql-ref-datatypes.html (zitiert auf S. 44).

[Apa23l] Apache Spark. *Datasets und DataFrames*. Aug. 2023. URL: https://spark.apache.org/docs/3.3.3/sql-programming-guide.html#datasets-and-dataframes (zitiert auf S. 30).

[Apa23m] Apache Spark. *Image data source*. Aug. 2023. URL: https://spark.apache.org/docs/3.3.3/ml-datasource.html#image-data-source (zitiert auf S. 47).

[Apa23n] Apache Spark. *RDD Programming Guide - Performance Impact*. Aug. 2023. URL: https://spark.apache.org/docs/3.3.3/rdd-programming-guide.html#performance-impact (zitiert auf S. 76).

[Apa23o] Apache Spark. *Scala Documentation: Dataset*. Aug. 2023. URL: https://spark.apache.org/docs/3.3.3/api/scala/org/apache/spark/sql/Dataset.html (zitiert auf S. 69).

[BG13] A. Bauer, H. Günzel. *Data-Warehouse-Systeme: Architektur, Entwicklung, Anwendung*. dpunkt. verlag, 2013 (zitiert auf S. 18).

[BH06] D. Boulton, M. Hammersley. „Analysis of unstructured data“. In: *Data collection and analysis* 2 (2006), S. 243–259 (zitiert auf S. 43).

[BP09] A. A. B. Saenz de Ugarte, R. Pellerin. „Manufacturing execution system – a literature review“. In: *Production Planning & Control* 20.6 (2009), S. 525–539. DOI: 10.1080/09537280902938613 (zitiert auf S. 18).

[CAG+23] J. Camacho-Rodríguez, A. Agrawal, A. Gruenheid, A. Gosalia, C. Petculescu, J. Aguilar-Saborit, A. Floratou, C. Curino, R. Ramakrishnan. *LST-Bench: Benchmarking Log-Structured Tables in the Cloud*. 2023. DOI: 10.48550/ARXIV.2305.01120 (zitiert auf S. 29, 33, 34, 36, 53, 54, 65).

[Cam23] J. Camacho-Rodríguez. *Merged Pull Request - Spark session client*. Juni 2023. URL: https://github.com/microsoft/lst-bench/pull/70 (zitiert auf S. 34).

[CD97] S. Chaudhuri, U. Dayal. „An overview of data warehousing and OLAP technology“. In: *ACM SIGMOD Record* 26.1 (März 1997), S. 65–74. DOI: 10.1145/248603.248616 (zitiert auf S. 15).

[Dat22] Databricks. *Binary file*. Okt. 2022. URL: https://docs.databricks.com/external-data/binary.html (zitiert auf S. 16).

[Dat23a] Databricks. *Image*. Okt. 2023. URL: https://docs.databricks.com/external-data/image.html (zitiert auf S. 16, 47, 49, 50, 80).

[Dat23b] Databricks. *Reference solution for image applications*. Okt. 2023. URL: https://docs.databricks.com/machine-learning/reference-solutions/images-etl-inference.html (zitiert auf S. 16).

[Dat23c] Databricks. *What is Delta Lake?* Okt. 2023. URL: https://docs.databricks.com/delta/index.html (zitiert auf S. 23).

[Del23] Delta Lake. *Delta Lake Integrations*. 2023. URL: https://delta.io/integrations (zitiert auf S. 24).

[Ebe+16] A. C. Eberendu et al. „Unstructured Data: an overview of the data of Big Data“. In: *International Journal of Computer Trends and Technology* 38.1 (2016), S. 46–50 (zitiert auf S. 43).

[GK06] M. Gupta, A. Kohli. „Enterprise resource planning systems and its implications for operations function“. In: *Technovation* 26.5-6 (Mai 2006), S. 687–696. DOI: 10.1016/j.technovation.2004.10.005 (zitiert auf S. 18).

[HGQ16] R. Hai, S. Geisler, C. Quix. „Constance: An Intelligent Data Lake System“. In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: ACM, Juni 2016, S. 2097–2100. ISBN: 9781450335317. DOI: 10.1145/2882903.2899389 (zitiert auf S. 20).

[Inm05] W. H. Inmon. *Building the data warehouse*. Third Edition. John wiley & sons, 2005 (zitiert auf S. 18).

[JKP+23a] P. Jain, P. Kraft, C. Power, T. Das, I. Stoica, M. Zaharia. „Analyzing and Comparing Lakehouse Storage Systems“. In: CIDR. 2023 (zitiert auf S. 27, 28, 35, 36, 53, 55, 65, 71).

[JKP+23b] P. Jain, P. Kraft, C. Power, T. Das, I. Stoica, M. Zaharia. *LHBench - Benchmarks*. Jan. 2023. URL: https://github.com/lhbench/lhbench/tree/main/src/main/scala/benchmark (zitiert auf S. 35, 36, 55, 56).

[JKP+23c] P. Jain, P. Kraft, C. Power, T. Das, I. Stoica, M. Zaharia. *LHBench - run-benchmark.py*. Jan. 2023. URL: https://github.com/lhbench/lhbench/blob/main/run-benchmark.py (zitiert auf S. 56).

[JKP+23d] P. Jain, P. Kraft, C. Power, T. Das, I. Stoica, M. Zaharia. *LHBench - Running This Benchmark Yourself*. Jan. 2023. URL: https://github.com/lhbench/lhbench/blob/main/experiment-instructions.md (zitiert auf S. 35, 55).

[JKP+23e] P. Jain, P. Kraft, C. Power, T. Das, I. Stoica, M. Zaharia. *LHBench - Scripts*. Jan. 2023. URL: https://github.com/lhbench/lhbench/tree/main/scripts (zitiert auf S. 56).

[KKR17] A. Kejariwal, S. Kulkarni, K. Ramasamy. *Real Time Analytics: Algorithms and Systems*. 2017. DOI: 10.48550/ARXIV.1708.02621 (zitiert auf S. 21).

[KW18] P. P. Khine, Z. S. Wang. „Data lake: a new ideology in big data era“. In: *ITM Web of Conferences* 17 (2018). Hrsg. von K. Eguchi, T. Chen. DOI: 10.1051/itmconf/20181703025 (zitiert auf S. 19, 20).

[Lub21] S. Luber. *Was ist Apache Hudi?* März 2021. URL: https://www.bigdata-insider.de/was-ist-apache-hudi-a-1006518/ (zitiert auf S. 27).

[Mic23] Microsoft. *LST-BENCH*. Okt. 2023. URL: https://github.com/microsoft/lst-bench (zitiert auf S. 33).

[Mon23] MongoDB. *Unstructured Data*. 2023. URL: https://www.mongodb.com/unstructured-data (zitiert auf S. 43).

[Net18] Netflix. *Iceberg*. Dez. 2018. URL: https://github.com/Netflix/iceberg (zitiert auf S. 26).

[SD21] P. N. Sawadogo, J. Darmont. „Benchmarking Data Lakes Featuring Structured and Unstructured Data with DLBench“. In: *Big Data Analytics and Knowledge Discovery*. Springer International Publishing, 2021, S. 15–26. DOI: 10.1007/978-3-030-86534-4_2 (zitiert auf S. 37).

[SDC+16] S. Salloum, R. Dautov, X. Chen, P. X. Peng, J. Z. Huang. „Big data analytics on Apache Spark“. In: *International Journal of Data Science and Analytics* 1.3-4 (Okt. 2016), S. 145–164. DOI: 10.1007/s41060-016-0027-9 (zitiert auf S. 29).

[SGL+23] J. Schneider, C. Gröger, A. Lutsch, H. Schwarz, B. Mitschang. „Assessing the Lakehouse: Analysis, Requirements and Definition“. In: *Proceedings of the 25th International Conference on Enterprise Information Systems (ICEIS 2023)*. Scitepress 2023, 2023 (zitiert auf S. 15, 19, 21–23, 40).

[SVD00] T. V. Scannell, S. K. Vickery, C. L. Droge. „Upstream supply chain management and competitive performance in the automotive supply industry“. In: *Journal of Business Logistics* 21.1 (2000), S. 23 (zitiert auf S. 18).

[VZ22] A. Vaisman, E. Zimányi. *Data Warehouse Systems*. Second Edition. Springer Berlin Heidelberg, 2022. DOI: 10.1007/978-3-662-65167-4 (zitiert auf S. 17, 18, 29–31).

[Whi12] T. White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012 (zitiert auf S. 30).

Alle URLs wurden zuletzt am 18. 10. 2023 geprüft.

Erklärung

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

Ort, Datum, Unterschrift