Masterarbeit im Studiengang Informatik der Fakultät Informatik

Methoden zur Persistierung eines dynamischen Datenmodells für OLTP Anwendungen

zur Erlangung des akademischen Grades eines Master of Science

angefertigt von

Matthias Strauß

ausgegeben am: 16.01.2021
abgegeben am: 14.07.2021

Betreuer:

Erstprüfer: Regensburger, Prof. Dr. rer. nat. Franz
Zweitprüfer: Göldner, Prof. Dr.-Ing. Ernst-Heinrich
EPOS CAT GmbH: Köll Thomas

Kinding, 14.07.2021

Abstract

Viele moderne online Anwendungen geben ihren Benutzern große Autonomität über das Modell der zu verwaltenden Daten. Dadurch entsteht die Notwendigkeit, das konkrete Datenmodell der Anwendung dynamisch und anpassbar zu machen. Das steht nicht in Einklang mit der Grundannahme gängiger Datenbankanwendungen, die ein festes Schema bevorzugen. Diese Arbeit beleuchtet neuartige NO-/ und NewSQL Datenbanken für die Verwendung unter des Online Transaction Processing (OLTP) Paradigma und analysiert die Performance zweier Lösungswege für relationale Datenbanken.

Erklärung

Ich erkläre hiermit, dass ich die Arbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Kinding, den 14.07.2021

(Matthias Strauß)

Abkürzungsverzeichnis

ACID Atomicity Consistency Isolation Durability

BLOB Binary Large Object

CLOB Character Large Object

DBMS Datenbank Management System

DCL Data Control Language

DDL Data Description Language

DML Data Modeling Language

DQL Data Query Language

EAV Entity-Attribute-Value

GIN Generalized Inverted Index

HTAP Hybrid Transactional and Analytics Processing

IDS Integrated Data Store

JAXB Java Architecture for XML Binding

JDBC Java Database Connectivity

JPA Jakarta Persistence API

JPQL Jakarta Persistence Query Language

JSON Javascript Object Notation

LOB Large Object

NOSQL Not Only SQL

NCLOB National Character Set Large Object

ODBC Open Database Connectivity

OLAP Online Analytical Processing

OLTP Online Transaction Processing

ORM object-relational mapping

POJO Plain Old Java Objects

SQL Structured Query Language

XML Extensible Markup Language

Inhaltsverzeichnis

1 Einleitung **8**

2 Dynamische Datenschemas in relationalen Datenbanken **10**
2.1 Geschichte und Grundlagen 10
2.2 Datenbank Normalisierung: Normalformen 12
2.3 Object Relational Mapping 13
2.4 Ansätze für dynamische Attribute in relationalen Datenbanken . . . 15
2.4.1 Bearbeitung des Schemas zur Laufzeit 15
2.4.2 Entity-Attribute-Value Modell 17
2.4.3 Speichern von komplexen Daten in LOBs 19
2.4.4 Weitere Lösungsansätze 22

3 Moderne Datenbanken **23**
3.1 NOSQL 23
3.1.1 Datenmodell 23
3.1.2 Datenkonsistenz Modell 26
3.1.3 Partitionierung 26
3.1.4 CAP Theorem 27
3.2 NewSQL 29

4 Performance Analyse relationaler Lösungswege **30**
4.1 Test-Dimensionen 30
4.1.1 Wahl der Datenbank 31
4.1.2 Anwendungsfälle 33
4.1.3 Indizierung 35
4.2 Testaufbau 36
4.2.1 Programmiertechnische Umsetzung 37
4.2.2 Wiederverwendbarkeit und Reproduktion der Anwendung und Tests 42
4.2.3 Daten 44
4.2.4 Hardware und Laufumgebung 46
4.3 Ergebnisse 46
4.3.1 Statistische Analyse der Ergebnisse 49
4.3.2 Interpretation der Ergebnisse 63
4.4 Fazit 67

5 Abschluss **70**

Abbildungsverzeichnis **71**

Quellcodeverzeichnis 73

Literaturverzeichnis 74

1 Einleitung

Sobald eine Anwendung ein gewisses Maß an Daten persistent ablegen muss, kommt man als Entwickler nicht an Datenbanksysteme vorbei. Sie ermöglichen performante Methoden, Daten zu speichern, zu löschen und zu durchsuchen. Die bekannteste Art von Datenbanken sind relationale. Sie speichern Daten in tabellarischen Strukturen ab, zwischen denen sich Einträge gegenseitig referenzieren können. Datentypen für bestimmte Spalten ermöglichen es, Daten effizient zu speichern und unlogische Werte zu vermeiden. Das relationale Modell ermöglicht es, viele verschiedene Aufgabenstellungen in ihm abzubilden. Zusätzlich existiert mit SQL eine mächtige Sprache, die alle Aspekte der Datenverwaltung übernehmen kann. Doch nicht jede Art von Informationen kann gut von dem relationalen Modell repräsentiert werden. Besonders unter den Herausforderungen, die das Internet an moderne Anwendungen stehlt, wurden neue Datenbanken benötigt. Dazu gehören Aspekte wie Ausfallsicherheit, extreme Mengen an Benutzer, die gleichzeitig auf die Datenbank zugreifen, und Daten oder auch andere Datenmodelle, die nicht in das relationale Modell passen. In dieser Arbeit werden Lösungen für ein Problem gesucht, das nicht direkt zu den Problemen, die moderne Datenbanken lösen wollen, gehört. Viele Arten von Informationen müssen ständig erweitert werden. Dazu gehören medizinische Daten, die Elementen neue Eigenschaften zuweisen müssen, e-commerce Systeme, die zwischen allen Kategorien von Daten tausende von Attribute verwalten müssen, und Team-Kollaboration Systeme, die Benutzern ermöglichen, eigene Strukturen anzulegen und zu erweitern. Diese Systeme haben die Anforderung, das Modell dynamisch zur Laufzeit anzupassen und Objekten neue Attribute mit neuen Werten hinzuzufügen oder

alte zu entfernen. Dies steht im Konflikt mit dem festen Schema von relationalen Datenbanken, das in der Regel nur bei Versionsänderungen der zugehörigen Anwendung geändert wird. Um verschiedene Ansätze, die diese Aufgabenstellung erfüllen, zu vergleichen, wurde ein Java-Programm geschrieben. Dieses Programm ermöglicht es, die Performance von typischen Anwendungsfällen, wie den CRUD-Operationen, zu vergleichen und kann beliebig mit neuen Lösungswegen erweitert werden. Der Programmcode dieses Programms ist frei unter der MIT Lizenz unter `https://github.com/matthias-epos/dynamic-datamodels` verfügbar. Das Programm wurde verwendet, um die Perfomance zweier Ansätze in zwei verschiedenen Datenbanksysteme zu testen und zu vergleichen.

Im Rahmen dieser Arbeit werden zunächst relevante Grundlagen relationaler Datenbanken erläutert. Im Anschluss werden die Aspekte von NO- und NewSQL Datenbanken näher betrachtet, um ihre Qualitäten und Schwächen besser einschätzen zu können. Dadurch wird ihre Tauglichkeit für die Aufgabenstellung untersucht. Im Hauptteil der Arbeit wird die Testumgebung beschrieben, mit der die relationalen Ansätze verglichen wurden. Dazu gehören die Architektur des Testprogramms, die verwendeten Technologien, Entscheidungsmerkmale und Stolpersteine, die die Entwicklung beeinflussten. Zum Abschluss werden die Ergebnisse präsentiert, statistisch untersucht und interpretiert.

2 Dynamische Datenschemas in relationalen Datenbanken

Um die Herausforderung, dynamische Datenschemas in Datenbanken abzuspeichern, besser zu verstehen, werden in diesem Kapitel die Grundlagen von relationalen Datenbanken näher betrachtet. Anschließend werden verschiedene Lösungswege in relationalen Datenbanken für die vorlegende Aufgabenstellung beschreiben.

2.1 Geschichte und Grundlagen

Da die direkte Ablage von Programmdaten in Dateien zur langfristigen Speicherung sich schnell als sehr unflexibel herausstellte, wurde bereits 1963 das erste Datenbank Management System (DBMS), das IDS, von Bachman entwickelt [26]. Obwohl damals die Hardware und die Speichermedien, hauptsächlich Tapes, die größeren Herausforderungen waren, besitzt Integrated Data Store (IDS) viele Aspekte moderner DBMS, unter anderem die Verwaltung von Metadaten über die gespeicherten Daten und die Möglichkeit, dass verschiedene Programme auf Basis der selben Datenbank arbeiten. Spätere Versionen beherrschten bereits die Verarbeitung von Online-Transaktionen. Daten in IDS wurden in einer hierarchischen Struktur abgelegt. Die heute dominierende Form von Datenbanken und deren relationales Modell wurden erst später von Ted Codd entwickelt [19]. Der Begriff Relation stammt aus der Mengenlehre und beschreibt eine Menge von n-Tupel, d.h. von Folgen mit einer festen Menge von Elementen. Die einzelnen Skalare bzw. Werte

an den Stellen eines Tupels können nur eine, für diese Stelle vorbestimmte, Form annehmen. Die jeweilige Stelle wird Attribut genannt und die möglichen Werte Wertebereich. Ein Tupel wird mit einem Primärschlüssel identifiziert. Mit diesem Schlüssel können Referenzen zwischen Tupeln in verschiedenen Relationen indiziert werden. Das relationale Datenmodell wird basierend auf den Strukturen, die sich aus den mathematischen Grundlagen ergeben, meist mit Tabellen beschrieben. Eine Relation ist eine Tabelle, die Tupel werden zu Reihen und Attribute werden mit Spalten visualisiert. Der erste Vertreter von relationalen Datenbanken, System R, verwendete die erste Implementation der DBMS-Interaktionssprache Structured Query Language (SQL) [16]. Modernes SQL beinhaltet folgende Sprachen die zur Datenhaltung benötigt werden [34]:

- Data Modeling Language (DML) enthält Befehle, mit denen Daten in der Datenbank modelliert, d.h. Einträge eingefügt und bearbeitet, werden können.
- Data Description Language (DDL) wird verwendet, um eine konkrete Instanz des Datenmodels zu erstellen. Das entstehende Konstrukt heißt Datenschema.
- Data Control Language (DCL) beinhaltet Befehle zur Anpassung von Rechten auf die Daten.

Die Kombination einer sehr flexiblen Sprache in SQL und einem Modell, das große Teile der benötigten Daten von Unternehmen hocheffizient abspeichern kann, ermöglichten den Aufstieg der relationale Datenbanken[1] zu ihrer heutigen Dominanz auf dem Markt. Trotzdem besitzt dieses Modell mehrere Schwächen.

Eine Schwäche ist die mangelnde Dynamik des Datenschemas, welche diese Arbeit motiviert. Änderungen am Schemas sind grundsätzlich möglich, sind jedoch

[1]Genau genommen handelt es sich um SQL Datenbanken. Die umgangssprachliche Verwendung der Bezeichnung *relationale Datenbank* entspricht nicht der Definition des Begriffs. Ein Unterschied zwischen der Definition und der heutigen Implementationen ist z.B. die Möglichkeit, NULL Werte abzuspeichern. Das ist jedoch nicht Teil der Spezifikation der relationalen Datenbank.

nicht sehr performant. Die ANSI-SPARC Architektur für Datenbankschemas verspricht logische Datenunabhängigkeit. In der Praxis sind jedoch meist externe Anwendungen und das Schema direkt gekoppelt. Für Softwareprojekte werden bei Datenbankänderungen oft Migrations-Scripts angelegt, die das Datenbankschema in einem großem Schritt umwandelt. Volatile und dynamische Datenstrukturen können nur mit Umwegen, die im Anschluss betrachtet werden, abgespeichert werden. Eine andere Schwäche ist die Skalierbarkeit der Datenbanken, verglichen mit speziell entwickelten verteilten Datenbanken [42]. Fehlende Partitionierungsmethoden erschweren die horizontale Skalierung und erfordern stattdessen stärkere Hardware um größere Datenmengen abzulegen und mehr Transaktionen zu verarbeiten.

2.2 Datenbank Normalisierung: Normalformen

Datenbanknormalisierung ist ein Vorgang während der Definition des Datenmodels zur Minimierung von Datenanomalien und Eliminierung von Redundanz. Mithilfe der Normalformen werden Strukturen definiert, die die Abhängigkeiten zwischen den Daten einschränkt. Die Normalisierung ist ein Vorgang mit Vor- und Nachteilen.

- **Datenanomalien**

  Es existieren verschiedene Arten von Datenanomalien. Update-Anomalien entstehen, wenn die Bearbeitung von Daten zu inkonsistenten Zuständen durch Duplikate führen kann. Einfüge-Anomalien beschreiben den Zustand, wenn neue Entities nicht ohne zusätzliche Attribute in die Datenbank eingefügt werden können. Lösch-Anomalien treten durch Datenverluste auf. Diese Anomalien lassen sich auf duplizierte und schlecht strukturierte Datenmodelle zurückführen, die durch die Normalisierung verhindert werden.

- **Speicherverbrauch**

  Ein Nebeneffekt der Eliminierung von Duplikaten in der Datenbank ist die Verringerung des Speicherbedarfs der Datenbank.

- **Performanz**

  Die Normalisierung teilt Informationen, die zunächst in einer einzigen Tabelle gespeichert waren, oft in mehrere Tabellen auf, je nach den Abhängigkeiten der Daten. Um diese Verhältnisse in einer Abfrage wieder darzustellen und abzulaufen müssen unperformante join Operationen durchgeführt werden. Dadurch steigt die durchschnittlichen Antwortzeit.

Es existieren mehrere Normalformen, die jeweils die Struktur des Modells immer weiter einschränken. Ein nicht-normalisiertes Modell wird auch nullte Normalform genannt. Die erste Normalform bringt die Einschränkung, das alle Attribute nur einfache Werte annehmen dürfen. Das bedeutet, dass ein Wert keine Menge sein darf und keine innere Struktur, die weiter verarbeitet werden muss, beinhalten darf. Diese Eigenschaft wird auch als atomar bezeichnet. Die bekannteste und häufig angestrebte Normalform ist die dritte nach Boyce/Codd Normalform.

> „3. Normalform (Boyce/Codd): Eine Relation R in l.NF ist auch in 3.NF, wenn jede Determinante in R ein Schlüsselkandidat ist. (Determinante heißt ein Attribut oder eine Attributskombination, von welchem irgend ein anderes Attribut von R voll abhängig ist.)“ [52, Seite 52]

Zusätzlich existieren höhere Normalformen, die die Strukturen weiter eingrenzen.

Höhere Normalformen sind in praktischen Umgebungen seltener in Gebrauch, da bereits die dritte Normalform alle Redundanzen entfernt. [52]

2.3 Object Relational Mapping

Datenbanksysteme müssen Informationen echter Problemstellungen abbilden können. Dazu erstellen Datenbankentwickler Schemas, die diese Informationen innerhalb des Datenbankmodells ablegen. In Objektorientierten Programmiersprachen

müssen die Informationen auch in ein Klassenschema eingefügt werden. Dieses unterscheidet sich von dem Datenbankschema, da auch die zugrunde liegende Modelle sich unterscheiden. Ein Programm muss trotz dieser Unterschiede die Informationen von Objekten in die Datenbank übertragen, um sie persistent abzuspeichern. Dieses Problem wird auch "object-relational impedance mismatch" genannt. [20] Eine Lösung dafür ist die Zuweisung von Objektdefinitionen zu einer relationalen Abbildung, bekannt unter dem Begriff object-relational mapping (ORM). Da die Persistenz von Programmdaten eine Grundaufgabe vieler Programme ist, ist dies ein bereits sehr erschlossenes Gebiet. ORM-Frameworks erleichtern die Arbeit, Datenbanksysteme und -modelle zu konfigurieren und relevante Elemente von Klassen zur Persistierung zu markieren. Ein Framework, dass im Java Bereich sehr viel Einsatz findet, ist Jakarta Persistence API (JPA)[2]. Es beschreibt einen Standard, der die genannten Aufgaben erfüllt. Dazu gehört ein Entity-Manager, der für die Verwaltung, darunter dem Erstellen und Löschen, der Objekte verantwortlich ist. Außerdem existiert eine spezielle Query-Sprache, genannt Jakarta Persistence Query Language (JPQL), mit der Datenbankoperationen durchgeführt werden können [38]. Da es sich nur um eine Spezifikation handelt, gibt es mehrere Implementationen. Gut bekannte Implementationen sind Hibernate [8] und Apache Cayenne [12]. Für das Thema dieser Arbeit sind mehrere Aspekte dieser Frameworks interessant: Zunächst betrachten wir, ob diese Frameworks unsere Aufgabenstellung erfüllen könnten. Dafür müssten sie flexible Felder ermöglichen oder Schemas dynamisch anpassen können. Sie sind jedoch weiterhin an feste Versionen der Schemas, die zu bestimmten Programmversionen gehören, gebunden. Datenbank-Schema Migration ist eine besondere Herausforderung. Zwar können Tools wie Liquibase [11] bei der Verwaltung von Datenbank-Schemas helfen, jedoch ermöglichen sie nicht komplett dynamische Schemas. Daher können ORM Frameworks die Aufgabenstellung nicht direkt erfüllen. Sie könnten uns jedoch bei der Betrachtung von Lösungswegen, die innerhalb eines festen Schemas umgesetzt werden, helfen. Solche Lösungswege könnten jedoch von speziellen Datenbank-

[2]Früher bekannt als Java Persistance API

Features, die nur von einzelnen Herstellern geboten werden, Gebrauch machen. Hibernate und ähnliche Frameworks erschweren die Verwendung dieser, da sie möglichst starke Entkopplung von einzelnen Datenbankanbietern bieten wollen. Daher entschieden wir uns, diese Frameworks nicht zu verwenden und uns näher an der Datenbankschicht zu bewegen. Dafür wurde stattdessen Java Database Connectivity (JDBC) verwendet, was den Einsatz von reinem SQL ermöglicht. Trotzdem bieten diese Frameworks eine gute Grundlage für das Design unserer Tests. Dazu gehören Modell-Klassen und Klassen die für die Verwaltung unserer Test-Entities zuständig sind. Ein weiteres Feature, welches solche Frameworks bieten, sind Caches. Caches können wiederholt angeforderte Daten zwischenspeichern und dadurch die Performance verbessern. Ein solches Feature wurde im Rahmen dieser Arbeit nicht umgesetzt, wäre jedoch für den Einsatz in einem realen Projekt ein Feature mit hoher Priorität.

2.4 Ansätze für dynamische Attribute in relationalen Datenbanken

2.4.1 Bearbeitung des Schemas zur Laufzeit

SQL, die Sprache zur Interaktion mit relationalen Datenbanken, ermöglicht nicht nur die Eingabe und Abfrage von Daten, sondern auch die Manipulation des Schemas. Dadurch kann zur Laufzeit eines Programmes das Schema angepasst werden. Dieses Feature können ORM Frameworks zur automatischen Erstellung des Schemas verwenden. Es wird jedoch oft davon abgeraten, dieses Feature zu verwenden, da das resultierende Schema oft angepasst werden muss oder nicht sehr stabil ist [35]. Trotzdem lässt sich dieser Ansatz für unsere Problemstellung verwenden. Hier haben wir die Wahl zwischen zwei Möglichkeiten. Eine sehr weite Tabelle, deren Spalten modifiziert und erweitert werden, oder viele Tabellen, eine für jede Variation der Attributkombinationen. Der Ansatz einer Tabelle mit einer Spalte pro möglichem Attribut, die zur Laufzeit weitere Spalten erhält, hat zwei

Probleme. Zum einen existiert in jedem Datenbanksystem eine Obergrenze, wie viele Spalten eine Tabelle maximal haben kann. Diese liegt z.B. bei PostgreSQL bei 1600 [44] und bei Oracle Database bei 1000 [40]. Dadurch wäre die maximale Anzahl an Attributen eingeschränkt. Das zweite Problem ist die "sparse"-heit der Tabelle, d.h. sie besteht aus Reihen, die größtenteils nur NULL-Werte enthalten. Manche Datenbankanbieter bieten Features, die Spalten dieser Art besser verarbeiten können. Dennoch haben sie einen schlechten Einfluss auf die Performance von Abfragen. [18] Die zweite Variante ersetzt eine einzige Tabelle durch viele. Sie hat aber die Voraussetzung, dass sich bestimmte Attributkombinationen wiederholen müssen, wie Klassen in einer objektorientierten Programmiersprache. Jede Klasse würde von einer Tabelle repräsentiert werden. Dies verhindert Reihen mit vielen NULL Einträgen, da nur Reihen in Tabellen mit passenden Spalten abgelegt werden. Beide Ansätze haben das Problem, dass während der Bearbeitung des Schemas der Datenbank, d.h. bei dem Hinzufügen und Entfernen von Spalten, kein Zugriff auf die Tabelle möglich ist. Dies ist ein Ausschlusskriterium für Anwendungen, die viele Aufrufe gleichzeitig und mit niedriger Latenz verarbeiten müssen. Daher eignen sich Methoden, die zur Laufzeit das Schema ändern, eher für Anwendungsfälle, bei denen die Daten der Tabellen erst ab einen bestimmten Zeitpunkt zugänglich sein müssen und sich anschließend nicht mehr oft ändern. Ein solcher Fall ist z.B. die Bereitstellung eines persistenten Speichers für Add-Ons für eine Anwendung [7]. Es existieren auch Lösungen, die bei der Änderung des Schemas eine Kopie der Tabelle erstellen, an dieser die Änderung durchführt und anschließend alle Daten kopiert, bevor die Originalversion gelöscht wird [24]. Eine weitere Gefahr ist der Verlust von Daten. Ein sich ständig änderndes Schema erschwert die Verwaltung von Backups und könnte zu Datenverlust führen, falls Attribute und dadurch Spalten entfernt werden würden. Da die untersuchten Anwendungsfälle nicht parallel getestet wurden, würden Lösungen dieser Art wahrscheinlich sehr gut abschneiden. Ihre Schwächen würden sich erst bei mehreren parallelen Zugriffen, die regelmäßig neue Attribute hinzufügen, zeigen. Daher wurde dieser Ansatz im Rahmen dieser Arbeit zwar nicht getestet, würde aber ein interessanter

Kandidat bei komplexeren Anwendungsfällen sein.

2.4.2 Entity-Attribute-Value Modell

Eine weitere Möglichkeit, ein dynamisches Modell in einer relationalen Datenbank abzulegen, ist das Entity-Attribute-Value (EAV)-Modell. Anstelle eines fest definierten Schemas für alle realen Entities der abzubildenden Informationen benutzt man ein Schema, in dem sich alle Daten generisch abspeichern lassen. Dieses Vorgehen fand bereits in verschiedenen Projekten, insbesondere in medizinischen Bereichen, die oft höchst dynamische Daten abspeichern müssen, Verwendung. Dazu gehören z.B. das Columbia-Presbyterian Medical Center [28] oder SenseLab, eine Webanwendung zur Verwaltung von Forschungsdaten über Neuronen und neuronale Systeme [31,33]. Das Modell besteht aus drei Teilen.

- **Entity** Das Objekt, dass beschrieben wird.
- **Attribut** Die Eigenschaft, die einem Objekt zugeschrieben werden kann. Dazu gehören Name, mögliche Werte und Typen, Beschreibungen und Ähnliches.
- **Value** Der jeweilige Wert, den ein Objekt für ein Attribut hat.

In einem relationalem Schema kann EAV wie folgt abgebildet werden:

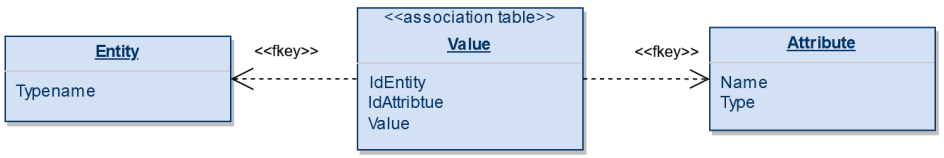

Abbildung 2.1: Datenbankmodell EAV Ansatz

Für jeden der drei Teile wird eine Tabelle angelegt. Alle Eigenschaften, die jedes Objekt besitzen soll, bekommen eine Spalte in der Entity Tabelle. Dazu gehört die Id, die in jeder Umsetzung existieren muss, oder auch optionale Werte wie der Zeitpunkt der Erstellung oder der Ersteller des Objekts. Die Definition der dynamischen Attribute, zu der der Name und logische Datentyp gehören, werden

in der Attribute Tabelle abgelegt. Die Value Tabelle wird verwendet, um Objekten dynamisch neue Attribute zusammen mit den zugehörigen Werten zuzuweisen. Ein Eintrag in dieser Tabelle benötigt als Fremdschlüssel die Id des Objektss, die Id des Attributs und ein Wert für dieses Attribut. So kann einem Objekt beliebige Attributwerte zugeordnet werden.

Der Einsatz von EAV in einer relationalen Datenbank wird meist als Anti-Pattern bezeichnet oder wird als Indikator für schlechtes Datenbankdesign genommen. Das ist insbesondere der Fall, wenn EAV verwendet wird, um ein höchst generisches Modell zu erstellen, in dem alle Daten abgelegt werden können. Hier spricht man von dem Inner Platform Effekt, wiederum ein Anti-Pattern und Folge von Soft-Coding [29]. In gewisser weise ist jedoch die Aufgabenstellung, dynamisch Attribute zu Objekten hinzuzufügen und zu entfernen, auch ein Indikator für den Inner Platform Effekt. Daher sollten diese Bedenken ignoriert werden.

Die Probleme bzw. Gefahren durch EAV sind vielfältig [46]:

Performance Durch die Aufteilung der Informationen einer Entity müssen bei jeder Query mehrere Joins durchgeführt werden. Außerdem ist es nicht möglich, Indizes auf bestimmte Attribute, die Teil besonders vieler Querys sind, zu legen.

Datenintegrität Je nach Komplexität der Umsetzung des EAV Models ist die Aufrechterhaltung von Datenintegrität erschwert. In der einfachsten Umsetzung werden alle Attributwerte als Text oder binär abgelegt. Dadurch gehen Tests für logische Werte auf Datenbankebene verloren. Administratoren könnten bei direkter Eingabe von Daten in die Datenbank unlogische Werte eintragen. Außerdem erschwert dieses Vorgehen die Einsicht der gespeicherten Daten. Bei Anwendungsfehlern müssen Support-Mitarbeiter oft basierend auf Anwendungsqualität die Daten auf Datenbankebene auf Fehler untersuchen. Dieses Vorgehen ist durch die Art wie EAV Daten ablegt erschwert.

Aussagen über Datenbestände Die Erstellung von Berichten über gespeicherte Daten ist nur mithilfe von speziellen SQL-Befehlen möglich. In normalen relatio-

nalen Datenbankschemas sind Aufgabenstellungen wie die Ausgabe aller Einträge einer Tabelle, d.h. einer logischen Struktur, einfacher umsetzbar.

Dennoch wird EAV als möglicher Lösungsweg für verschiedene Probleme vorgeschlagen. Dazu gehören zum einen das Thema dieser Arbeit, dynamische Schemas, aber auch dünn besetzte Daten. Ein großes Einsatzfeld von EAV sind Datenbanken für Daten aus klinischen Studien. Dazu gehören sowohl Open-Source Datenbanken wie Open Source Vista oder kommerzielle Produkte wie ClinTrial oder Clinical. Ein anderes Beispiel ist SenseLab, ein Projekt das Daten zu bestimmten biologischen Objekten im Bereich des Gehirn einmal abspeichert und dazu verschiedene Webmasken anbietet, die je nach Kontext die Daten unterschiedlich präsentieren. Das unterliegende Datenmodell wird in Organization of heterogeneous scientific data using the EAV/CR representation ausführlich beschrieben [36]. Es handelt sich um eine Erweiterung des EAV Modells durch Klassen und Relationen.

2.4.3 Speichern von komplexen Daten in LOBs

Relationale Datenbanken besitzen neben einfachen Datentypen wie `int` oder `varchar` auch meist einen Typ für komplexe Objekte wie Bilder oder Binärdateien, die nur semi-/ oder nicht strukturiert sind. Dieser Typ wird in der Regel Large Object (LOB) genannt. Manche Datenbankanbieter stellen verschiedene Variationen dieses Typs zur Verfügung, etwa Oracles Binary Large Object (BLOB), Character Large Object (CLOB) und National Character Set Large Object (NCLOB) Typen.

Dieser Datentyp kann dazu verwendet werden, Daten mit dynamischen Schema abspeichern. Die binäre Repräsentation des Objekts aus dem Programm direkt abzuspeichern ist nicht empfehlenswert, da sich diese je nach Programmiersprache, Betriebssystem und Programmversion unterscheiden können. Stattdessen werden Austauschformate verwendet. Dafür gibt es zwei oft verwendete Standards: Extensible Markup Language (XML) und Javascript Object Notation (JSON).

XML XML ist ein Standard des World Wide Web Consortiums für den Austausch von strukturierten Informationen über das Internet. XML Dokumente müssen ei-

ner strikten Form von sich öffnenden und schließenden Tags folgen. Diese Tags markieren Elemente, die bestimmte Attribute haben und auch weitere Elemente enthalten können. Mithilfe der Elemente werden die benötigten Informationen des Dokuments abgebildet. [3]

Rund um XML haben sich verschiedene Tools entwickelt, die die Arbeit mit dem Format erweitern können. Eines ist XPath, eine Abfragesprache um Informationen aus einem XML Dokument zu extrahieren [2]. Wohlgeformte XML Dokumente, d.h. Dokumente die allen Regeln der XML Definition folgen, bilden eine Baumstruktur. XPath erlaubt es, Knoten basierend auf ihren Namen, Attributen und Position in der Struktur auszuwählen und auszulesen. Ein weiteres Werkzeug für XML ist XSD, eine Schemabeschreibungssprache für XML Dokumente [4]. XSD ermöglicht es, das Schema bzw. die Struktur von einer Klasse von XML Dokumenten zu definierten. Dafür wird wiederum ein XML Dokument verwendet. Damit lassen sich Dokumente z.B. validieren und ohne Fehler automatisch auslesen.

JSON Laut des offiziellen ECMA-Standards ist:

> „JSON (...) a text format that facilitates structured data interchange between all programming languages. JSON is syntax of braces, brackets, colons, and commas that is useful in many contexts, profiles, and applications.“ [21, Seite iii]

JSON basiert auf dem Objektformat von Javascript und ermöglicht es, mit Text Objekte zu beschrieben. Diese Objekte bestehen aus Key-Value Paaren, die als Wert wiederum Objekte, Arrays oder nur einfache Werte haben. JSON wird oft zum Datenaustausch in Webumgebungen verwendet etwa in REST Schnittstellen. Durch das einfache und für den Menschen verständliche Format hat die Popularität von JSON seit der Definition 2006 stetig zugenommen und ersetzt inzwischen XML in manchen Einsatzgebieten.

Da diese Datenformate generell für viele Einsatzzwecke verwendet werden, bieten Datenbanksysteme oft spezielle Mechanismen um sie in LOBs abzulegen. Entweder durch einen eigenen Datentyp, wie es bei `XMLType` für Oracle Database und `JSONB`

Listing (2.1) Beispiel XML

```xml
<users>
  <user>
    <firstname>Max</firstname>
    <lastname>Fischer</lastname>
  </user>
  <user>
    <firstname>Lisa</firstname>
    <lastname>Schmidt</lastname>
  </user>
</users>
```

(a) per XML

Listing (2.2) Beispiel JSON

```json
{
  "users" : [
    { "firstName" : "Max",
      "lastName"  : "Fischer"
    },
    { "firstName" : "Lisa",
      "lastName"  : "Schmidt"
    }
  ]
}
```

(b) per JSON

Abbildung 2.2: Vergleich des Aufbau

für Postgres der Fall ist, oder Constraints, wie der JSON check in Oracle Database. Dadurch unterscheidet sich die Umsetzung von Datenbank zu Datenbank. Das macht es um so schwieriger, die Features durch eine allgemeine Schnittstelle, wie z.B. JDBC in einer Java-Umgebung oder Open Database Connectivity (ODBC), einzusetzen.

Der Vorteil eines solchen Datentyps ist jedoch, dass dadurch auch Unterstützung durch spezielle Indizes zur Verfügung gestellt werden.

Datenbankindizes werden dazu verwendet, um Suchanfragen an die Datenbank zu beschleunigen. Ohne einen Index muss die gesamte Tabelle nach passenden Einträgen durchsucht werden. Dieser Vorgang ist als full table scan oder sequentieller Scan bekannt. Dieser Scan ist meistens die letzte Wahl. Es existieren verschiedene Index Typen, die verschiedene Datenstrukturen und Einsatzzwecke haben. Am meisten Einsatz finden B-Tree Indizes, die besonders gut geeignet sind, um Einträge mit genauen Werten oder Werten in einem bestimmten Bereich zu finden. Genauer gesagt wird im Index hinterlegt, in welchem Speicherblock mögliche Treffer zu finden sind.

Die Auswahl von Datenbankensystemen soll basierend auf verfügbare Indizes eingeschränkt werden, da ansonsten keine natürlichen Methoden zur Verbesserung der Performance existieren. Da diese Textformate als spezielle Datentypen abgelegt

werden, sollten auch spezielle Indizes verfügbar sein.

Für die Testanwendung wurde JSON verwendet. Diese Entscheidung basiert auf den selben Faktoren, die zur Priorisierung von JSON über XML im Bereich der Webentwicklung geführt haben. Dazu gehören in diesem Gebiet die niedrigere Verbosität von JSON, was im selben Zug zu niedrigerem Speicherverbrauch führt, und die höhere Serialisierungsgeschwindigkeit [37].

Zusätzlich ist zu erwähnen, dass dieser Ansatz die erste Normalform missachtet und dadurch auch keine höheren Normalformen erfüllen kann. Dies wird in Kauf genommen, ähnlich wie die vierte und fünfte Normalform auch oft ignoriert werden. Daher muss mit fehlender Redundanz der Datensätze und den daraus resultierenden Anomalien gerechnet werden. Daher sollte dieser Ansatz mit spezieller Logik erweitert werden, die diese Anomalien verhindern soll.

2.4.4 Weitere Lösungsansätze

Eine weiterer Lösungsweg wäre der sogenannte “h-store” von PostgreSQL. h-store beschreibt einen Datentyp, in dem mehrere Key-Value Paare gespeichert werden können. Um diesen Ansatz jedoch komplett umzusetzen müsste zusätzlich ein Weg gefunden werden, den Attributtyp eines Paares abzuspeichern und zu referenzieren. Ein weiteres Problem wären Attribute, die mehrere Werte haben. h-store kann nur einfache Datentypen und keine Listen ablegen. Zu Beginn der Arbeit wurden beide Features als wichtig eingestuft und haben auch das Design des Testprogramms und das JSON Format der serialisierten Objekte beeinflusst. Herausforderungen bei der Umsetzung haben diese Features jedoch in den Hintergrund geschoben. Beide sind in dem Design noch sichtbar, besitzen jedoch keine praktischen Features. Hierzu wird genauer in Kapitel 4.1.3 Testaufbau eingegangen.

3 Moderne Datenbanken

Seit dem Aufstieg des Internets müssen sich Datenbankentwickler neuen Herausforderungen stellen. Global verfügbare Anwendungen und Dienstleistungen, die von Millionen von Benutzern gleichzeitig verwendet werden, haben andere Anforderungen als die gewohnten tabellarischen Systeme. Dazu gehören größere Datenmengen, komplizierte Beziehungen zwischen Entities, Ausfallresistenz oder dynamische Datenstrukturen. Um diese Herausforderungen zu lösen wurden Abstriche in den gewohnten Stärken von DBMS, etwa der Konsistenz der Daten oder das fest definierte relationale Modell, in Kauf genommen. Diese neuen Produkte werden oft unter dem Begriff Not Only SQL (NOSQL)-Datenbanken gelistet. Aber auch Datenbanken mit relationalen Modell wurden weiterentwickelt und es entstanden neue Produkte mit neuen Stärken.

3.1 NOSQL

Ohne Überblick über die verschiedenen Merkmale von NOSQL-Datenbanken fällt es schwer, die Aussage zu treffen, ob eine diese Datenbanken zu dem jeweils vorliegendem Anwendungsfall passt. Daher werden in diesem Abschnitt vier Merkmale dieser Datenbanken, basierend auf der Arbeit von Davoudian et al. [20] vorgestellt, um anschließend zu entscheiden, ob eine dieser Lösungen für unser Problem passt.

3.1.1 Datenmodell

Der prägnanteste Unterschied, der bereits aus der Bezeichnung NOSQL ersichtlich ist, sind die verschiedenen verfügbaren Datenmodelle. Sie ermöglichen es Daten,

die nicht in das tabellarische Format passen, abzubilden. Aktuell umfasst der Begriff NOSQL im Groben vier verschiedene bekannte Modelle. Es gibt jedoch auch bereits Produkte, die verschiedene Modelle vereinen oder die den hier genannten nur ähneln.

Key-Value Stores Key-Value Stores sind die unrestriktivsten Datenbanken bezüglich ihres Inhalts. Sie legen die Dateien in einem Schlüssel - Wert Format ab. Der Schlüssel, meist ein Merkmal zur Identifikation, ist das einzige Medium zur Interaktion mit der Datenbank und steht für einen Eintrag in der Datenbank. Der Value, der unter einem Key abgelegt wird, kann ein beliebiges Format annehmen. Die Inhalte der Datenbank lassen sich per Schlüssel abfragen, Querys nach bestimmten Werten sind nicht oder nur dank spezieller Features einzelner Datenbanken möglich. Daher finden Key-Value Stores oft in Fällen Einsatz, in denen schnell komplexe Daten abgelegt und abgerufen werden müssen, etwa zur Verwaltung von User-Sessions. Sie lassen sich außerdem weiter unterscheiden nach der Art des Speichermediums. In-Memory Key-Value Stores eignen sich für Caching und ähnliche Anwendungsfälle, in denen die Daten flüchtig sind. Nicht-flüchtige Daten können in persistenten oder hybriden, die sowohl im Arbeitsspeicher als auch auf einer Festplatte agieren, Key-Value Stores abgelegt werden. Bekannte und beliebte Systeme sind Redis, DynamoDB und Azure Cosmos DB.

Dokumentendatenbanken Dokumentendatenbanken verfolgen denselben Ansatz wie Key-Value Stores, ermöglichen es aber, nur Teile von Werten auszulesen und auch nach bestimmten Werten in Querys zu suchen. Dies geschieht, indem die Werte festgesetzten Formaten folgen. Standardmäßig werden hierfür oft die Formate XML oder JSON verwendet, da diese durch das Web-Umfeld bereits viel Einsatz finden. Relationen zwischen einzelnen Einträge in der Datenbank können mit Fremdschlüsseln abgebildet werden oder direkt ineinander abgelegt werden.

Wide-Column-Store Wide-Column-Stores, auch Column-Family-Stores genannt, ähneln von allen NOSQL Datenmodellen am meisten den der relationalen Daten-

banken. Daten werden als Kombination von Reihen und Spalten-Familien, die einer Menge von Spalten entspricht, die in der Regel zusammen aufgerufen werden, dargestellt. Diese Spalten-Familien bilden auch die physische Speichereinheiten.

Beliebte Vertreter dieser Datenbankenart fokussieren sich auf extreme Datenmengen und die Aufteilung dieser Daten auf verschiedene Knoten in einem Cluster. Sowohl Cassandra als auch HBase empfehlen eine Hardwarekonfiguration mit mindestens 3 Knoten.

Graph-Database Graph-Datenbanken legen den Fokus des Daten-Modells auf die Verhältnisse zwischen Entities. Das Ablaufen von Verhältnissen, was in einer SQL Datenbank mit vielen Performance intensiven join-Operationen verbunden wäre, ist eine besondere Stärke dieser Datenbanken. Sie sind in ihrem Datenmodel sehr von der Graphentheorie inspiriert. Dank spezieller Algorithmen können Netzwerkdaten effizient gespeichert werden, wobei es unterschiedliche Verfahren gibt, die jeweils für verschiedene Einsatzzwecke optimiert sind.

Alle NOSQL-Datenmodelle haben ähnliche Anforderungen gegenüber der Modellierung der Informationen. In relationalen Modellen orientiert sich ein Datenbankarchitekt an den Vorgaben der Normalform, um Duplikate in der Datenbank zu vermeiden. Beim Design von NOSQL Schemas liegen die Use-Cases im Vordergrund. Daten werden so abgelegt, das sie performant wieder aufgerufen werden können. Durch die limitierten Abfragewege in NOSQL Datenbanken müssen alle gewünschten Querys abgedeckt werden. Daher ist es manchmal hilfreich, Daten mehrfach in verschiedenen Strukturen abzulegen. Die Schemafreiheit von manchen NOSQL-Datenbanken bedeutet nicht komplette Freiheit von Datenstrukturen. Damit Daten automatisch von Programmen verarbeitet werden können, muss eine Struktur definiert sein. Dieser Schritt wird in relationalen Datenbanken per SQL erledigt. Datenbankadministratoren beschreiben das Datenbank Schema mithilfe von Datenbank Werkzeugen. NOSQL Datenbanken jedoch verschieben die Verantwortung von der Datenbankebene auf die Anwendungsebene. Die Struktur der Daten wird entweder als richtig impliziert oder muss mithilfe einer bestimmten

DDL identifiziert und validiert werden. Dadurch wird der Anwendungscode komplexer und unflexibel, eine Eigenschaft, deren Lösung eigentlich DBMS sein sollten. Der Vorteil dieses Vorgehens ist jedoch, dass das Ablegen der Daten, sollten sie beim Abspeichern nicht auf ein Schema getestet werden, schneller erfolgen kann. Der Moment der Validierung wird vom Speicher- zum Lesevorgang verschoben.

3.1.2 Datenkonsistenz Modell

Neben dem Datenmodell ist ein weiterer wichtiger Punkt für die Auswahl einer Datenbank die Konsistenz der Daten. Dies ist besonders bei Datenbanken der Fall, die über mehrere Knoten aufgeteilt sind, ein Feature vieler NOSQL Datenbanken. Bei gleichzeitigen Lese- und Schreibvorgängen auf verschiedenen Knoten, die Duplikate der selben Datensätze speichern, muss das Verhalten der Datenbank voraussagbar sein. Viotti und Vukolic [51] haben über 50 verschiedene Konsistenz-Modelle untersucht und in Relation zueinander gesetzt. Gröber betrachtet lassen sich diese in verschiedene Kategorien eingliedern. Der strikteste Konsistenz ist starke Konsistenz, auch als Liniearisierbarkeit bekannt, in der die Reihenfolge der Operationen strikt und fest ist. Am anderen Ende des Spektrums liegt Eventual Consistency, ein Model das den Fokus auf Performance und Verfügbarkeit setzt. Dass Dateneinträge eventuell veraltet sein können, wird ignoriert und man erwartet nur, dass irgendwann am Schluss alle Einträge konsistent sind.

3.1.3 Partitionierung

Partitionierung beschreibt die Möglichkeit, Daten auf mehrere Knoten aufzuteilen. Dies geschieht meist auf horizontaler Ebene, d.h. Aufteilung der Daten auf mehrere Maschinen. Dies ermöglicht schnelleren Zugriff basierend auf geographischen Faktoren, als auch parallelen Zugriff auf unterschiedliche Datensätze. Dieser Vorgang, auch als Sharding bekannt, ist mit den meisten relationalen Datenbanken nicht möglich. Man kann Datenbanken basierend auf verschiedenen Partitionierungsfokusse unterscheiden, etwa zur Verteilung der Daten um die Auslastung von Maschinen auszugleichen oder um eine Mindestanzahl an Backups versichern zu

können.

3.1.4 CAP Theorem

Das CAP Theorem beschreibt das Verhältnis von Konsistenz, Verfügbarkeit und Aufteilungstoleranz.

- Consistency (C) - Die Konsistenz der Daten, vgl. 3.1.2
- Availability (A) - Die Verfügbarkeit und Reaktionsgeschwindigkeit des Systems, um sowohl Lese- als auch Schreibanweisungen durchzuführen
- Partition Tolerance (P) - Die Toleranz, die das System gegenüber Teilausfällen, z.B. einzelner Nodes, zeigt.

Das Theorem besagt:

> „You can have at most two of these properties for any shared-data system“ [14, Folie 14]

So lassen sich verteilte Datensysteme als CA, CP oder AP einstufen. Traditionelle relationale Single Node Datenbanken zum Beispiel fallen unter CA, d.h. nur konsistent und verfügbar, da ein Ausfall eines Knotens das gesamte System stilllegt. Genau betrachtet wäre das jedoch keine Spaltung sondern ein Komplettausfall. Systeme, die eine gewisse Toleranz gegenüber Spaltung erreichen wollen, müssen Abstriche in einem der beiden anderen Themen in Kauf nehmen. Dies basiert auf der Verteiltheit der Daten. Um die aktuellste Version eines Datensatzes zu bekommen, müssen alle zuständigen Nodes abgefragt werden. Dies erhöht jedoch die Laufzeit von Anfragen. Ein Administrator muss also basierend auf dem Anwendungsfall die Wahl zwischen Datenbanksystemen bzw. Datenbank-Konfigurationen treffen.

Das Theorem, die wissenschaftliche Untersuchung des Theorems und der Umgang damit wird jedoch auch kritisiert. Viele Datenbanken erfüllen nicht die strengen Anforderungen des Beweises des Theorems, identifizieren sich jedoch trotzdem

als eine der Variationen. Es kann auch sein, dass Datenbanken diesen Anforderungen nur entsprechen, falls sie in einem bestimmten Modus laufen. [30]

Es lohnt sich dennoch es zu betrachten, da es hilft, grundlegende Unterschiede von verteilten Datenbanksystemen zu erkennen.

Die beschriebenen Eigenschaften von NOSQL Datenbanken geben Einblick über ihre Fokusse. Diese liegen hauptsächlich auf verteilten Datenbanken, mit eingeschränkter Konsistenz der Daten und maximaler Anzahl von Querys. Sie eignen sich für extreme Datenmengen und Daten mit speziellen Format. Diese Daten können meist nur in einer bestimmten Form abgerufen werden, da nur in bestimmten Datenmodellen Ad-Hoc Querys unterstützt werden. Um das zu umgehen, werden oft die selben Daten mehrfach abgespeichert, was den Grundgedanken der Normalisierung des Modells in traditionellen Datenbanken ignoriert. Weitere ignorierte Faktoren, die durch die Spezialisierung auf besondere Aufgabenstellungen entsteht und Administratoren von traditionellen Datenbanken am Herzen liegen, sind u.A. der Einhaltung des Atomicity Consistency Isolation Durability (ACID) Modells und der Verlust von bekannten und verinnerlichten Tools. Das macht es oft schwer, NOSQL Datenbanken neu in bereits etablierten Umgebungen einzuführen, insbesondere da die Änderung von Maxime stets mit Skepsis verbunden ist.

Im Idealfall werden NOSQL Datenbanken parallel mit relationalen Datenbanken verwendet, wobei auch dieses Vorgehen durch Konsistenz und Ausfalls-Verhalten eingeschränkt und erschwert wird. Und mit Steigerung der Komplexität der gesamten Architektur von Anwendungen steigt auch die Möglichkeit von Fehlern. Da Online Transaction Processing (OLTP) Anwendungen, besonders wenn Rechnungs- oder Bankdaten bearbeitet werden, meist auf relationalen Datenbanken und ihren Stärken basieren, konzentriert sich diese Arbeit in der praktischen Analyse auf Methoden zur Implementierung des dynamischen Datenmodells in relationalen Datenbanken. Mit genügend Planung und wenn der Verwendungszweck nicht perfekte Konsistenz erlaubt, würden sich insbesondere Wide-Column- und Key-Value-Stores auch für die Aufgabenstellung qualifizieren.

3.2 NewSQL

Ein weiterer Begriff, der sich in den 2010er Jahren etablierte, ist NewSQL. Unter diesem Schlagwort werden alle Datenbanken eingegliedert, die Lehren aus den neuen Technologien der NOSQL Datenbanken ziehen, selbst aber wie relationale Datenbanken agieren. Da NOSQL für Not Only SQL steht, ist NewSQL eine Unterkategorie von NOSQL. Stonebraker [49] identifiziert zwei Anwendungsgebiete von New SQL. Zum einen Anwendungen, die extreme Datenmengen verarbeiten müssen, aber nicht auf Features wie SQL als Abfragesprache und die ACID Eigenschaften von Transaktionen verzichten wollen. Dazu gehören typische BigData Szenarien wie z.B. Webanwendungen mit extrem vielen Benutzern und Transaktionen pro Minute.

Der andere Fall sind Anwendungen, die sowohl analytische als auch transaktionale Use-Cases unterstützen sollen. Systeme dieser Art, die OLTP und Online Analytical Processing (OLAP) verbinden, werden auch Hybrid Transactional and Analytics Processing (HTAP) genannt. Sie ermöglichen die Echtzeitanalyse von Geschäftsdaten. Diese Anforderung lässt sich auf verschiedenen Wegen implementieren. Giceva und Sadoghi [22] identifizierten zwei Kernpunkte, in denen sich HTAP Systeme unterscheiden können. Als erstes bei Datenrepräsentation, d.h. der Einsatz von entweder Row- oder Columnstores oder eine Kombination der beiden. Und zum anderen bei Daten Replikation, insbesondere ob nur eine oder mehrere Kopien der Daten im physikalischen Speicher vorliegt.

Diese beiden Szenarien treffen jedoch nicht genau unsere Problemstellung. Trotzdem gäbe es interessante Kandidaten. Insbesondere In-Memory Stores, d.h. Datenbanken, die einen Teil der Daten in flüchtigen und dadurch in besonders schnellen Speicher hält, könnten den der benötigten Unterschied für eine akzeptable Performance ausmachen. Diese Datenbanken haben dadurch jedoch auch größere Anforderungen an die benötigte Hardware, besonders bei der Menge an Arbeitsspeicher. Da die verfügbare Testmaschine nicht diese Anforderungen erfüllen konnte, wurden In-Memory Datenbanken nicht weiter in Erwägung gezogen.

4 Performance Analyse relationaler Lösungswege

Um die Performance der im zweiten Kapitel beschriebenen Lösungswege zu testen, wurde ein Programm geschrieben. Zunächst wird beschreiben, auf welche Parameter die verschiedenen Lösungswege untersucht wurden. Anschließend betrachten wir den Testaufbau, d.h. die verwendeten Technologien und die Struktur des Testprogramms. Dabei wird auf mehrere Probleme bei der Umsetzung und auf Lösungen für oder Einschränkungen durch diese eingegangen. Am Ende des Kapitels werden die Laufzeiten verglichen und Aussagen zu den Ergebnissen getroffen.

4.1 Test-Dimensionen

Um die Effizienz unserer Lösungswege zu bestimmen, müssen mehrere Aspekte untersucht werden. Da jeder dieser Aspekte die Anzahl an Testergebnissen multiplikativ erhöht, werden sie Test-Dimensionen genannt.

Die erste Dimension ist die Anzahl der Elemente in der Datenbank. Durch die Untersuchung mit unterschiedlicher Anzahl von Elementen in der Datenbank lassen sich Aussagen über die Skalierbarkeit der Lösungsansätze treffen. Für die konkreten Tests wurde beschlossen, die Fälle mit jeweils 100.000 und 1.000.000 Elementen in der Datenbank zu untersuchen.

4.1.1 Wahl der Datenbank

Um eine starke Lösung zu finden müssen nicht nur unterschiedliche Ansätze, sondern auch die jeweiligen Datenbanken, in denen die Ansätze durchgeführt werden, gezielt ausgewählt werden. Um Vergleichbarkeit zu schaffen sollten die Datenbanken sowohl für den Einsatz von EAV als auch der JSON Methode geeignet sein. Hier ist eher die Unterstützung von JSON Elementen interessant, da der EAV Ansatz keine besonderen Voraussetzungen besitzt. Die Auswahl kann via dem Kriterium, ob eine Datenbank Indizies für den JSON Datentyp bereitstellt, eingegrenzt werden.

Nach Betrachtung der beschriebenen Anforderungen wurden zwei Datenbanken gewählt.

Die erste Wahl fiel auf PostgreSQL. PostgreSQL ist eine freie Open-Source-Datenbank. Die verwendete Version war die zum Zeitpunkt des Beginns der Entwicklung aktuellste Version, 13.1 (Debian 13.1-1.pgdg100+1). PostgreSQL qualifizierte sich durch extra Datentypen für JSON Elemente und eines speziellen Index für diese Datentypen. Zusätzlich erweitert PostgreSQL die SQL Syntax mit speziellen Befehlen, die für die Verarbeitung von JSON Elementen erstellt wurden.

Um JSON Elemte in der Datenbank abzulegen, existieren zwei Datentypen: `json` und `jsonb`. `json` legt den übergebenen Wert direkt in der Datenbank ab, während `jsonb` den Wert in ein Binärformat umwandelt. Das erleichtert den Umgang mit den gespeicherten Daten, vor allem da dadurch der Einsatz von Indizes ermöglicht wird. Daher wurde in der Testanwendung auch der Typ `jsonb` verwendet.

Für die beiden JSON-Querys wurde folgender Befehl verwendet:

Listing 4.1: PostgreSQL Querys

```
SELECT * FROM pages WHERE attributes::jsonb @> ?::jsonb
```

Das Fragezeichen steht als Platzhalter für einen Parameter, der unseren Suchtext beinhaltet. Der Operator `@>` ist ein Containment Operator. Er überprüft, ob das erste Element das Zweite beinhaltet. Das Element, das für die Suche in die Query eingefügt wird, wurde per Hilfsmethode aus den gesuchten Parametern erstellt. [43]

Die zweite Datenbank, die untersucht wird, ist Oracle Database. Im speziellen Oracle Database Express Edition (Oracle Database XE) 18c. Die Lizensierung der Oracle Database erschwert es, das neuste Release, zur Zeit der Veröffentlichung dieser Arbeit 21c, der Datenbank zu testen. Oracle stellt regelmäßig bestimmte Versionen, deren Lizenz nicht käuflich erworben werden muss, frei zur Verfügung. Diese Releases werden Express Edition genannt.

Eine Einschränkung dieser Version ist eine geringe maximale Speichergröße von 12 GB. Diese Begrenzung mach es unmöglich, den Test mit 1,000,000 Einträgen durchzuführen, sowohl mit dem EAV als auch mit dem JSON Ansatz. Daher konnte in der Oracle Umgebung nur die kleineren Tests mit weniger Testdaten durchgeführt werden. [9]

Oracle Database bietet einen Check für die JSON-Konformität für Spalten und spezielle SQL Syntax für die Abfrage von JSON Elementen. Zusätzlich können Indizes basierend auf bestimmten Funktionen oder für Volltextsuche für JSON optimiert angelegt werden. [39]

Um eine Spalte als JSON-Spalte zu verwenden, muss ein Constraint verwendet werden. Der Befehl hierzu lautet:

Listing 4.2: Oracle Database JSON Constraint

```
CONSTRAINT "ENSURE_JSON_ATTRIBUTE" CHECK (attributes IS JSON)
```

Der Typ der Spalte attributes, der verwendet wurde, ist clob.

Die Syntax für JSON Querys ist anders als bei PostgreSQL. Anstelle eines neuen Operators werden komplette Funktionen verwendet. Operatoren sind jedoch nur syntaktischer Zucker und sollte nicht als wichtiges Argument für oder gegen eine Lösung stehen. Schwerwiegender ist jedoch, dass die Funktionen den Einsatz von Prepared Statements erschweren (vgl. 4.2.1 JDBC). Die Befehle zur Suche von Elementen mit einem bestimmten Attribut und einem bestimmten Attributwert haben den Aufbau

Listing 4.3: Oracle Database, FindByAttribute-Query

```
SELECT * FROM pages WHERE json_exists(attributes, '$[*]?(@.name ==
    ↪ "attributeName")')
```

respektive

Listing 4.4: Oracle Database, FindByValue-Query

```
SELECT * FROM pages WHERE json_exists(attributes, '$[*]?(@.name ==
    ↪ "attributeName" && @.values[*].value.Type == "value.Value")'
```

Der Suchtext liest sich in etwa: Existiert ein Element in einem Array (`$[*]`), welches einen Schlüssel mit dem Namen "name" und dem gesuchten Attributnamen als Wert enthält (`?(@.name == "attributeName")`); und zusätzlich bei der Suche nach einem bestimmten Attributwert ein Array mit der gewünschten Typ-Wert als Schlüssel-Wert Kombination enthält (`&& @.values[*].valueType == ↪ "valueValue"`).

Die Indizes der beiden Datenbanken werden in Abschnitt 4.1.2 näher beschrieben.

Zu den ausgeschlossenen Datenbanken zählt u.a. MySQL, welches zwar einen JSON Datentyp und Funktionen für Querys bereitstellt [5], aber keine Möglichkeit bietet, JSON Felder direkt zu indizieren. Stattdessen wird empfohlen, Indizes auf generierten Spalten zu erstellen [6]. Dies bezieht sich aber auf feste Schlüssel und keine Array-Werte.

4.1.2 Anwendungsfälle

Die nächste Dimension der Testfälle sind die untersuchten Anwendungsfälle. Diese basieren auf den grundlegenden Operationen in der Persistenzschicht von Anwendungen: Create, Read, Update und Delete, oft unter dem Akronym CRUD zusammengefasst. Daraus lassen sich folgende Anwendungsfälle ableiten:

- Create: Erstellen eines neuen Elements ohne Attribute
- Read: Abfragen nach Elementen mit bestimmten Attributen und nach Elementen mit bestimmten Attributwerten[1]

[1] Für die Simulation einer Webanwendung und zur Minimierung der Testlaufzeiten wurde entschieden, dass in Queries eine maximale Fetch-Größe von 100 Einträgen gesetzt ist. Dieses Vorgehen entspricht einer Seiten-basierter Ansicht, bei der weitere Ergebnisse erst nach Aufruf geladen werden. In den Tests wurde jeweils nur die ersten 100 Ergebnisse von der Datenbank

- Update: Änderung eines einzelnen Attributwert
- Delete: Löschen eines kompletten Elements inklusive aller Attributwerte

Zusätzlich wurde ein Testfall mit einem äußert großem Attributwert untersucht. Für Testzwecke wurde der extreme Fall, einen 2 Megabyte(MB) großen String als ein Attribut zu speichern und anschließend weiteren Text anzuhängen, gewählt. Gemessen wurde die benötigte Zeit, um eine Änderung an diesem Text durchzuführen. Dieser Testfall war durch bekannte Problemfälle in existierenden Anwendungen motiviert, führte jedoch zu einer Einschränkung der Testergebnisse. Durch die Notwendigkeit, ein sehr großes Element speichern zu können, mussten teilweise die Datentypen angepasst werden. Dies hatte auch zur Folge, das in der Oracle Umgebung mit dem Ansatz EAV keine Tests mit Indizes durchgeführt werden konnten. Das gleiche gilt für den EAV-Ansatz in PostgreSQL, der mit Indizes nur einen Teil der Anwendungsfälle durchführen konnte. Die restlichen Fälle konnten hier untersucht werden, indem der Fall mit dem großen Attributwert übersprungen wurde (vgl. Absatz 4.1.2).

Die zugehörigen Bezeichnungen für diese Anwendungsfälle in der Auswertung lauten:

- Create: “CreatePage”
- Read: “FindByAttribute”: Suche nach Elementen, die ein Attribut mit diesem Namen haben
  “FindByValue”: Suche nach Elementen, die ein Attribut mit einem bestimmten Namen und Wert haben
- Update: “UpdatePage”
- Delete: “DeletePage”
- Änderung eines großen Attributs: “ChangeBigAttribute”

aufgerufen. Hier war zu beachten, das die Fetch-Größe explizit gesetzt werden muss und bei PostgreSQL die Verbindung nicht auf Auto-Commit stehen darf. Außerdem kann bei besonders vielen Treffern ohne eine Fetch-Size eine Query bei einer PostgreSQL Datenbank zu einem OutOfMemory Fehler führen.

4.1.3 Indizierung

Wie bereits beschrieben bieten verschiedene Datenbanken unterschiedliche Indizes an. Um die Wirksamkeit dieser Indizes zu überprüfen wurden die Messungen jeweils einmal mit und einmal ohne spezielle Indizes durchgeführt. Ohne Index heißt in diesem Fall nur mit einem Index auf den jeweiligen Primärschlüssel, der immer und automatisch existiert. Das ist die vierte und letzte Dimension unserer Tests.

Oracle Für das Indizieren von JSON Daten unterscheidet die Oracle Dokumentation zwischen zwei Fällen: Indizierung von festen Attributen innerhalb fester JSON Strukturen, ein Format das dem Standardfall ähnlich den von normalen Spalten entspricht, und einem JSON search index. Da in Rahmen dieser Arbeit beliebig neue Attribute in der JSON Struktur abgelegt werden müssen, ist der JSON search index die erste Wahl. Oracle empfiehlt diesen Index für ad-hoc Queries und full-text search.

Um den Ersten, den Value Index, zu verwenden muss zum Zeitpunkt der Erstellung des Index bereits das zu suchende Attribut bekannt sein. Daher müssten wir, um diesen Index zu verwenden, für jedes neue Attribut zur Laufzeit einen neuen Index anlegen. Das kann je nach Art der verwalteten Daten zu großen Perfomance-Problemen führen, da die Anzahl an Indizes zusammen mit der Zahl der Attributen im System steigt. Dadurch könnte die Leistung bei Create und Update Operationen leiden, falls die Indizes sofort die neuen Einträge enthalten müssen. Eine Möglichkeit, diese Gefahr zu umgehen, wäre ein regelmäßiger Job, der die Indizes zu Zeiten aktualisiert, bei denen die Grundlast sehr niedrig ist, etwa während der Nacht.

Der Search Index dagegen kann für beliebige Werte verwendet werden. Er kann genauso wie der Index für feste Werte asynchron verwendet werden. [39]

Der EAV Ansatz würde hingegen keine speziellen Indizes benötigen, sondern nur Indizes die weitere Spalten beinhalten. Leider konnten für diesen Ansatz keine Tests mit Indizes durchgeführt werden. Der Testfall “ChangeBigAttribute” schränkt den verwendeten Datentyp, der mindestens etwas mehr als 2MB speichern können muss, für Attributwerte stark ein. Um einen möglichst großen Attributwert

abspeichern zu können wurde der Datentyp clob verwendet. Dieser kann sehr große Datenmengen enthalten, ist jedoch für die Verwendung von den benötigten Indizes nicht geeignet. Da nicht wie bei PostgreSQL ein Fehler erst bei der Durchführung dieses Anwendungsfalls, sondern bereits bei der Erstellung des Index auftrat, wurden auch die restlichen Anwendungsfälle nicht durchgeführt.

PostgresSQL Auch PostgreSQL bieten mehrere Indizes für JSON bzw. Volltextsuche an. Empfohlen wird der sogenannte Generalized Inverted Index (GIN) [43]. Hier handelt es sich um einen Index, der speziell für Werte verwendet werden kann, die aus Teilwerten bestehen. Dazu gehört der verwendete Typ jsonb, aber auch Arrays vieler verschiedener Typen und Typen für Textsuche. Ein GIN legt einen Eintrag in dem Index für jedes Teilelement der zu indizierenden Daten an. Die Grundstruktur ähnelt der eines B-tree Index. Auf Blatt Ebene speichert ein GIN anstelle der Spaltenwerte der Reihe, die indiziert wird, eine Liste oder einen weiteren binär Baum mit Pointern auf Reihen, die diesen Teilwert besitzen. Dabei ist jede Wert-Pointer Kombination einzigartig. Die Wahl, ob eine Liste oder eine Binärbaum von tids, die der Datentyp für solche Pointer ist,verwendet wird, hängt von der Anzahl an Reihen mit diesen Teilwerten ab. Diese heißen "posting list" respektive "posting tree". Da das Einfügen neuer Daten in diese Struktur sehr aufwendig ist, existiert eine weitere Liste, genannt "pending list", in der neue Einträge zwischengespeichert werden. Erst wenn diese Liste vollläuft werden die Änderungen in den Index verschmolzen. Davor werden bei der Verwendung des Index sowohl der Baum als auch die "pending list" durchsucht. [25]

4.2 Testaufbau

Um die Performance aller Tests aufzunehmen und vergleichbar zu machen wurde eine Java Anwendung programmiert. Mithilfe einer kompletten Anwendung lassen sich alle Verarbeitungsschritte zusammen messen und komplexe Versuche einfacher aufgestellt werden.

4.2.1 Programmiertechnische Umsetzung

Auf dem "Popularity of Programming Language index" liegt Java zum Zeitpunkt dieser Arbeit auf Platz zwei [1]. Außerdem existieren im Java-Umfeld bereits mehrere Frameworks, die ähnliche Problemstellungen lösen, i.e. Hibernate. Zusätzlich kann Dank JDBC Treibern leicht Datenbankverbindungen hergestellt werden. Daher wurde Java als Programmiersprache für die Testanwendung verwendet.

Das gesamte Programm kann zur genauen Untersuchung unter `https://github.com/matthias-epos/dynamic-datamodels` gefunden werden.

Java Database Connectivity (JDBC) JDBC und ähnliche Schnittstellen erlauben es, Code allgemein für verschiedene Datenbanken zu schreiben. Dank spezieller Klassen und Interfaces kann die Kopplung an bestimmten Datenbanken minimiert werden. Voraussetzung dazu ist, dass der Datenbankanbieter eine Implementation des JDBC Treibers zur Verfügung stellt. Der Treiber bildet die Brücke zwischen einer Java-Anwendung und eines Datenbankmanagementsystems. Die Interaktion mit dem Treiber wird über die JDBC API gesteuert. Sie liefert Klassen für alle Aspekte, die für die Interaktion benötigt werden. Dazu gehören unter Anderem `Datasources`, die für Datenbankinstanzen stehen und `Connections`, mit denen SQL Statements erstellt und ausgeführt werden können [45]. Ein SQL Befehl kann mit passender Konfiguration so ausgeführt werden:

Listing 4.5: Prepared Statement mit JDBC

```
// conn ist ein vorbereitetes Connection Objekt
stLoadPage = conn.prepareStatement("SELECT * FROM pages WHERE id =
    ↪ ?");
stLoadPage.setLong(1, pageId);

rsLoadPage = stLoadPage.executeQuery();
```

Hier handelt es sich um ein Prepared Statment. Diese Struktur erlaubt es, Parameter auf eine Art in eine Query einzufügen, die SQL-Injection Angriffe unmöglich macht. Angriffe dieser Art entstehen, wenn ein SQL-Befehl mithilfe eines Strings, der zur Laufzeit zusammengebaut wird, erstellt wird. Sie funktionieren, indem der

beabsichtigte Befehl mithilfe eines ; Zeichens verfrüht abgeschlossen wird und ein anderer Befehl, der z.B. beliebige Informationen auslesen könnte, mit angehängt wird.

Innerhalb des Texts, mit dem das Prepared Statement erstellt wird, markieren Fragezeichen Stellen, an denen Parameter eingefügt werden müssen. Dies führt bei den JSON Implementationen der Datenbanken zu Problemen. PostgreSQL benutzt das Fragezeichen als Operator. Um den Operator in einem JDBC Prepared Statement zu verwenden, kann das Fragezeichen mit einem zweitem Fragezeichen escaped werden (??). Oracle Database hat ein ähnliches Problem mit den JSON-Funktionen, die für die Querys benötigt werden. Hier gibt es jedoch keine Art, das Problem zu umgehen. Stattdessen muss der Text des SQL Statements dynamisch zusammengebaut werden, was die Anwendung verwundbar gegenüber SQL Injection macht, falls keine weitere Validation der Eingabewerte erfolgt [48].

Java Architektur Die Anwendung besteht aus mehreren Teilen. Modell-Klassen definieren den Grundaufbau unserer Entities und befinden sich im Package model.

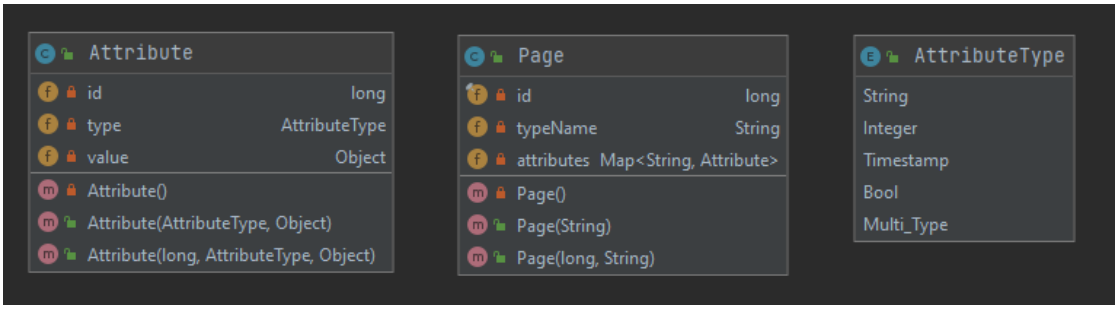

Abbildung 4.1: Klassenmodell des EAV Ansatzes

Die Klasse Page ist die Klasse, deren Objekte mit beliebigen Attributen zur Laufzeit von dem Benutzer erweitert werden kann. Diese Objekte werden in dem Feld attributes, eine <String, Attribute> Map hinterlegt. Ansonsten besitzt sie nur einen Typ-Namen und eine Id. Beim Erstellen eines neuen Page Objekts wird der Wert der Id auf -1 gesetzt. Die echte Id wird erst von der Datenbank generiert und anschließend auf das Objekt übertragen. Die Klasse Attribute enthält auch ein Feld für die Id. Diese Id wird nur bei dem EAV Ansatz verwendet und wird

im Json Ansatz ignoriert. Der Wert und der Typ des Attributs werden in weiteren Felder gespeichert. Die verfügbaren Attribut-Typen werden in dem Enum AttributeType verwaltet. Diese Typen würden in einem realen Programm Anwendung finden, die diese Typen für extra Logik verwenden könnte. Beide untersuchten Ansätze speichern die Attributwerte in einem Textfeld ab. Der gespeicherte Attributwert könnte dazu verwendet werden, um diese Textrepräsentation wieder in passende Objekte umzuwandeln, auch bekannt als Unmarshalling oder Deserialisierung. Für die Tests wurden nur inhärent serialisierbare Objekte verwendet. Für eine komplette Anwendung müssten die Attributwerte auf Serialisierbarkeit untersucht werden.

Objekte, die mit diesen Modell-Klassen erstellt wurden, werden mithilfe von Datenbank-Adaptern in der Datenbank abgelegt und wieder aufgerufen. Diese befinden sich im dem Package approachImpl.

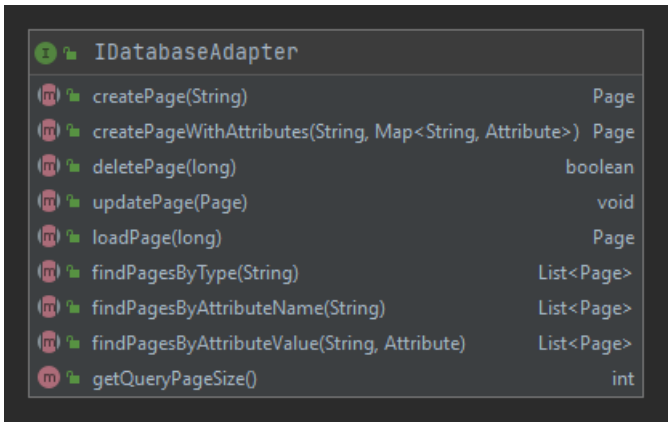

Abbildung 4.2: Outline Datenbankadapter Interface

Das Ziel war es, ein Interface für die Datenbankfunktionen zu definieren, welches jeden Use Case, der definiert wurde, erfüllen konnte. Daher enthält das Interface IDatabaseAdapter Methoden für die CRUD Operationen und verschiedene Query Methoden. Anschließend wurden Implementationen dieses Interfaces für jeden Ansatz programmiert. Hier ist zu beachten, das der EAV Ansatz für Oracle und PostgreSQL Datenbank fast identisch ist und wiederverwendet werden konnte. Die Klasse im Programm heißt EAV_DatabaseAdapter. Die Implementierung unterscheidet nur beim Auslesen von genierten Primärschlüsseln und der SQL Query

zum Suchen von Einträgen mit bestimmten Attributwerten zwischen den Datenbanken. Da sich die Syntax von Queries mit JSON Inhalten in Oracle und PostgreSQL unterscheiden, wurde für beide Datenbanken jeweils eine JSON Implementation geschrieben. Die zugehörigen Klassen heißen JSON_Oracle_DatabaseAdapter und JSON_Postgres_DatabaseAdapter.

Das Package connection enthält alle Klassen, die die Verbindung zu den Datenbanken verwalten. Auch hier gibt es eine Klasse, AbstractConnectionManager, die den Grundaufbau dieser Klassen enthält. Alle Tests wurden nur mit einer einzigen Connection zu der jeweiligen Datenbank durchgeführt, d.h. es finden keine parallelen Zugriffe statt. Oracle Database und PostgreSQL verwalten mehrere Datenbanken auf einer Instanz unterschiedlich. In Oracle Database lassen sich mehrere Datenbanken am einfachsten mit mehreren Usern umsetzen. PostgreSQL kann stattdessen mehrere Datenbanken mit nur einem Benutzer anlegen. Ein weiterer Unterschied ist der Connection-String bzw. die URL der Datenbank. Diese beiden Besonderheiten werden von den jeweiligen Implementationen beachtet. Es existiert zusätzlich ein Connection-Manager für eine H2 Datenbank. H2 Database Engine ist eine In-Memory Datenbank, die oft zum Testen von Software und deren Datenbankschnittstellen verwendet wird. Auch in diesem Projekt wurde H2 verwendet, um die Umsetzung des EAV Ansatzes, der komplexe Logik enthält, zu testen. Diese Tests haben den Vorteil, dass sie ohne eine existierende Datenbank durchgeführt werden können. Dies war besonders zum Anfang der Entwicklung, bevor fest beschlossen war, wie die Datenbanken verwaltet werden, sehr hilfreich. Da die JSON Ansätze immer spezielle Syntax für die jeweilige Datenbank besitzen lohnte es sich nicht, Unit-Tests mit H2 für sie zu schreiben. Stattdessen existieren zusätzlich Integrations-Tests für jede Ansatz-Datenbank Kombination. Die Performance Tests, mit denen die Messwerte generiert wurden, haben sich aus diesen Integration Tests heraus gebildet und besitzen eine ähnliche Umsetzung. Die Klasse, die für die Perfomancetests verwendet wurde, heißt CommonCaseSetup.

Serialisierung von Objekten Für die Verwendung von JSON Datentypen in Datenbanken müssen Objekte serialisiert werden. Dafür wurde das Framework *gson* verwendet [23]. Es ermöglicht, Plain Old Java Objects (POJO)s in JSON umzuwandeln. Die Komplexität des Einsatzes dieses Frameworks steigt mit der Komplexität der Objekte. Da die umgewandelten Objekte ein bestimmtes Format haben sollen, musste ein spezieller Type-Adapter registriert werden. Type-Adapter erlauben es, spezielle selbst konfigurierte Serialisierer und Deserialisierer bei bestimmten Objekten zu verwenden. Dies ist der verwendete Aufbau der JSON Objekte:

Listing 4.6: Attribute-JSON

```
[
    {
        "name": "firstAttribute",
        "values": [
            {
                "String": "firstValue"
            }
        ]
    },
    {
        "name": "secondAttribute",
        "values": [
            {
                "String": "Mustermann"
            }
        ]
    },
    {
        "name": "anotherAttribute",
        "values": [
            {
                "String": "Berlin"
            }
        ]
    }
]
```

Dieser Aufbau basiert auf mehreren Ansprüchen. Das Root-Element ist ein Array, das alle gewünschten Attribute enthalten kann. Die Objekte in dem Array haben zwei Schlüssel-Wert Paare. Das erste beschreibt den Namen des Attributs.

Das zweite hat als Wert ein Array aus einzelnen Schlüssel-Wert Paaren. Hier wurde ein Array gewählt, um in Zukunft das Speichern von mehreren Werten unter einem Attributnamen zu ermöglichen. In der Praxis wurde diese Fähigkeit nicht verwendet oder untersucht. Das gleiche gilt für den Attributtyp, der den Schlüssel in dem Wert-Objekt bildet. Dieser ermöglicht theoretisch, spezielle Logik basierend auf dem Typ des Attributs anzuwenden, etwa Tests auf Datenintegrität bei einem Typ wie Datum. Letztendlich existierten die Enums für verschiedene Typen, doch das Testprogramm machte keine weiteren Unterscheidungen basierend auf diesen. Der Wert für den Typ-Schlüssel ist schließlich der gespeicherte Wert für das Attribut.

4.2.2 Wiederverwendbarkeit und Reproduktion der Anwendung und Tests

Der Wunsch, die Testumgebung möglichst wiederverwendbar und einfach aufstellbar zu machen, zeigt sich in mehreren Aspekten der Testapplikation.

Zur Organisation der Abhängigkeiten und des Build-Prozesses wurde Maven verwendet. Maven ist neben Gradle und Apache Ant das am häufigsten verwendete Build-Tool in Java Umgebungen [50]. Maven automatisiert den Download und die Einrichtung aller benötigten Frameworks und Treibern. Die Testanwendung benötigte unter anderem Frameworks zur Durchführung von Unit-Tests, Logging von Ergebnissen, Serialisierung von Objekten und der Verwaltung von Docker Containern. Zusätzlich werden auch passende JDBC Treiber für unsere Datenbank verwaltet, d.h. heruntergeladen und registriert. Eine weitere Hilfe ist Maven für einen benutzerdefinierten Build-Process. In der Testanwendung werden die verwendeten Standard-Schritte validate, compile, test, package und verify mit einem pre-integration-test Schritt erweitert, der die benötigten Docker Images mithilfe bereitgestellter Skripts erstellt. Durch diese Anpassungen wurde die Ausführung des Programms an neuen Maschinen erleichtert.

Eine weitere Entscheidung, die durch diese Mentalität geprägt war, ist, alle Testdatenbanken innerhalb von Docker Containern zu betreiben. Docker ist ein

Tool zur Virtualisierung von Containern, die virtuellen Maschinen ähneln. Docker hat jedoch den Vorteil, das alle Container den Kernel teilen können und jeweils nur mit den nötigen Bibliotheken und Tools ausgestattet werden. Dadurch werden Speicherplatz und Performance eingespart. Sogenannte Dockerfiles ermöglichen es, Images für Container basierend auf existierenden Images zu erstellen. Diese Technologie wird verwendet, um die Datenbanken reproduzierbar zu konfigurieren. Sowohl PostgreSQL als auch Oracle Database stellen Basis-Images zur Verfügung, die beliebig erweitert werden können. Für die Performancetests wurden für jede Datenbankanwendung jeweils ein Dockerfile angelegt. Diese fallen sehr einfach aus. Beide Basis-Images stellen einen Ordner, `/docker-entrypoint-initdb.d/` bzw. `/docker-entrypoint-initdb.d/setup`, für Skripts bereit. Alle sql-Dateien, die sich in diesem Ordner befinden, werden zur Initialisierung des Containers, sobald die Datenbank bereit ist, automatisch ausgeführt. Die Test-Dockerfiles erweitern die Basis-Images der Datenbanken nur mit einem Befehl, der unsere Skripte, die die Befehle zur Erstellung der benötigten Benutzer und Tabellen enthalten, in diesen Ordner kopieren.

Da Oracle Database keine freie Datenbank ist, ist der Vorgang zur Erstellung des Basis-Images komplizierter als der des PostgreSQL-Images, welches automatisch von DockerHub bezogen werden kann. Stattdessen muss das Image per Hand mit einem Skript, verfügbar unter `https://github.com/oracle/docker-images/blob/main/OracleDatabase/SingleInstance/README.md`, gebaut werden. Bei diesem Schritt werden in der Regel die Binaries der Datenbank benötigt. Das ist bei der verwendeten Version (XE) nicht der Fall, da dies die einzige frei verfügbare ist. Daher befinden sich im Projektarchiv der Testanwendung auch alle benötigten Dateien, um die benötigte Version automatisch und ohne Download des oben verlinkten Archivs zu bauen. Da dies ein sehr zeitintensiver Vorgang ist, testet das Skript zunächst, ob auf der Maschine bereits das fertige Image existiert. Falls nicht, wird lokal das Image mithilfe dieser Dateien gebaut. Dieses Verfahren hat den Nachteil, das eventuelle Änderungen von Oracle an dem Image bzw. dem Dockerfile des Images nicht automatisch übernommen werden können. So können alle

benötigten Images automatisch während des Build-Prozesses von maven erstellt werden. Starten und Stoppen von Containern basierend auf diesen Images wurde mithilfe des Framework Testcontainers erledigt [10]. Dieses kann automatisch neue und existierende Container starten und stoppen und stellt die ansonsten dynamischen Ports mithilfe einer statischen Funktion für die Datenbank-Verbindung zur Verfügung.

Wie bereits in der Einleitung beschrieben ist das Testprogramms auf GitHub unter `https://github.com/matthias-epos/dynamic-datamodels` verfügbar. Das Projekt sollte sowohl unter Windows als auch Linux Umgebungen ausführbar sein. Nach dem Download des Repositories kann es mithilfe von mvn integration-tests gebaut und getestet werden. Hier muss beachtet werden, dass von jedem Datenbanksystem maximal eine Instanz laufen soll, da es ansonsten zu Fehlern kommen kann. Die Performance Tests wurden durch Aufruf der Klassen innerhalb einer IDE ausgeführt. Diese sind auch per Kommandozeilen Befehl erreichbar. Um Neu-Initialisiergungen der Datenbank und der Testdaten, die besonders bei dem Testfall mit 1.000.000 Einträgen sehr lange dauert, zu vermeiden, muss in dem Benutzer-Ordner ein Testcontainers-Properties-File mit Name `.testcontainers.properties` und der Zeile `testcontainers.reuse.enable=true` angelegt werden. Dadurch prüft das verwendete Framework zur Verwaltung der laufenden Container, ob bereits eine laufende Instanz existiert und löscht diese nicht mehr am Ende des Tests.

4.2.3 Daten

Für eine möglichst genaue Simulation einer echten Anwendung wären realistische Daten ein großer Vorteil. Es erspart den Notwendigkeit, einen Algorithmus zu schreiben, der passende Daten generiert. Außerdem hat die Art der gespeicherten Daten einen Einfluss auf die Performance der Datenbank. Ohne bekannte Metriken kann nur geschätzt werden, wie viele Einträge existieren, wie viele Attribute sie jeweils haben und wie groß die gespeicherten Attributwerte sind. Es ist jedoch selbst wenn echte Daten existieren meist nicht möglich sie für Forschungsarbeiten zu benutzten, wenn die gesamte Arbeit nicht unter Verschluss bleibt. Selbst

dann muss genau geprüft werden, ob Datenschutzbestimmungen eingehalten werden können. Sobald etwa persönliche Daten enthalten sind, müssen die Benutzer laut Datenschutz-Grundverordnung dem Zweck von Analyse zur Verbesserung zugestimmt haben (vgl. Art. 6 DSVGO). Ein Umweg, die Anonymisierung der Daten, ist auch mit viel Aufwand und Gefahr durch Unachtsamkeit verbunden.

In Anbetracht dieser Tatsachen wurde entscheiden, die Testdaten selbst zu generieren. Da die Daten mehrmals für verschiedene Datenbanken generiert werden mussten, werden die Daten basierend auf einem Seed erstellt. Durch die Wiederverwendung dieses Seeds werden stets die selben Daten erstellt und in den Datenbanken abgelegt. So sind die Versuche leichter nachzustellen und die Performance von verschiedenen Datenbanksystemen vergleichbar.

Der Datenerstellungsalgorithmus nimmt vier Parameter:

- NUMBER_OF_START_ENTITIES
  Die Anzahl der Objekte mit dynamischer Anzahl an Attributen, die in der Datenbank abgelegt werden. Verwendet: **100.000 und 1.000.000**

- NUMBER_OF_START_ATTRIBUTES
  Die Anzahl an Attributen, die für die generierten Daten verwendet werden. Es wird davon ausgegangen, dass bestimmte Attribute mehrfach verwendet werden. Daher existiert eine Menge von Attributen, von denen bei Erstellung neuer Objekte zufällig welche ausgewählt werden. Verwendet: **1000**

- MEAN_NUMBER_OF_ATTRIBUTES
  Die durchschnittliche Anzahl an Attributen, die ein Objekt haben wird. Die Verteilung der Anzahl ist normalverteilt. Verwendet: **100**

- MAX_NUMBER_OF_ATTRIBUTES
  Die maximale Anzahl an Attributen, die ein Objekt haben kann. Ein einfaches Limit, um Extremfälle zu vermeiden. Verwendet: **200**

Die erstellten Daten haben bloß einfache Werte. Da bei beiden Ansätzen die Werte auf Textwerte umgewandelt werden wurde entschieden, auch nur Textwerte zu erstellen. Die Werte sind alle komplett zufällige Zeichenketten der Länge 16.

Zusätzlich zu den komplett zufälligen Attributen wurden auch sogenannte Performance-Attribute erstellt. Diese werden zum Testen der Performance der Queries verwendet. Ein Performance-Attribute hat einen bestimmten Namen, eine Funktion zur Erstellung des Wertes und eine Wahrscheinlichkeit. Der Wahrscheinlichkeitswert beschreibt den Prozentsatz der existierenden Objekte, die dieses Attribut bekommen sollen. Die Werte-Funktion gibt in dem konkreten Testaufbau mit gleicher Wahrscheinlichkeit `true` oder `false` zurück. Dies ermöglicht den Anwendungsfall "FindByValue" zu untersuchen. Ein Performance-Attribut mit den Aufbau `("testAttribut", () -> String.valueOf(random.nextBoolean()), 50%))` zum Beispiel fügt der Hälfte aller Objekte ein neues Attribut mit dem Namen `"testAttribut"` hinzu. Davon hat eine Hälfte den Wert `true` und die Andere `false`. Diese Attribute wurden dann für die Performance Tests verwendet, da genau vorhergesagt werden konnte, wie viele Treffer es jeweils ungefähr geben muss.

4.2.4 Hardware und Laufumgebung

Die Tests wurden auf einer Windows-Maschine mit der Version Windows 10 20H2, Build 19042.928 durchgeführt. Es besitzt einen Intel Core i5-8250U CPU mit 1.60GHz und stellt 16GB Arbeitsspeicher zur Verfügung. Die Festplatte ist eine SK hynix SC311 SSD mit eine mit `winsat` gemessenen Leistung von 456.44 MB/s Lesen und 437.89 MB/s Schreiben. Alle Tests wurden 30 mal durchgeführt. Jeder Testdurchlauf begann mit einer Warm-up-Phase, bei der die Aufgaben fünf mal ausgeführt wurden, bevor ein weiterer Durchgang gemessen wurde. Der Warm-up Schritt wurde durchgeführt, da sich zeigte, dass sich nach diesem die gemessenen Zeiten eingependelt haben.

4.3 Ergebnisse

Für den allgemeinen Überblick präsentieren die folgenden Grafiken die durchschnittlichen Laufzeiten der verschiedenen Testaufbauten. Die Ergebnisse werden

im nächsten Kapitel genauer analysiert und interpretiert.

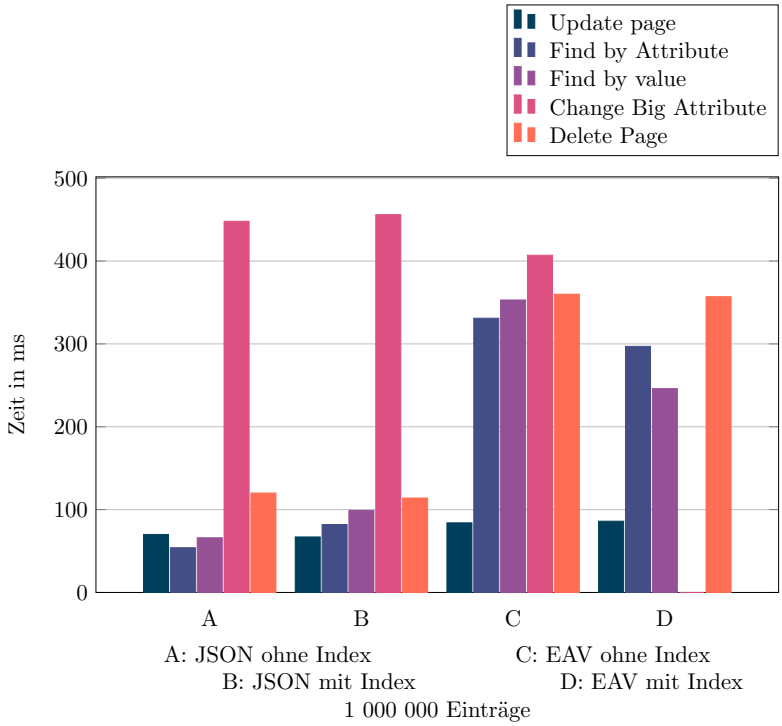

Abbildung 4.3: Ergebnisse PostgreSQL 1.000.000 Einträge

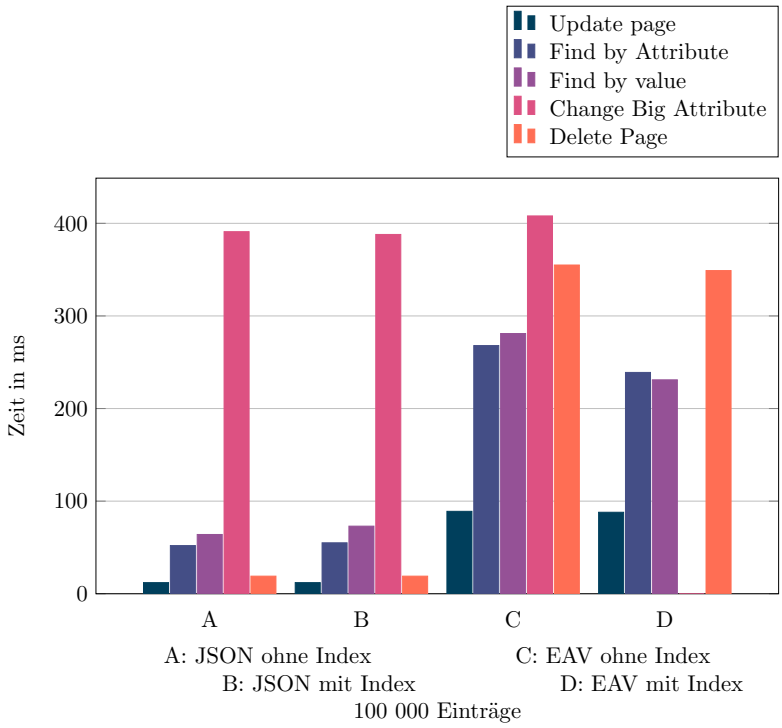

Abbildung 4.4: Ergebnisse PostgreSQL 100.000 Einträge

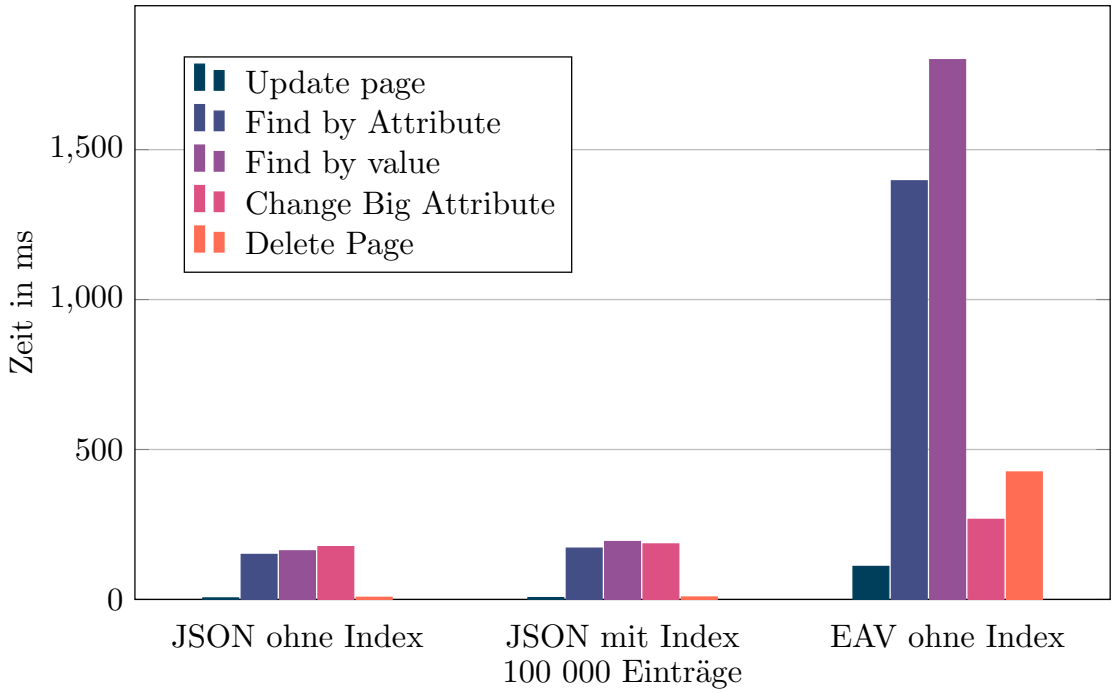

Abbildung 4.5: Ergebnisse Oracle Database 100.000 Einträge

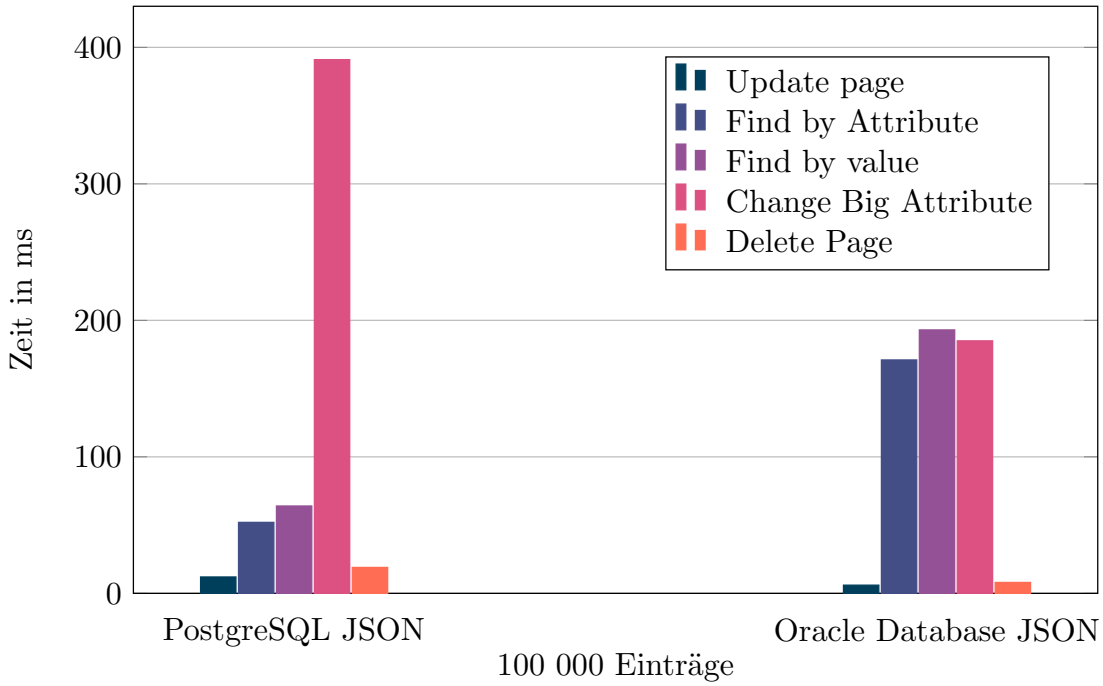

Abbildung 4.6: Vergleich Ergebnisse PostgreSQL und Oracle Database 100.000 Einträge

4.3.1 Statistische Analyse der Ergebnisse

Bei den vorher gezeigten Testwerten handelt es sich nur um Mittelwerte der Stichproben. Jeder Durchlauf der Test-Suite liefert leicht unterschiedliche Ergebnisse. Diese Eigenschaft ist inhärent von Computern und kann nicht umgangen werden. Um die Ergebnisse auch empirisch zu belegen, muss eine Aussage über die Gesamtgrundmengen, d.h. eine imaginäre Menge unendlich vieler Testdurchgänge, erstellt werden. Dafür kann der Welch-Test für mehrere Ansätze und deren Performance-Ergebnisse angewendet werden. Der Welch-Test ist eine Unterkategorie der t-Tests, eine Reihe von Hypothesentests, mit denen ein Mittelwertvergleich von verschiedenen Gesamtmengen, von denen nur eine Teilmenge bekannt ist, durchgeführt werden kann.

Die Nullhypothese des t-Tests ist, dass beide Teilmengen aus derselben Grundmenge entnommen sind. Dies würde für unsere Untersuchung bedeuten, dass unterschiedliche Ansätze keinen Einfluss auf die Performance unserer Testfälle haben. Da wir den Mittelwert zweier verschiedener Mengen vergleichen wollen, würde der Zweistichproben-t-Test verwendet werden. Da die Varianz der Stichproben nicht identisch ist, müsste der Welch-Test angewendet werden. Andere Varianten ermöglichen den Vergleich von einander anhängigen Paaren oder die Abweichung eines bekannten Erwartungswertes gegenüber dem Mittelwert einer Stichprobe. Weitere Voraussetzungen für die Verwendung des t-Tests ist, dass

- die Standardabweichung der Grundgesamtmenge nicht bekannt ist,
- die Stichprobe zufällig entnommen und
- die Grundgesamtheit der Daten normal verteilt ist; oder basierend auf dem zentralem Grenzwertsatz eine Mindestmenge an Proben existiert.

Die Normalverteilung unserer Testwerte kann mithilfe des Shapiro-Wilk-Test bewiesen werden. Die Nullhypothese H_0 dieses Tests nimmt an, dass die Grundmenge, d.h. die Laufzeit unendlicher Testdurchgänge, normalverteilt ist. Die Alternativhypothese H_1 ist, dass keine Normalverteilung vorliegt. Der Test kann nicht dazu

verwendet werden, einzuschätzen, in welcher Art die Daten von der Normalverteilung abweichen. [27]

Tabelle 4.1: P-Werte des Shapiro-Wilsk-Test für alle Test-Szenarien(gerundet auf 3 signifikante Stellen)

Ansatz	UpdatePage	FindByAttribute	FindByValue	ChangeBigAttribute	DeletePage	CreatePage
Postgres-JSON-mit Index-1m	0,00212	0,003	**0,17**	1,38e-05	2,06e-07	5,64e-07
Postgres-JSON-ohne Index-1m	0,00041	0,0118	0,0363	**0,0605**	7,82e-09	0,00154
Postgres-EAV-mit Index-1m	0,000515	1,46e-05	0,00717	-	6,13e-09	0,000144
Postgres-EAV-ohne Index-1m	**0,255**	0,0214	**0,152**	0,0324	5,18e-07	3,09e-05
Postgres-JSON-mit Index-100t	6,55e-09	0,00161	0,000327	1,6e-05	2,19e-06	4,24e-06
Postgres-JSON-ohne Index-100t	8,05e-08	0,00917	**0,454**	0,00355	0,000256	7,44e-11
Postgres-EAV-mit Index-100t	0,00341	0,000383	5,16e-07	-	0,000216	6,01e-06
Postgres-EAV-ohne Index-100t	**0,195**	0,00797	0,000779	**0,81**	0,00147	4,48e-07
Oracle-JSON-mit Index-1m	3,08e-07	0,0412	1,11e-06	2,21e-09	1,16e-06	2,02e-05
Oracle-JSON-ohne Index-100t	9,86e-10	0,054	0,000174	1,35e-09	9,89e-06	3,43e-07
Oracle-EAV-ohne Index-100t	0,0134	**0,139**	0,000445	**0,864**	1,29e-06	4,59e-08

Tabelle 4.1 listet die p-Werte des Shapiro-Wilsk Tests für alle Testdimensionen auf. Der p-Wert gibt an, mit welcher Wahrscheinlichkeit die Nullhypothese nicht abgelehnt wird. Sobald dieser Wert unter das Signifikanzniveau α fällt, wird die Nullhypothese abgelehnt. Die Tabelle markiert alle Einträge, in denen der p-Wert kleiner als $\alpha = 5\%$ ist. Die Auswertung des Tests zeigt, dass die Testwerte nur sehr spärlich normalverteilt sind. So können bei allen Use-Cases im Testfall PostgreSQL Datenbank Ansatz Json mit Index und 100.000 Einträgen mit 95% Konfidenz H_0 abgelehnt werden. Stattdessen wird H_1 angenommen, dass die Daten nicht normalverteilt sind. Alle Ergebnisse, die laut der eben verwendeten Punkte normalverteilt sind, sind in Tabelle 4.1 fett markiert. Das deckt sich mit der Beobachtung von Tianshi Chen et al. [17]. Ähnlich Ihrer Beobachtungen sind die Performancewerte nicht normalverteilt sondern haben vermehrt Ausreißer nach rechts, d.h. die Laufzeit ist besonders groß. Dies ist auch erkennbar an den Boxplots der Testdaten, die im Anschluss näher betrachtet werden, die hauptsächlich nur Ausreißer nach oben verzeichnen.

Dieses Verhalten ist erklärbar. Computer haben eine Mindestlaufzeit für bestimmte Operationen, die jedoch leicht durch unterschiedliche Ereignisse verzögert werden können. Das ist besonders bei dem Testsystem der Fall, bei dem im Hintergrund andere Programme durch Events oder ständigen Aufgaben Rechenleistung in Anspruch nehmen. Aber auch die Datenbanksysteme haben Routinen, die im

Hintergrund ausgelöst werden können und selbst CPU-Performance Tests zeigen ähnliche Verhalten. Tianshin Chen et al. [17] untersuchten auch die nötige Anzahl an Testdaten, um basierend auf dem Zentralem Grenzwertsatz Normalverteilung anzunehmen. Das Ergebnis war, dass mehrere hundert Testdurchläufe durchgeführt werden müssten. Daher kann auch die Alternative der Voraussetzung von Normalverteilung, eine gewisse Menge an Testdaten, nicht eingesetzt werden.

Da basierend auf diesen Beobachtungen der t-Test nicht bzw. nur mithilfe von Vorbereitung der Daten angewendet werden kann, muss ein Alternativtest angewendet werden. Hier eignet sich der Mann-Whitney-U-Test[2] [15]. Dies ist ein Rang-Analyse Test und wird meistens dann verwendet, wenn die Voraussetzungen für den t-Test nicht gegeben sind. Alle p-Werte des Mann-Whitney-U-Test, die in den folgenden Tabellen aufgelistet sind, wurden auf 3 signifikante Stellen gerundet.

Betrachtung des Einfluss des Index

Zunächst betrachten wir die Unterschiede zwischen indizierten und nicht indizierten Ansätzen. Da bei dem EAV Ansatz in der Oracle Umgebung keine Indizes erstellt werden konnten überspringen wir in den Vergleich in diesem Aufbau. Außerdem wird auch der Anwendungsfall "CreatePage" in den Boxplots nicht betrachtet, da die Performance bei allen untersuchten Variationen fast identisch ist und so schnell ist, dass Unterschiede ignoriert werden können.

PostgreSQL EAV 100.000 Elemente Bei einer Gesamtzahl von 100.000 Elementen sieht der Vergleich bei dem Ansatz EAV wie folgt aus:

[2] Auch bekannt als Wilcoxon-Mann-Whitney-Test

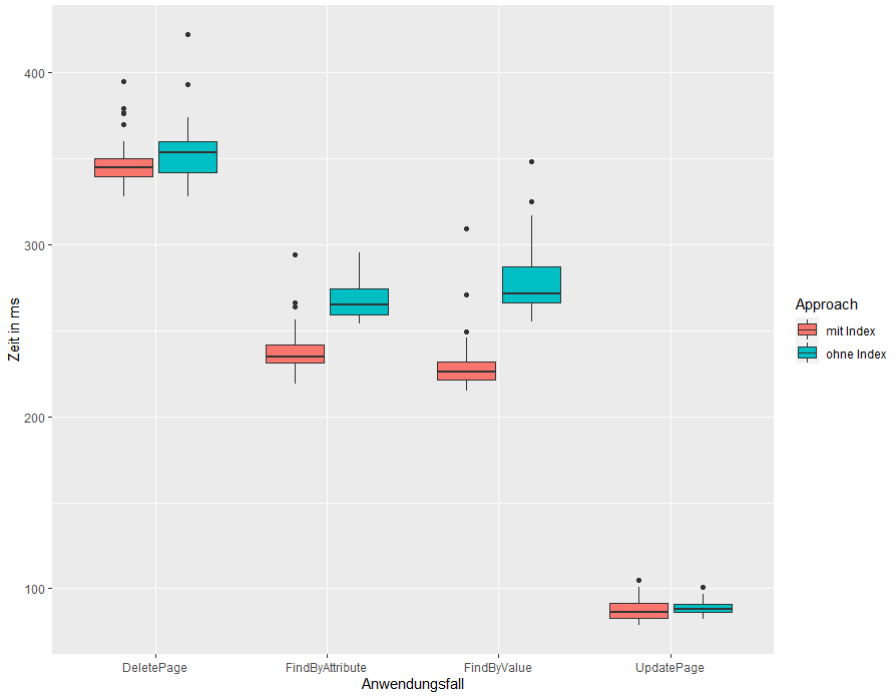

Abbildung 4.7: Vergleich Performance mit bzw. ohne Index
PostgreSQL, EAV, 100.000 Einträge

Hier ist zu beachten, dass der UseCase “ChangeBigAttribute” mit einem Index nicht durchgeführt werden konnte, da das zu indizierende Element die maximalen Größe eines Indexeintrags nicht einhalten kann.

Unterschiede in der Performance sind vor allem bei den Querys, d.h. den Anwendungsfällen “FindByAttribute” und “FindByValue”, erkennbar. Der Ansatz mit Indizes besitzt anscheinend eine bessere Performance. Ob der Unterschied signifikant ist, kann mithilfe des Mann-Whitney-U-Tests bestimmt werden. Der p-Wert, die für alle Vergleiche in diesem Abschnitt in Tabelle 4.3 aufgelistet sind, gibt Auskunft über die Wahrscheinlichkeit, dass die Nullhypothese korrekt ist. In der Regel wird angenommen, dass ein p-Wert von unter 5% einen signifikanten Unterschied zwischen den Mengen beweist. Der p-Wert im Anwendungsfall ”UpdatePage“ liegt bei 0,21. Dieser Wert ist größer als die 5% Grenze. Daher kann die Nullhypothese nicht abgelehnt werden und es kann kein signifikanter Unterschied festgestellt werden. Dass gleiche gilt für den Anwendungsfall “DeletePage”, bei dem der Vergleich der Stichproben einen p-Wert von 0,11 ergab. Das deutet an, dass

der Index keinen Einfluss auf die Performance bei diesen Anwendungsfällen hat. Bei den Anwendungsfällen mit Querys hingegen zeigen die Indizes deutlichen und signifikanten Einfluss. Das spiegelt sich auch in den p-Werten: 0,00000000996 für "FindByAttribute" und 0,00000000166 für "FindByValue". In diesen Fällen ist die Chance, dass die Nullhypothese korrekt ist, verschwindend gering.

In den Beschreibungen der Ergebnisse der meisten folgenden Vergleiche werden die p-Werte nicht explizit aufgelistet. Sie befinden sich jedoch alle am Ende des jeweiligen Abschnitts. Dabei stehen fett markierte Einträge für alle Werte, die sich unter 5% befinden und daher einen signifikanten Unterschied anzeigen.

PostgreSQL JSON 100.000 Elemente Abbildung 4.8 zeigt den Boxplot für den JSON Ansatz mit 100.000 Elementen.

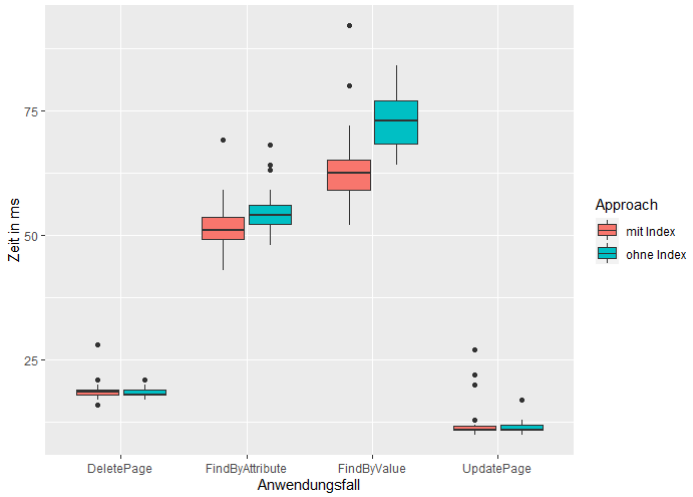

Abbildung 4.8: Vergleich Performance mit bzw. ohne Index
PostgreSQL, JSON, 100.000 Einträge

Da die Performance des Anwendungsfalls "ChangeBigAttribute"wesentlich langsamer ist, betrachten wir ihn in einer extra Grafik 4.9.

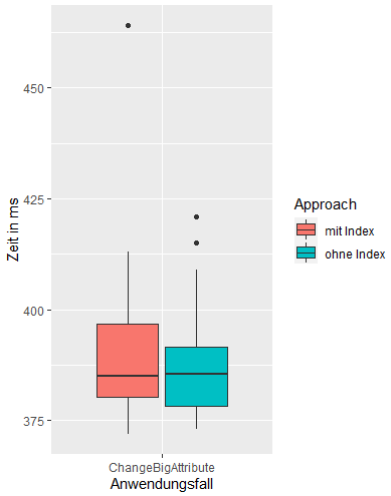

Abbildung 4.9: Vergleich Performance mit bzw. ohne Index
PostgreSQL, JSON, 100.000 Einträge, Anwendungsfall "ChangeBigAttribute"

Es zeigt sich ein ähnliches Bild wie bei dem EAV Ansatz. Die Performance der Query-UseCases FindByAttribute und FindByValue profitieren von dem Index, während die anderen UseCases nur geringe Unterschiede haben. Dies belegt auch der Mann-Whitney-U-Test (vgl. Tabelle 4.3).

PostgreSQL EAV 1.000.000 Elemente Um die Performance der Indizes auch für größere Datenmengen einzuschätzen, betrachten wir nun den Vergleich bei einer Million Einträge.

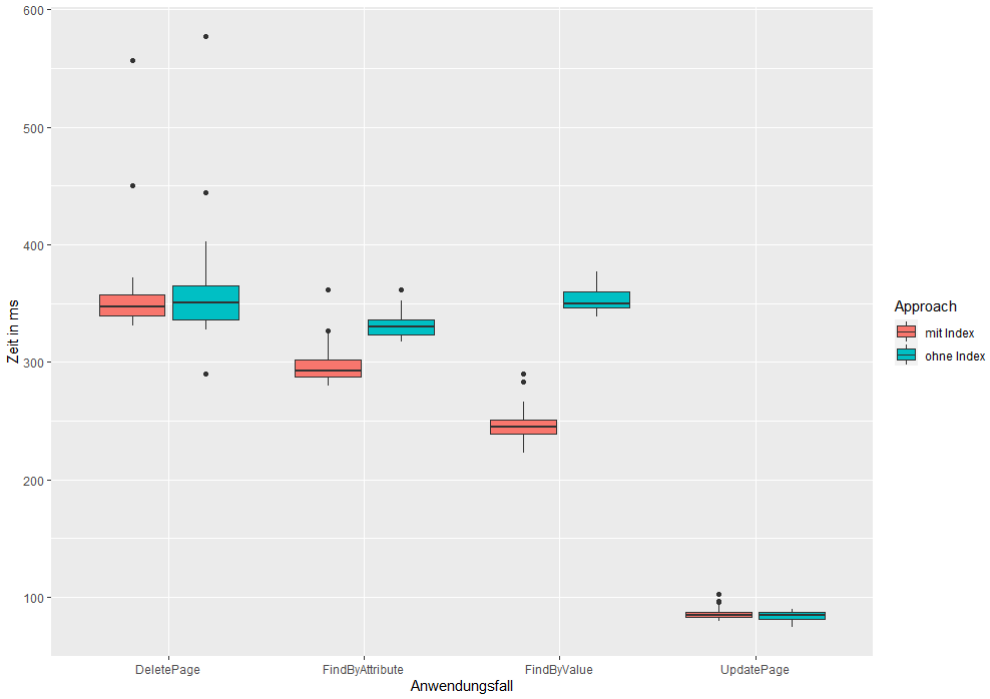

Abbildung 4.10: Vergleich Performance mit bzw. ohne Index
PostgreSQL, EAV, 1.000.000 Einträge

Bei dem EAV Ansatz bleibt das Ergebnis ähnlich. Die Querys profitieren und die restlichen Anwendungsfällen verhalten sich fast gleich (vgl. Tabelle 4.3). Interessanterweise zeigt sich auch ein signifikanter Unterschied der Performance bei der Erstellung von Seiten (in den Boxplots nicht dargestellt). Die Differenz der Medians(mit Index: 4ms, ohne Index: 3ms) von 1ms ist jedoch verglichen mit den anderen Ergebnissen winzig und wird keinen starken Einfluss auf die Gesamtperformance eines Programms haben.

PostgreSQL JSON 1.000.000 Elemente Interessanter ist die Betrachtung des JSON Ansatzes.

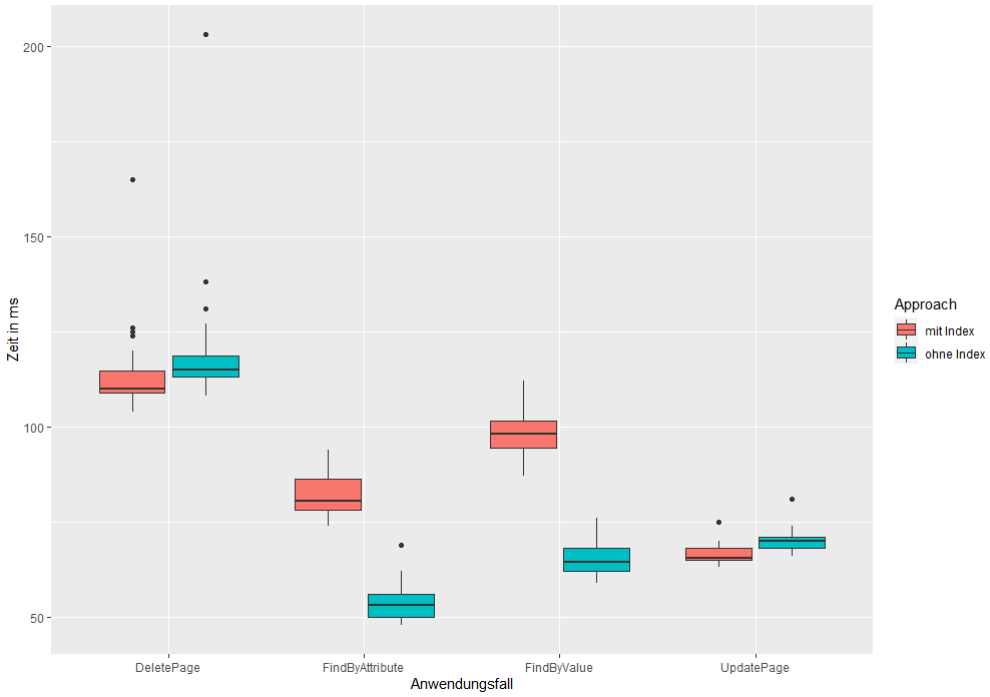

Abbildung 4.11: Vergleich Performance mit bzw. ohne Index
PostgreSQL, JSON, 1.000.000 Einträge

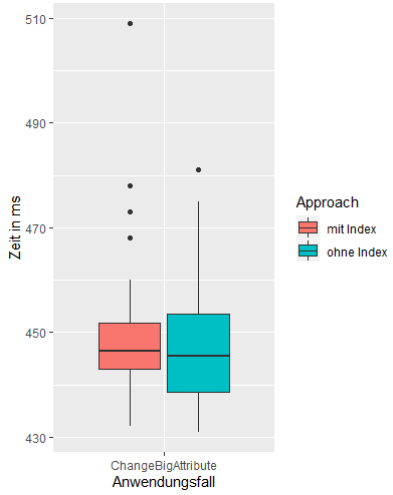

Abbildung 4.12: Vergleich Performance mit bzw. ohne Index
PostgreSQL, JSON, 1.000.000 Einträge, Anwendungsfall "ChangeBigAttribute"

Der Boxplot hier zeigt das Gegenteil der erwarteten Ergebnisse basierend auf den Bisherigen. Während die anderen Anwendungsfälle weiterhin ähnlich bleiben, ist die Situation bei den Querys invertiert. Die Querys mit Index benötigen wesent-

lich mehr Zeit zum Abschluss. Der Anwendungsfall "ChangeBigAttribute" ist auf erstem Blick nach wie vor kaum von dem Index betroffen und scheint sich in beiden Fällen ähnlich zu verhalten. Der zweiseitige Mann-Whitney-U-Test gibt Einblick über den Vergleich der Werte (vgl. Tabelle 4.3). Die Tests ergeben, dass bei allen Anwendungsfällen bis auf ChangeBigAttribute und CreatePage signifikante Unterschiede existieren. Da laut Boxplot unterschiedliche Ansätze jeweils besser sind betrachten wir in diesem Fall zusätzlich die Alternativhypothesen "greater" und "less".

Tabelle 4.2: P-Werte der Alternativhypothesen des Mann-Whitney-U-Tests für den Testaufbau Postgres, 1.000.000 Einträge, JSON; Vergleich mit Index gegenüber ohne Index

Alternativhypothese	UpdatePage	FindByAttribute	FindByValue	ChangeBigAttribute	DeletePage	CreatePage
"less"	**6,12e-06**	1	1	0,711	**0,00197**	0,747
"greater"	1	**1,43e-11**	**1,44e-11**	0,295	0,998	0,258

Durch unterschiedliche Alternativhypothesen können genauere Aussagen getroffen werden. Während die Hypothese "two.sided" nur überprüft, ob ein Unterschied existiert, können die Hypothesen "less" und "greater" bestimmen, ob die Werte der erste Menge gegnüber der zweiten geringer bzw. größer sind.

Oracle JSON 100.000 Elemente Als letztes betrachten wir den Unterschied der Performance in der Oracle Umgebung. Basierend auf den bereits beschriebenen Faktoren, ist in dieser Umgebung nur der Vergleich mit dem Ansatz JSON mit 100.000 Einträgen möglich.

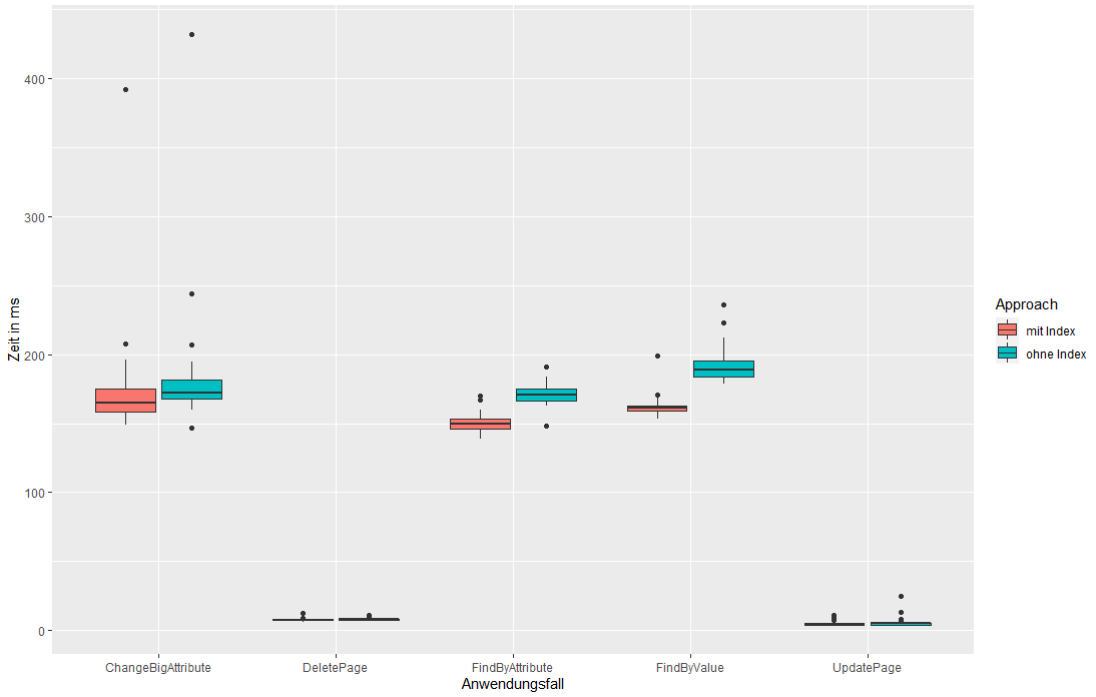

Abbildung 4.13: Vergleich Performance mit bzw. ohne Index
Oracle Database, JSON, 100.000 Einträge

Hier ist wieder ein ähnliches Bild wie in den anderen Testkonfigurationen zu sehen. Wie erwartet verbessert der Index die Suche nach bestimmten Einträgen.

Gesamt betrachtet lässt sich sagen, dass sich der Einsatz von Indizes meistens lohnt. Sie haben keinen signifikanten Einfluss auf Update und Delete Zeiten und verbessern meistens die Performance von Querys. Die einzige Ausnahme ist der GIN Index bei einer Million Einträgen. Bei diesem Fall verschlechtert sich die Performance signifikant.

Tabelle 4.3: P-Werte des Mann-Whitney-U-Test mit zweiseitiger Alternativhypothese für den Vergleich zwischen mit und ohne Index(gerundet auf 3 signifikante Stellen)

Ansatz	UpdatePage	FindByAttribute	FindByValue	ChangeBigAttribute	DeletePage	CreatePage
Postgres-EAV-100.000	0,218	**9,96e-09**	**1,66e-09**	-	0,11	0,432
Postgres-JSON-100.000	0,379	**0,0062**	**5,43e-07**	0,584	0,876	0,336
Postgres-EAV-1m	0,276	**3,94e-09**	**2,96e-11**	-	0,796	**0,00503**
Postgres-JSON-1m	**1,22e-05**	**2,86e-11**	**2,87e-11**	0,589	**0,00393**	0,517
Oracle-JSON-100.000	0,308	**9,34e-10**	**2,77e-10**	**0,0308**	**0,0163**	0,094

Betrachtung des Einfluss des Ansatzes

Nach dem Vergleich der Performance von Indizes werden nun die Unterschiede zwischen den Ansätzen näher betrachtet. Die Wahl, ob der Aufbau mit oder ohne Indizes verwendet wird, wird auf Basis der vorausgehenden Ergebnisse getroffen. Das bedeutet, dass für den Vergleich bei dem Aufbau mit 100.000 Einträgen die Variationen mit Index verglichen werden. Selbst bei Betrachtung des Aufbaus mit 1.000.000 Einträgen wird nur mit Index verglichen, da die bekannten Unterschiede bei dem JSON Ansatz nicht groß genug sind, um Einfluss auf diesen Vergleich zu haben. Außerdem kann bei dem Anwendungsfall "ChangeBigAttribute" und Ansatz EAV nur die Werte ohne Index verwendet werden, da der Test nicht mit Indizes durchgeführt werden konnte.

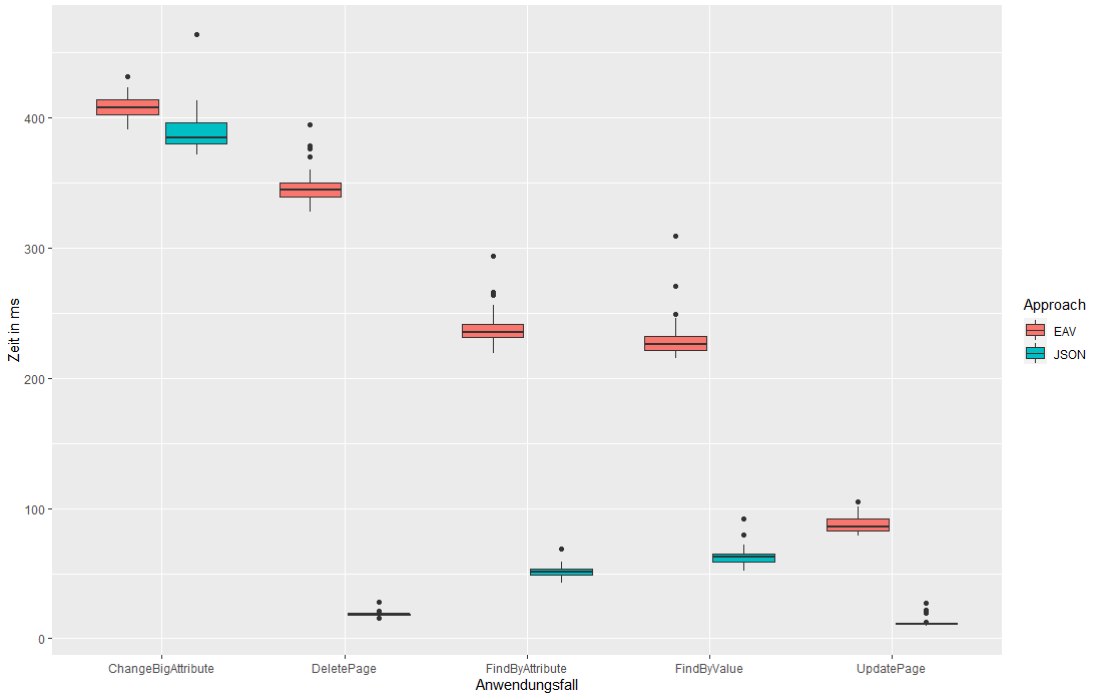

Abbildung 4.14: Vergleich Ansätze EAV / JSON
PostgreSQL, JSON, 100.000 Einträge

PostgreSQL 100.000 Einträge Auf ersten Blick ist direkt erkennbar, dass der JSON Ansatz den EAV Ansatz in fast allen Fällen deutlich schlägt. Die einzige Außnahme ist der Anwendungsfall "ChangeBigAttribute", der relativ ähnlich lange

benötigt. Auch die Ergebnisse des Mann-Whitney-U-Test (vgl. Tabelle 4.4) belegen einen signifikanten Unterschied der Testmengen. Selbst bei dem Anwendungsfall "ChangeBigAttribute" gibt es nur eine Chance von 0,00000000156%, dass die die Nullhypothese korrekt ist und nicht zugunsten der Alternativhypothese "two.sided" abgelehnt wird.

Oracle 100.000 Einträge Nach dem Vergleich der Performance bei 100.000 Einträgen bei PostgreSQL wird nun der selbe Vergleich bei Oracle Database betrachtet.

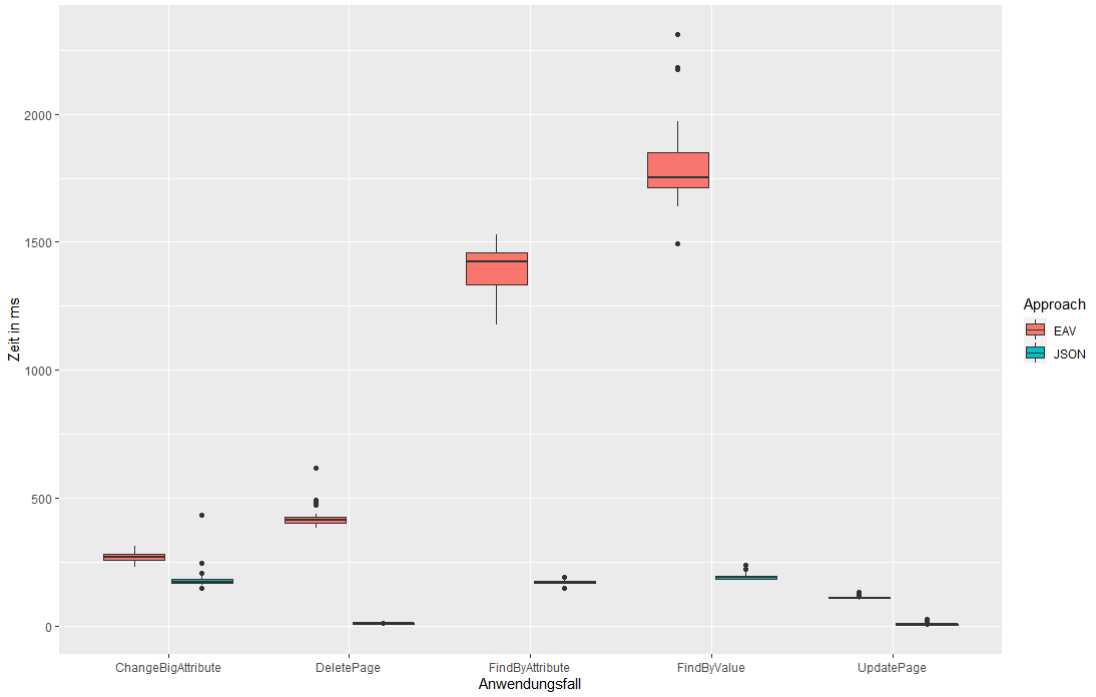

Abbildung 4.15: Vergleich Ansätze EAV / JSON
Oracle Database, 100.000 Einträge

Hier zeigt sich ein ähnliches Bild wie bei dem Vergleich der Ansätze in der PostgreSQL Umgebung. Die benötigte Zeit beim Ansatz EAV für die Querys fallen sogar extremer aus und benötigen länger als der Anwendungsfall "ChangeBigAttribute", der bisher immer als langsamster Testfall auffiel. Die Beobachtungen durch den Boxplot werden weiterhin von dem Mann-Whitney-U-Test gedeckt (vgl. Tabelle 4.4).

PostgreSQL 1.000.000 Einträge Als letztes werden die Ansätze bei der PostgreSQL Instanz mit 1.000.000 Einträgen verglichen.

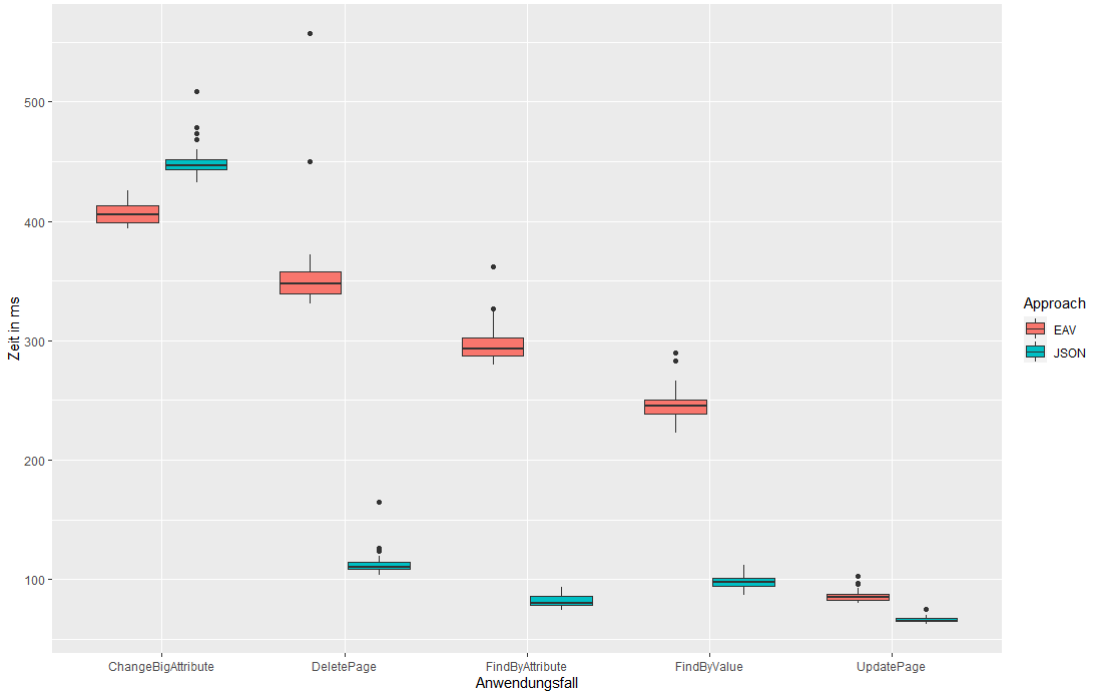

Abbildung 4.16: Vergleich Ansätze EAV / JSON
PostgreSQL, JSON, 1.000.000 Einträge

Hier finden wir die einzige Instanz, in der die JSON Variante langsamer als die zugehörige EAV Variante ist. Das ist bei diesem Vergleich bei dem Anwendungsfall "ChangeBigAttribute" der Fall.

Tabelle 4.4: P-Werte des Mann-Whitney-U-Tests mit Alternativhypothese "two.sided"; Vergleich der Ansätze EAV und JSON, jeweils mit Index

Aufbau	UpdatePage	FindByAttribute	FindByValue	ChangeBigAttribute	DeletePage	CreatePage
Postgres-100k	**1,56e-11**	**2,88e-11**	**2,93e-11**	**1,58e-06**	**2,52e-11**	**0,0117**
Oracle-100k	**1,95e-11**	**2,94e-11**	**2,96e-11**	**8,05e-10**	**1,87e-11**	**0,0445**
Postgres-1m	**2,51e-11**	**2,93e-11**	**2,92e-11**	**2,97e-11**	**2,85e-11**	**0,0305**

Betrachtung des Einfluss des Datenbanksystems

Zum Abschluss werden die Unterschiede zwischen den Datenbanksystemen betrachtet. Hierfür verwenden wir zuerst die Ergebnisse der Versuche der Datenban-

kumgebungen mit Indizes und dem JSON Ansatz, da diese eine bessere Performance bei 100.000 Einträgen gezeigt haben.

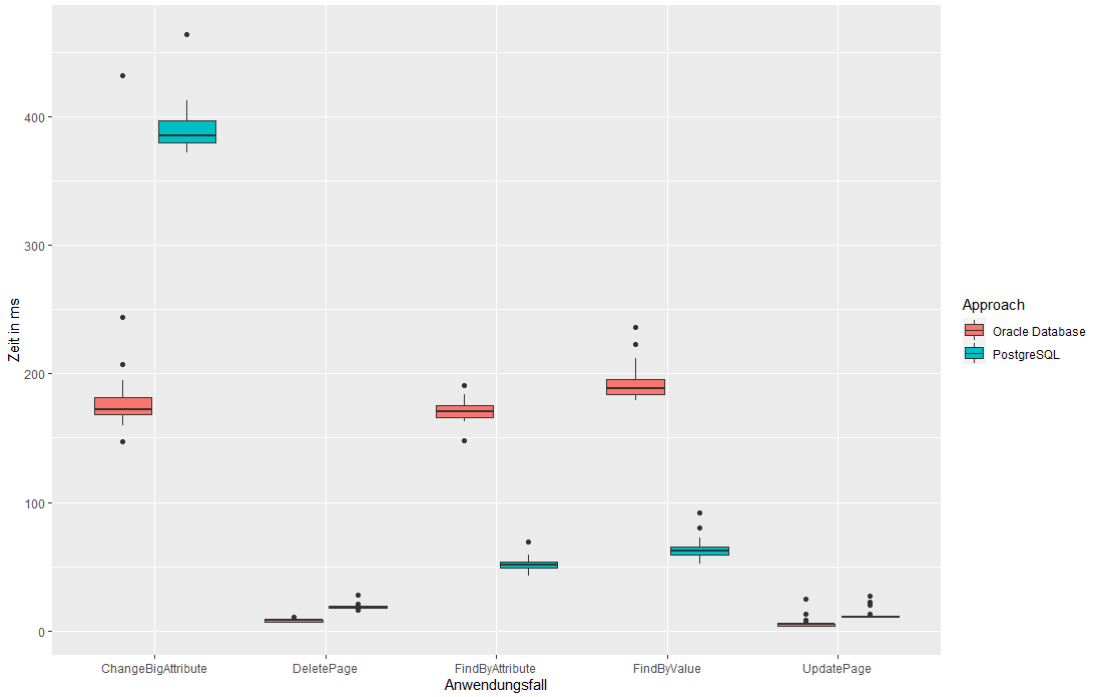

Abbildung 4.17: Vergleich Datenbankensysteme Oracle Database / PostgreSQL Ansatz JSON, 100.000 Einträge

Während der Werte die Anwendungsfälle "DeletePage" und "UpdatePage" zwar signifikant unterschiedlich sind, sind die absoluten Zeiten verglichen mit den Anderen Fällen relativ ähnlich, wobei Oracle Database schneller ist. PostgreSQL kann besonders bei den Querys überzeugen. Oracle Database hingegen schlägt sich viel besser bei dem Fall "ChangeBigAttriubte".

Tabelle 4.5: P-Werte der Alternativhypothesen des Mann-Whitney-U-Tests für den Vergleich PostgreSQL und Oracle Database, Ansatz JSON

Alternativhypothese	UpdatePage	FindByAttribute	FindByValue	ChangeBigAttribute	DeletePage	CreatePage
"less"	1	**1,41e-11**	**1,46e-11**	1	1	**2,76e-07**
"greater"	**1,19e-09**	1	1	**2,51e-10**	**7,89e-12**	1

Vergleich EAV Ansatz Obwohl sich der EAV Ansatz meistens schlechter schlägt als der JSON Ansatz werden zum Abschluss noch die Ergebnisse der Datenbanken mit EAV verglichen.

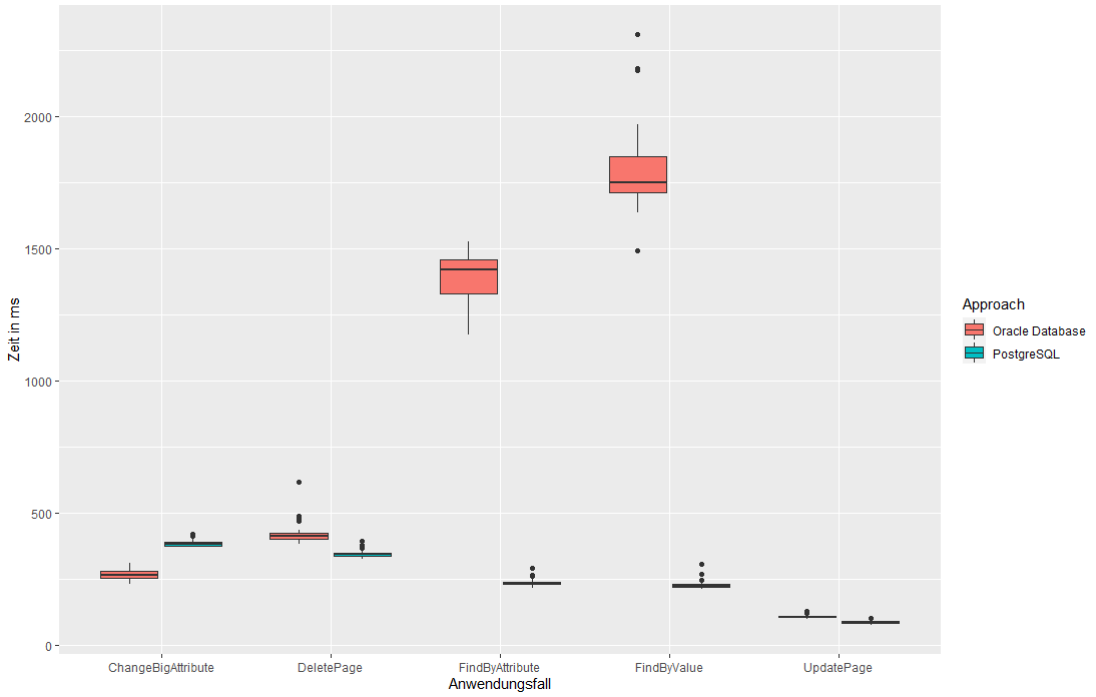

Abbildung 4.18: Vergleich Datenbankensysteme Oracle Database / PostgreSQL Ansatz EAV, 100.000 Einträge

Das auffälligste am Vergleich ist die schlechtere Performance von Oracle Database bei den Querries. Hier ist zu beachten, dass bei Oracle Database keine zusätzlichen Indizes eingesetzt werden konnten. Ansonsten ist bemerkenswert, dass die Verhältnisse der Anwendungsfälle “DeletePage” und ”UpdatePage” verglichen mit dem Ansatz JSON invertiert haben.

Tabelle 4.6: P-Werte der Alternativhypothesen des Mann-Whitney-U-Tests für den Vergleich PostgreSQL und Oracle Database, Ansatz EAV

Alternativhypothese	UpdatePage	FindByAttribute	FindByValue	ChangeBigAttribute	DeletePage	CreatePage
“less”	**4,48e-11**	**1,5e-11**	**1,49e-11**	1	**2,56e-11**	**1,57e-06**
“greater”	1	1	1	**1,49e-11**	1	1

4.3.2 Interpretation der Ergebnisse

Die statistische Analyse der Ergebnisse hat sowohl erwartete als auch unerwartete Resultate aufgedeckt. Sie beweist auch den Nutzen mehrerer Testdurchläufe. Eine hohe Zahl an Ausreißern nach oben, d.h. Stichproben die untypisch lange

dauerten, kann dazu führen, dass Ansätze als gleich bewertet werden. Ein wahrer Performancevergleich kann nur mit genug Stichproben stattfinden. In diesem Abschnitt werden verschiedene Erklärungen und Lösungsvorschläge für Auffälligkeiten in den Ergebnissen in Erwägung gezogen. Sie können als Grundlage für weitere Forschung in Betracht gezogen werden.

Unterschiede Performance PostgreSQL und Oracle Database Der Unterschied der Performance zwischen PostgreSQL und Oracle Database mit Ansatz JSON lässt sich leicht kategorisieren. PostgreSQL schlägt Oracle Database bei den Querys, während Oracle bessere Performance bei den restlichen Anwendungsfällen zeigt. Die Erklärung für diese Beobachtung könnte ähnlich wie bei dem CAP-Theorem an den Fokusen der Indizes liegen. Desto schneller Querys basierend auf einem Index sein sollen, desto komplexer ist es, den Index aktuell zu halten. Ein einfacherer Index hingegen kann leichter aktualisiert werden.

Der GIN von PostgreSQL ist standardmäßig mit einer "fast update list" ausgestattet. In dieser Liste werden Änderungen an den Daten im Index zwischengespeichert, bevor sie mit dem Index zusammengeführt werden. Dies geschieht entweder per Hand oder automatisch, wenn die Liste eine bestimmte Größe überschreitet. Diese Eigenschaft macht es unwahrscheinlicher, dass dies der Grund ist. Ein anderer Grund könnten die verwendeten Datentypen sein. Während Oracle Database einen Constraint benutzt, besitzt PostgreSQL einen eigenen Datentyp für JSON Felder. Alle Daten, die in einem `jsonb` Feld abgelegt werden, werden automatisch in eine Binärrepresentation umgewandelt. Der zusätzliche Aufwand dieser Umwandlung könnte der Einfluss sein, der die Querys erleichtert und die Update Funktion langsamer macht.

Ein weiterer Einfluss auf die Ergebnisse ist das Tuning der Datenbanken. In den Tests wurden die Docker Images der Hersteller benutzt, ohne sie speziell auf die Aufgabenstellung vorzubereiten. Standardeinstellungen, die in diesen Images vorhanden sind, können die Performance auf verschiedene Art und Weise beeinflussen. Außerdem wurden für die Tests die Express Edition von Oracle Database

verwendet, die im Gegensatz zu der Enterprise Edition in den Features, etwa der maximalen Größe der Daten, eingeschränkt ist. Auch grundlegende Vergleiche der Performance dieser Datenbanken in anderen Anwendungsfällen werden weiterhin durchgeführt und zeigen Unterschiede in der Performance auf [32].

Schlechte Performance des GIN Index bei vielen Einträgen Ein besonders interessantes Ergebnis ist die schlechte Performance des Generalized Inverted Index (GIN) der PostgreSQL Datenbank. Eigentlich wird von einem Index erwartet, dass er die Laufzeit von Querys verringert, sobald eine gewisse Menge an Daten vorhanden sind. Die praktischen Tests ergaben jedoch, dass der Index mit steigender Anzahl von Einträgen zu langsameren Querys führte. Während bei 100.000 Einträgen der Anwendungsfall "FindByValue" der Testaufbau mit Index beim Vergleich der Mediane etwa 10,5ms schneller ist, fällt die Performance bei einer Millionen Einträgen und ist 33,5ms langsamer als das Gegenstück ohne Index.

Eine erste Annahme für den Grund für diese Performance könnte der Aufbau der Testdaten sein. Indizes für Volltextsuche sind meistens für die Verwendung mit echten Texten optimiert. Unsere Testdaten sind jedoch komplett zufällig generiert. Die Werte der dynamischen Attribute sind eine Folge von 16 zufälligen Zeichen. Dadurch vermindert sich die Chance, dass sich Werte wiederholen. Die Werte echter Datensätze würden aus einfachen Zahlen und sich wiederholende Wörter oder Zeichenketten bestehen. Die einzigen Werte, die sich in dem Testdatensatz wiederholen, sind die Attributnamen und die Schlüssel. Eine genaue Untersuchung der Funktionsweise des GIN, den PostgreSQL in der JSON Umgebung verwendet, hilft bei der Einschätzung dieser Annahme. Bei der Erstellung des Indizes gibt es die Wahl zwischen zwei Klassen von GIN, eine Standardversion und die Version `jsonb_path_ops`. Hier ist zu beachten, dass der `jsonb_path_ops` Index nur den Operator `@>` unterstützt. Der Unterschied liegt in der Art und Weise, wie der Pfad der Elemente in der JSON Struktur in dem Index abgelegt werden. Die Dokumentation des Index benutzt folgendes Beispiel [43]: Das JSON-Objekt `{"foo": {"bar": "baz"}}` würde bei dem `jsonb_path_ops` GIN Index zu einem

einzelnen Eintrag führen, der aus einem Hash der Pfadelemente und dem Wert selbst besteht. Die andere Variante erstellt einen Eintrag für jeden Schlüssel auf dem Weg und einen extra für den Wert. Eine Suche für genau dieses Element würde dann einen Eintrag suchen, der alle drei Elemente enthält. Ein kurzer Test vor den gesamten Testläufen, die in dieser Arbeit besprochen werden, konnte die Angabe der Dokumentation, dass der GIN mit der Klasse `jsonb_path_ops` schneller ist, nachweisen. Das deckt sich auch mit den Angaben einer Präsentation von Cristophe Pettus von 2015 [41].

Die Dokumentation des `jsonb` Dateityps legt auch nahe, dass der Containment Check, d.h. die Operation, die für die Querys benutzt werden, besser für JSON Objekte als für JSON Arrays geeignet ist [43]. Leider wird nicht darauf eingegangen, ob diese Eigenschaft auch zu tragen kommt, wenn ein GIN verwendet wird. Das Root-Element des JSON Elements, das die Attribute abspeichert, ist ein Array. Die Dokumentation besagt auch, dass Arrays linear durchsucht werden. Da jedoch die durchschnittliche Anzahl an Attributen bei den beiden Test-Setups gleich ist, sollte dies nicht der Grund sein. Selbst falls dies der Fall ist, kann das Array, welches die Attribute beinhaltet, nicht geändert werden, da das gesamte Konzept auf der Fähigkeit der Arrays, eine beliebige Anzahl von Elementen zu enthalten, aufbaut.

Einen weiteren Einfluss auf die Performance des Index haben die Informationen, die dem Querry Builder zur Verfügung stehen. Hier existiert das Problem, das PostgreSQL keine dynamischen Statistiken für Dokumentfelder, darunter der in den Tests verwendete Typ `jsonb`, anlegt. Stattdessen wird einfach angenommen, dass der Index 1% der Reihen trifft [13, 47]. Dies verfälscht die Wirksamkeit des Index und führt zu schlechten Entscheidungen des Query Planers.

Schlechte Performance des Ansatz EAV bei dem Anwendungsfall "Delete-Page" Bereits bei dem ersten untersuchten Vergleich, PostgreSQL EAV 100.000 Einträge mit und ohne Index, fällt die äußerst schlechte Performance beim Löschen von Elementen. Dieser Vorgang dauert sogar länger als die Suche nach Elementen.

Diese Eigenschaft zieht sich durch alle Test-Variationen mit dem Ansatz EAV. Sie kann durch die primitive Umsetzung dieser Methode erklärt werden. Jeder Attributwert nimmt eine Zeile in der Tabelle AttributValues ein. Beim Löschen eines Elements aus der Datenbank wird über die Attribute dieses Elements iteriert und jeder Eintrag in einem eignen SQL-Befehl gelöscht. Ein einziger SQL-Befehl, der alle Einträge mit der Id des zu löschenden Elements entfernt, könnte bereits zu einer Leistungssteigerung führen.

4.4 Fazit

Die Programmierung der Testumgebung und das Auswerten der Ergebnisse hat viele interessante Erfahrungen ermöglicht. Zwar konnte ein funktionales Programm erstellt werden, mit dem dynamische Attribute abgespeichert werden können, doch ein Teil der Features ist mit der Zeit in den Hintergrund gerückt. Dazu gehören die Möglichkeiten, mehrere Werte unter einem Attribut abzuspeichern und eine praktischere Verwendung der Attributtypen. Beide Faktoren sind noch im Design der Anwendung oder des JSON-Elements sichtbar, sind jedoch nicht mit besonderer Logik versehen und könnten mit weiterer Problematiken und Performanceeinbußen verbunden sein. Für den Einsatz in einem realen Programm könnten Stakeholder weitere Features, etwa das Speichern von komplexen Objekten als Attribute oder einer Fuzzy-Suche nach Attributwerten fordern, die das gesamte System noch komplexer machen. Diese Faktoren zeigen zusammen mit den Testergebnissen, warum dies ein Feature mit schlechtem Ruf ist. Dennoch hat sich bei der Recherche nach bekannten Umsetzungen gezeigt, dass es genug reale Fälle gibt, die genau dieses Feature benötigen, etwa im medizinischen Bereich.

Der Vergleich der Ansätze und die Untersuchung des Einfluss des Datenbanksystems und der Indizes haben mehrere Beobachtungen ermöglicht. Der Einsatz von EAV ist mit mehr Programmcode verbunden, ist dadurch fehleranfälliger und hat eine schlechtere Leistung als der JSON Ansatz. Dennoch sollte EAV nicht komplett ignoriert werden. Erstens haben nicht alle Datenbanksysteme so funktionsfähige

JSON Unterstützung, wie die beiden die untersucht wurden. Zusätzlich gibt es mehrere Verbesserungspunkte für die Umsetzung. Neben der schlechten Lösch-Performance, die bereits besprochen wurde, könnte der Grundaufbau verbessert werden. Existierende Systeme haben den EAV Ansatz mit einem Klassensystem erweitert. Dies würde den Einsatz speziellere Indizes ermöglichen. Auch die Datenbankfunktion Views, mit denen die Performance häufiger Joins verbessert werden kann, zu erstellen, wurde noch nicht untersucht und könnte den Unterschied in der Performance verbessern.

Der JSON Ansatz im Gegenzug war einfacher umzusetzen. Dies könnte an Erfahrungen mit dieser Datenrepräsentationsart liegen, da JSON auch bei der Programmierung von Webschnittstellen oder in Konfigurationsdateien Einsatz findet. Außerdem wurden in den Tests keine komplexen Objekte verwendet, was die Serialisierung erleichterte. Ein deutlicher Nachteil ist jedoch, dass jede Implementation stark an des verwendete Datenbanksystem gekoppelt ist. Beide Datenbankanbieter, die untersucht wurden, hatten eine eigene Syntax für Querys für Inhalte dieses Typs und spezielle Datentypen. Daher muss bei der Programmierung das Zielsystem bekannt sein. Diese Eigenschaft beeinflusst auch die vorhandene Dokumentation. Desto spezieller ein Feature ist, desto schwieriger ist es, Lösungen für häufige Probleme oder Anleitungen zu finden. Ein Problem, welches sich bei den Tests zeigte, ist die schlechte Performance des GIN der PostgreSQL Datenbank. Recherche für dieses Problem führte zu weiteren Schwächen. Fehlende Statistiken für den Index erschweren richtige Query Planung, was zu schlechterer Performance führt. Hier wäre der Vergleich zu dem Oracle Full-Text-Search Index sehr interessant gewesen, was zu einem weiteren Problem führt. Das Lizenzmodel der Oracle Datenbank erschwert die Forschung mit dieser. Die Einschränkung, dass die freie Datenbank maximal 12GB groß sein darf, machte die Tests mit mehr Testeinträgen unmöglich und verwehrt die Möglichkeit, die Indizes in genau der Umgebung zu testen, in der ihre Performance besonders wichtig ist.

Generell kann basierend auf den Testergebnissen folgende Entscheidung empfohlen werden: Falls die verwendete Datenbank einer der beiden untersuchten ist,

hat sich der JSON Ansatz in allen Anwendungsfällen als besser erwiesen. Die Wahl zwischen den Datenbanksystemen kann basierend auf der Wichtigkeit oder Menge der Anwendungsfälle getroffen werden. Liegt der Fokus auf Querys, d.h. der Suche nach Elementen mit bestimmen Attributen oder Attributwerten, empfiehlt sich PostgreSQL. Falls jedoch die Dauer von Update und Delete Operationen wichtiger ist, fällt die Wahl eher auf Oracle Database.

Hier werden natürlich viele weitere Faktoren, die die Wahl der Datenbank beeinflussen können, ignoriert. Eine solche Eigenschaft ist die Erst-Einrichtung der Datenbank. In der Docker-Umgebung, in der die Tests durchgeführt wurden, war die Einrichtung der PostgreSQL Datenbank einfacher, da das Basisimage bereits fertig gebaut auf DockerHub verfügbar ist. Andere Faktoren sind z.B. die Kosten, existente Kenntnisse mit den Datenbanken, die Verwaltungstools, Zugriffsverwaltung und viele mehr.

Falls keine der beiden Datenbanken verfügbar ist, ist der EAV Ansatz trotzdem nicht zu unterschätzen. Die Performance ist jedoch viel stärker von der Implementation als von der Datenbank abhängig, da die Verwaltung der Attributwerte in der Datenbank komplexer ist und per Hand programmiert werden muss. Dadurch entstehen mehr Möglichkeiten, Fehler und schlecht performanten Code zu schreiben.

5 Abschluss

Letztendlich konnte in dieser Arbeit nur ein kleiner Teil der möglichen Umsetzungsarten untersucht werden. Der Grundaufbau des Testprojekts, welches nun frei im Internet verfügbar ist, würde es möglich machen, es mit vielen weiteren Lösungswegen zu erweitern und diese zu vergleichen. Besonders wichtig wäre die Untersuchung der NO- und New SQL Datenbanken. Obwohl nur ein kleiner Teil dieser Datenbanken ein passendes Datenmodell und Fokus haben, wären diese Datenbanken wichtige Testkandidaten. Dazu gehören z.B. der Wide-Column Store Cassandra oder die In-Memory Datenbank Hana. Leider wird der Einsatz moderner und nicht stark etablierter Lösungen oft durch Angst vor Integrations- oder Sicherheitsproblemen in etablierten Organisationen verhindert oder erschwert. Hier besteht die Hoffnung, das wachsende Bekanntheit und wachsendes Wissen über die Technologien die Adaption dieser Datenbanken in Zukunft leichter machen.

Ein weiteres Gebiet, das Betrachtung verdient, ist der Einfluss des Aufbaus der Testdaten auf die Performance. Die schlechte Leistung des GIN könnte ein Indiz dafür sein. Desto näher sich die Testdaten an realen Daten orientieren, desto wertvoller sind alle Aussagen, die mit den Tests getroffen werden. Es wird nie eine perfekte Struktur, die alle Anwendungsgebiete abdeckt, geben. Doch jede mögliche Annäherung stärkt das Vertrauen in die Ergebnisse.

Auch die untersuchten Anwendungsfälle könnten erweitert werden. Die Performance von CRUD Operationen ist ein Grundbaustein der Performance. Zusätzlich könnte der Einfluss von parallelen Zugriffen mit mehreren Benutzern und der Einfluss durch das Hinzufügen oder Entfernen von Attributen gemessen werden. Der zweite Fall würde in den untersuchten Ansätzen nur geringen Einfluss haben,

könnte aber Andere stärker treffen.

Gesamt gesehen gibt es viele Möglichkeiten, dieses Thema weiter zu untersuchen. Vielleicht kann diese Arbeit eine Starthilfe für die weitere Betrachtung sein.

Abbildungsverzeichnis

2.1 Datenbankmodell EAV Ansatz 17
2.2 Vergleich des Aufbau 21

4.1 Klassenmodell des EAV Ansatzes 38
4.2 Outline Datenbankadapter Interface 39
4.3 Ergebnisse PostgreSQL 1.000.000 Einträge 47
4.4 Ergebnisse PostgreSQL 100.000 Einträge 47
4.5 Ergebnisse Oracle Database 100.000 Einträge 48
4.6 Vergleich Ergebnisse PostgreSQL und Oracle Database 100.000 Einträge 48
4.7 Vergleich Performance mit bzw. ohne Index PostgreSQL, EAV, 100.000 Einträge 52
4.8 Vergleich Performance mit bzw. ohne Index PostgreSQL, JSON, 100.000 Einträge 53
4.9 Vergleich Performance mit bzw. ohne Index PostgreSQL, JSON, 100.000 Einträge, Anwendungsfall "ChangeBigAttribute" 54
4.10 Vergleich Performance mit bzw. ohne Index PostgreSQL, EAV, 1.000.000 Einträge 55
4.11 Vergleich Performance mit bzw. ohne Index PostgreSQL, JSON, 1.000.000 Einträge 56
4.12 Vergleich Performance mit bzw. ohne Index PostgreSQL, JSON, 1.000.000 Einträge, Anwendungsfall "ChangeBigAttribute" 56
4.13 Vergleich Performance mit bzw. ohne Index Oracle Database, JSON, 100.000 Einträge 58

4.14 Vergleich Ansätze EAV / JSON PostgreSQL, JSON, 100.000 Einträge 59
4.15 Vergleich Ansätze EAV / JSON Oracle Database, 100.000 Einträge 60
4.16 Vergleich Ansätze EAV / JSON PostgreSQL, JSON, 1.000.000 Einträge 61
4.17 Vergleich Datenbankensysteme Oracle Database / PostgreSQL Ansatz JSON, 100.000 Einträge 62
4.18 Vergleich Datenbankensysteme Oracle Database / PostgreSQL Ansatz EAV, 100.000 Einträge 63

Quellcodeverzeichnis

2.1 Beispiel XML 21
2.2 Beispiel JSON 21

4.1 PostgreSQL Querys 31
4.2 Oracle Database JSON Constraint 32
4.3 Oracle Database, FindByAttribute-Query 32
4.4 Oracle Database, FindByValue-Query 33
4.5 Prepared Statement mit JDBC 37
4.6 Attribute-JSON 41

Literaturverzeichnis

[1] *PYPL PopularitY of Programming Language index.* https://pypl.github.io/PYPL.html. Version: 01.07.2021

[2] *XML Path Language (XPath) 3.1.* https://www.w3.org/TR/2017/REC-xpath-31-20170321/. Version: 06.04.2021

[3] *Extensible Markup Language (XML) 1.0 (Fifth Edition).* https://www.w3.org/TR/xml/. Version: 09.10.2018

[4] *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures.* https://www.w3.org/TR/xmlschema11-1/. Version: 09.10.2018

[5] *MySQL :: MySQL 8.0 Reference Manual :: 11.5 The JSON Data Type.* https://dev.mysql.com/doc/refman/8.0/en/json.html. Version: 10.07.2021

[6] *MySQL :: MySQL 8.0 Reference Manual :: 13.1.20.9 Secondary Indexes and Generated Columns.* https://dev.mysql.com/doc/refman/8.0/en/create-table-secondary-indexes.html#json-column-indirect-index. Version: 10.07.2021

[7] *Active Objects.* https://developer.atlassian.com/server/framework/atlassian-sdk/active-objects/. Version: 12.07.2021

[8] *Hibernate ORM.* https://hibernate.org/orm/. Version: 13.07.2021

[9] *Oracle Database Express Edition | Oracle Deutschland.* https://www.oracle.com/de/database/technologies/appdev/xe.html. Version: 13.07.2021

[10] *Testcontainers*. https://www.testcontainers.org/. Version: 13.07.2021

[11] *Liquibase | Open Source Version Control for Your Database*. https://www.liquibase.org/. Version: 15.06.2021

[12] *Apache Cayenne*. https://cayenne.apache.org/. Version: 19.03.2021

[13] Brandstetter, Erwin: *Inconsistent statistics on jsonb column with btree index - Database Administrators Stack Exchange*. https://dba.stackexchange.com/questions/167525/inconsistent-statistics-on-jsonb-column-with-btree-index. Version: 08.07.2021

[14] Brewer, Eric A.: Towards Robust Distributed Systems. https://sites.cs.ucsb.edu/~rich/class/cs293-cloud/papers/Brewer_podc_keynote_2000.pdf. In: *Symposium on Principles of Distributed Computing 2000*

[15] Bridge, Patrick D. ; Sawilowsky, Shlomo S.: Increasing Physicians' Awareness of the Impact of Statistics on Research Outcomes. In: *Journal of Clinical Epidemiology* 52 (1999), Nr. 3, S. 229–235. http://dx.doi.org/10.1016/S0895-4356(98)00168-1. – DOI 10.1016/S0895–4356(98)00168–1. – ISSN 08954356

[16] Chamberlin, Donald D. ; Astrahan, Morton M. ; Blasgen, Michael W. ; Gray, James N. ; King, W. F. ; Lindsay, Bruce G. ; Lorie, Raymond ; Mehl, James W. ; Price, Thomas G. ; Putzolu, Franco ; Selinger, Patricia G. ; Schkolnick, Mario ; Slutz, Donald R. ; Traiger, Irving L. ; Wade, Bradford W. ; Yost, Robert A.: A history and evaluation of System R. In: *Communications of the ACM* 24 (1981), Nr. 10, S. 632–646. http://dx.doi.org/10.1145/358769.358784. – DOI 10.1145/358769.358784. – ISSN 0001–0782

[17] Chen, Tianshi ; Guo, Qi ; Temam, Olivier ; Wu, Yue ; Bao, Yungang ; Xu, Zhiwei ; Chen, Yunji: Statistical Performance Comparisons of Computers.

In: *IEEE Transactions on Computers* 64 (2015), Nr. 5, S. 1442–1455. http://dx.doi.org/10.1109/TC.2014.2315614. – DOI 10.1109/TC.2014.2315614. – ISSN 1557–9956

[18] Chu, Eric ; Beckmann, Jennifer ; Naughton, Jeffrey: The case for a wide-table approach to manage sparse relational data sets. In: Zhou, Lizhu (Hrsg.): *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. New York, NY : ACM, 2007. – ISBN 9781595936868, S. 821

[19] Codd, E. F.: A relational model of data for large shared data banks. In: *Communications of the ACM* 13 (1970), Nr. 6, S. 377–387. http://dx.doi.org/10.1145/362384.362685. – DOI 10.1145/362384.362685. – ISSN 0001–0782

[20] Davoudian, Ali ; Chen, Liu ; Liu, Mengchi: A Survey on NoSQL Stores. In: *ACM Computing Surveys (CSUR)* 51 (2018), Nr. 2, S. 1–43. http://dx.doi.org/10.1145/3158661. – DOI 10.1145/3158661. – ISSN 0360–0300

[21] Ecma International: *ECMA-404: The JSON Data Interchange Syntax*. https://www.ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf. Version: 04.02.2021

[22] Giceva, Jana ; Sadoghi, Mohammad: Hybrid OLTP and OLAP. Version: 2018. http://dx.doi.org/10.1007/978-3-319-63962-8_179-1. In: Sakr, Sherif (Hrsg.) ; Zomaya, Albert (Hrsg.): *Encyclopedia of Big Data Technologies*. Cham : Springer International Publishing, 2018. – DOI 10.1007/978–3–319–63962–8_179–1. – ISBN 978–3–319–63962–8, S. 1–8

[23] GitHub: *google/gson*. https://github.com/google/gson. Version: 05.07.2021

[24] GitHub: *github/gh-ost*. https://github.com/github/gh-ost. Version: 12.07.2021

[25] Grandjonc, Louise: *PostgreSQL's indexes - GIN | Louise Grandjonc, Python/SQL developer but also a human.* http://www.louisemeta.com/drafts/indexes-gin/. Version: 24.10.2020

[26] Haigh, Thomas: How Charles Bachman invented the DBMS, a foundation of our digital world. In: *Communications of the ACM* 59 (2016), Nr. 7, S. 25–30. http://dx.doi.org/10.1145/2935880. – DOI 10.1145/2935880. – ISSN 0001–0782

[27] Hedderich, Jürgen ; Sachs, Lothar: *Angewandte Statistik: Methodensammlung mit R.* 17., überarbeitete und ergänzte Auflage. Berlin : Springer Spektrum, 2020. http://dx.doi.org/10.1007/978-3-662-62294-0. http://dx.doi.org/10.1007/978-3-662-62294-0. – ISBN 978–3–662–62293–3

[28] Johnson, Stephen B. ; Cimino, James J. ; Friedman, Carol ; Hripcsak, George ; Clayton, Paul D.: Using Metadata to Integrate Medical Knowledge in a Clinical Information System. In: *Proceedings of the Annual Symposium on Computer Application in Medical Care* (1990), 340–344. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2245431/. – ISSN 0195–4210

[29] Jon Heller: Avoid Anti-Patterns. Version: 2019. http://dx.doi.org/10.1007/978-1-4842-4517-0_15. In: *Pro Oracle SQL Development.* Apress, Berkeley, CA, 2019. – DOI 10.1007/978–1–4842–4517–0_15, 355–378

[30] Kleppmann, Martin: *Please stop calling databases CP or AP.* https://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html. Version: 2015

[31] Marenco, L. ; Nadkarni, P. ; Skoufos, E. ; Shepherd, G. ; Miller, P.: Neuronal database integration: the Senselab EAV data model. In: *Proceedings. AMIA Symposium* (1999), S. 102–106. – ISSN 1531–605X

[32] Martins, Pedro ; Tomé, Paulo ; Wanzeller, Cristina ; Sá, Filipe ; Abbasi, Maryam: Comparing Oracle and PostgreSQL, Performance and Optimization.

In: Rocha, Álvaro (Hrsg.) ; Adeli, Hojjat (Hrsg.) ; Dzemyda, Gintautas (Hrsg.) ; Moreira, Fernando (Hrsg.) ; Ramalho Correia, Ana M. (Hrsg.): *Trends and Applications in Information Systems and Technologies*. Cham : Springer International Publishing, 2021. – ISBN 978–3–030–72651–5, S. 481–490

[33] McDougal, Robert A. ; Morse, Thomas M. ; Carnevale, Ted ; Marenco, Luis ; Wang, Rixin ; Migliore, Michele ; Miller, Perry L. ; Shepherd, Gordon M. ; Hines, Michael L.: Twenty years of ModelDB and beyond: building essential modeling tools for the future of neuroscience. In: *Journal of computational neuroscience* 42 (2017), Nr. 1, S. 1–10. http://dx.doi.org/10.1007/s10827-016-0623-7. – DOI 10.1007/s10827–016–0623–7

[34] Mertens, Peter ; Bodendorf, Freimut ; König, Wolfgang ; Schumann, Matthias ; Hess, Thomas ; Buxmann, Peter: *Grundzüge der Wirtschaftsinformatik*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2017. http://dx.doi.org/10.1007/978-3-662-53362-8. http://dx.doi.org/10.1007/978-3-662-53362-8. – ISBN 978–3–662–53361–1

[35] Mihalcea, Vlad ; Ebersole, Steve ; Boriero, Andrea ; Morling, Gunnar ; Badner, Gail ; Cranford, Chris ; Bernard, Emmanuel ; Grinovero, Sanne ; Meyer, Brett ; Ferentschik, Hardy ; King, Gavin ; Bauer, Christian ; Andersen, Max R. ; Maesen, Karel ; Vansa, Radim ; Jacomet, Louis: *Hibernate ORM 5.5.0.Final User Guide*. https://docs.jboss.org/hibernate/orm/5.5/userguide/html_single/Hibernate_User_Guide.html

[36] Nadkarni, P. M. ; Marenco, L. ; Chen, R. ; Skoufos, E. ; Shepherd, G. ; Miller, P.: Organization of heterogeneous scientific data using the EAV/CR representation. In: *Journal of the American Medical Informatics Association : JAMIA* 6 (1999), Nr. 6, S. 478–493. http://dx.doi.org/10.1136/jamia.1999.0060478. – DOI 10.1136/jamia.1999.0060478. – ISSN 1067–5027

[37] Nurseitov, Nurzhan ; Paulson, Michael ; Reynolds, Randall ; Izurieta, C.: Comparison of JSON and XML Data Interchange Formats: A Case Study. In: *CAINE* (2009). https://www.semanticscholar.org/paper/Comparison-of-JSON-and-XML-Data-Interchange-A-Case-Nurseitov-Paulson/84321e662b24363e032d680901627aa1bfd6088f

[38] O'Conner, John: *Using the Java Persistence API in Desktop Applications.* https://www.oracle.com/technical-resources/articles/javase/persistenceapi.html. Version: 2007

[39] Oracle Help Center: *JSON Developer's Guide: 1 JSON in Oracle Database.* https://docs.oracle.com/en/database/oracle/oracle-database/18/adjsn/json-in-oracle-database.html#GUID-A2BC5593-7F70-482E-89D1-B32C3798E0E4. Version: 2019

[40] Oracle Help Center: *Database Reference: A.3 Logical Database Limits.* https://docs.oracle.com/en/database/oracle/oracle-database/19/refrn/logical-database-limits.html. Version: 2021

[41] Pettus, Christophe: *PostgreSQL and JSON: 2015.* https://thebuild.com/presentations/json2015-pgconfus.pdf. Version: 2015

[42] Pokorny, Jaroslav: NoSQL databases: a step to database scalability in web environment. In: *International Journal of Web Information Systems* 9 (2013), Nr. 1, S. 69–82. http://dx.doi.org/10.1108/17440081311316398. – DOI 10.1108/17440081311316398. – ISSN 1744–0084

[43] PostgreSQL Documentation: *8.14. JSON Types.* https://www.postgresql.org/docs/13/datatype-json.html#JSON-INDEXING. Version: 2021

[44] PostgreSQL Documentation: *Appendix K. PostgreSQL Limits.* https://www.postgresql.org/docs/12/limits.html. Version: 2021

[45] Reese, George: *Database programming with JDBC and Java.* 2. ed. Beijing : O'Reilly, 2000 (Safari Books online). http://proquest.tech.safaribooksonline.de/1565926161. – ISBN 1565926161

[46] Smithers Mike: *The Anti-Pattern – EAV(il) Database Design ?* https://mikesmithers.wordpress.com/2013/12/22/the-anti-pattern-eavil-database-design/. Version: 2013

[47] Solovyov, Vsevolod: *Pitfalls of JSONB indexes in PostgreSQL.* https://vsevolod.net/postgresql-jsonb-index/. Version: 08.07.2021

[48] Stack Overflow: *json - How to use Oracle's JSON_VALUE function with a PreparedStatement - Stack Overflow.* https://stackoverflow.com/questions/56948001/how-to-use-oracles-json-value-function-with-a-preparedstatement. Version: 14.07.2021

[49] Stonebraker, Michael: New SQL: An Alternative to NoSQL and Old SQL For New OLTP Apps. In: *Communications of the ACM* (2011)

[50] Sulír, Matúš ; Porubän, Jaroslav: A quantitative study of Java software buildability. In: Anslow, Craig (Hrsg.): *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools.* New York, NY : ACM, 2016. – ISBN 9781450346382, S. 17–25

[51] Viotti, Paolo ; Vukolić, Marko: *Consistency in Non-Transactional Distributed Storage Systems.* http://arxiv.org/pdf/1512.00168v4

[52] Zehnder, Carl A.: *Informationssysteme und Datenbanken.* Wiesbaden : Vieweg+Teubner Verlag, 1985. http://dx.doi.org/10.1007/978-3-663-14082-5. http://dx.doi.org/10.1007/978-3-663-14082-5. – ISBN 978–3–519–02480–4