TU Clausthal

EINSATZ EINER GRAPHDATENBANK FÜR ANALYSEN UND SUCHFUNKTIONEN VON SAMMELKARTEN ANHAND VON MAGIC: THE GATHERING

BACHELOR THESIS

vorgelegt von

PASCAL KLEINDIENST

Abteilung für Databases and Information Systems
Institut für Informatik
Technische Universität Clausthal

ES-Mooo

[draft created on 10. Februar 2017 at 10:58 (delete `drafting` option in `thesis-config.tex` to remove this line)]

Pascal Kleindienst: *Einsatz einer Graphdatenbank für Analysen und Suchfunktionen von Sammelkarten anhand von Magic: the Gathering*

MATRIKELNUMMER
402592

GUTACHTER
Erstgutachter: Prof. Dr. Sven Hartmann
Zweitgutachter: Prof. Dr. Zweitgutachter

TAG DER EINREICHUNG
30. März 2016

[draft created on 10. Februar 2017 at 10:58 (delete `drafting` option in `thesis-config.tex` to remove this line)]

EIDESSTATTLICHE VERSICHERUNG

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, wurden als solche kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsstelle im Sinne von §11 Absatz 5 lit. b) der Allgemeinen Prüfungsordnung vorgelegen.

Hiermit erkläre ich mich zudem damit einverstanden, dass meine Bachelor Thesis in der Instituts- und/oder der Universitätsbibliothek ausgelegt und zur Einsichtnahme aufbewahrt werden darf.

Clausthal-Zellerfeld, den 30. März 2016

Pascal Kleindienst

[draft created on 10. Februar 2017 at 10:58 (delete `drafting` option in `thesis-config.tex` to remove this line)]

[draft created on 10. Februar 2017 at 10:58 (delete drafting option in thesis-config.tex to remove this line)]

ABSTRACT

It is often difficult to quickly analyze cards, decks or tournaments in relational databases such as MySQL. The database queries get complex rather quickly and contain many JOIN commands, which adversely affects scalability. The many JOIN commands result from the fact that cards and decks have many attributes and therefore many relationships. Particularly larger tournaments contain many games, which increases the size of the data set fast and thereby having a negative effect on the runtime due to the bad scaling. Therefore the solution is the use of graph databases as these can represent such data superb and work with many relationships. Through the use of a graph database, tournaments can be analyzed well, but it is not quite as good for text searches. It has also been shown that searching in arrays in Neo4j is slower than joining and looking for data with the JOIN command.

ZUSAMMENFASSUNG

Es ist oftmals schwer Karten, Decks oder Turniere in relationalen Datenbanken wie MySQL schnell zu analysieren oder zu durchsuchen. Die Datenbankabfragen werden schnell kompliziert und enthalten viele JOIN-Befehle, was sich negativ auf die Skalierbarkeit auswirkt. Die vielen JOIN-Befehle resultieren daraus, dass Karten und Decks viele Attribute und damit Verknüpfungen haben. Besonders größere Turniere enthalten viele Spiele wodurch der Datensatz schnell steigt, was sich aufgrund der schlechten Skalierung negativ auf die Laufzeit auswirkt. Als Lösung bietet sich daher die Verwendung von Graph-Datenbanken an, da diese Daten mit vielen Verknüpfungen gut darstellen und darauf arbeiten können. Durch den Einsatz einer Graph-Datenbank lassen sich Turniere gut analysieren, aber für Textsuchen eignen sie sich nicht ganz so gut. Außerdem hat sich gezeigt, dass das Suchen in Arrays in Neo4j langsamer ist als die Daten per JOIN einzubinden und darauf zu suchen.

[draft created on 10. Februar 2017 at 10:58 (delete `drafting` option in `thesis-config.tex` to remove this line)]

[draft created on 10. Februar 2017 at 10:58 (delete drafting option in thesis-config.tex to remove this line)]

INHALTSVERZEICHNIS

[draft created on 10. Februar 2017 at 10:58 (delete `drafting` option in `thesis-config.tex` to remove this line)]

ABBILDUNGSVERZEICHNIS

ABKÜRZUNGSVERZEICHNIS

UML Unified Modeling Language

ER-Modell Entity-Relationship-Modell

OOP objektorientierte Programmierung

RDBMS relationales Datenbankmanagementsystem

MtG Magic: the Gathering

TCG Trading Card Game

[draft created on 10. Februar 2017 at 10:58 (delete drafting option in thesis-config.tex to remove this line)]

CRUD Create Read Update Delete

NoSQL Not Only SQL

JSON JavaScript Object Notation

CSV Comma-separated values

JIT Just-in-time

1 EINLEITUNG

Sammelkartenspiele auch Trading Card Game (TCG) genannt sind Spiele bei denen es meist mehrere hunderte Karten verschiedener Seltenheitsstufen gibt. Man kann sich aus den Karten eine Auswahl zusammenstellen und mit anderen Spielern gegeneinander spielen. Einige der bekanntesten Vertreter von Sammelkartenspielen sind etwa *Magic: the Gathering*, *Pokemon*, *Yu-Gi-Oh!* und *Hearthstone: Heroes of Warcraft*.

Die Anzahl an Karten eines TCG wie Magic: the Gathering steigt von Jahr zu Jahr an. Bei Magic: the Gathering erscheinen pro Jahr zwei bis vier neue Sets mit durchschnittlich zwischen 200 bis 400 Karten. Die Menge der Karten nimmt also jedes Jahr beständig zu *(aktuell gibt es >15.000 verschiedene Karten und insgesamt >25.000 Karten verschiedener Versionen)*[1]. Mit `http://mtgjson.com` gibt es einen open-source Datensatz aller Kartendaten, welcher unter anderem auch in [6, 15] benutzt wird, so dass dieser auch hier zum Einsatz kommen soll.

Online-Shops benötigen oft nur einen Bruchteil der Informationen, die sich auf einer Karte befinden und speichern diese tabellarisch, d.h. in einer relationalen Datenbank [10]. Gerade für kleinere Online-Shops eignet sich dieser Ansatz, da hierbei der Wartungsaufwand gering gehalten wird [10]. Karten-Suchmaschinen erfordern aber hoch strukturierte Daten, da alle Informationen einer Karte effizient gespeichert, verwaltet und Suchanfragen in Echtzeit beantwortet werden müssen.

Ein weiteres Einsatzgebiet ist das verwalten und analysieren von Decks und Turnieren. Turniere werden nach dem Schweizer-System, einer Sonderform des Rundenturniers, gespielt, wodurch sich ein Netzwerk aus Paarungen der verschiedenen Runden ergibt. Aus diesem Grund werden Ergebnisse von Turnieren oftmals der Einfachheit halber nur als Rangfolge gespeichert und nicht jedes Rundendergebnis. Mit letzterem Ansatz ließen sich verschiedene Analysen über die Decks und Spieler, wie zum Beispiel Matchups, das heißt welche Gewinn- oder Verlustchancen ein Deck gegen ein anderes Deck hat, erstellen.

1 Basierend auf http://mtgjson.com

[draft created on 10. Februar 2017 at 10:58 (delete `drafting` option in `thesis-config.tex` to remove this line)]

[draft created on 10. Februar 2017 at 10:58 (delete drafting option in thesis-config.tex to remove this line)]

2 GRUNDLAGEN

Magic: the Gathering (MtG) ist ein strategisches TCG, bei dem zwei oder mehr Spieler mit einem individuellen Deck von Magic-Karten gegeneinander spielen. Im Laufe des Spiels werden verschiedene Karten, wie Länder, Kreaturen, Hexereien und andere Zauber eingesetzt, um das Spiel zu gewinnen. [13]

2.1 AUFBAU EINER KARTE

Die einfachste Möglichkeit Karten zu gruppieren ist anhand ihrer Farbe. Es gibt 5 verschiedene Farben in MtG: *weiß, blau, schwarz, grün* und *rot*. Außerdem gibt es auch Karten die keiner Farbe angehören und daher als *farblos* bezeichnet werden oder aber mehr als einer Farbe angehören, zum Beispiel gibt es Karten die sowohl *rot* als auch *grün* sind. [13]

2.1.1 *Bestandteile einer Karte*

KARTENNAME Jede Karte hat einen eindeutigen Namen.

TYP Jede Karte hat einen bestimmten Typ. Außerdem kann eine Karte zusätzlich noch einen *Subtyp/Untertyp* oder *Supertyp* haben. Zum Beispiel: Der Shivan Dragon in Abbildung 2.1 hat den Typ *Kreatur* und als Untertyp *Drache*. [13]

REGELTEXT Enthält die *Fähigkeiten* einer Karte, sofern vorhanden.

FLAVOR-TEXT Enthält Informationen über den Welt von MtG und hat keinerlei Auswirkungen auf das Spielgeschehen.

SAMMLERNUMMER Hilft dabei die Karten einer Edition zu sortieren/organisieren.

KÜNSTLER Der Künstler von dem das Bild für die Karte stammt.

STÄRKE UND WIDERSTANDSKRAFT Jede *Kreatur* hat einen Wert für Stärke und Widerstandskraft. Die erste Zahl gibt die Stärke an und die zweite die Widerstandskraft.

EDITIONS-SYMBOL Gibt Auskunft über die Edition aus der die Karte stammt. Die Farbe des Symbols verrät die Seltenheit: schwarz für häufige, silbern für nicht ganz so häufige, golden für seltene und rot-orange für sagenhafte Karten. [13]

[draft created on 10. Februar 2017 at 10:58 (delete drafting option in thesis-config.tex to remove this line)]

Abbildung 2.1: Bestandteile einer Karten

MANAKOSTEN Die Hauptressource in MtG ist *Mana*, welches von *Ländern* produziert und für das spielen von *Zaubern* benötigt wird. [13]

2.1.2 *Kartentypen und Fähigkeiten*

Jede Karte in MtG hat einen oder mehrere von den insgesamt sechs Typen *(Land, Artefakt, Kreatur, Verzauberung, Planeswalker, Spontanzauber und Hexerei)*. An dem Kartentyp erkennt man wie sich die Karte verhält, das heißt wann man sie spielen kann und was danach mit ihr geschieht. [13]

Viele Karten haben Texte, die sich auf das Spielgeschehen auswirken können, sogenannte Fähigkeiten. Aufgrund der großen Anzahl an Karten und der damit verbunden hohen Anzahl an verschiedenen Fähigkeiten werden diese in *statische*, *ausgelöste* und *aktivierte Fähigkeiten* unterteilt. [13]

STATISCHE FÄHIGKEITEN heißen so, da sie, solange sich die Karte im Spiel befindet, sich nicht verändern und konstant aktiv ist.

AUSGELÖSTE FÄHIGKEITEN sind Fähigkeiten, die durch ein bestimmtes Ereignis ausgelöst werden. Es gibt mehrere Formulierungen anhand derer man diese Art identifizieren kann. Die häufigsten sind *wenn*, *immer wenn* und *zu*.

AKTIVIERTE FÄHIGKEITEN erkennt man am Doppelpunkt. Sie werden in der Form `[Kosten] : [Effekt]` angegeben, das heißt man muss etwas einsetzen, um die Fähigkeit zu aktivieren.

SCHLÜSSELWORTE Fähigkeiten, die auf ein einzelnes Wort oder Satz gekürzt sind und eventuell einen Erinnerungstext haben, heißen *Schlüsselwort-Fähigkeiten*. Meistens handelt es sich um *statische Fähigkeiten*, aber *ausgelöste* und *aktivierte Fähigkeiten* sind auch möglich.

2.2 AUFBAU EINES DECKS

Ein Deck muss aus mindestens 60 Karten bestehen, wobei nach oben hin keine Grenzen gesetzt sind. Aus strategischer Sicht ist es aber sinnvoll möglichst nicht mehr als 60 Karten zu verwenden, da sonst die Wahrscheinlichkeit sinkt, die passende Karte zu ziehen. Des Weiteren ist es verboten mehr als vier Exemplare einer Karte im Deck zu haben, mit Ausnahme von Standardland-Karten, welche beliebig oft im Deck sein dürfen. Daraus ergeben sich folgende Richtlinien: [13]

LÄNDER Ein Deck sollte zwischen 35% und 40% aus Ländern bestehen

KREATUREN Ein Deck sollte zwischen 25% und 40% aus Kreaturen bestehen. Dabei sollte beachtet werden, dass die Mana-Kosten gut verteilt sind

SONSTIGES Alles was nicht Land oder Kreatur ist und das Deck unterstützt

Turniere bieten eine Besonderheit, da es hier verschiedene Formate gibt. Bei manchen sind fast alle Karten erlaubt wohingegen bei anderen Formaten nur Karte der letzten paar Jahre erlaubt sind [3].

DIE GOLDENE REGEL Falls eine MtG Karte dem Regelbuch widerspricht, hat die Karte Vorrang. [13]

2.2.1 *Deck-Typen*

Durch die große Menge an Karten und die goldene Regel ergeben sich viele verschiedene Spielweisen und Strategien nach denen man sein Deck ausrichten kann. Dadurch entstehen einige Decktypen, die die oben genannten Richtlinien für ihre Strategie anpassen. Decks die eine ähnliche Strategie haben oder gleich aufgebaut sind, werden unter einem Deck-Typ zusammengefasst [8]. Besonders starke Deck-Typen entstehen häufig durch die Analyse der einzelnen Karten und den Ergebnissen eines Decks in Turnieren [8]. Ein wichtiger Begriff ist hierbei die sogenannte Manakurve. Sie gibt an wie ausgewogen die Mana-Kosten in einem Deck sind - eine "hohe Manakurve"bedeutet, dass das Deck viele teurer Karten hat, wohingegen eine "niedrige Manakurve"bedeutet, dass viele günstige Karten enthalten sind.

AGGRO Bei der Aggro-Strategie geht es darum den Gegner möglichst früh mit den eigenen Kreaturen zu überrennen. Es wird durch die eigenen Kreaturen möglichst früh Druck auf den Gegener ausgeübt, um ihn so in die Verteidigung zu bringen. Entscheidend für die Strategie sind eine geringe Manakurve und viele kleine Kreaturen. [3]

CONTROL Die Control-Strategie ist das genaue Gegenteil der Aggro-Strategie. Ziel ist es den Gegner solange in seiner Strategie zu stören, bis man im späten Spielverlauf die Partie mit eigenen wenigen starken Kreaturen das Spiel dominieren kann. Kontrolldecks setzten in erster Linie auf defensive Zauberkarten und wenige teure Kreaturen. [3]

MIDRANGE Midrange-Decks sind eine Kombination aus Aggro- und Kontrolldecks. Sie sind sehr effizient und können sich an die Strategie des Gegners anpassen. Anstatt sich auf eine Strategie festzulegen wechseln Midrange-Decks je nach Situation zwischen aggressiver und defensiver Spielweise. [3]

KOMBO Kombo-Decks nutzen Synergien die aus dem Zusammenspiel mehrerer Karten ergeben. Die Decks sind komplett darauf ausgerichtet, diese Synergien noch weiter zu unterstützen und dadurch die Oberhand in der Partie zu gewinnen. Aufgrund der Vielzahl an Karten gibt es unendlich viele Karten-Kombinationen. [3]

2.3 TURNIERE

Turniere werden ähnlich wie beim Schach nach dem Schweizer System gespielt. Dies ist eine Variante des Rundensystems bei der die Teilnehmer jede Runde spielen und die Paarungen sich anhand der bisherigen Spiel-Ergebnisse ergeben. Dadurch wird sichergestellt, dass Spieler mit ähnlichen Ergebnissen gegeneinander antreten [18]. Wie in Tabelle 2.1 zu sehen, ist die Anzahl der gespielten Runden nach der Anzahl der Teilnehmer gestaffelt.

Eine Runde wird nach dem *Best-of-Three* Modus gespielt, das heißt die Runde ist beendet, sobald ein Spieler 2 Spiele gewonnen hat.

Tabelle 2.1: Anzahl der Runden die ab einer Teilnehmerzahl gespielt werden [4]

Spieler	Runden	Spieler	Runden
2	1	129-212	8
3-4	2	213-384	9
5-8	3	385-672	10
9-16	4	673-1248	11
17-32	5	1249-2272	12
33-64	6	2273+	13
64-128	7		

2.3.1 *Match-Up*

Match-Up ist ein Begriff, welcher aus der Deckanalyse stammt und die Gewinn- und Verlustchancen eines Decks gegen ein anderes beschreibt. Der Wert wird oft in Prozentangaben wie 60:40 angegeben oder auch einfach nur als schlechtes oder gutes Match-Up.

2.4 GRAPH-DATENBANKEN

Ein Graph ist eine Sammlung von Objekten *(Knoten)* und deren Verbindungen *(Kanten)*. Graphen lassen sich vielfältig einsetzen, vom Straßennetz bis zur Krankengeschichte von Populationen [16]. In Abbildung 2.2 sieht man eine einfache Unternehmenshierarchie als Graph dargestellt.

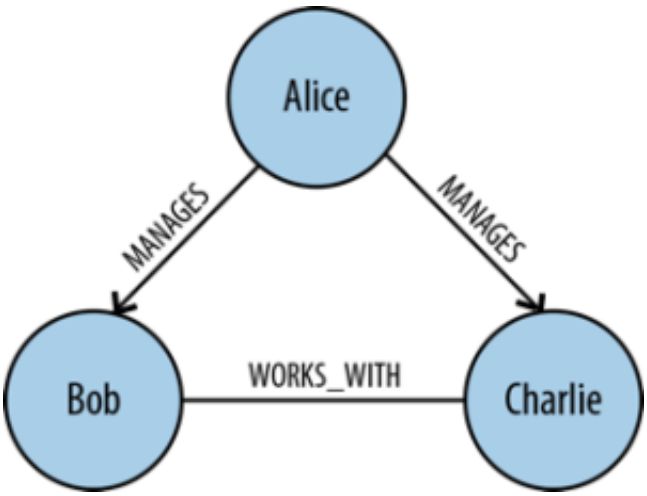

Abbildung 2.2: Einfacher Graph [2]

Ein *Graph Datenbank Management System* (kurz *Graph-Datenbank*) ist ein Datenbanken-Management-System, welches die Create Read Update Delete (CRUD) Eigenschaften unterstützt und ein Graph Daten-Modell bereitstellt [16].

Beziehungen in Graph-Datenbanken sind, im Gegensatz zu anderen Datenbank-Systemen, sogenannte *first class citizens*. Dies bedeutet, dass Objekte mit einander verbunden werden können ohne dazu Fremdschlüssel oder ähnliche Konstrukte zu verwenden. Dadurch sind die Datenmodelle einfacher und ausdrucksvoller als Modelle von relationalen oder anderen Not Only SQL (NoSQL) Datenbanken. [16]

2.4.1 *Stärken von Graph-Datenbanken*

Aufgrund der Tatsache, dass Abfragen nur auf einem Teil des Graphen arbeiten, bleibt die Performance einer Graph-Datenbank relativ konstant auch wenn der Datensatz wächst. Bei relationalen Datenbank Abfragen mit vielen

joins hingegen verschlechtert sich die Abfrageperformance bei zunehmenden Datensatz. [16]

Eine weitere Stärke von Graph-Datenbanken ist ihr flexibles Datenmodell. Es können neue Beziehungen, Knoten oder Untergraphen zu bestehenden Strukturen hinzugefügt werden ohne, dass es dabei zu Konflikten mit vorhandene Abfragen kommt. Dies bedeutet, dass das Datenmodell nicht ausführlich mit allen Details vor Projektbeginn vorhanden sein muss, sondern an geänderte Anforderungen im laufe des Projekts angepasst werden kann. [16]

[draft created on 10. Februar 2017 at 10:58 (delete drafting option in thesis-config.tex to remove this line)]

3 METHODE

3.1 ANALYSE DER ANFORDERUNGEN

3.1.1 *Kartensuche*

Aufgrund der großen Anzahl an Karten *(>15.000 [17])* ist eine Suchfunktion hervorragend geeignet, um bestimmte oder ähnliche Karten zu finden. Wie in 2.1 beschrieben, enthält eine Karte viele verschiedenen Attribute. Daher ist es sinnvoll bei einer Suchfunktion die *Verknüpfung dieser Attribute* zu erlauben, sodass zum Beispiel nach allen Karten eines Künstlers, die in einem bestimmten Set erschienen sind, gesucht werden kann.

Da eine solche Suchfunktion oftmals webbasiert ist, muss es möglich sein, dass ein Benutzer die Ergebnisse in Echtzeit erhält.

3.1.2 *Deckbau*

Wie in 2.2 zu sehen ist, gibt es verschiedene Richtlinien die beim Deckbau zu beachten sind, sofern man ein gutes Deck erstellen möchte. Diese Richtlinien können sich jedoch abhängig vom Deck-Typen stark unterscheiden. Ein hilfreiches Werkzeug beim Deckbau ist daher die Deck-Analyse.

Die Berechnung der Kostenverteilung, das heißt die Manakurve, hilft ... Auch die Verteilung der einzelnen Kartentypen, das heißt die Anzahl der Kreaturen, Zauber, usw., ist für verschiedene Deck-Typen wichtig. So ist für ein Aggro-Deck eine hohe Anzahl an Kreaturen wichtig, wohingegen ein Control-Deck eher eine hohe Anzahl an Zaubern enthält. Da sich viele Decks aber nicht nur auf eine Farbe beschränken, ist es auch wichtig zu wissen, wie die Farben im Deck verteilt sind, um so die Manaquellen im Deck entsprechend zu verteilen.

Beim Deckbau werden in der Regel nur die Anzahl der Karte im Deck und ihr Name angegeben. Aus diesem Grund ist es wichtig den Inhalt des Decks mit den Karten-Daten zu verknüpfen, um Zugriff auf die benötigten Attribute zu haben. Außerdem sollte auch hier die Analyse die Ergebnisse in Echtzeit liefern, damit sich die Funktion für webbasierte Anwendungen eignet.

3.1.3 *Turniere*

Sowohl für professionelle Turnierspieler als auch für Amateure ist die Analyse vergangener Turniere wichtig, um ihre Decks bestmöglich an potentiell überlegende Decks anzupassen. Dazu ist die Berechnung des Match-Ups für

[draft created on 10. Februar 2017 at 10:58 (delete drafting option in thesis-config.tex to remove this line)]

ein Deck-Typ wichtig. Dazu müssen die einzelnen Ergebnisse einer Runde eines Turniers mit den Decks verknüpft werden.

Ein weitere interessante Information ist die des erfolgreichsten Spielers oder des erfolgreichsten Deck(-Typen) über eine Auswahl an Turnieren.

Wie in den vorherigen Fällen ist auch hier eine webbasierte Anwendung wünschenswert und damit die Berechnung der Resultate in Echtzeit.

3.2 POTENTIELLE ANSÄTZE UND PROBLEME

3.2.1 *Relationale Datenbanken*

Ein Ansatz, welcher sich für kleine Online-Shops eignet, ist die Karten-Daten in einer relationalen Datenbank zu speichern [10]. Da ein Online-Shop nur einen Teil der Attribute einer Karte, wie Name, Seltenheit und Set, benötigt, kann das Datenbank Schema aus ein bis zwei Tabellen bestehen, die diese Informationen speichern. Für eine komplexe Suchfunktion, die alle Attribute einer Karte berücksichtigt, eignet sich dieser Ansatz allerdings nicht, da hier viele Tabellen benötigt werden, um die Daten zu speichern. Bei Daten mit vielen Beziehungen untereinander sorgen Relationale Datenbanken für komplexe Abfragen mit vielen JOINS, da diese hoch-strukturierte Daten nicht unterstützen [16]. In 1 ist beispielhaft eine komplexe Suchanfrage mit 7 benötigten JOINS angegeben.

```
1  SELECT * FROM `translations` AS `t`
2  JOIN `cards` ON `cards`.`id` = `t`.`card_id`
3  JOIN `cards_in_set` ON `cards_in_set`.`card_id` = `t`.`card_id`
4  JOIN `types` ON `cards`.`type_id` = `types`.`id`
5  JOIN `card_colors` ON `cards`.`id` = `card_colors`.`card_id`
6  JOIN `card_has_ability` ON `cards`.`id` = `card_has_ability`.`card_id`
7  JOIN `keyword_abilities` ON `card_has_ability`.`ability_id` = `
       keyword_abilities`.`id`
8  JOIN `sets` ON `sets`.`id` = `cards_in_set`.`set_id`
9  WHERE
10     `sets`.release > "2001-01-01" AND
11     `cards_in_set`.`rarity` = "RARE" AND
12     `t`.`lang` = "German" AND
13     `card_colors`.color = "BLUE" AND
14     `types`.`type` = "Creature" AND
15     `keyword_abilities`.`name` = "Flying" AND
16     `cards`.`cmc` <= 5
```

Listing 1: Komplexe SQL-Abfrage

3.2.2 *NoSQL Datenbanken*

NoSQL Datenbanken wie Key-Value-, Document- oder Column-oriented Stores eignen sich nicht für Daten mit vielen Verknüpfungen, da sie die Daten

nicht verbunden speichern. Eine Ausnahme sind die Graph-Datenbanken, da sie die Daten als Graph speichern, unterstützen sie daher Daten mit vielen Beziehungen untereinander von vornherein. [16]

[9] [20]

3.3 SOFTWARE-DESIGN

3.3.1 *Daten-Schemata*

Um das Daten-Schema für die relationale Datenbank darzustellen, wird Unified Modeling Language (UML) verwendet. Die Vorteile von UML liegen im Gegensatz zum Entity-Relationship-Modell (ER-Modell) darin, dass es weit verbreitet, standardisiert ist und sich gut für objektorientierte Programmierung (OOP) eignet [19]. In Abbildung 3.1 befindet sich das Daten-Schema für die relationale Datenbank.

Wie viele Graph-Datenbanken nutzt auch Neo4j das Property-Graph-Modell um Daten darzustellen. Dieses Modell ist ein Untertyp des mathematischen Graph-Modells. Property-Graph-Modelle sind einfacher, aussagekräftiger und geben Beziehungen explizit an. relationales Datenbankmanagementsystem (RDBMS) verwenden Fremdschlüssel, um Beziehungen implizit anzugeben [11]. In Abbildung 3.2 befindet sich ein möglicher Ansatz für ein Schema.

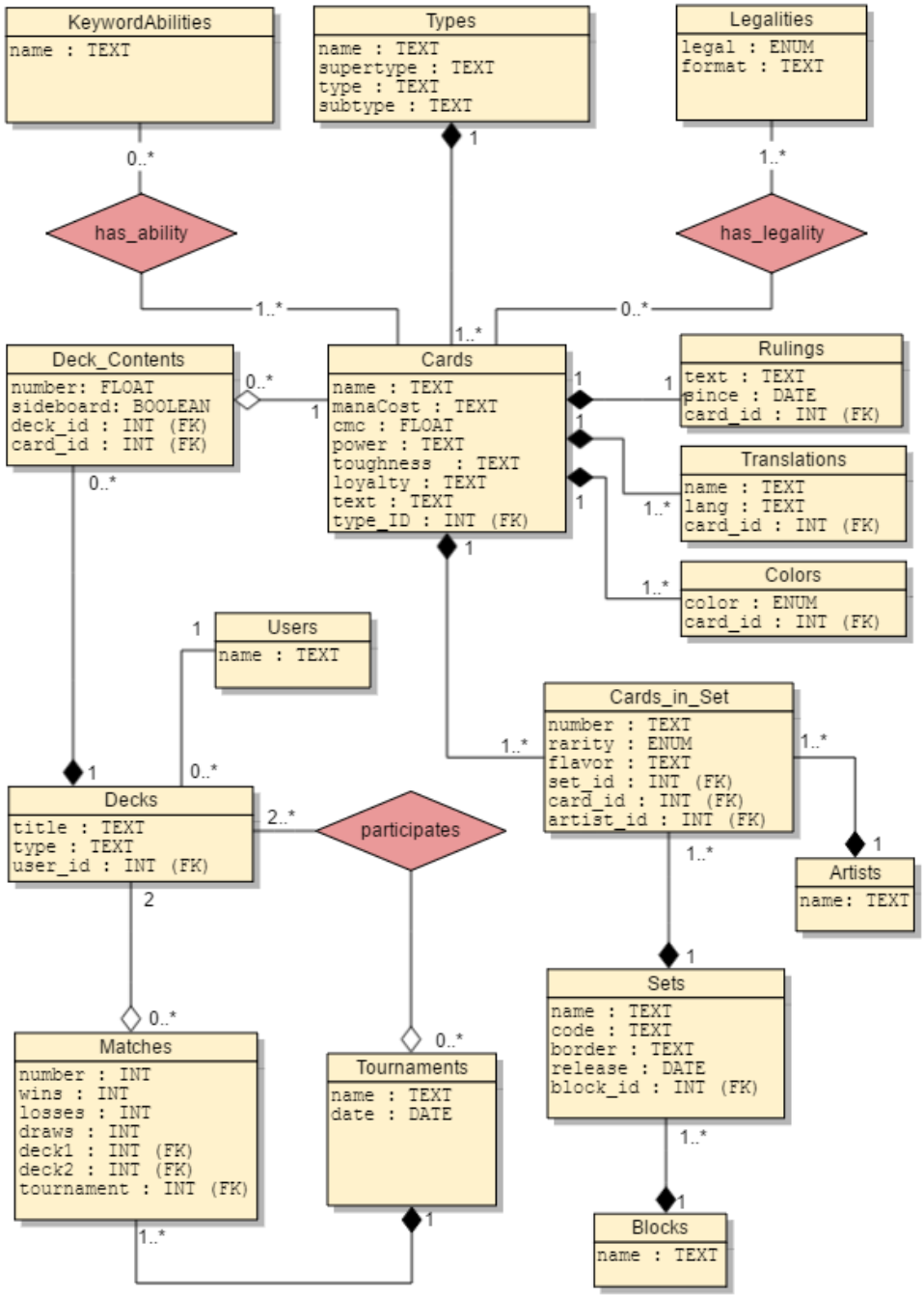

Abbildung 3.1: Schema für relationale Datenbank

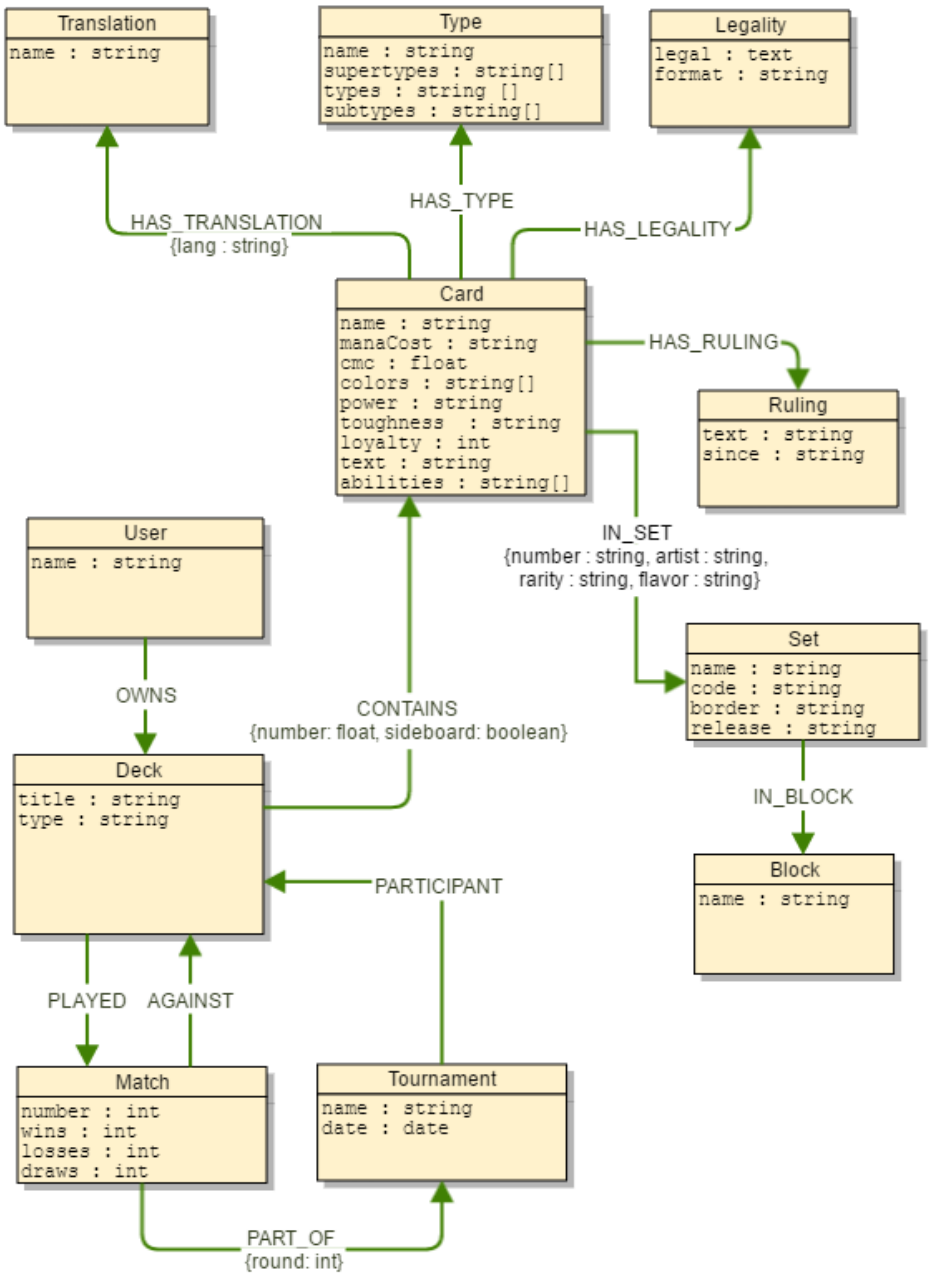

Abbildung 3.2: Schema für Graph-Datenbank

3.3.2 *Beschreibung der Software-Komponenten*

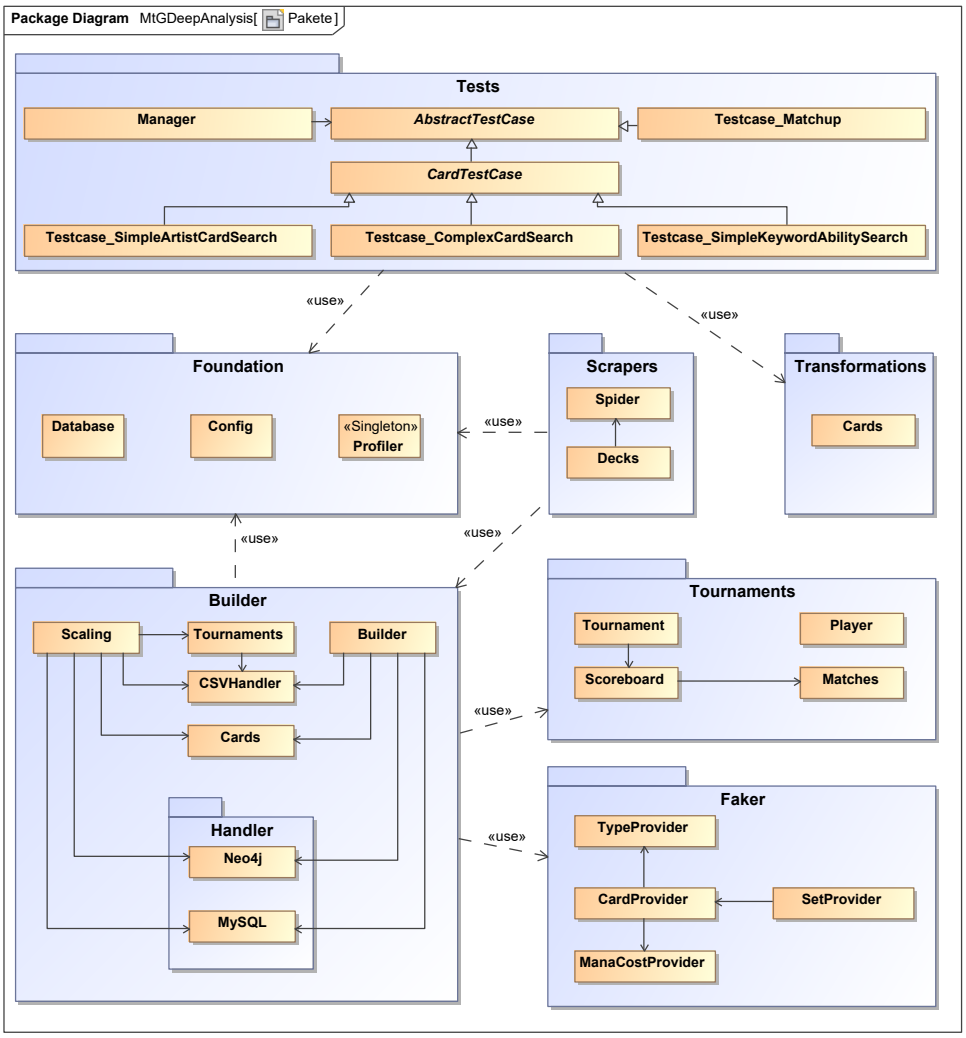

Abbildung 3.3: Software-Komponenten

BUILDER

Das Package Builder ist zuständig für das erstellen und befüllen der Datenbanken.

BUILDER.HANDLERS

Das Package Builder.Handlers enthält die Implementierungen für die konkreten Datenbanken

SCRAPERS

Das Package Scrapers lädt Decks von mtgtop8 [1] herunter, um einen Datensatz für Decks und Turniere zu erstellen

1 mtgtop8 - http://mtgtop8.com/

TESTS
: Enthält die konkreten Testfälle

TRANSFORMERS
: Das Package Transformers ist zuständig für das Transformieren von Daten in eine vorgegebene Datenstruktur.

3.3.3 *Beschreibung der Schnittstellen*

Builder

Die Komponente `Builder` nutzt die folgenden Schnittstellen:

FOUNDATION wird genutzt um ein Datenbank-Verbindungen zu erstellen

Scrapers

Die Komponente `Scrapers` nutzt die folgenden Schnittstellen:

FOUNDATION wird genutzt um auf die Konfiguration zuzugreifen

BUILDER wird genutzt um die Decks in Comma-separated values (CSV) Dateien zu speichern

Tests

Die Komponente `Tests` nutzt die folgenden Schnittstellen:

FOUNDATION wird genutzt um ein Datenbank-Verbindungen zu erstellen und mit dem `Profiler` die Testfälle zu überwachen

TRANSFORMATIONS um die ausgelesenen Daten in eine passende Datenstruktur zu überführen

3.4 IMPLEMENTIERUNG

3.4.1 *Klassendiagramme*

3.4.1.1 *Builder.Builder*

Die Klasse `Builder.Builder` hat die folgenden Schnittstellen:

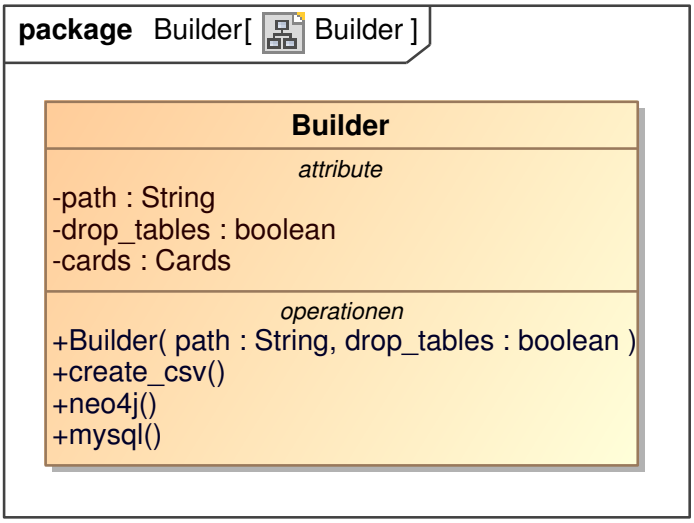

Abbildung 3.4: Klassendiagramm Builder.Builder

CONSTRUCTOR(PATH, DROP_TABLES)
: Einrichten der Datenstruktur `Cards` und speichern der übergebenen Argumente. Die Liste der Argumente befindet sich in Tabelle 3.1

CREATE_CSV()
: Lädt die Kartendaten aus der JavaScript Object Notation (JSON)-Datei und speichert die aufbereiteten Ergebnisse in CSV-Dateien

NEO4J()
: Erstellt und befüllt die Neo4j Datenbank mit den Daten aus den CSV-Dateien.

MYSQL()
: Erstellt und befüllt die MySQL Datenbank mit den Daten aus den CSV-Dateien.

Tabelle 3.1: Builder.Builder::constructor(path : string, drop_tables : boolean)

EINGABE	BESCHREIBUNG
`path : string`	Pfad zu der JSON-Datei, welche die Karten-Daten enthält
`drop_tables : boolean`	Angabe ob vor dem Einfügen der Daten in die Datenbank, alte Daten gelöscht werden sollen

3.4.1.2 *Builder.Cards*

Die Klasse `Builder.Cards` hat die folgenden Schnittstellen:

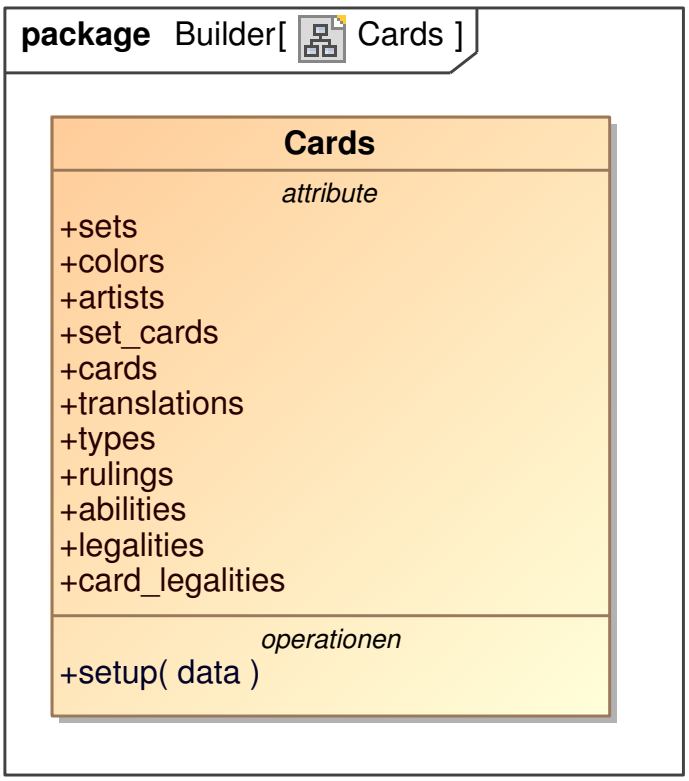

Abbildung 3.5: Klassendiagramm Builder.Cards

CONSTRUCTOR()
: Erstelle Listen und Daten-Container.

SETUP(DATA)
: Bearbeitet Kartendaten aus JSON-Datei so, dass diese als JSON-Dateien gespeichert werden können für den späteren Datenbank-Import. Die Liste der Argumente befindet sich in Tabelle 3.2

Tabelle 3.2: Builder.Cards::setup(data : Dictionary[])

EINGABE	BESCHREIBUNG
`data : Dictionary[]`	Liste mit allen Kartendaten die aufbereitet werden sollen

3.4.1.3 *Builder.CSVHandler*

Die Klasse `Builder.CSVHandler` hat die folgenden Schnittstellen:

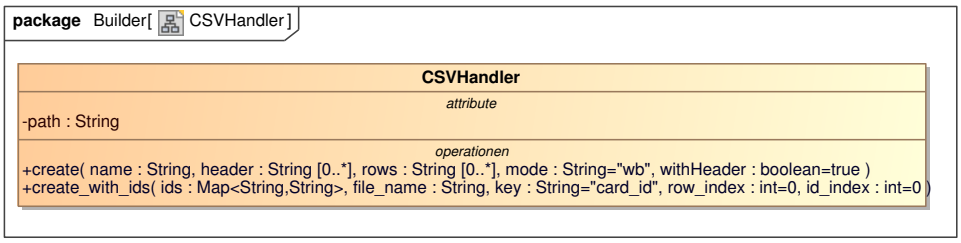

Abbildung 3.6: Klassendiagramm Builder.CSVHandler

3.4.1.4 *Builder.Tournaments*

Die Klasse `Builder.Tournaments` hat die folgenden Schnittstellen:

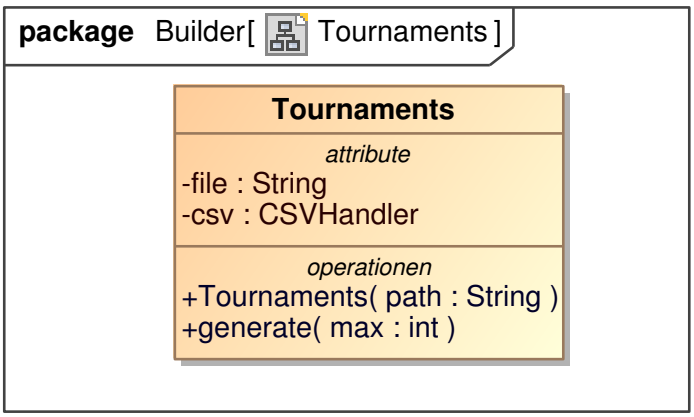

Abbildung 3.7: Klassendiagramm Builder.Tournaments

3.4.1.5 *Builder.Handlers.MySQLBuilder*

Die Klasse `Builder.Handlers.MySQLBuilder` hat die folgenden Schnittstellen:

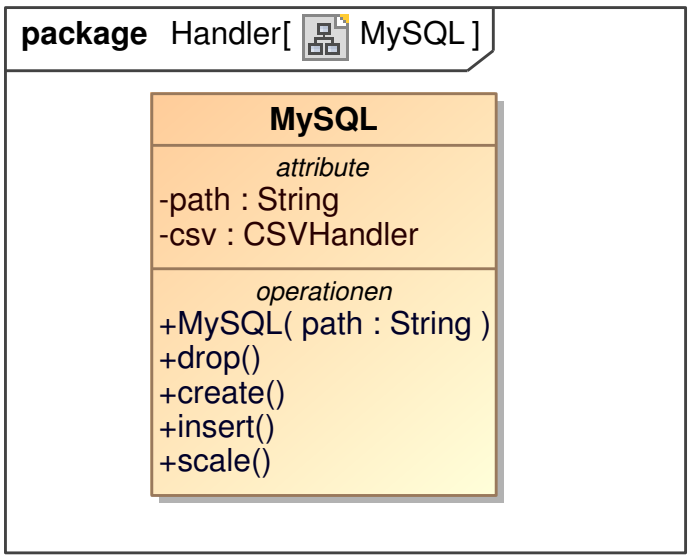

Abbildung 3.8: Klassendiagramm Builder.Handlers.MySQLBuilder

CONSTRUCTOR(PATH)
: Speichern der übergebenen Argumente. Die Liste der Argumente befindet sich in Tabelle 3.3

DROP()
: Löscht Tabellen.

CREATE()
: Erstellt Tabellen.

INSERT()
: Importiert Daten aus den CSV-Dateien.

Tabelle 3.3: Builder.Handlers.MySQLBuilder.constructor::setup(path : string)

EINGABE	BESCHREIBUNG
`path : string`	Pfad zu dem Verzeichnis welches die JSON und CSV Dateien enthält

3.4.1.6 *Builder.Handlers.Neo4jBuilder*

Die Klasse `Builder.Handlers.Neo4jBuilder` hat die folgenden Schnittstellen:

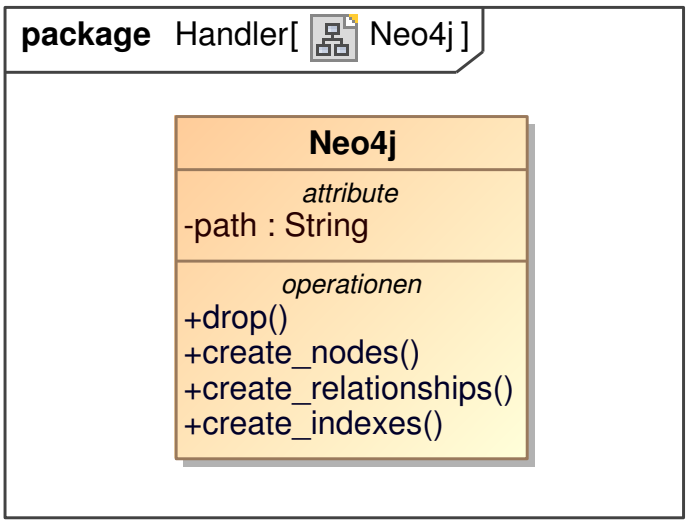

Abbildung 3.9: Klassendiagramm Builder.Handlers.Neo4jBuilder

CONSTRUCTOR(PATH)
: Speichern der übergebenen Argumente. Die Liste der Argumente befindet sich in Tabelle 3.4

DROP()
: Löscht Datenbank.

CREATE_NODES()
: Erstellt alle Knoten mit Daten aus den CSV-Dateien.

CREATE_INDEXES()
: Erstellt Indizes.

CREATE_RELATIONSHIPS()
: Erstellen Beziehungen zwischen Daten aus den CSV-Dateien.

Tabelle 3.4: Builder.Handlers.Neo4jBuilder.constructor::setup(path : string)

EINGABE	BESCHREIBUNG
`path : string`	Pfad zu dem Verzeichnis welches die JSON und CSV Dateien enthält

3.4.1.7 *Foundation.Config*

Die Klasse `Foundation.Config` hat die folgenden Schnittstellen:

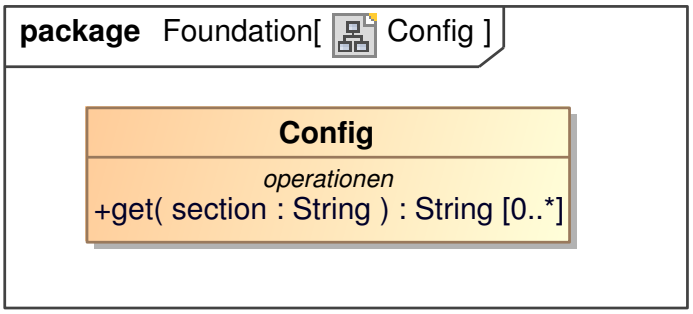

Abbildung 3.10: Klassendiagramm Foundation.Config

GET(SECTION)
: Gibt einen Konfigurationsabschnitt aus der Datei `config.ini` zurück. Die Liste der Argumente befindet sich in Tabelle 3.5

Tabelle 3.5: Foundation.Config::get(section : string) : string[]

EINGABE	BESCHREIBUNG
`section : string`	Name des Abschnitts der geladen werden soll

AUSGABE	BESCHREIBUNG
`string[]`	Konfigurationswerte des Abschnitts

3.4.1.8 *Foundation.Database*

Die Klasse `Foundation.Database` hat die folgenden Schnittstellen:

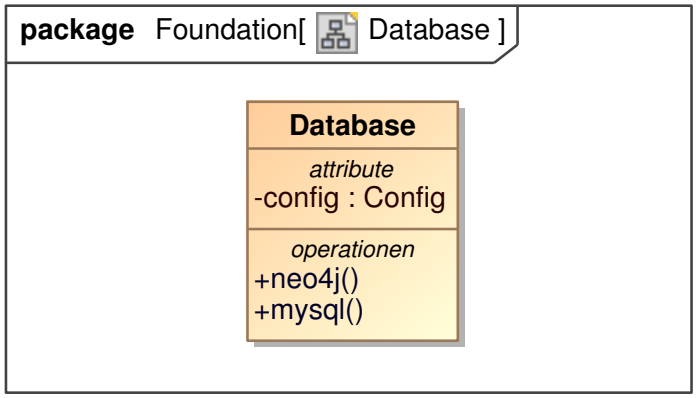

Abbildung 3.11: Klassendiagramm Foundation.Database

CONSTRUCTOR()
: Erstellt eine neue `Foundation.Config` Instanz und speichert diese in `config`.

NEO4J()
: Gibt eine neue Neo4j Datenbank-Instanz zurück.

MYSQL()
: Gibt eine neue Mysql Datenbank-Instanz zurück.

3.4.1.9 *Foundation.Profiler*

Die Klasse `Foundation.Profiler` hat die folgenden Schnittstellen:

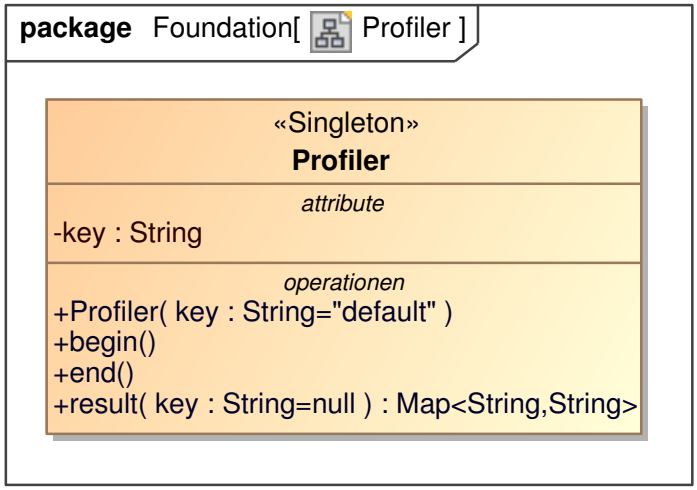

Abbildung 3.12: Klassendiagramm Foundation.Profiler

3.4.1.10 *Transformations.Cards*

Die Klasse `Transformations.Cards` hat die folgenden Schnittstellen:

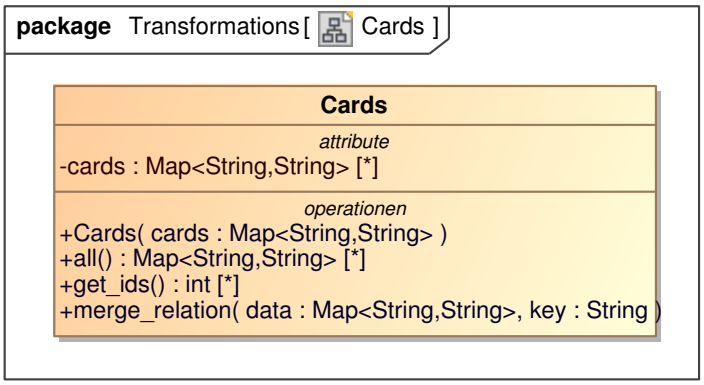

Abbildung 3.13: Klassendiagramm Transformations.Cards

CONSTRUCTOR(CARDS)
: Speichert Karten in `cards`. Die Liste der Argumente befindet sich in Tabelle 3.6

ALL()
: Ausgabe aller Karten

GET_IDS()
: Gibt eine List von allen Karten-IDs zurück.

MERGE_RELATION(DATA, KEY, ITEM_CALLBACK)
: Fügt die Daten einer Beziehung zu den Karten-Daten hinzu. Die Liste der Argumente befindet sich in Tabelle 3.7

Tabelle 3.6: Transformations.Cards::constructor(cards : Dictionary[])

EINGABE	BESCHREIBUNG
`cards : Dictionary[]`	Zu bearbeitende Karten

Tabelle 3.7: Transformations.Cards::merge_relation(data : Dictionary[], key : string, item_callback : Closure)

EINGABE	BESCHREIBUNG
`data : Dictionary[]`	Daten der Verknüpfung
`key : string`	Name unter dem die Verknüpfung in den Karten-Daten verfügbar sein soll
`item_callback : Closure`	Funktion, um Elemente der Verknpüfung zu bearbeiten bevor diese zu den Karten-Daten hinzugefügt werden

3.4.1.11 *Tests.AbstractTestCase*

Die Abstrakte Klasse `Tests.AbstractTestCase` hat die folgenden Schnittstellen:

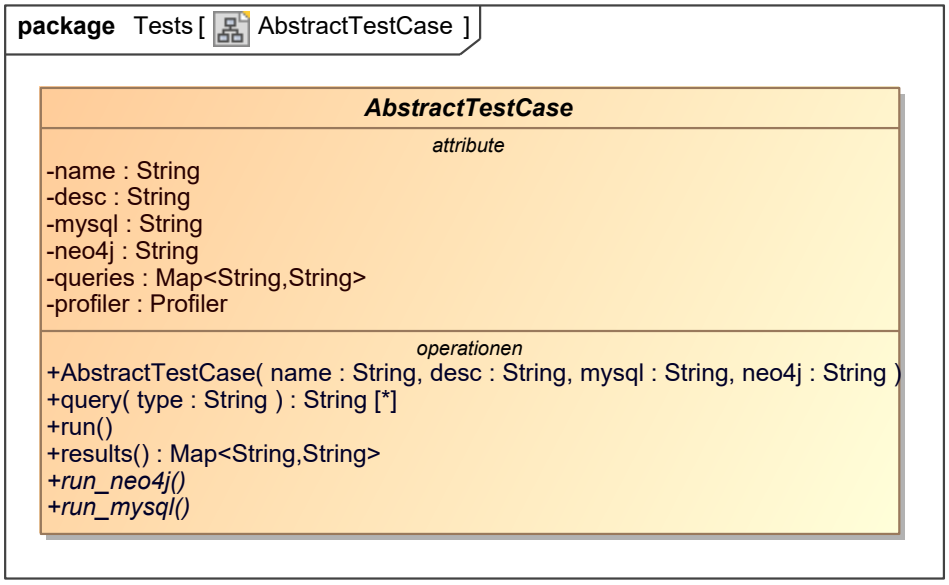

Abbildung 3.14: Klassendiagramm Tests.AbstractTestCase

CONSTRUCTOR(NAME, DESC, MYSQL, NEO4J)
: Setzt den Namen `name` und die Beschreibung `desc` des Testfalls. Außerdem werden die Schlüssel `mysql` und `neo4j` festgelegt, welche für die Testergebnisse benutzt werden. Die Liste der Argumente befindet sich in Tabelle 3.8

QUERY(TYPE)
: Gibt alle Abfragen für `type = [mysql, neo4j]` die in dem Test ausgeführt werden zurück.

RUN()
: Führt Test durch.

RESULTS()
: Gibt aufgezeichnete Ergebnisse zurück nachdem der Test ausgeführt wurde.

RUN_NEO4J()
: Testfall für Neo4j-Datenbank.

RUN_MYSQL()
: Testfall für MySQL-Datenbank.

3.4.1.12 *Tests.Manager*

Die Klasse `Tests.Manager` hat die folgenden Schnittstellen:

Tabelle 3.8: Tests.AbstractTestCase::constructor(name : string, desc : string, mysql : string, neo4j : string)

EINGABE	BESCHREIBUNG
`name : string`	Name des Testfalls
`desc : string`	Beschreibung des Testfalls
`mysql : string`	Schlüssel für MySQL-Testergebnisse
`neo4j : string`	Schlüssel für Neo4j-Testergebnisse

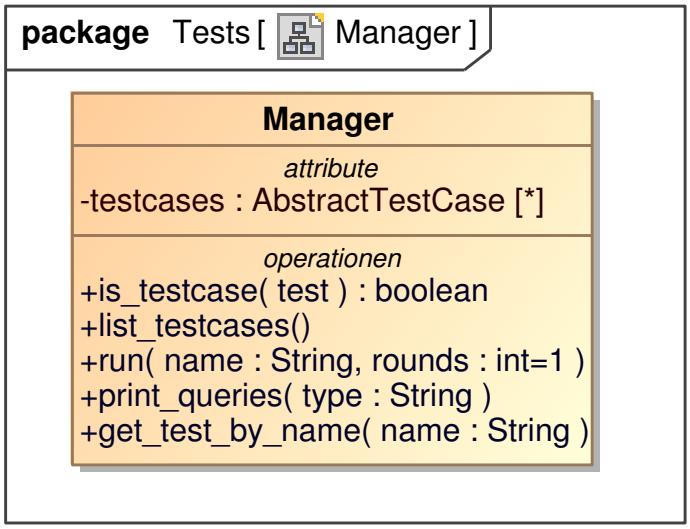

Abbildung 3.15: Klassendiagramm Tests.Manager

CONSTRUCTOR
: Lädt alle Testfälle und speichert diese in `testcases`

IS_TESTCASE(TEST)
: Prüft ob ein Objekt `test` ein Testfall ist, das heißt von `Tests.AbstractTestCase` abgeleitet ist.

LIST_TESTCASES()
: Gibt eine Liste der verfügbaren Tests in `testcases` aus

RUN(NAME, ROUNDS)
: Führt den Testfall `name` `rounds`-mal hintereinander aus

PRINT_QUERIES(NAME)
: Gibt alle Abfragen des Testfalls `name` aus.

GET_TEST_BY_NAME(NAME)
: Gibt den Testfall `name` zurück, sofern dieser sich in `testcases` befindet

Tabelle 3.9: Tests.Manager::constructor(name : string, desc : string, mysql : string, neo4j : string)

EINGABE	BESCHREIBUNG
`name : string`	Name des Testfalls
`desc : string`	Beschreibung des Testfalls
`mysql : string`	Schlüssel für MySQL-Testergebnisse
`neo4j : string`	Schlüssel für Neo4j-Testergebnisse

3.4.2 *Schnittstellenrealisierung*

3.4.2.1 *Fetch*

Fetch decks from mtgtop8.com [12]

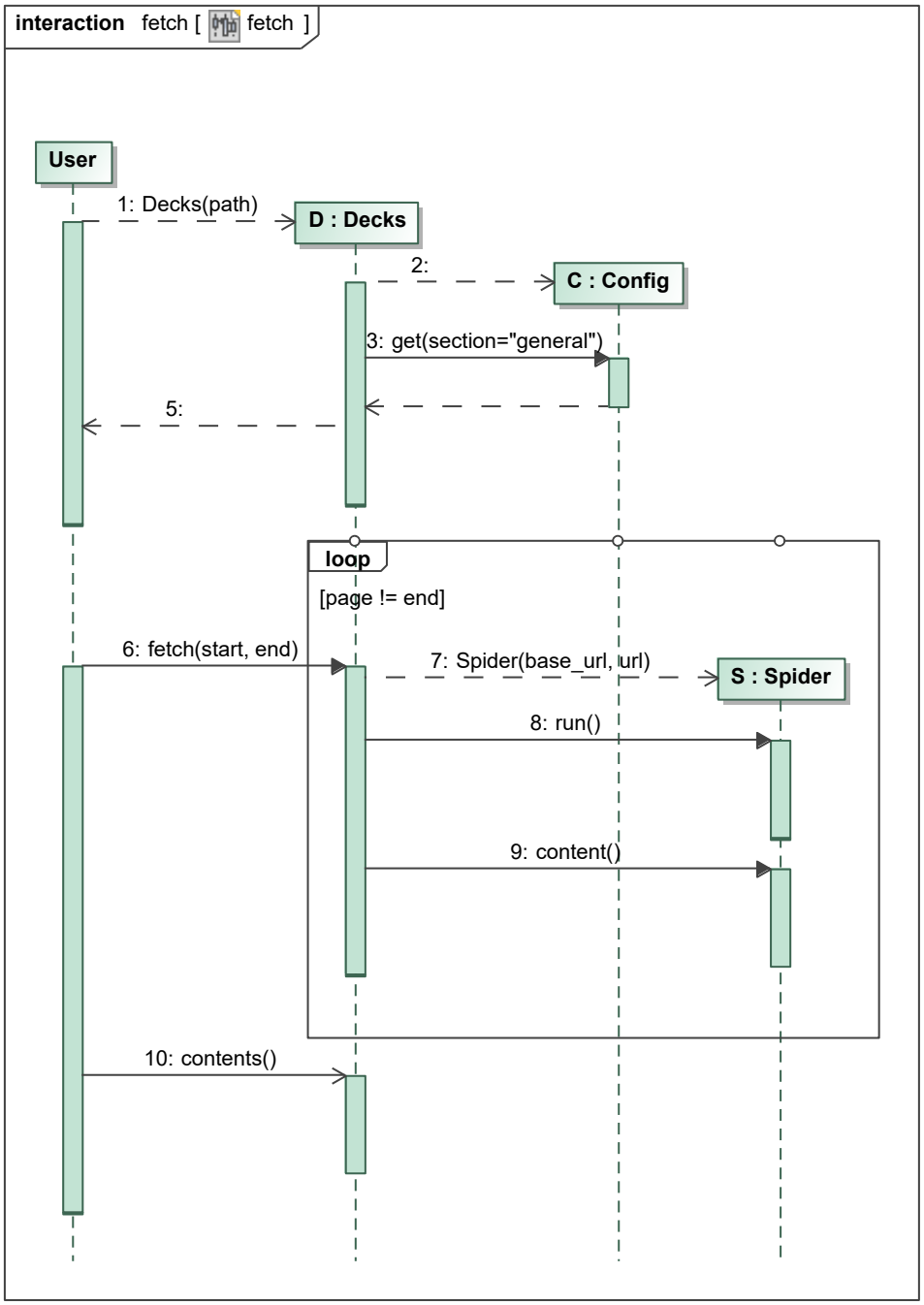

Abbildung 3.16: Sequenzdiagramm Fetch

3.4.2.2 *Build*

Build [12]

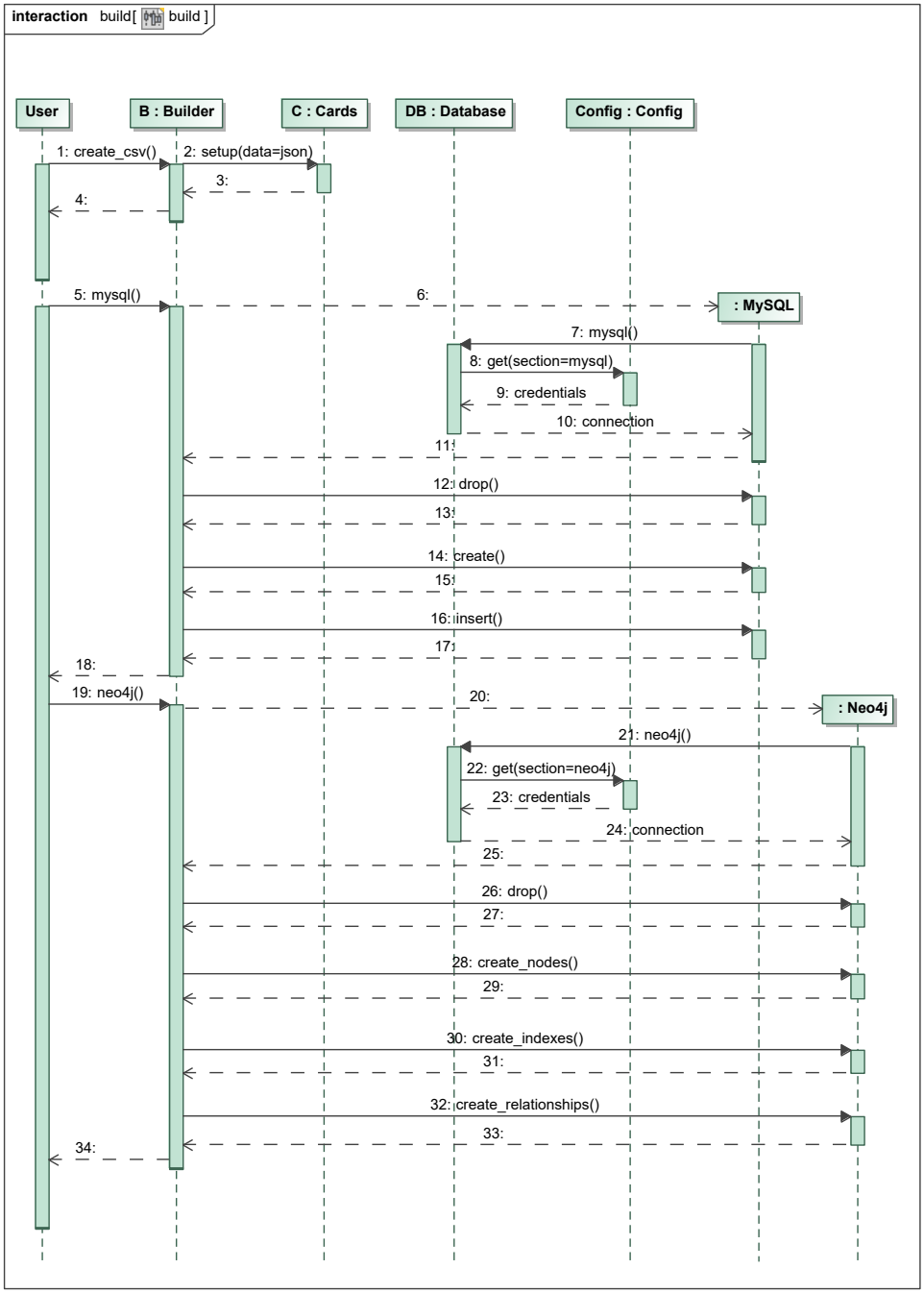

Abbildung 3.17: Sequenzdiagramm build

3.5 AUSGEWÄHLTER ANSATZ UND DETAILLÖSUNGEN

3.5.1 *Datenbasis*

Als Basis der Kartendaten wird mtgjson[2] benutzt, welche MtG Kartendaten als JSON bereitstellt. Die Decks hingegen werden von mtgtop8[3] bezogen.

3.5.2 *Datenbanken*

Als Graph-Datenbank wird *Neo4j* eingesetzt, da diese eine hoch skalierbare Datenbank ist, welche auf allen gängigen Betriebssystemen läuft [7]. Des Weiteren besitzt Neo4j mit *Cypher* eine ausdrucksstarke Abfragesprache. Ein Vorteil von Cypher ist, dass ab Neo4j 2.1 der Import aus CSV-Dateien unterstützt wird. [14]

Als RDBMS wird MariaDB[4] eingesetzt, welche vom Urheber von MySQL entwickelt wird. MariaDB ist ein Fork von MySQL und gilt als dessen evolutionäre Nachfolger. [1]

3.5.2.1 *Import der Daten*

Da eine große Menge an Daten importiert werden müssen, zum Beispiel über 90.000 Übersetzungen, ist eine schnelle Import-Funktion hilfreich. In 2 befindet sich ein Kommando, um Übersetzungen nach Neo4j zu importieren und in 3 ein Kommando, um nach MySQL zu importieren.

```
1  USING PERIODIC COMMIT 1000
2  LOAD CSV WITH HEADERS FROM "file:translations.csv" AS row
3  CREATE (:Translation { name: row.name });
```

Listing 2: Importieren von Übersetzungen

```
1  LOAD DATA LOCAL INFILE 'translations.csv' INTO TABLE `translations`
2  CHARACTER SET UTF8
3  FIELDS TERMINATED BY ','
4  ENCLOSED BY '"'
5  LINES TERMINATED BY '\r\n'
6  IGNORE 1 LINES
7  SET (@card, name, lang, card_id);
```

Listing 3: Importieren von Übersetzungen

2 http://mtgjson.com
3 http://mtgtop8.com
4 https://mariadb.com

3.5.3 *Programmiersprache*

Als Programmiersprache wird *PyPy*[5], eine alternative *Python* Implementierung, verwendet. Die Vorteile von PyPy sind der integrierte Just-in-time (JIT) Compiler, wodurch sich eine schnellere Ausführungszeit ergibt, und der geringere Speicherverbrauch. [5]

3.5.3.1 *Profiling*

Mit einem *Profiler* können verschiedene Werte wie *Ausführungszeit* oder *Speicherverbrauch* gemessen werden [5]. Um vergleichen zu können, wie gut die Datenbanksysteme in den einzelnen Testfällen abschneiden, werden die Ausführungszeit und der Speicherverbrauch gemessen.

Für die Ausführungszeit kann eine einfache Timer-Klasse[6] benutzt werden. Um den Speicherverbrauch zu messen wird die Python-Bibliothek psutil benutzt[7], da diese eine einfache und plattformübergreifende Methode bietet den Speicherverbrauch auszulesen.

5 http://pypy.org
6 https://www.huyng.com/posts/python-performance-analysis
7 http://fa.bianp.net/blog/2013/different-ways-to-get-memory-consumption-or-lessons-learned-from-memory_profiler/

4 ERGEBNISSE

4.1 VALIDIERUNG DES GESAMTKONZEPTES

In Tabelle 4.1 befindet sich eine Auflistung der Probleme die in Abschnitt 3.1 beschrieben wurden und welche Testfälle erforderlich sich, um die Anforderung zu überprüfen.

Jeder Test wurde 1000mal sowohl auf Neo4j als auch auf MySQL ausgeführt, um den Einfluss von Messfehlern zu reduzieren *(Gesetz der Großen Zahlen)*. Bei jedem Test wurde die Ausführungszeit in Millisekunden (ms) und der Speicherverbrauch in MByte gemessen. Um sicherzugehen, dass Caching oder Systemprozesse die Ergebnisse nicht beeinflussen, wurden die 10 längsten und kürzesten Zeiten verworfen. Die übriggebliebenen Werte wurden gemittelt und die Standardabweichung wurde berechnet, um zu schauen wie groß die Fehlergrenze ist.

Um zu erfahren wie gut die beiden Datenbanken skalieren, wurden außerdem die Testfälle stückweise auf größeren Datensätzen ausgeführt. Diese Datensätze wurden zufällig generiert und basieren nicht auf echten Daten. Als Staffelung für die Tests wurden *100.000, 250.000, 500.000 und 1.000.000* Karten/Turniere gewählt.

4.2 BESCHREIBUNG UND MOTIVATION DER TESTFÄLLE

4.2.1 *Testfall 1: Schlüsselwort-Fähigkeit*

Eine einfache Suche, um die ersten 300 Karten mit dem Schlüsselwort-Fähigkeit *Flying* zu erhalten. Wie in Abbildung 3.1 zu sehen ist, werden die Fähigkeiten in MySQL in einer eigenen Tabelle gespeichert. In Neo4j hingegen können diese als Attribut von *Card* hinterlegt werden *(siehe Abbildung 3.2)*, da Neo4j den Datentyp *Array* unterstützt. Mit diesem Test lässt sich also vergleichen wie gut sich Arrays im Vergleich zu joins beim durchsuchen eignen.

4.2.2 *Testfall 2: Textsuche*

Eine einfache Suche nach den Karten, die den Namen *Forest* enthalten und die entweder von *Aleksi Briclot* oder *John Avon* gezeichnet wurden. Da Karten anhand ihres Namens identifiziert werden, ist es sinnvoll zu testen wie gut sich eine Textsuche in den beiden Datenbanksystemen funktioniert.

[draft created on 10. Februar 2017 at 10:58 (delete drafting option in thesis-config.tex to remove this line)]

Tabelle 4.1: Erforderliche Testfälle

ANFORDERUNG	TESTFALL
Kartensuche: Schlüsselwort-Fähigkeit	Suche alle Karten mit einer bestimmten Fähigkeit
Kartensuche: Textsuche	Suche nach einer Zeichenkette die in einem Kartennamen vorkommt
Kartensuche: Verknüpfungen	Suche anhand verschiedener Karten-Attribute
Turnier: Matchup Analyse	Berechne das Matchup eines Deck-Typen
Turnier: Top 10 Decks	Berechne die 10 besten Decks

4.2.3 *Testfall 3: Verknüpfungen*

Eine komplexe Suche nach Karten, welche bestimmte Kriterien erfüllen *(siehe 1)*. Es wird nach Karten gesucht, die folgende Eigenschaften erfüllen:

- in einem Set nach dem *01.01.2001* erschienen
- als Seltenheit *RARE* besitzen
- in *deutscher* Sprache erschienen
- als Farbe *Blau* haben
- vom Typ *Kreatur* sind
- die Fähigkeit *Flying* besitzen
- umgewandelte Manakosten von *maximal 5* haben

4.2.4 *Testfall 4: Matchup Analyse*

Berechnung des Matchup für den Deck-Typ *Bant*. Es werden alle Matches analysiert an denen ein Deck vom Typ *Bant* beteiligt war und überprüft ob das Deck gewonnen oder verloren hat. Die Ergebnisse werden dann anhand der gegnerischen Deck-Typen gruppiert und geordnet. Insgesamt gibt es im Standard-Testfall 1000 Turniere, die jeweils zwischen 2 und 6 Matches enthalten.

4.2.5 *Testfall 5: Top 10 Decks*

Für jedes Deck wird das Verhältnis zwischen den gewonnen und verlorenen Matches berechnet und dann der Größe nach sortiert und die ersten 10 Decks gewählt. Je höher der Wert ist, desto besser ist das Deck unabhängig von der

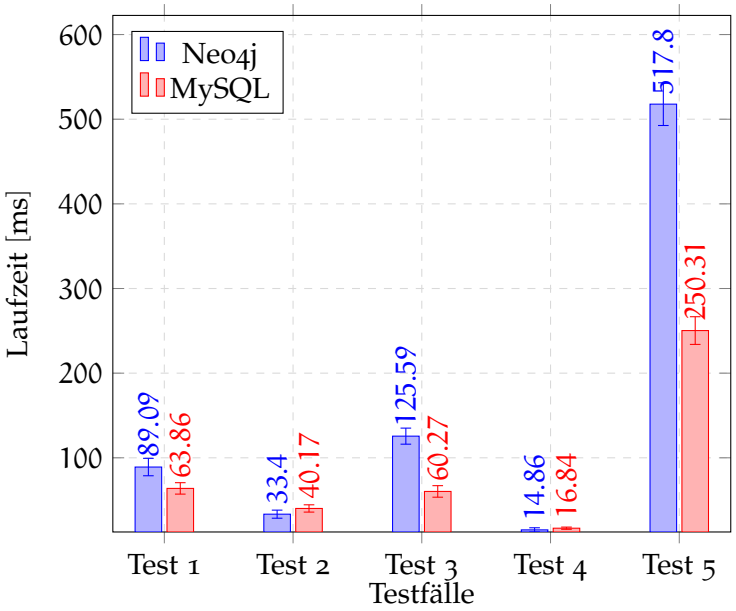

Abbildung 4.1: Ausführungszeit der einzelnen Testfälle

Anzahl der gespielten Matches. Insgesamt gibt es im Standard-Testfall 1000 Turniere, die jeweils zwischen 2 und 6 Matches enthalten.

4.3 ÜBERSICHT UND BEWERTUNG DER ERZIELTEN ERGEBNISSE

4.3.1 *Laufzeit*

Die gemessenen Laufzeiten samt ihrer Standardabweichung sind in Abbildung 4.1 als Fehlerbalkendiagramm dargestellt. Wie zu erwarten war Neo4j in Testfall 4 schneller als die MySQL-Abfrage mit JOINS. Überraschenderweise war Neo4j auch in Testfall 2 etwas schneller als MySQL, obwohl aufgrund der langjährigen Optimierungen der Textsuche in MySQL zu erwarten war, dass dieser schneller sei. Der Grund dafür, dass Neo4j in den Testfällen 1 und 3 langsamer ist als MySQL liegt vermutlich daran, dass als Datentyp Arrays für *color*, *abilities* und *types* benutzt wurde. Anscheinend ist in Neo4j die Suche innerhalb eines Arrays langsamer als die Suche mit einem JOIN in MySQL.

4.3.2 *Skalierbarkeit*

Wie in Abbildung 4.2 zu sehen ist, skalieren sowohl Neo4j als auch MySQL gut, aber auch hier zeigt sich, dass Neo4j in allen gemessenen Größen langsamer ist. Die Steigung ist bei beiden Datenbanken ungefähr gleich. Die Vermutung, dass die Suche in Arrays langsamer ist als mit JOINS bestätigt sich in Testfall 3. Wie in Abbildung 4.4 zu sehen ist, hat die Laufzeit bei Neo4j ein lineares Wachstum, wohingegen sie bei MySQL relativ konstant bleibt: bei doppelter Menge an Karten, verdoppelt *(ungefähr)* sich die Laufzeit in Neo4j.

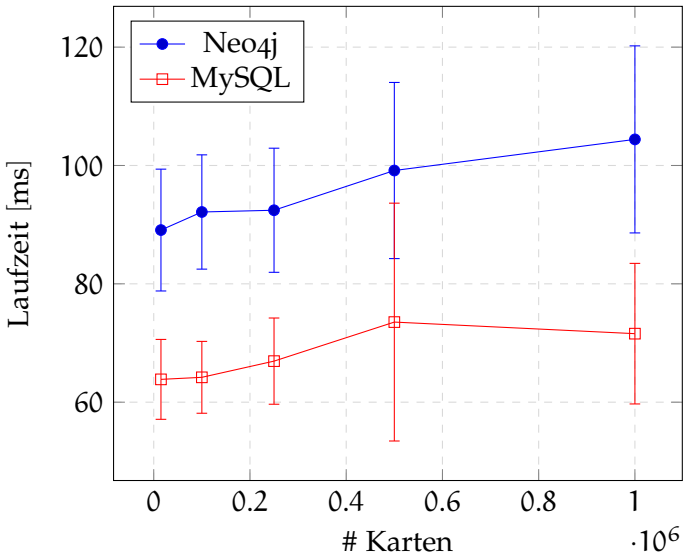

Abbildung 4.2: Skalierung Testfall 1

Es bietet sich an zu überprüfen, ob mit einem anderen Schema, welches auf Arrays verzichtet, Neo4j eine bessere Laufzeit und Skalierbarkeit hat.

In Testfall 2 bestätigt sich, dass MySQL bei der Textsuche optimierter ist als Neo4j, auch wenn Neo4j mit dem normalen Datensatz ein wenig schneller ist. In Abbildung 4.3 ist zu sehen, dass sich auch hier mit Verdopplung der Karten die Laufzeit der Neo4j-Abfrage um den Faktor 1.5 - 2 steigt, wohingegen die Laufzeit bei MySQL nur um 5% - 10% pro gemessenem Punkt steigt. MySQL ist also wie erwartet besser für Textsuchen geeignet als Neo4j.

In Abbildung 4.5 zeigt sich, dass die Laufzeit der MySQL-Abfrage für Testfall 4 ein exponentielles Wachstum hat. Bei der Neo4j Abfrage hingegen steigt die Laufzeit linear, das heißt für Turnier/Matchup-Analysen ist Neo4j besser geeignet als MySQL. Allerdings gilt dies nicht für Testfall 5, denn Abbildung 4.6 zeigt, dass hier Neo4j langsamer ist als MySQL. Sowohl MySQL als auch Neo4j haben ein lineares Wachstum, auch wenn es bei Neo4j nach einem beschränkten Wachstum aussieht. Ein beschränktes Wachstum erscheint aber für die Laufzeit nicht sinnvoll. Des Weiteren hat Neo4j ein stärkeres Wachstum als MySQL. Die Tatsache, dass beide Datenbanken bei den Abfragen deutlich langsamer sind, ist in diesem Testfall vernachlässigbar. Es ist ausreichend die besten Decks nur einmal am Tag oder wenn ein neues Turnier hinzugefügt wurde zu berechnen und das Ergebnis zu speichern: wenn kein neues Turnier hinzugefügt wird, ändert sich auch nicht die Top 10 Liste. Aus diesem Grund muss die Abfrage nicht in Echtzeit geschehen sondern kann durchaus einige Sekunden in Anspruch nehmen.

4.3.3 *Speicherverbrauch*

Der gemessene Speicherverbrauch in MByte unterscheidet sich zwischen MySQL und Neo4j faktisch nicht, das heißt er beträgt weniger als 0.1%. Des Weiteren bleibt dieser auch bei der Skalierung der Datenbank relativ kon-

stant mit ±1MByte Abweichung. Aus diesem Grund wird hier nicht weiter auf den Speicherverbrauch eingegangen.

[draft created on 10. Februar 2017 at 10:58 (delete drafting option in thesis-config.tex to remove this line)]

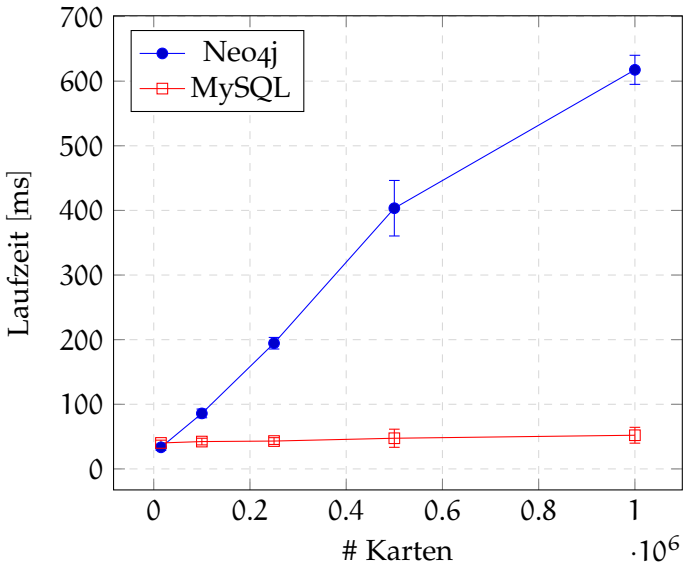

Abbildung 4.3: Skalierung Testfall 2

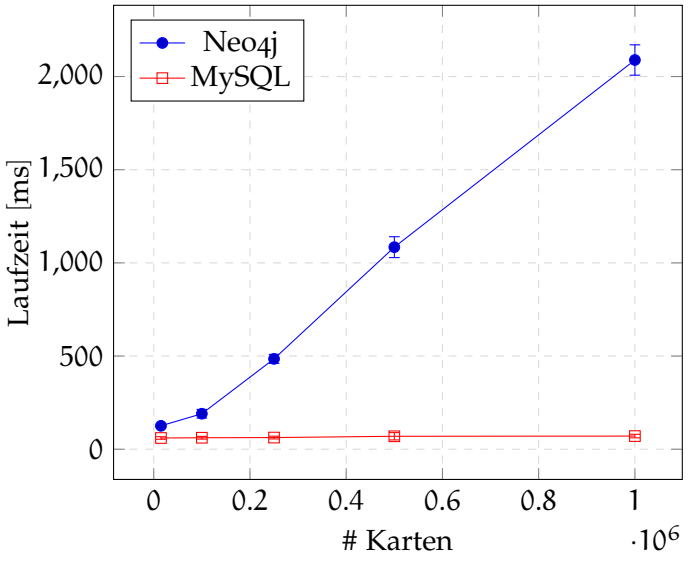

Abbildung 4.4: Skalierung Testfall 3

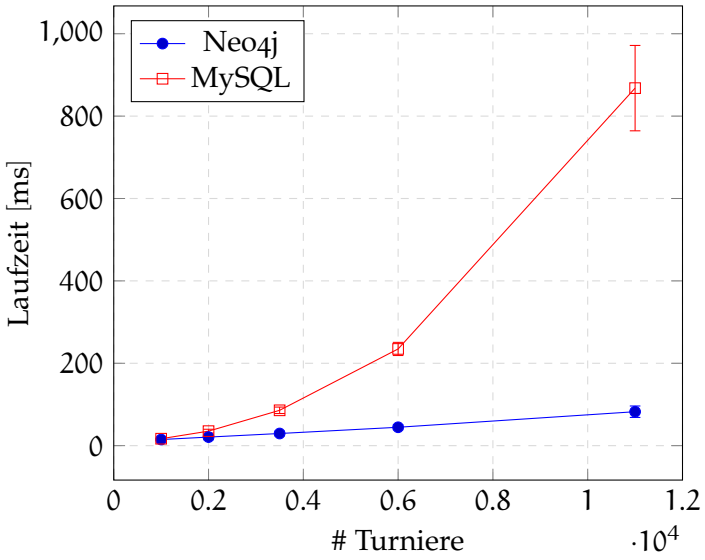

Abbildung 4.5: Skalierung Testfall 4

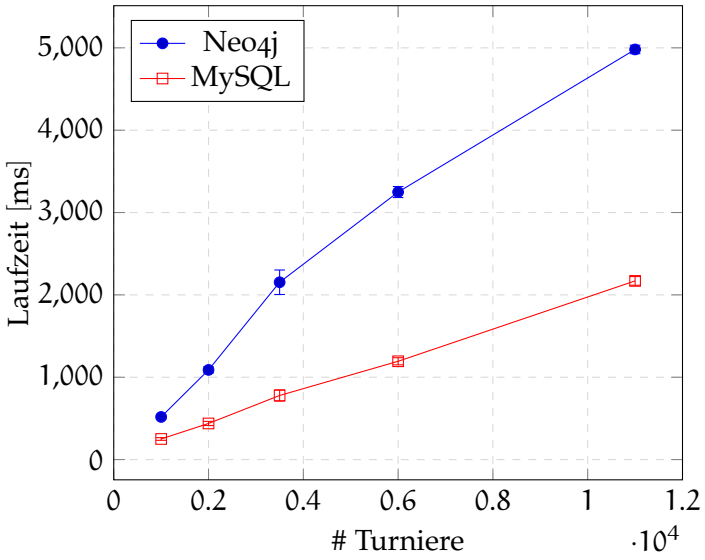

Abbildung 4.6: Skalierung Testfall 5

[draft created on 10. Februar 2017 at 10:58 (delete drafting option in thesis-config.tex to remove this line)]

5 ZUSAMMENFASSUNG UND AUSBLICK

5.1 ZUSAMMENFASSUNG

5.2 AUSBLICK

[draft created on 10. Februar 2017 at 10:58 (delete drafting option in thesis-config.tex to remove this line)]

[draft created on 10. Februar 2017 at 10:58 (delete drafting option in thesis-config.tex to remove this line)]

LITERATURVERZEICHNIS

[1] D. Bartholomew. *Getting Started with MariaDB*. Community experience distilled. Packt Publishing, 2013. ISBN: 9781782168102.

[2] Joy Chao. *Graph Databases for Beginners: Graph Theory & Predictive Modeling*. 2016. URL: https://neo4j.com/blog/graph-theory-predictive-modeling.

[3] Wizards of the Coast. *SPIELINFOS/GAMEPLAY STELLE DEIN EIGENES MAGIC-DECK ZUSAMMEN*. 2016. URL: http://magic.wizards.com/de/game-info/gameplay/how-to-build-a-deck.

[4] Wizards of the Coast. *WHAT ARE SWISS PAIRINGS?* 2016. URL: http://magic.wizards.com/en/game-info/products/magic-online/swiss-pairings.

[5] F. Doglio. *Mastering Python High Performance*. Packt Publishing, 2015. ISBN: 9781783989317.

[6] Dustin Fink, Benjamin Pastel und Neil Sapra. "Predicting the strength of Magic: The Gathering cards from card mechanics". In: ().

[7] A. Goel. *Neo4j Cookbook*. Quick answers to common problems. Packt Publishing, 2015. ISBN: 9781783287260.

[8] Roger Hau, Evan Plotkin und Hung Tran. *Magic: The Gathering Deck Performance Prediction*. Techn. Ber. Stanford University.

[9] Garima Jaiswal und Arun Prakash Agrawal. "Comparative analysis of Relational and Graph databases". In: *IOSR Journal of Engineering (IOSRJEN)* (2013).

[10] G Johnson, Z Wang und X Zhang. "An Online Database System for Card Stores". In: *Proceedings of the International Conference on e-Learning, e-Business, Enterprise Information Systems, and e-Government (EEE)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering und Applied Computing (WorldComp). 2013, S. 120.

[11] M. Lal. *Neo4j Graph Data Modeling*. Packt Publishing, 2015. ISBN: 9781784397302.

[12] *MTGTOP8 - netdecking with the stars*. URL: http://mtgtop8.com.

[13] Del Laugel Matt Tabak und Kelly Digges. *Magic: the Gathering Basic Rulebook 2013*, S. 4–7.

[14] Onofrio Panzarino. *Learning Cypher*. Packt Publishing, 2014. ISBN: 1783287756, 9781783287758.

[15] Boris A Perkhounkov, Cooper Gates Frye und Emily Margaret Franklin. "Financial Magic". In: ().

[16] I. Robinson, J. Webber und E. Eifrem. *Graph Databases: New Opportunities for Connected Data*. O'Reilly Media, 2015. ISBN: 9781491930861.

[draft created on 10. Februar 2017 at 10:58 (delete drafting option in thesis-config.tex to remove this line)]

[17] Robert Schults und Sergio Moura. *MTG JSON*. URL: http://mtgjson.com.

[18] StarCityGames.com. *What is a Swiss-style tournament?* 2016. URL: http://sales.starcitygames.com/FAQ.php?ID=92.

[19] T.J. Teorey, S.S. Lightstone, T. Nadeau und H.V. Jagadish. *Database Modeling and Design: Logical Design*. The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 2011. ISBN: 9780123820211.

[20] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen und Dawn Wilkins. "A comparison of a graph database and a relational database: a data provenance perspective". In: *Proceedings of the 48th annual Southeast regional conference*. ACM. 2010, S. 42.