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

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. Michael Prilla

TAG DER EINREICHUNG
30. März 2017

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 2017

Pascal Kleindienst

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.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

ABKÜRZUNGSVERZEICHNIS

API Application Programming Interface

UML Unified Modeling Language

ER-Modell Entity-Relationship-Modell

OOP objektorientierte Programmierung

RDBMS relationales Datenbankmanagementsystem

MtG Magic: the Gathering

TCG Trading Card Game

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]. Aufgrund der großen Menge an Karten sind daher Applikation sinnvoll mit denen man anhand bestimmter Kriterien nach Karten suchen kann. Besonders für professionelle Turnierspieler ist eine gute Suchfunktion wichtig, um die passenden Karten für ihr Deck zu finden. Andererseits ist auch die Funktion das Deck-Analysieren zu lassen wichtig, um es optimieren zu können und somit konkurrenzfähig zu bleiben.

Das Problem hierbei ist, dass Datenbankabfragen in relationalen Datenbanken, wie MySQL, aufgrund der vielen Verknüpfungen der Attribute einer Karte, sehr schnell an Komplexität zunehmen. Daraus folgt, dass die Abfragen viele `JOIN`-Befehle enthalten, was sich wiederrum sowohl negativ auf die Lauzeit und besonders die Skalierbarkeit als auch die Lesbarkeit auswirkt. Bei den meisten Applikation, die Suchfunktionen oder Deck- und Turnieranalysen anbieten, handelt es sich um Webapplikation, wodurch eine eine kurze Laufzeit sehr wichtig ist.

Turniere werden nach dem *Schweizer-System*, einer Sonderform des Rundenturniers, gespielt. Dadurch ergibt sich ein Netzwerk aus Paarungen der verschiedenen Runden. Aus diesem Grund steigt die Anzahl der gespielten Partien schnell an. Bei einem Turnier mit 50 Spielern werden 6 Runden gespielt, wovon jede Runde pro Paarung zwei bis drei Spiele gespielt werden. Bei 6 Runden und 50 Spielern ergeben sich somit zwischen 300 und 450 gespielte Partien. Daher speichern Anbieter wie `http://mtgtop8.com` oder `https://mtgdecks.net` Ergebnisse von Turnieren oftmals der Einfachheit halber nur als Rangfolge und nicht jedes Rundendergebnis. Mit letzterem Ansatz ließen sich jedoch verschiedene Analysen über die Decks und Spieler ausführen, wie zum Beispiel die Berechnung von Matchups, das heißt welche Gewinn- oder Verlustchancen ein Deck gegen ein anderes Deck hat. Diese Problematik wird auch in [9] behandelt, allerdings mit einem Schwerpunkt auf der Häufigkeit der benutzen Karten und der Synergie zwischen diesen. Unter Einbeziehung von Rundenergebnissen, welche aber nicht zur

1 Basierend auf `http://mtgjson.com`

Verfügung standen, könnte die Stärke eines Decks noch besser eingeschätzt werden [9].

Graph-Datenbanken haben in den letzten paar Jahren viel an Bedeutung gewonnen, da sie einen einfachen Vorteil gegenüber herkömmlichen relationalen Datenbank haben [1]. Bei Graph-Datenbanken dreht sich alles um die Verbindungen zwischen Datenpunkten und auch bei wachsenden Datensätzen bleibt die Performance relativ konstant [18].

In [10, 14, 21] wird Neo4j mit Graph-Datenbanken verglichen. Sowohl [21] als auch [10] kamen zu dem Ergebnis, dass beide Systeme eine gute Performanz hatten und bei nur einer Beziehung die Laufzeit fast gleich war. Außerdem hat sich gezeigt, dass falls in der Abfrage mehrere Beziehungen beteiligt oder verschachtelte Abfragen enthalten sind, die Graph-Datenbank eine geringere Laufzeit hatte [10]. Der Ansatz einer Graph-Datenbank scheint daher geeignet für Kartenspiele und Turniere, welche viele Verknüpfungen aufweisen oder ein großes Wachstum an Daten haben, so dass diese näher zu betrachten ist.

Um die relationale Datenbank mit der Graph-Datenbank vergleichen zu können, werden als Basis die Informationen zu allen erschienen Karten und Editionen benötigt. Das Softwarepaket *mtgjson*[2] ist ein open-source Datensatz aller Kartendaten und Editionen. Dieser stellt eine Art Standard dar, welcher unter anderem auch in [7, 16, 17] benutzt wird, so dass er auch hier zum Einsatz kommen soll.

Die Daten für Decks wurden von der Seite `http://mtgtop8.com`, welche Decks von Turnieren der ganzen Welt veröffentlicht, entnommen. MtGTop8 ist einer der größten Anbieter mit über 75.000 Decks und wurde unter anderem von Pawlicki et. al. benutzt um Preisvorhersagen über Karten zu machen [16]. Jedoch bietet MtGTop8 weder eine herunterladbare Liste der Decks noch ein öffentliches Application Programming Interface (API) an. Leider kann MtGTop8 nicht benutzt werden um auch als Datenbasis für Turniere zu dienen, da hier wie bei praktisch allen großen Anbieter nur die Endergebnisse veröffentlicht werden und die einzelnen Rundenergebnisse nicht vorhanden sind. Aus diesem Grund müssen simulierte Turniere als Datenbasis verwendet werden.

Online-Shops benötigen oft nur einen Bruchteil der Informationen, die sich auf einer Karte befinden und speichern diese in einer relationalen Datenbank in ein bis drei Tabellen [11]. Gerade für kleinere Online-Shops eignet sich dieser Ansatz, da hierbei der Wartungsaufwand gering gehalten wird [11] und die Abfragen nicht viele `JOIN` Befehle enthalten und damit performant sind. Karten-Suchmaschinen erfordern aber, dass alle Attribute einer Karte gespeichert werden, damit nach jedem Attribut gesucht *(auch kombiniert)* gesucht werden kann. Daher ist es erforderlich, dass alle Informationen einer Karte effizient gespeichert und verwaltet werden und außerdem Suchanfragen in Echtzeit beantwortet werden können.

Wie schon erwähnt speichern bisher die großen Anbieter von Turnier- und Deck-Daten lediglich die Endplatzierungen der Decks und Spieler und

2 `http://mtgjson.com`

meistens auch maximal nur die ersten 10 Spieler. Dieser Ansatz macht es aber praktisch unmöglich Deck-Analysen zu erstellen, da es hierfür erforderlich ist die einzelnen Spiele-Paarungen und Rundenergebnisse zu betrachten. Daher werden in diesem Ansatz nicht die Endergebnisse der Turniere sondern jedes Ergebnis jeder gespielten Partie gespeichert, sodass es möglich ist Decks und ihre Stärken und Schwächen besser zu analysieren.

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* [13]. 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 [13].

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

REGELTEXT Enthält die *Fähigkeiten* einer Karte, sofern vorhanden [13].

FLAVOR-TEXT Enthält Informationen über den Welt von MtG und hat keinerlei Auswirkungen auf das Spielgeschehen [13].

SAMMLERNUMMER Hilft dabei die Karten einer Edition zu sortieren/organisieren [13].

KÜNSTLER Der Künstler von dem das Bild für die Karte stammt [13].

STÄRKE UND WIDERSTANDSKRAFT Jede *Kreatur* hat einen Wert für Stärke und Widerstandskraft. Die erste Zahl gibt die Stärke an und die zweite die Widerstandskraft [13].

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].

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 [13].

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* [13].

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 [13].

SCHLÜSSELWORTE Fähigkeiten, die auf ein einzelnes Wort oder Satz gekürzt sind und eventuell einen Erinnerungstext haben, heißen *Schlüsselwort-Fähigkeiten* [13]. Meistens handelt es sich um *statische Fähigkeiten*, aber *ausgelöste* und *aktivierte Fähigkeiten* sind auch möglich [13].

2.2 AUFBAU EINES DECKS

Ein Deck muss aus mindestens 60 Karten bestehen, wobei nach oben hin keine Grenzen gesetzt sind [13]. 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 [13]. Daraus ergeben sich folgende Richtlinien:

- Ein Deck sollte zwischen 35% und 40% aus Ländern bestehen [4]
- Ein Deck sollte zwischen 25% und 40% aus Kreaturen bestehen. Dabei sollte beachtet werden, dass die Manakosten gut verteilt sind [4]
- Rest: Alles was nicht Land oder Kreatur ist und das Deck unterstützt [4]

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 [4].

2.2.1 *Deck-Typen*

Durch die große Menge an Karten und die goldene Regel *(falls eine Karte dem Regelbuch widerspricht, hat die Karte Vorrang)* 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 [4]. Decks die eine ähnliche Strategie haben oder gleich aufgebaut sind, werden unter einem Deck-Typ zusammengefasst [9]. Besonders starke Deck-Typen entstehen häufig durch die Analyse der einzelnen Karten und den Ergebnissen eines Decks in Turnieren [9]. Ein wichtiger Begriff ist hierbei die sogenannte Manakurve. Sie gibt an wie ausgewogen die Manakosten 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 [4]. Es wird durch die eigenen Kreaturen möglichst früh Druck auf den Gegner ausgeübt, um ihn so in die Verteidigung zu bringen [4]. Entscheidend für die Strategie sind eine geringe Manakurve und viele kleine Kreaturen.

Tabelle 2.1: Anzahl der Runden die ab einer Teilnehmerzahl gespielt werden [5]

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		

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 [4]. Kontrolldecks setzten in erster Linie auf defensive Zauberkarten, um den Gegner zu stören, und wenige teure Kreaturen, um damit das Spiel zu gewinnen [4].

MIDRANGE Midrange-Decks sind eine Kombination aus Aggro- und Kontrolldecks. Sie sind sehr effizient und können sich an die Strategie des Gegners anpassen [4]. Anstatt sich auf eine Strategie festzulegen wechseln Midrange-Decks je nach Situation zwischen aggressiver und defensiver Spielweise [4].

KOMBO Kombo-Decks nutzen Synergien die aus dem Zusammenspiel mehrerer Karten ergeben [4]. Die Decks sind komplett darauf ausgerichtet, diese Synergien noch weiter zu unterstützen und dadurch die Oberhand in der Partie zu gewinnen [4]. Aufgrund der Vielzahl an Karten gibt es unendlich viele Karten-Kombinationen [4].

2.3 TURNIERE

Turniere werden ähnlich wie beim Schach nach dem Schweizer System gespielt. Dies ist eine Variante des Rundensystem bei der die Teilnehmer jede Runde spielen und die Paarungen sich anhand der bisherigen Spiel-Ergebnisse ergeben [19]. Dadurch wird sichergestellt, dass Spieler mit ähnlichen Ergebnissen gegeneinander antreten [19]. 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.

2.3.1 *Matchup*

Matchup 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 Matchup.

2.4 GRAPH-DATENBANKEN

Ein Graph ist eine Sammlung von Objekten *(Knoten)* und deren Verbindungen *(Kanten)* [18]. Graphen lassen sich vielfältig einsetzen, vom Straßennetz bis zur Krankengeschichte von Populationen [18]. In Abbildung 2.2 sieht man eine einfache Unternehmenshierarchie als Graph dargestellt.

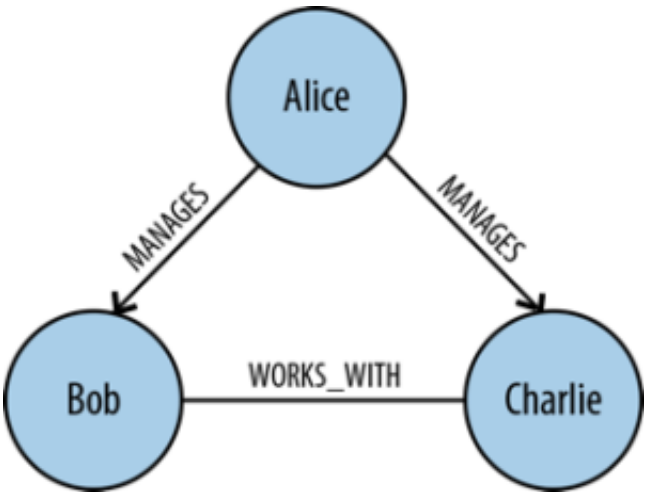

Abbildung 2.2: Einfacher Graph [3]

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 [18].

Beziehungen in Graph-Datenbanken sind, im Gegensatz zu anderen Datenbank-Systemen, sogenannte *first class citizens* [18]. Dies bedeutet, dass Objekte mit einander verbunden werden können ohne dazu Fremdschlüssel oder ähnliche Konstrukte zu verwenden [18]. Dadurch sind die Datenmodelle einfacher und ausdrucksvoller als Modelle von relationalen oder anderen Not Only SQL (NoSQL) Datenbanken [18] .

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 [18]. Bei relationalen Datenbank Abfragen mit

vielen `JOIN`-Abfragen hingegen verschlechtert sich die Abfrageperformance bei zunehmenden Datensatz [18].

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 [18]. 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 [18].

3 METHODE

3.1 ANALYSE DER ANFORDERUNGEN

3.1.1 *Kartensuche*

Aufgrund der großen Anzahl an Karten *(>15.000 [1])* 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 dabei Karten zu wählen die der Strategie des Decks entsprechen. Bei einem Aggro-Deck kann man so im Auge behalten eine niedrige Manakurve zu haben, um nicht zu viele Karten mit hohen Kosten in das Deck aufzunehmen. 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.

1 basierend auf `mtgjson.com`

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 Matchups für einen Deck-Typ wichtig. Dazu müssen die einzelnen Ergebnisse einer Runde eines Turniers mit den Decks verknüpft werden und das Verhältnis zwischen gewonnenen und verlorenen Spielen berechnet werden.

Ein weitere interessante Information ist die des erfolgreichsten Spielers oder 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 [11]. Da ein Online-Shop nur einen Teil der Attribute einer Karte, wie Name, Seltenheit und Set, benötigt, kann das Datenbank Schema aus ein bis drei Tabellen bestehen, die diese Informationen speichern [11]. 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 `JOIN`-Befehlen, da diese hoch-strukturierte Daten nicht unterstützen [18]. Des Weiteren sorgen viele `JOIN`-Befehle in einer Abfrage dafür, dass diese schlecht skaliert. Dies ist in Neo4j nich der Fall [18]. In Listing 1 ist beispielhaft eine komplexe SQL Suchanfrage mit 7 benötigten `JOIN`-Befehlen angegeben.

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 [18]. Eine Ausnahme sind die Graph-Datenbanken, da sie die Daten als Graph speichern, unterstützen sie daher Daten mit vielen Beziehungen untereinander von vornherein [18]. Außerdem skalieren Graph-Datenbanken sehr gut, da sie immer nur auf einem Teil des Graphen agieren und nicht den ganzen Graph betrachten müssen [18]. In [10] wurde außerdem gezeigt, dass für Abfragen mit mehr als einem `JOIN`-Befehl oder verschachtelten Abfragen, Neo4j mit der Abfragesprache Cypher besser geeignet ist. Zu einem ähnlichen Ergebnis, dass Neo4j bei Strukturabfragen besser geeignet ist, kamen auch Vicknair et. al. in [21]. Aufgrund der vielen Attribute einer Karte und den damit verbunden Verknüpfungen erscheint Neo4j als Datenbank gut geeignet zu sein.

```
SELECT * FROM `translations` AS `t`
JOIN `cards` ON `cards`.`id` = `t`.`card_id`
JOIN `cards_in_set` ON `cards_in_set`.`card_id` = `t`.`card_id`
JOIN `types` ON `cards`.`type_id` = `types`.`id`
JOIN `card_colors` ON `cards`.`id` = `card_colors`.`card_id`
JOIN `card_has_ability` ON `cards`.`id` = `card_has_ability`.`card_id`
JOIN `keyword_abilities` ON `card_has_ability`.`ability_id` = `
    keyword_abilities`.`id`
JOIN `sets` ON `sets`.`id` = `cards_in_set`.`set_id`
WHERE
    `sets`.release > "2001-01-01" AND
    `cards_in_set`.`rarity` = "RARE" AND
    `t`.`lang` = "German" AND
    `card_colors`.color = "BLUE" AND
    `types`.`type` = "Creature" AND
    `keyword_abilities`.`name` = "Flying" AND
    `cards`.`cmc` <= 5
```

Listing 1: Komplexe SQL-Abfrage

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 [20]. 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 [12]. Dieses Modell ist ein Untertyp des mathematischen Graph-Modells. Property-Graph-Modelle sind einfacher, aussagekräftiger und geben Beziehungen explizit an [12]. relationales Datenbankmanagementsystem (RDBMS) verwenden Fremdschlüssel, um Beziehungen implizit anzugeben [12]. In Abbildung 3.2 befindet sich ein möglicher Ansatz für ein Schema.

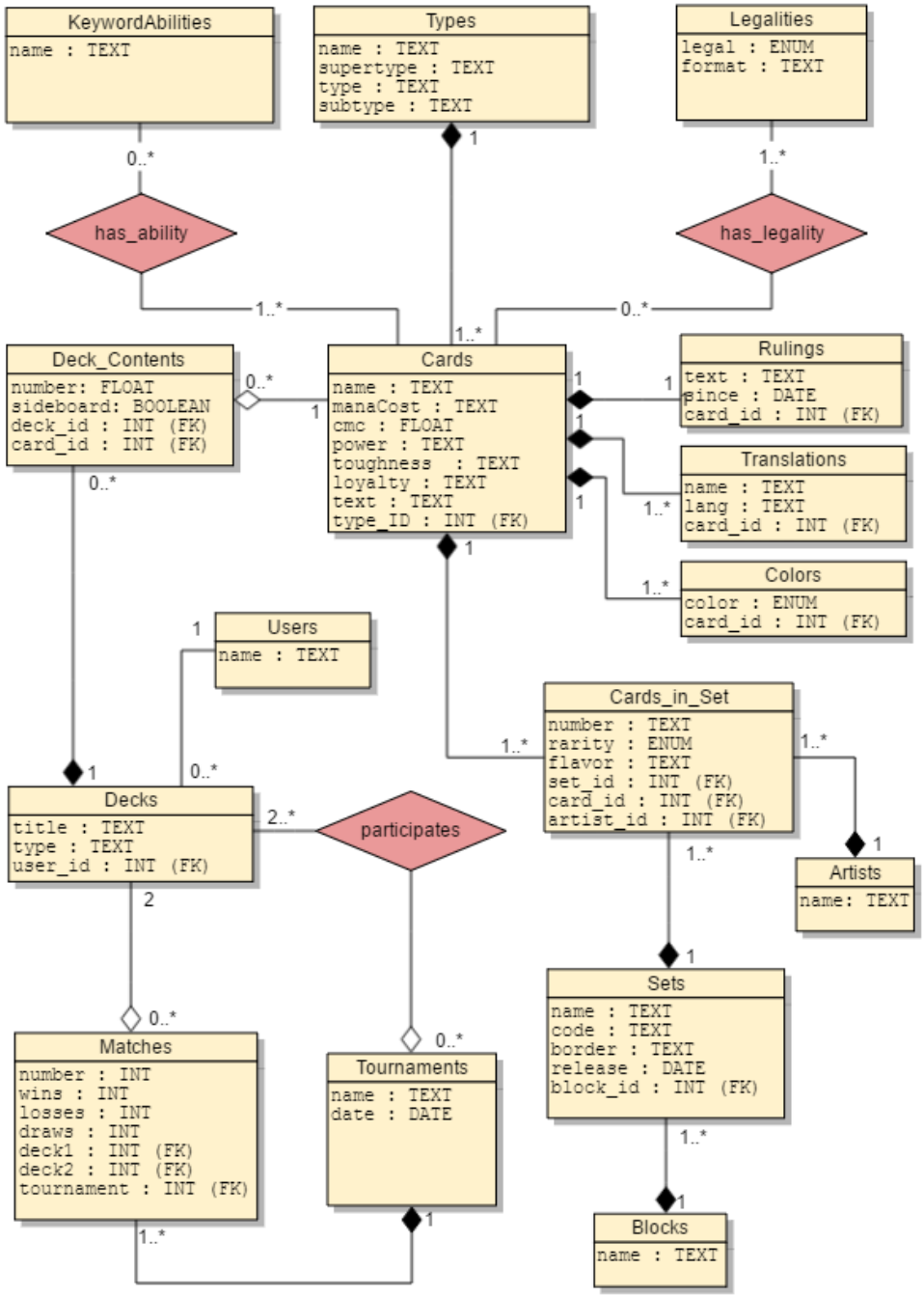

Abbildung 3.1: Schema für relationale Datenbank

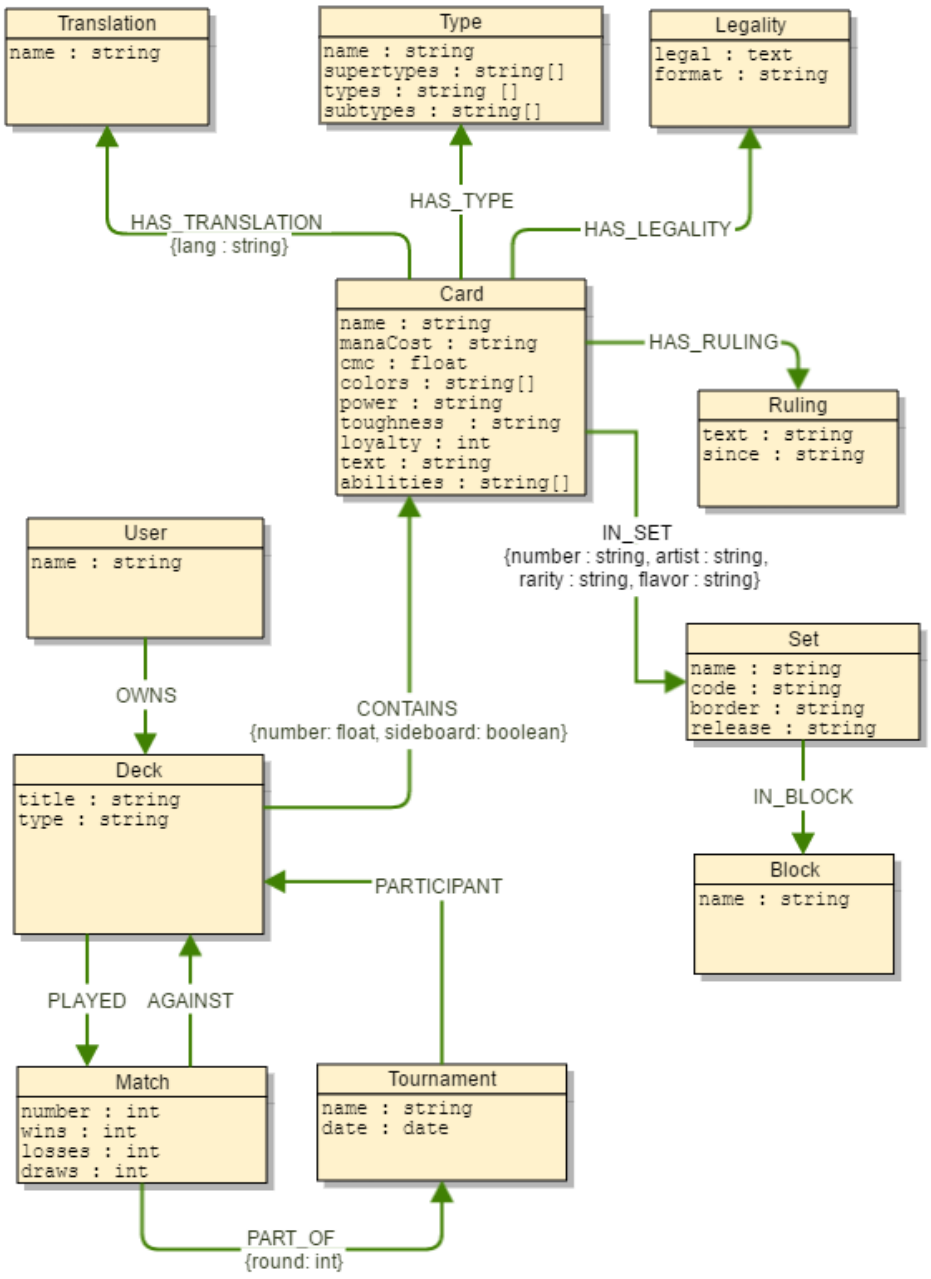

Abbildung 3.2: Schema für Graph-Datenbank

3.3.2 *Beschreibung der Software-Komponenten*

TESTS

Enthält die konkreten Testfälle und einen Manager um diese auszuführen

FOUNDATION

Enthält das Grundgerüst, welches von den meisten anderen Paketen benutzt wird

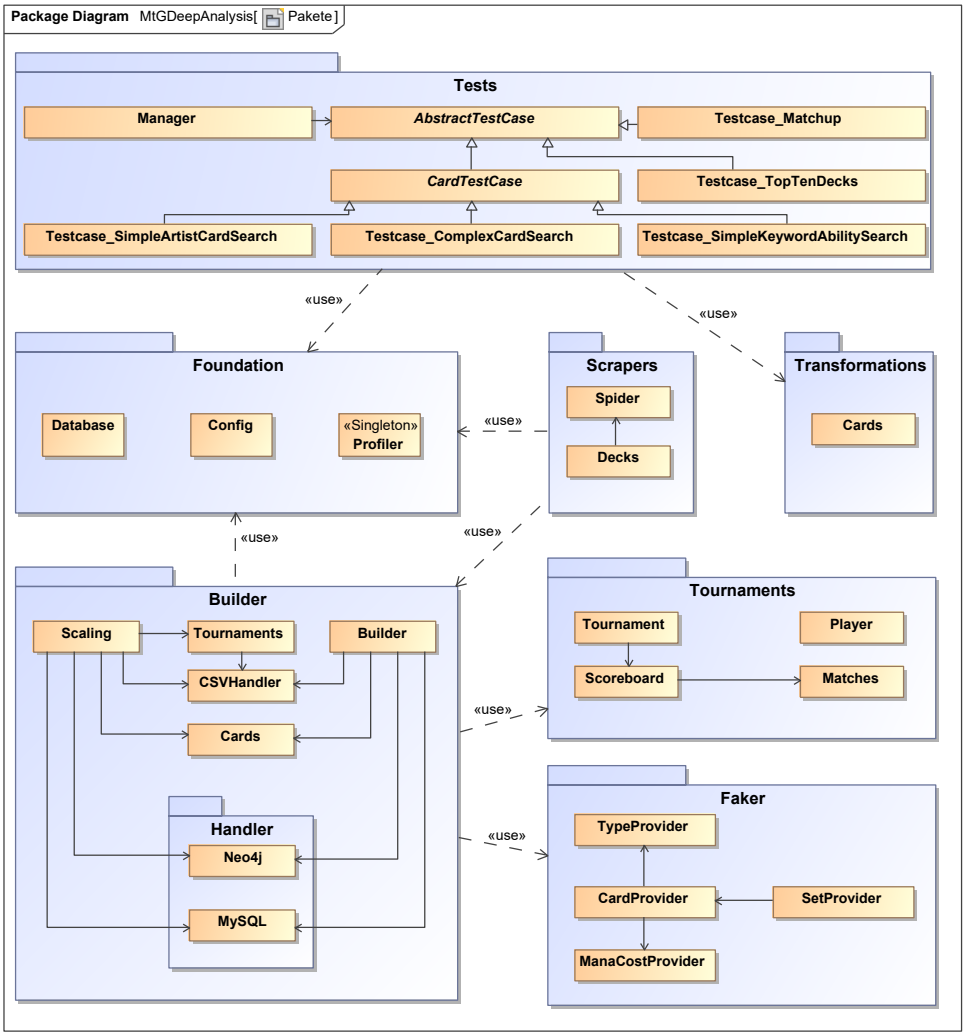

Abbildung 3.3: Software-Komponenten

SCRAPERS
: Ist dafür zuständig Decks von mtgtop8 [2] herunterzuladen, um einen Datensatz für Decks zu erstellen

TRANSFORMATIONS
: Ist zuständig für das Transformieren von Daten in eine vorgegebene Datenstruktur

BUILDER
: Ist zuständig für das erstellen und befüllen der Datenbanken

BUILDER.HANDLERS
: Enthält die Implementierungen für die konkreten Datenbanken

2 http://mtgtop8.com/

TOURNAMENTS
: Enthält Datenstrukturen und Algorithmen um Turniere nach dem Schweizer-System zu erstellen und auszuführen

FAKER
: Ist zuständig für die Erstellung von Dummy-Daten, um die Skalierung der Datenbanken zu testen

3.3.3 *Beschreibung der Schnittstellen*

Tests

Die Komponente `Tests` nutzt die folgenden Schnittstellen:

FOUNDATION wird genutzt, um ein Datenbank-Verbindungen zu erstellen und mit dem `Profiler` die Laufzeit und Speicherverbrauch zu messen

TRANSFORMATIONS wird genutzt, um die ausgelesenen Daten in eine passende Datenstruktur zu überführen

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

Builder

Die Komponente `Builder` nutzt die folgenden Schnittstellen:

FOUNDATION wird genutzt, um ein Datenbank-Verbindungen zu erstellen

TOURNAMENTS wird genutzt, um Turnierdaten basierend auf den heruntergeladenen Decks zu erstellen

FAKER wird genutzt, um ein Dummy-Daten für die Skalierungstests zu erstellen und zu importieren

3.4 IMPLEMENTIERUNG

3.4.1 *Klassendiagramme*

3.4.1.1 *Tests.AbstractTestCase*

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

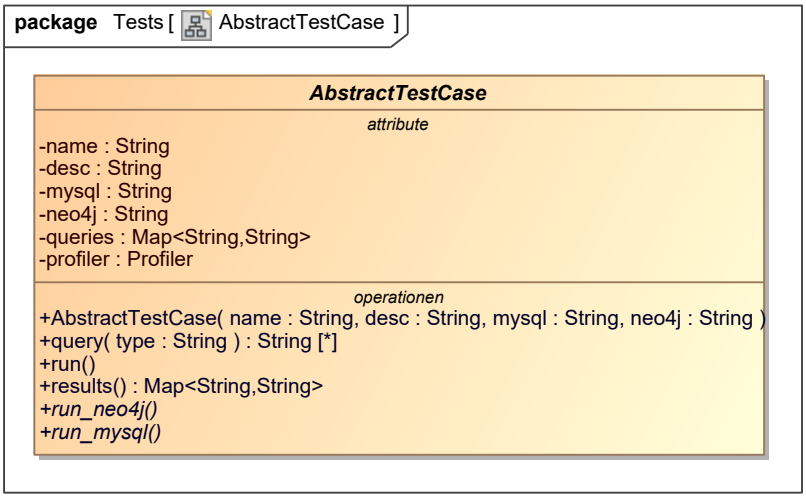

Abbildung 3.4: Klassendiagramm: Tests.AbstractTestCase

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 3.1

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 ausführen

RUN_MYSQL()
: Testfall für MySQL-Datenbank ausführen

3.4.1.2 *Tests.CardTestcase*

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

Tabelle 3.1: 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

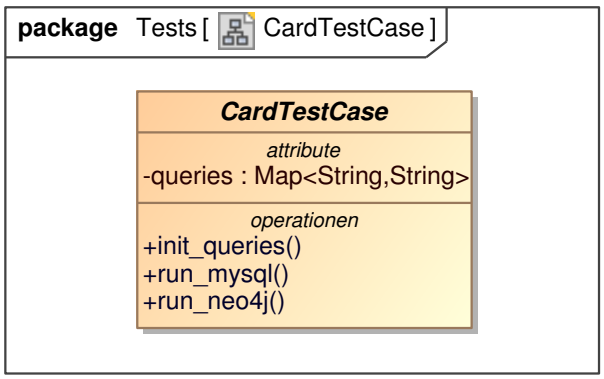

Abbildung 3.5: Klassendiagramm: Tests.CardTestcase

INIT_QUERIES()
: Fügt Standardabfragen zu `queries` hinzu, welche für die Kartentests gebraucht werden

RUN_NEO4J()
: Führt Cypher-Abfragen aus und überführt Karten in Datenstruktur `Transformations.Cards`

RUN_MYSQL()
: Führt SQL-Abfragen aus und überführt Karten in Datenstruktur `Transformations.Cards`

3.4.1.3 *Tests.Manager*

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

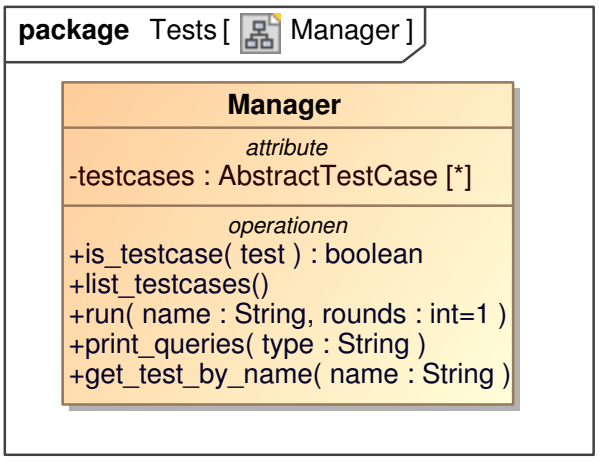

Abbildung 3.6: Klassendiagramm: Tests.Manager

MANAGER()
: 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

3.4.1.4 *Tests.Testcase_ComplexCardSearch*

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

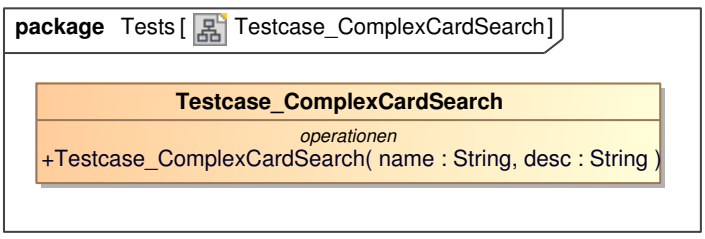

Abbildung 3.7: Klassendiagramm: Tests.Testcase_ComplexCardSearch

TESTCASE_COMPLEXCARDSEARCH(NAME, DESC)
Fügt Abfragen des Testfalls zu `queries` hinzu

3.4.1.5 *Tests.Testcase_Matchup*

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

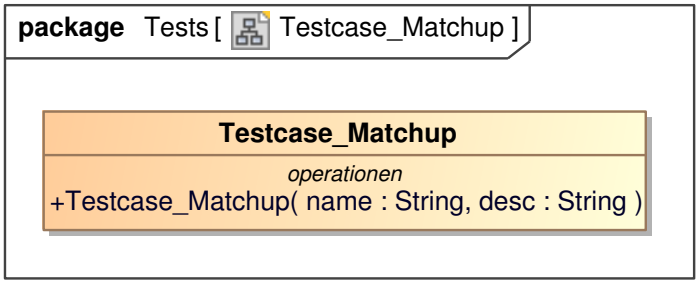

Abbildung 3.8: Klassendiagramm: Tests.Testcase_Matchup

TESTCASE_MATCHUP(NAME, DESC)
Fügt Abfragen des Testfalls zu `queries` hinzu

3.4.1.6 *Tests.Testcase_SimpleArtistCardSearch*

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

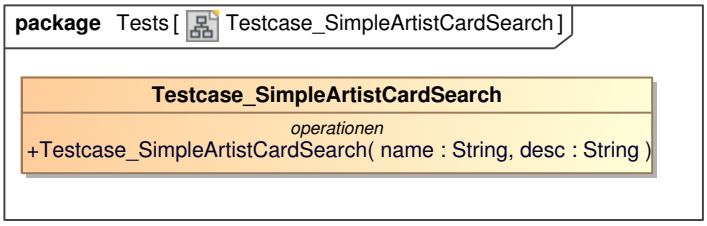

Abbildung 3.9: Klassendiagramm: Tests.Testcase_SimpleArtistCardSearch

TESTCASE_SIMPLEARTISTCARDSEARCH(NAME, DESC)
Fügt Abfragen des Testfalls zu `queries` hinzu

3.4.1.7 *Tests.Testcase_SimpleKeywordAbilitySearch*

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

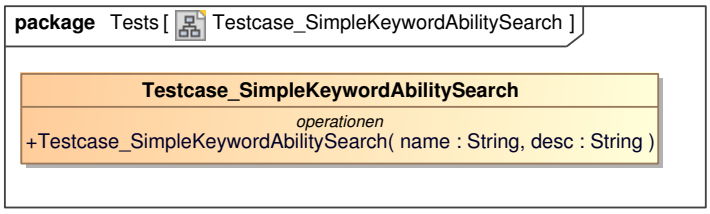

Abbildung 3.10: Klassendiagramm: Tests.Testcase_SimpleKeywordAbilitySearch

TESTCASE_SIMPLEKEYWORDABILITYSEARCH(NAME, DESC)
Fügt Abfragen des Testfalls zu `queries` hinzu

3.4.1.8 *Tests.Testcase_TopTenDecks*

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

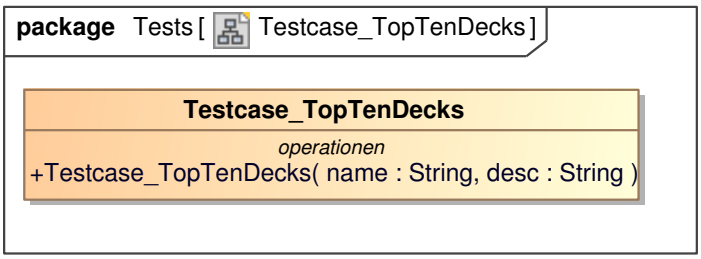

Abbildung 3.11: Klassendiagramm: Tests.Testcase_TopTenDecks

TESTCASE_TOPTENDECKS(NAME, DESC)
Fügt Abfragen des Testfalls zu `queries` hinzu

3.4.1.9 *Foundation.Config*

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

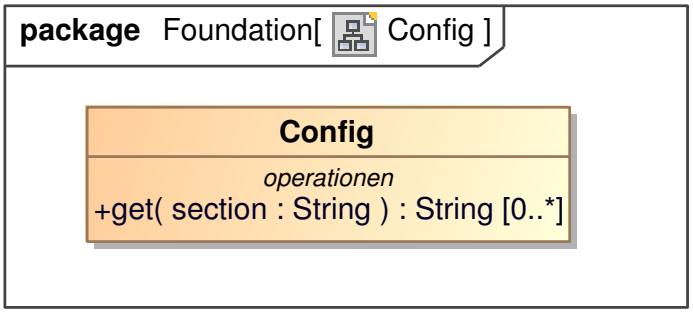

Abbildung 3.12: Klassendiagramm: Foundation.Config

CONFIG()
Lädt die Konfigurationsdatei Datei `config.ini`

GET(SECTION)
: Gibt einen Konfigurationsabschnitt `section` als `string[]` aus der Datei `config.ini` zurück

3.4.1.10 *Foundation.Database*

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

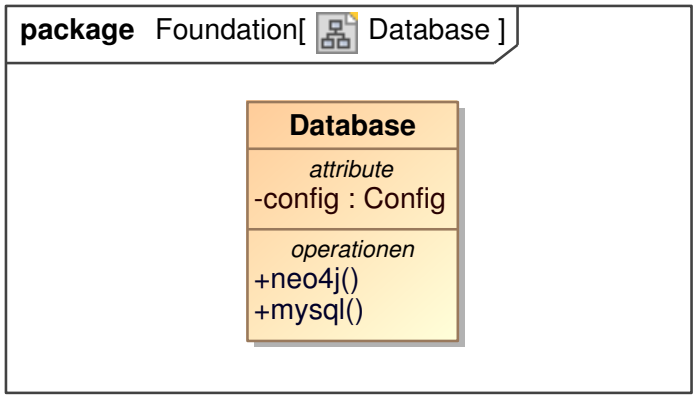

Abbildung 3.13: Klassendiagramm: Foundation.Database

DATABASE()
: 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.

3.4.1.11 *Foundation.Profiler*

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

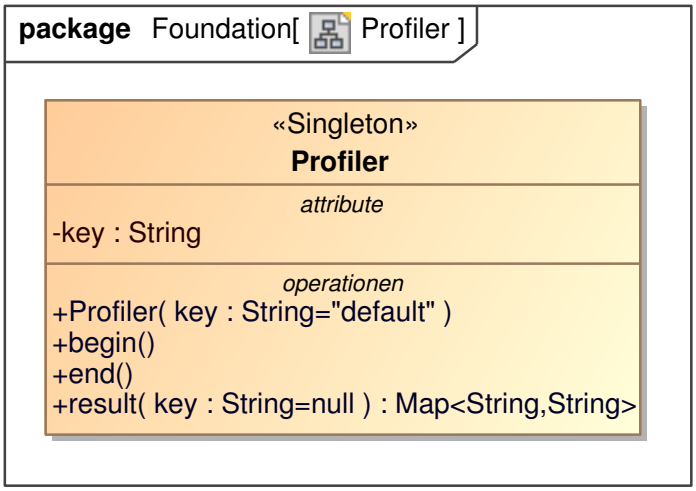

Abbildung 3.14: Klassendiagramm: Foundation.Profiler

PROFILER(KEY)
: Erstellt eine neue `Profiler` Instanz mit dem Schlüssel `key` falls noch keine Vorhanden ist

BEGIN()
: Startet den Profiler

END()
: Beendet der Profiler und speichert die gemessene Laufzeit und Speicherverbrauch

RESULT(KEY : STRING)
: Gibt das Ergebnis des Profilers `key` zurück

3.4.1.12 *Scrapers.Spider*

Die Klasse `Scrapers.Spider` hat die folgenden Schnittstellen:

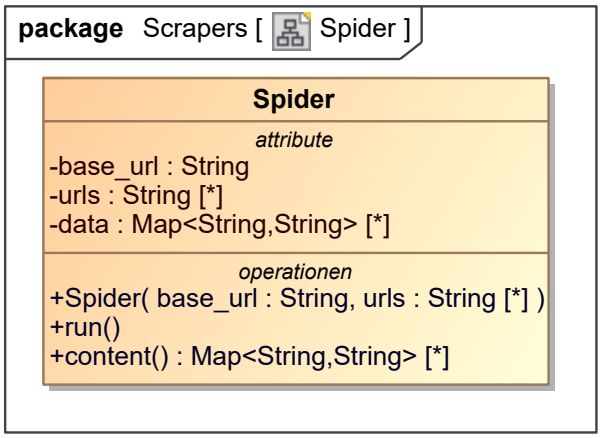

Abbildung 3.15: Klassendiagramm: Scrapers.Spider

SPIDER(BASE_URL, URLS)
: Initiiert die `Spider` Klasse

RUN
: Lädt die Decks von `urls` und speichert diese

CONTENT()
: Gibt die heruntergeladenen Decks zurück

3.4.1.13 *Scrapers.Decks*

Die Klasse `Scrapers.Decks` hat die folgenden Schnittstellen:

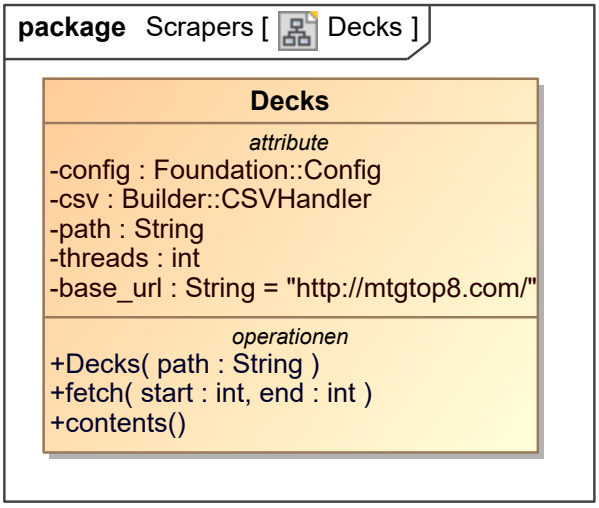

Abbildung 3.16: Klassendiagramm: Scrapers.Decks

DECKS(PATH)
: Initiiert den `CSVHandler` mit `path`

FETCH(START, END)
: Lädt die Decks per `Spider` und speichert diese per `CSVHandler` in einer `decks.csv`

CONTENTS()
: Extrahiert die Karten aus den heruntergeladenen Decks und speichert diese in eigener CSV Datei

3.4.1.14 *Transformations.Cards*

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

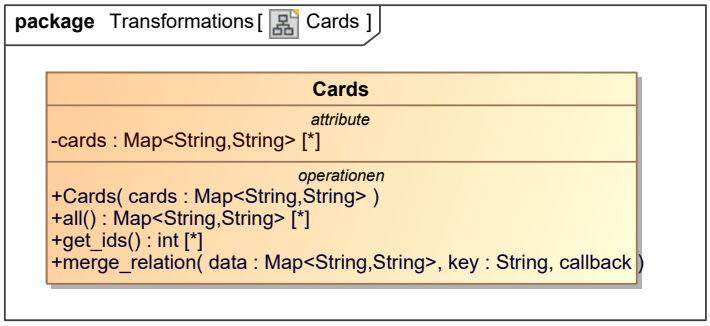

Abbildung 3.17: Klassendiagramm: Transformations.Cards

CONSTRUCTOR(CARDS)
: Speichert die Karten `cards`

ALL()
: Ausgabe aller Karten

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

MERGE_RELATION(DATA, KEY, CALLBACK)
: Fügt die Daten einer Beziehung zu den Karten-Daten hinzu. Die Liste der Argumente befindet sich in Tabelle 3.2

Tabelle 3.2: Transformations.Cards::merge_relation(data : Dictionary[], key : string, callback : Closure)

EINGABE	BESCHREIBUNG
`data : Map<String,String>`	Daten der Verknüpfung
`key : string`	Name unter dem die Verknüpfung in den Karten-Daten verfügbar sein soll
`callback : Closure`	Funktion, um Elemente der Verknpüfung zu bearbeiten bevor diese zu den Karten-Daten hinzugefügt werden

3.4.1.15 *Builder.Builder*

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

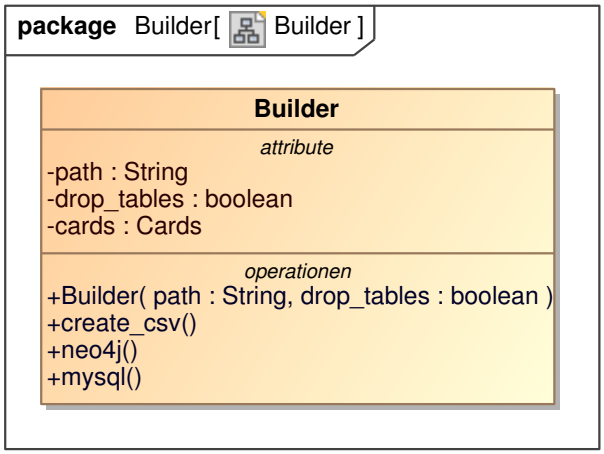

Abbildung 3.18: Klassendiagramm: Builder.Builder

CONSTRUCTOR(PATH, DROP_TABLES)
: Einrichten der Datenstruktur `Cards`, speicher des Pfades zur JavaScript Object Notation (JSON)-Datei, welche die Karten-Daten enthält

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

NEO4J()
: Erstellt und befüllt die Neo4j Datenbank mit den Daten aus den CSV-Dateien. Löscht alte Daten falls `drop_tables = True`

MYSQL()
: Erstellt und befüllt die MySQL Datenbank mit den Daten aus den CSV-Dateien. Löscht alte Daten falls `drop_tables = True`

3.4.1.16 *Builder.CSVHandler*

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

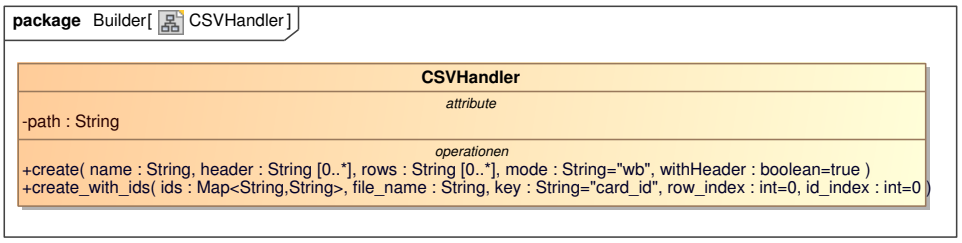

Abbildung 3.19: Klassendiagramm: Builder.CSVHandler

CSVHANDLER(PATH)
: Speichert Pfadangabe

CREATE(NAME, HEADER, ROWS, MODE="WB", WITHHEADER=TRUE)
: Erstellt eine CSV-Datei mit dem Namen `name` und dem Header `header` und befüllt diese mit den Reihen `rows`. Mit `mode` kann der Schreibmodus *(überschreiben, anhängen)* angegeben werden und mit `withHeader` ob der `header` hinzugefügt werden soll.

CREATE_WITH_IDS(IDS, FILE_NAME, KEY="CARD_ID", ROW_INDEX=0, ID_INDEX=0)
: Erstellt eine CSV-Datei mit dem Namen `file_name` welche die IDs vom Typ `key` hat. `row_index` gibt den Index der CSV Spalte an, die die ID erhält und `id_index` den Index an dessen Stelle die ID sich in `ids` befindet.

3.4.1.17 *Builder.Cards*

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

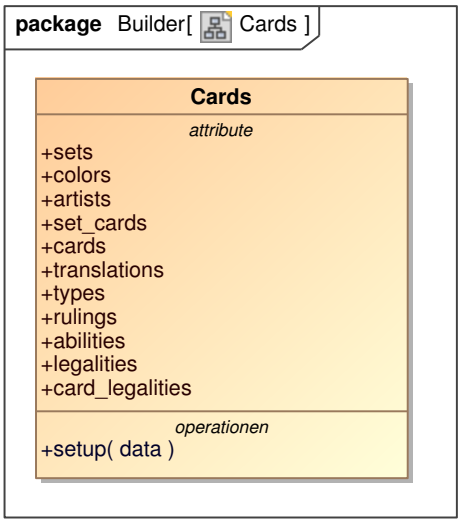

Abbildung 3.20: Klassendiagramm: Builder.Cards

CARDS()
: Erstelle Listen und Daten-Container.

SETUP(DATA)
: Bearbeitet Kartendaten `data`, so dass diese als CSV-Dateien gespeichert werden können für den späteren Datenbank-Import.

3.4.1.18 *Builder.Tournaments*

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

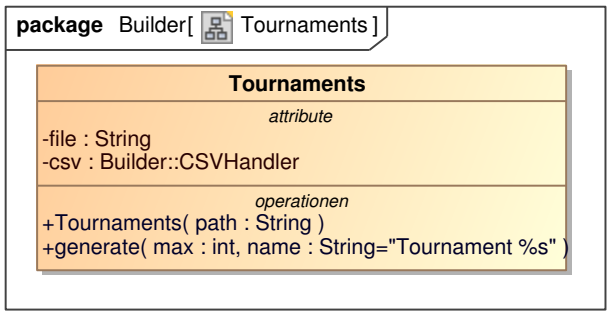

Abbildung 3.21: Klassendiagramm: Builder.Tournaments

TOURNAMENTS(PATH)
: Initiiert den `CSVHandler` mit `path`

GENERATE(MAX, NAME="TOURNAMENT %S")
: Generiere `max` Turniere mit den Namen `name`

3.4.1.19 *Builder.Scaling*

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

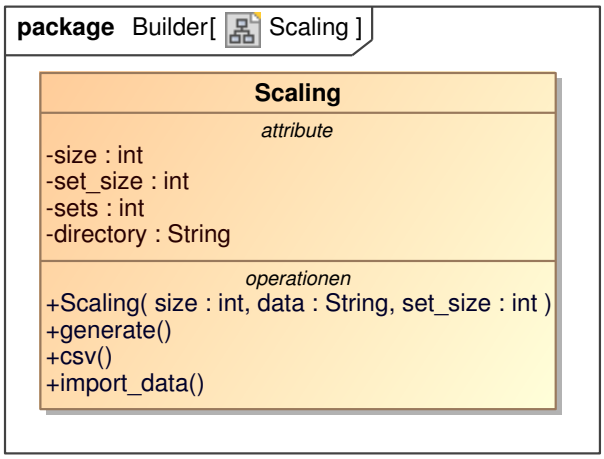

Abbildung 3.22: Klassendiagramm: Builder.Scaling

SCALING(SIZE, DATA, SET_SIZE=250)
: Berechne die Anzahl der zu generierenden Sets anhand von `size` und `set_size` und erstelle die nötigen Verzeichnisse in `data`.

GENERATE()
: Generiere Dummy-Data und speichere diese als JSON Dateien

CSV()
: Lade Dummy-Data aus JSON Dateien und speichere diese als CSV Dateien, sodass sie von der Datenbank importiert werden können

IMPORT_DATA()
: Importiere die generierten CSV in die beiden Datenbanken

3.4.1.20 *Builder.Handlers.MySQLBuilder*

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

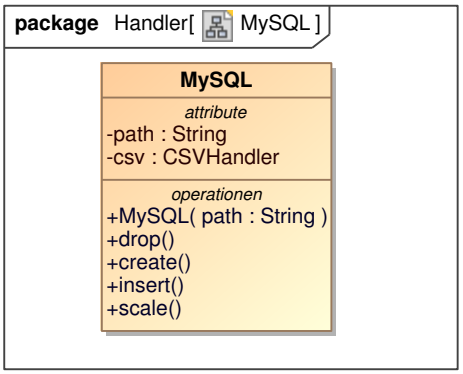

Abbildung 3.23: Klassendiagramm: Builder.Handlers.MySQLBuilder

MYSQL(PATH)
: Initiiere den `CSVHandler` mit `path` und baue die Datenbankverbindung auf

DROP()
: Löscht Tabellen

CREATE()
: Erstellt Tabellen

INSERT()
: Importiert Daten aus den CSV-Dateien

INSERT()
: Importiert Dummy-Daten aus den erstellten Skalierungs-CSV-Dateien

3.4.1.21 *Builder.Handlers.Neo4jBuilder*

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

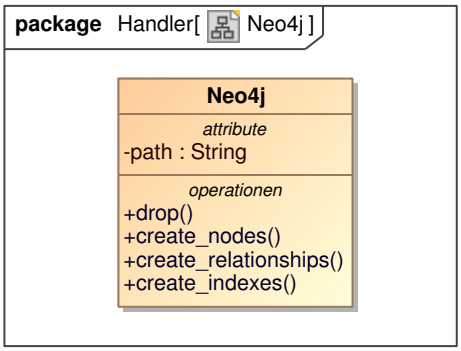

Abbildung 3.24: Klassendiagramm: Builder.Handlers.Neo4jBuilder

CONSTRUCTOR(PATH)
: Speichere die Pfadangabe `path` zu den CSV-Dateien und baue die Datenbankverbindung auf

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

3.4.1.22 *Tournaments.Matches*

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

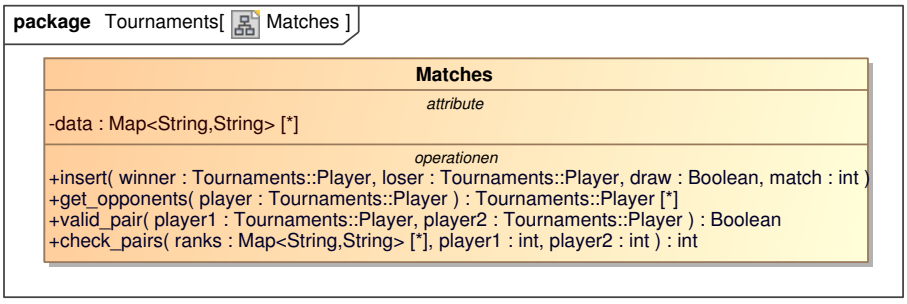

Abbildung 3.25: Klassendiagramm: Tournaments.Matches

INSERT(WINNER, LOSER, DRAW, MATCH)
: Hinzufügen eines neuen Matches

GET_OPPONENTS(PLAYER)
: Gibt eine List mit allen bisherigen Gegnern von `player` zurück

VALID_PAIR(PLAYER1, PLAYER2)
: Überprüft ob eine valide Paarung vorliegt, das heißt die beiden Spieler noch nicht gegeneinander gespielt haben

CHECK_PAIRS(RANKS, PLAYER1, PLAYER2)
: Überprüft ob eine valide Paarung vorliegt, falls nicht suche rekursiv nach einem möglichen Spieler 2 für `player1`

3.4.1.23 *Tournaments.Player*

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

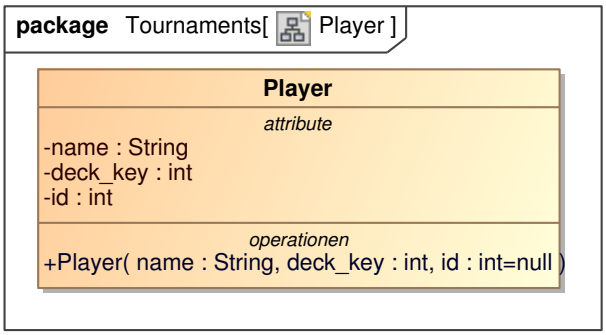

Abbildung 3.26: Klassendiagramm: Tournaments.Player

PLAYER(NAME, DECK_KEY, ID)
: Anlegen eines Spielers `name` mit der ID `id` *(falls vorhanden)* und dem Deck `deck_key`

3.4.1.24 *Tournaments.Scoreboard*

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

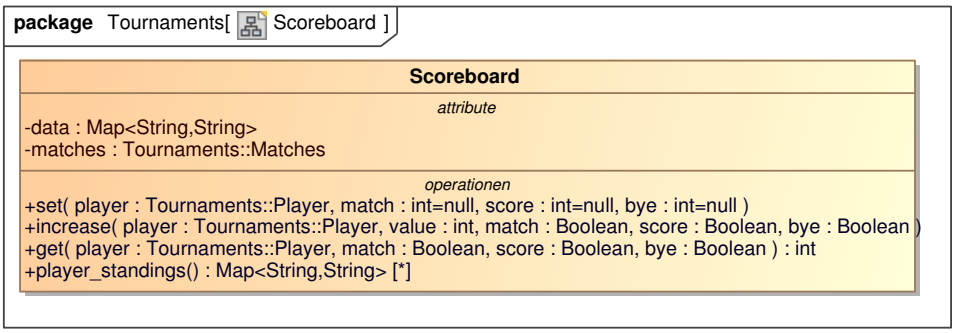

Abbildung 3.27: Klassendiagramm: Tournaments.Scoreboard

(PLAYER, SCORE, MATCH, BYE)
: Setze das Ergebnis, Match oder den Bye eines Spielers `player`

SET(PLAYER, VALUE, SCORE, MATCH, BYE)
: Erhöhe das Ergebnis, Match oder den Bye eines Spielers `player` um `value`

GET(PLAYER, SCORE, MATCH, BYE)
: Gibt entweder das Ergebnis, Match oder den Bye eines Spielers `player` zurück

PLAYER_STANDINGS()
: Gibt eine Liste der Spieler und ihrer Gewinn-Historie zurück, sortiert nach Siegen. Der erste Eintrag in der Liste sollte der Spieler an erster Stelle sein

3.4.1.25 *Tournaments.Tournament*

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

Abbildung 3.28: Klassendiagramm: Tournaments.Tournament

COUNT_PLAYERS()
: Liefert die Anzahl der Spieler des Turniers

REGISTER_PLAYER(PLAYER)
: Fügt dem Turnier einen neuen Spieler hinzu

DELETE_SCOREBOARD()
: Löscht den Spielstand und die Spielerliste

REPORT_MATCH(WINNER, LOSER, DRAW)
: Aufzeichnen der Ergebnisse eines einzelnen Spiels zwischen zwei Spielern

HAS_BYE(PLAYER)
: Prüft ob der Spieler Bye hat

REPORT_BYE(PLAYER)
: Füge dem Spieler Punkte für einen Bye hinzu

CHECK_BYES(RANKS, PLAYER)
: Überprüfe ob der Spieler schon ein Bye hat, falls ja wird der erste Spieler zurückgegeben der noch kein Bye hat

PAIRINGS()
: Gibt eine Liste von Spielernamen für die nächste Runde eines Spiels zurück. Unter der Annahme, dass es eine gerade Anzahl von registrierten Spielern gibt, erscheint jeder Spieler genau einmal in den Paarungen. Jeder Spieler wird mit einem anderen Spieler mit einem gleichen oder fast gleichen Gewinn Historie gepaart.

3.4.1.26 *Faker.ManaCostProvider*

Die Klasse `Faker.ManaCostProvider` hat die folgenden Schnittstellen:

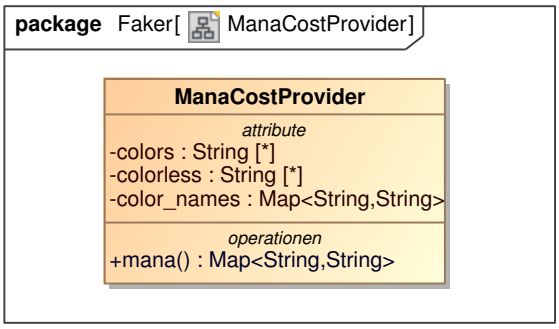

Abbildung 3.29: Klassendiagramm: Faker.ManaCostProvider

MANA()
: Liefert Mana-Kosten, Umgewandelte Manakosten und Farbe zurück

3.4.1.27 *Faker.TypeProvider*

Die Klasse `Faker.TypeProvider` hat die folgenden Schnittstellen:

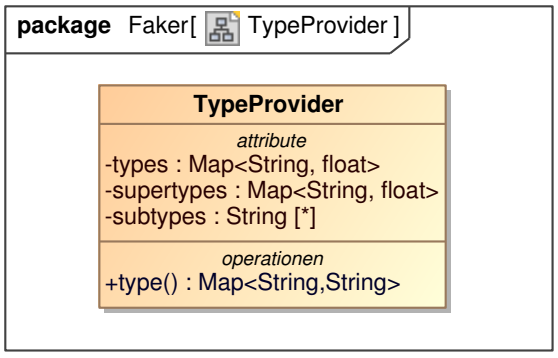

Abbildung 3.30: Klassendiagramm: Faker.TypeProvider

TYPE()
: Liefert Typ, Supertyp und Subtyp zurück

3.4.1.28 *Faker.CardProvider*

Die Klasse `Faker.CardProvider` hat die folgenden Schnittstellen:

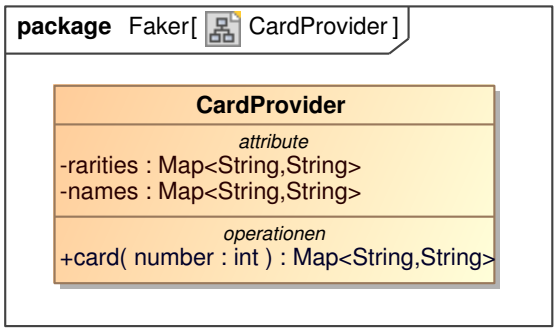

Abbildung 3.31: Klassendiagramm: Faker.CardProvider

CARD()
: Liefert Daten zu einer Karte mit Nummer `number` zurück

3.4.1.29 *Faker.SetProvider*

Die Klasse `Faker.SetProvider` hat die folgenden Schnittstellen:

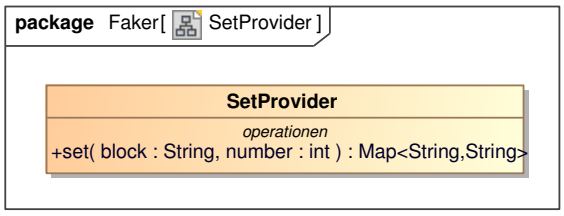

Abbildung 3.32: Klassendiagramm: Faker.SetProvider

CARD(BLOCK, NUMBER)
: Liefert ein Set im `block` Block mit `number` Karten zurück

3.4.2 *Schnittstellenrealisierung*

3.4.2.1 *Decks herunterladen*

Lade Decks von `http://mtgtop8.com` herunter, dabei ist `path` ist die Pfadangabe zum Verzeichnis in dem die Decks gespeichert werden sollen. `start` und `end` geben die Seitennummern an auf denen gestartet und gestoppt werden soll.

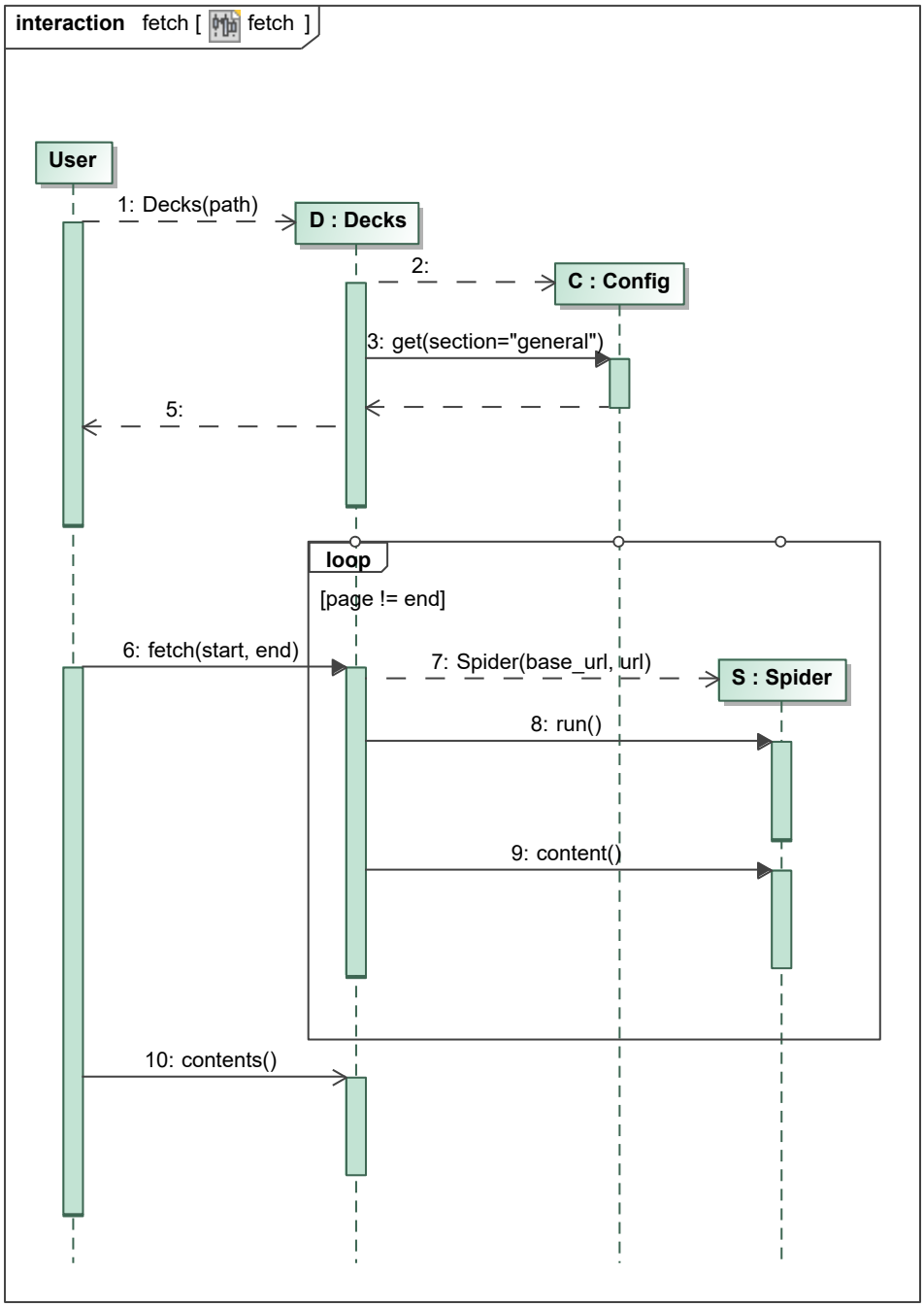

Abbildung 3.33: Sequenzdiagramm: Decks herunterladen

3.4.2.2 *Generieren der Turniere*

Generiere `n` Turniere und speichere diese als `tournaments.csv` in dem Verzeichnis unter `path`.

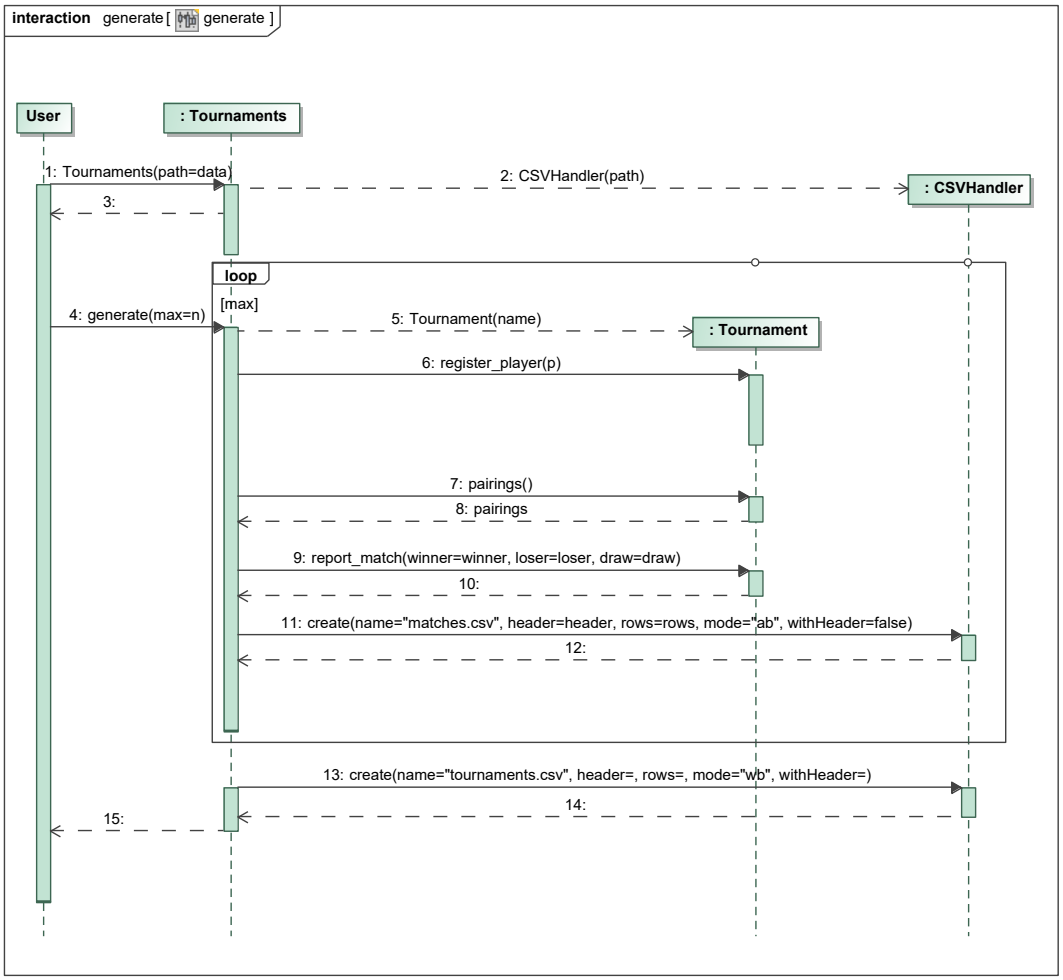

Abbildung 3.34: Sequenzdiagramm: Generierung der Turniere

3.4.2.3 *Importieren der Daten*

Erstellen des Datenbankschemas und importieren der Daten.

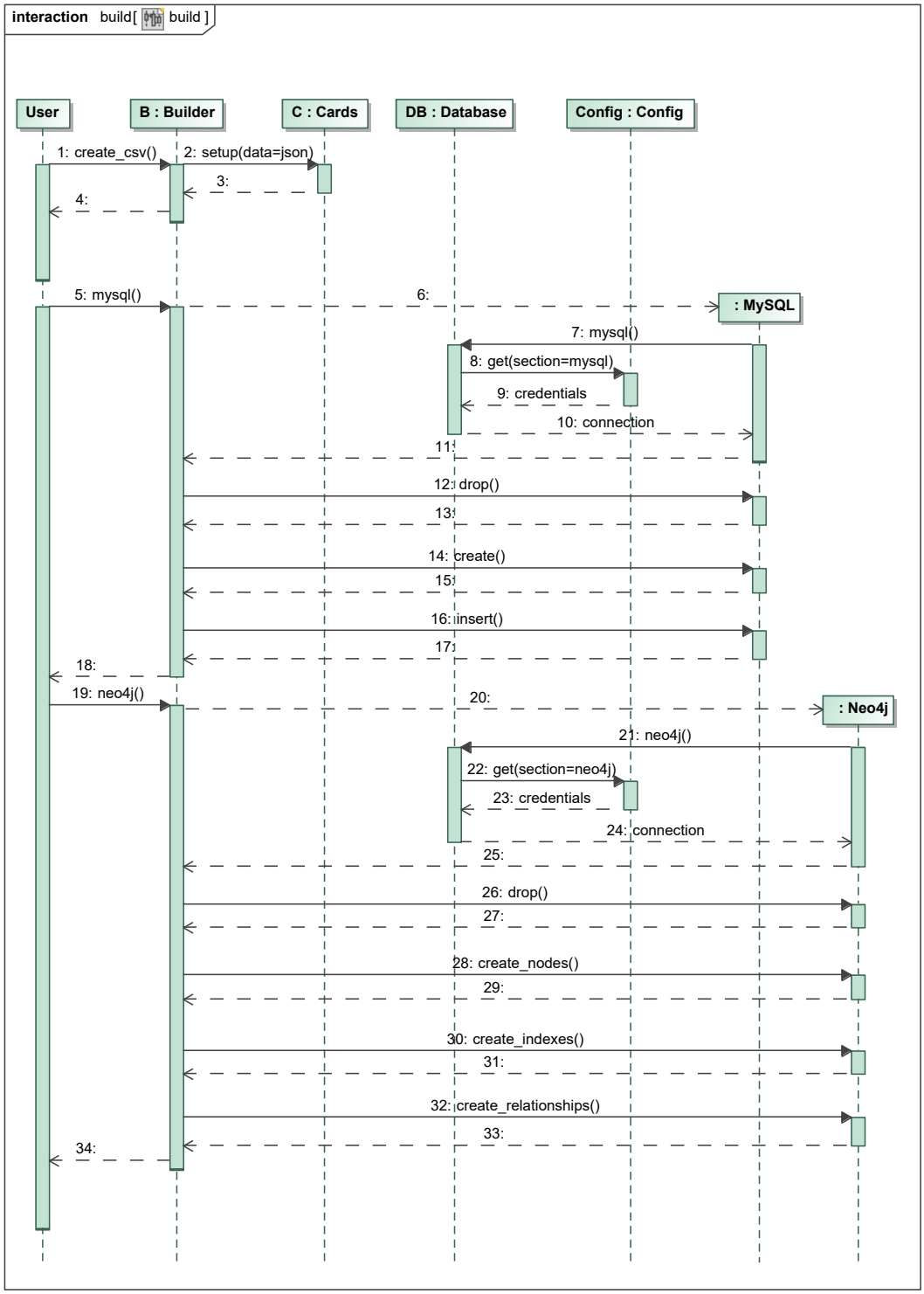

Abbildung 3.35: Sequenzdiagramm: Erstellen der Datenbank und Import des Datensatzes

3.4.2.4 *Ausführen eines Tests*

Ausführen des Testfalls mit Namen `test`. Falls `queries == True` sein sollte, wird nicht der Testfall ausgeführt sondern die verwendeten SQL- und Cypher-Abfragen auf der Konsole ausgegeben.

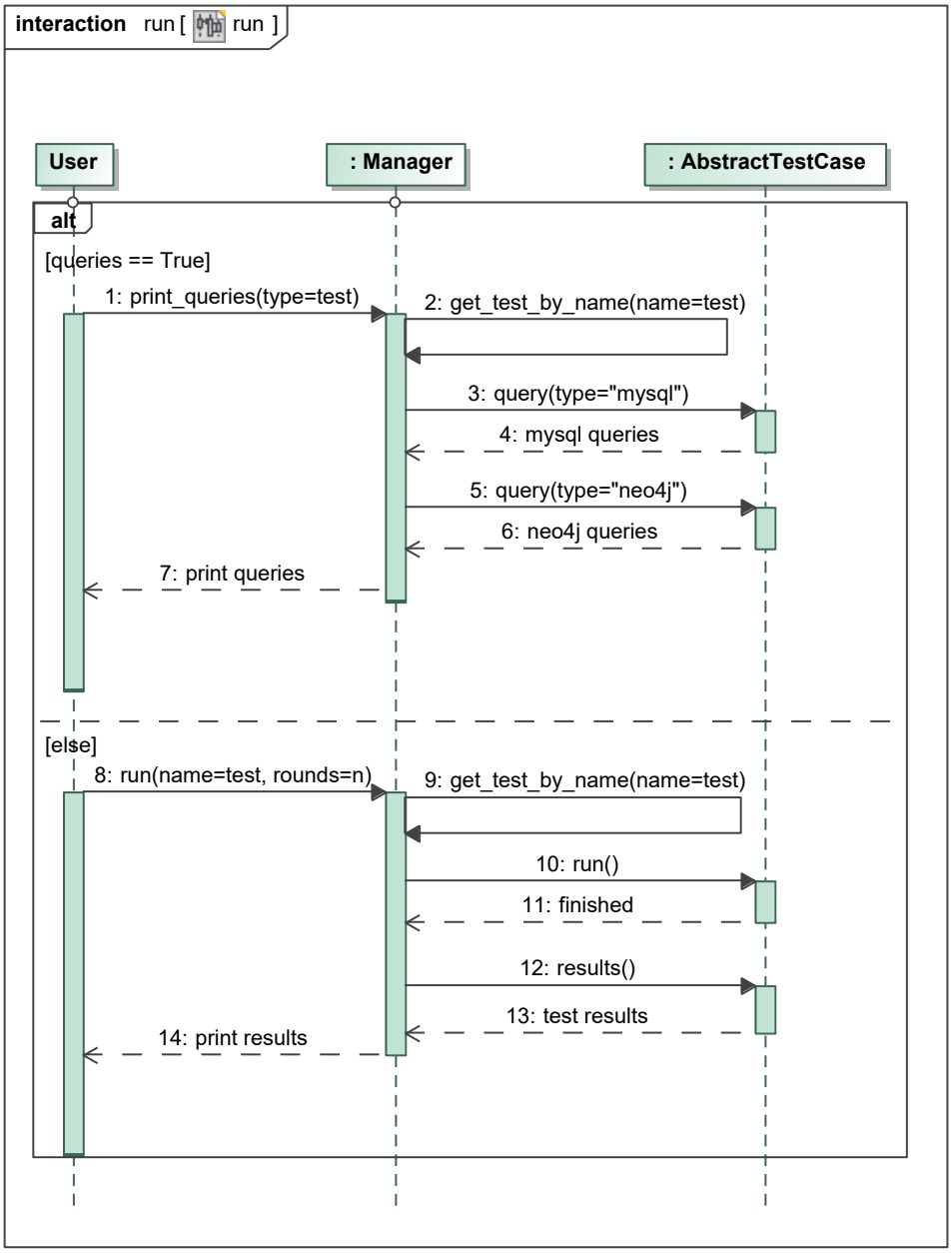

Abbildung 3.36: Sequenzdiagramm: Ausführen eines Test

3.4.2.5 *Skalierung*

Generiert und importiert Dummy-Daten, um die Skalierbarkeit der einzelnen Testfälle zu testen.

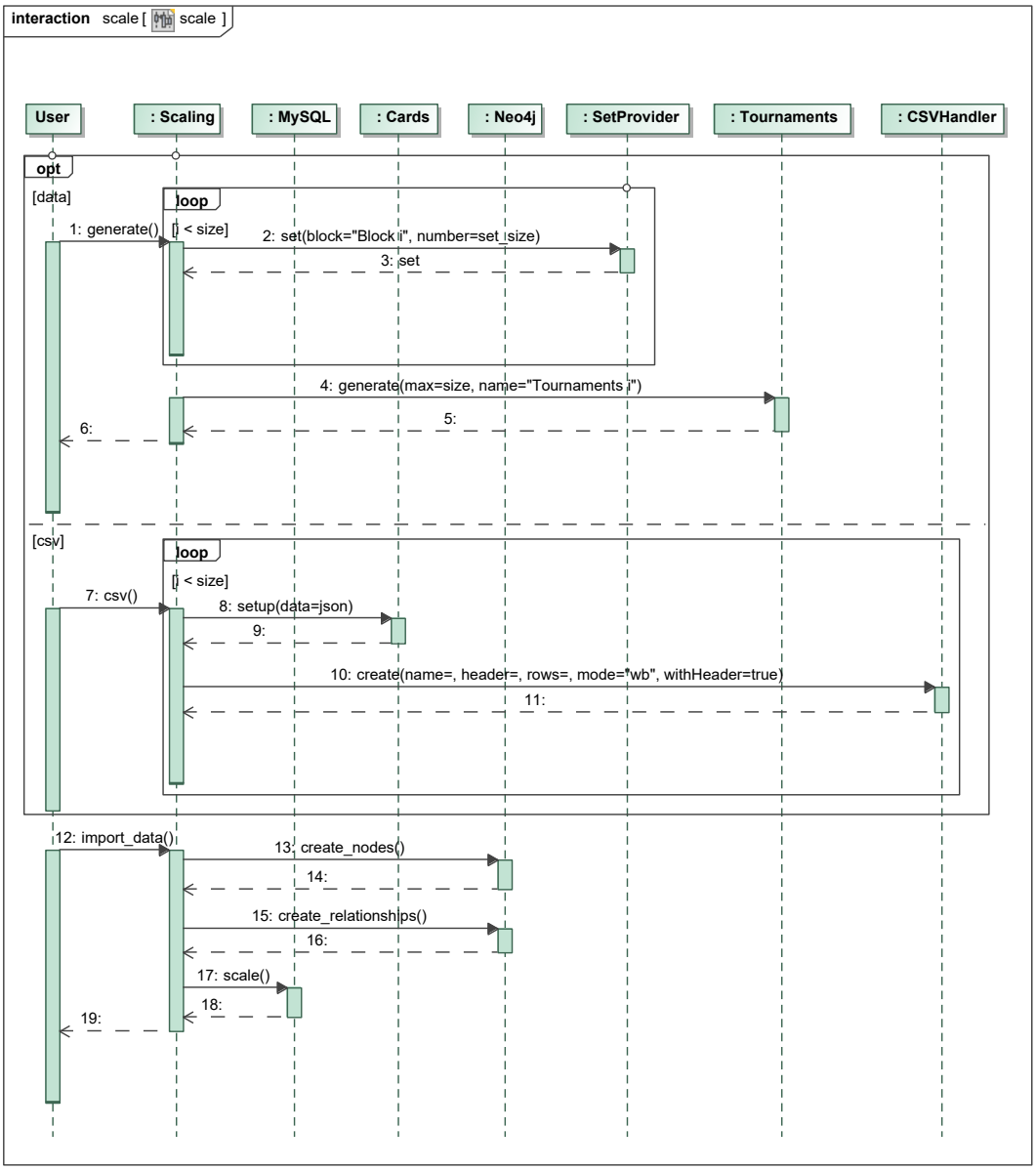

Abbildung 3.37: Sequenzdiagramm: Skalierung

3.4.2.6 *Auflisten der vorhandenen Testfälle*

Listet die vorhandenen Testfälle mit Namen und Beschreibung auf und gibt diese auf der Konsole aus.

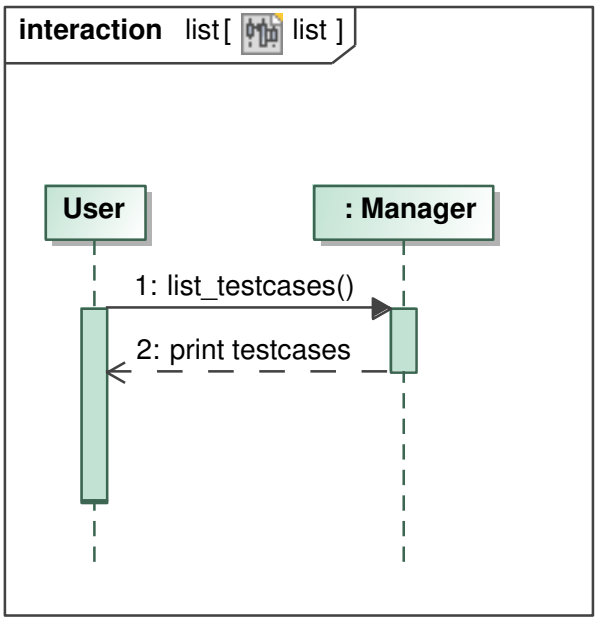

Abbildung 3.38: Sequenzdiagramm: Auflisten der vorhandenen Testfälle

3.5 AUSGEWÄHLTER ANSATZ UND DETAILLÖSUNGEN

3.5.1 *Datenbasis*

Als Basis der Kartendaten wird mtgjson[3] benutzt, welche MtG Karten und Editions-Daten als JSON-Datei bereitstellt. Die Decks hingegen werden von mtgtop8[4] bezogen.

3.5.2 *Datenbanken*

Als Graph-Datenbank wird *Neo4j* Version 3.1 eingesetzt, da diese eine hoch skalierbare Datenbank ist, welche auf allen gängigen Betriebssystemen läuft [8]. 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 [15].

Als RDBMS wird MariaDB[5] Version 10.1 eingesetzt, welche vom Urheber von MySQL entwickelt wird. MariaDB ist ein Fork von MySQL und gilt als dessen evolutionäre Nachfolger [2].

3.5.2.1 *Import der Daten*

Da eine große Menge an Daten importiert werden müssen, zum Beispiel über 100.000 Übersetzungen, ist eine schnelle Import-Funktion hilfreich. Die

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

```
1 USING PERIODIC COMMIT 1000
2 LOAD CSV WITH HEADERS FROM "file:translations.csv" AS row
3 CREATE (:Translation { name: row.name });
4
5 USING PERIODIC COMMIT
6 LOAD CSV WITH HEADERS FROM "file:%s/translations.csv" AS row
7 MATCH (c:Card { name: row.card })
8 MATCH (t:Translation { name: row.name })
9 CREATE (c)-[:HAS_TRANSLATION { lang: row.language }]->(t);
```

Listing 2: Importieren von Übersetzungen in Neo4j

```
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 in MySQL/MariaDB

schnellste und einfachste Möglichkeit dafür, welche beide Systeme unterstützen, ist die Daten als CSV-Dateien zu importieren. In Listing 2 befindet sich ein Kommando, um Übersetzungen nach Neo4j zu importieren und in Listing 3 das Äquivalent für den Import in MySQL/MariaDB.

3.5.3 *Programmiersprache*

Als Programmiersprache wird *PyPy*[6], eine alternative *Python* Implementierung, die eine hohe Kompatibilität zu Python hat, verwendet. Die Vorteile von PyPy sind der integrierte Just-in-time (JIT) Compiler, wodurch sich eine schnellere Ausführungszeit ergibt, und der geringere Speicherverbrauch [6]. Als Version wird *PyPy2.7 v5.6.0* verwendet, welches der Python Version 2.7 entspricht.

3.5.4 *Betriebssystem*

Die Tests wurden auf einem Ubuntu 14.04 Server mit 5120 MB Arbeitsspeicher und 2 CPU Kernen getestet. Die zur Verfügung stehenden CPU Kerne stammen von einem AMD FX-4100 Prozessor, welcher mit 3.5GHz getaktet ist.

6 http://pypy.org

3.5.5 *Schweizer System*

Aufgrund der Tatsache, dass keine Turnier Daten zur Verfügung stehen, müssen die Turniere nach dem Schweizer System simuliert werden, um einen geeigneten Datensatz zu erstellen. Als Grundlage hierfür wird das Projekt *Swiss Tournament Pairing System - Relational Databases*[7] von Ben Brandt verwendet. Hierbei handelt es sich um ein Schweizer Turnier System für MtG, welches in Python geschrieben ist.

3.5.5.1 *Profiling*

Mit einem *Profiler* können verschiedene Werte wie *Ausführungszeit* oder *Speicherverbrauch* gemessen werden [6]. Um vergleichen zu können, wie gut die Datenbanksysteme in den einzelnen Testfällen abschneiden, werden die Ausführungszeit und der Speicherverbrauch gemessen.

Für die Ausführungszeit wird eine einfache Timer-Klasse[8] benutzt, welche die Zeit mittels der `datetime` Bibliothek misst. Um den Speicherverbrauch zu messen wird die Python-Bibliothek `psutil` benutzt[9], da diese eine einfache und plattformübergreifende Methode bietet den Speicherverbrauch auszulesen.

7 https://benjaminbrandt.com/relational-databases-final-project/
8 https://www.huyng.com/posts/python-performance-analysis
9 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/MariaDB 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 Fehlergrenzen sind.

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 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/MariaDB 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.

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 mit etwas über 6500 Decks, 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

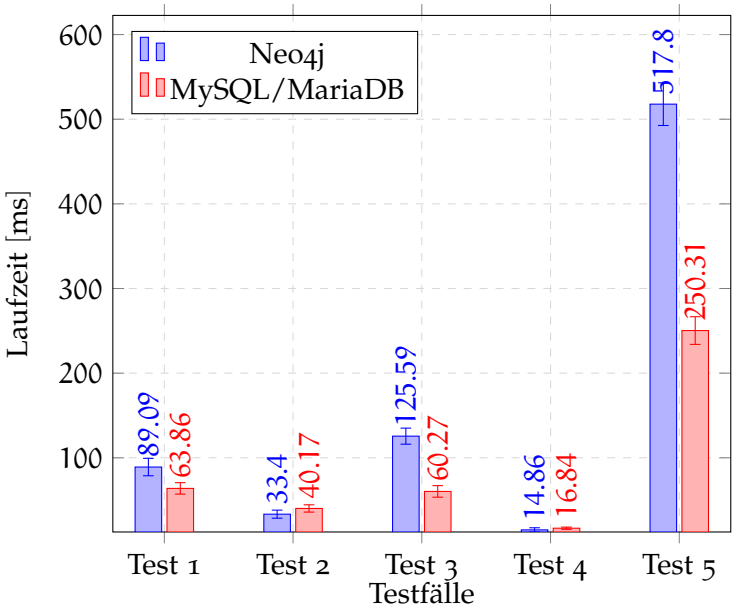

Abbildung 4.1: Ausführungszeit der einzelnen Testfälle

Anzahl der gespielten Partien. Insgesamt gibt es im Standard-Testfall 1000 Turniere mit etwas über 6500 Decks, 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 ist Neo4j in Testfall 4 schneller als die SQL-Abfrage mit `JOINS`. Überraschenderweise war Neo4j auch in Testfall 2 etwas schneller als MySQL/MariaDB, obwohl aufgrund der langjährigen Optimierungen der Textsuche in MySQL/MariaDB zu erwarten war, dass diese schneller sei. Der Grund dafür, dass Neo4j in den Testfällen 1 und 3 langsamer ist als MySQL/MariaDB liegt vermutlich daran, dass als Datentyp Arrays für *color*, *abilities* und *types* benutzt wurden. Anscheinend ist in Neo4j die Suche innerhalb eines Arrays langsamer als die Suche mit einem `JOIN` in SQL.

4.3.2 *Skalierbarkeit*

Wie in Abbildung 4.2 zu sehen ist, skalieren sowohl Neo4j als auch MySQL/MariaDB gut, aber auch hier zeigt sich, dass Neo4j in allen gemessenen Punkten 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.

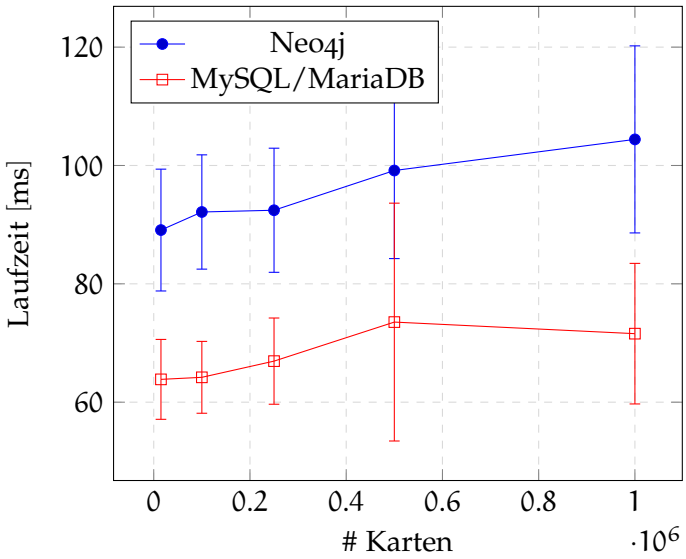

Abbildung 4.2: Skalierung Testfall 1

In Testfall 2 bestätigt sich, dass MySQL/MariaDB 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/MariaDB nur um 5% - 10% pro gemessenem Punkt steigt. MySQL/MariaDB ist also wie erwartet besser für Textsuchen geeignet als Neo4j.

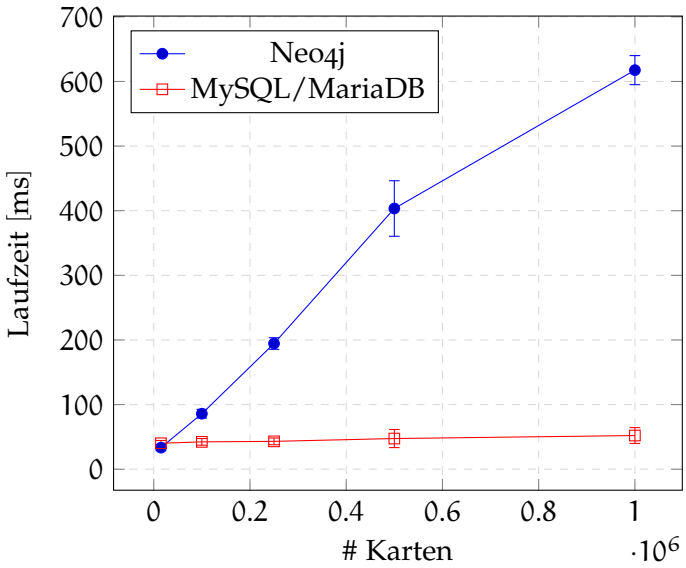

Abbildung 4.3: Skalierung Testfall 2

Wie in Abbildung 4.4 zu sehen ist, hat die Laufzeit bei Neo4j ein lineares Wachstum, wohingegen sie bei MySQL/MariaDB relativ konstant bleibt: bei doppelter Menge an Karten, verdoppelt sich *(ungefähr)* die Laufzeit in Neo4j.

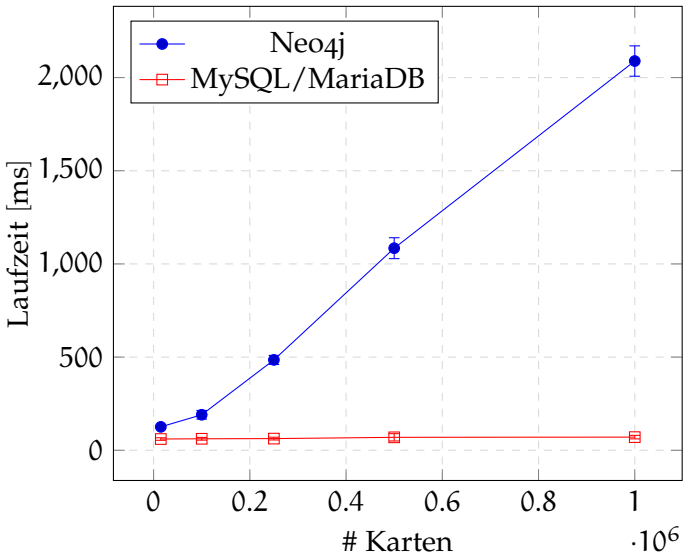

Abbildung 4.4: Skalierung Testfall 3

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

In Abbildung 4.5 zeigt sich, dass die Laufzeit der SQL-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/MariaDB.

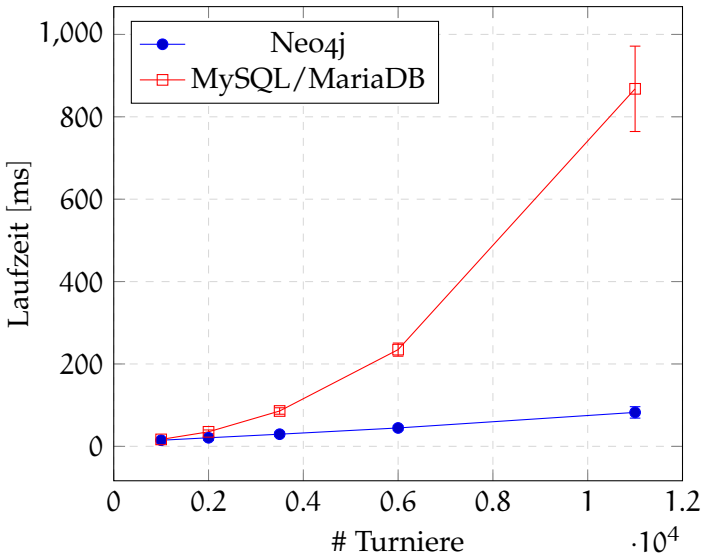

Abbildung 4.5: Skalierung Testfall 4

Allerdings gilt dies nicht für Testfall 5, denn Abbildung 4.6 zeigt, dass hier Neo4j langsamer ist als MySQL/MariaDB. Sowohl MySQL/MariaDB 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

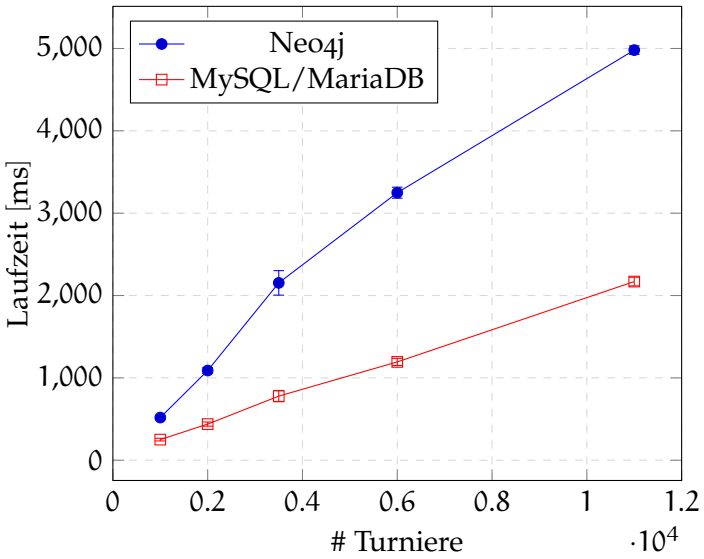

Abbildung 4.6: Skalierung Testfall 5

Wachstum als MySQL/MariaDB. Die Tatsache, dass beide Datenbanken bei den Abfragen deutlich langsamer sind, ist in der Praxis 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/MariaDB und Neo4j faktisch nicht, das heißt er beträgt weniger als 0.1%. Des Weiteren bleibt dieser auch bei der Skalierung der Datenbank relativ konstant mit ±1MByte Abweichung. Aus diesem Grund wird hier nicht weiter auf den Speicherverbrauch eingegangen.

5 ZUSAMMENFASSUNG UND AUSBLICK

5.1 ZUSAMMENFASSUNG

Die anfängliche Vermutung, dass eine Graphdatenbank besser für die Matchup-Analyse eines Decks geeignet ist, hat sich, wie im Testfall 4 zu sehen, bestätigt. Bei der ursprünglichen Datenmenge von 1000 Turnieren unterschied sich die Laufzeit bei beiden Systemen nur minimal. Bei der Skalierung aber wies die Laufzeit der Neo4j Abfragen nur ein leichtes lineares Wachstum auf, wohingegen die Laufzeit bei MySQL/MariaDB ein exponentielles Wachstum hatte.

Bei der Berechnung der Top 10 Decks war eigentlich davon auszugehen, dass hier die Graphdatenbank wieder besser abschneiden würde. Der Grund für die Vermutung war, dass hier wieder Matchups berechnet werden müssen und dies bei dem vorherigen Testfall für die Matchup Analyse der Fall war. Allerdings ergab sich in diesem Testfall, dass Neo4j eine längere Laufzeit hatte als MySQL/MariaDB. Beide Datenbanksysteme hatten in diesem Fall ein lineares Wachstum, aber bei Neo4j war die Steigung größer als bei MySQL/MariaDB. Ein Grund für das unerwartete Ergebnis könnte sein, dass die genutzte Cypher Abfrage nicht die beste war und es noch einen besseren Weg gibt.

Wie zu erwarten war, skalierte die Textsuche mittels `LIKE` bei SQL besser als das äquivalent `CONTAINS` bei Cypher. Während die Laufzeit bei MySQL/-MariaDB kaum bis gar nicht stieg, stieg sie pro gemessenem Punkt in Neo4j um den Faktor 1.5 bis 2.0 - hatte also ein starkes lineares Wachstum. Was allerdings unerwartet war, war das Neo4j beim ursprünglichen Datensatz sogar leicht schneller war als die MySQL/MariaDB Textsuche.

Bei *Testfall 3*, wo es um Abfragen mit vielen Verknüpfungen ging, kam es zu zwei unerwarteten Ergebnissen. Eigentlich war davon auszugehen, dass Neo4j hier sehr gut skaliert wohingegen MySQL/MariaDB, aufgrund der vielen `JOINS`, bei der Skalierung einen großen Wachstumsfaktor in der Laufzeit aufweisen würde. Nichtsdestotrotz trat genau der entgegengesetzte Fall ein: Die Laufzeit bei MySQL/MariaDB blieb relativ konstant ungeachtet der Anzahl an Karten und Neo4j skalierte schlecht und war in jedem Messpunkt langsamer als MySQL. Ein möglicher Grund könnte sein, dass in dem gewählten Graph-Schema die Karten-Typen, Fähigkeiten und Farben als Array gespeichert wurden. Dadurch wurden zwar zwei Abfragen weniger gebraucht als bei MySQL/MariaDB, aber anscheinend war dies der Grund für die schlechte Skalierung von Neo4j.

In dem Test mit den *Schlüsselwort-Fähigkeiten* skalierten beide Datenbanksysteme ungefähr gleich gut. Dies lag vermutlich daran, dass die Anzahl der Fähigkeiten sich bei der Skalierung nicht änderte. Des Weiteren war aber

auch hier Neo4j langsamer als MySQL/MariaDB. Der Grund dafür liegt vermutlich an der Tatsache, dass wie bei dem vorherigen Testfall mit den Verknüpfungen für Fähigkeiten Arrays benutzt wurden, denn dies ist den beiden Testfällen gemein.

Es hat sich also gezeigt, dass Neo4j, entgegen der anfänglichen Annahmen, nicht in allen Punkten besser war als eine relationale Datenbank wie MySQL/MariaDB, um Sammelkarten effizient zu speichern und zu analysieren. Die einzige Ausnahme war im Testfall der *Matchup Analyse*, hier lag Neo4j deutlich vor MySQL/MariaDB was die Skalierbarkeit anbelangt. Für Kartenspiele, deren Turniere nach dem Schweizer System ablaufen, eignet sich also Neo4j eher als MySQL/MariaDB. Insgesamt waren aber die Abfragen mit Cypher sowohl kürzer und damit auch lesbarer als die entsprechenden Abfragen in SQL, was ein Vorteil von Neo4j ist. Entgegengesetzt der anfänglichen Annahme unterschied sich der Speicherverbrauch der beiden Systeme nur um maximal 0.1% und war somit irrelevant.

5.2 AUSBLICK

Wie zuvor schon erwähnt tauchten bei Neo4j unerwartete Ergebnisse in Zusammenhang mit der Benutzung von Arrays auf. Weiterführend bietet sich also an andere Daten-Schemata zu verwenden, welche keine Arrays nutzen, um zu überprüfen, ob Arrays der Grund für die schlechte Performance sind. Es bietet sich weiterhin an dabei zu untersuchen, wie groß der Einfluss bei den einzelnen Attributen *(colors, abilities und types)* ist. Aufgrund der Tatsache, dass hier 3 weitere Schemata *(für colors, abilities und types)* getestet werden müssten, konnte dies aus zeitlichen Gründen nicht untersucht werden.

Es wäre interessant zu untersuchen, warum die Cypher Abfrage im *Top 10 Decks Testfall* langsamer ist als die SQL-Abfrage und auch schlechter skaliert. Ein möglicher Grund könnte sein, dass die Abfrage nicht optimal ist, was sehr wahrscheinlich ist, und es einen besseren Weg gibt die Abfrage zu schreiben. Ein weiterer Grund könnte sein, dass das gewählte Graph-Schema für diese Art von Abfragen nicht geeignet ist und ein anderes Schema besser geeignet wäre. Sofern man das Daten-Schema ändert, muss außerdem der Testfall 4 *(Matchup-Analyse)* erneut getestet werden, da es sein könnte, da sich hier die Laufzeiten ebenfalls positiv oder negativ ändern können. Die Abfrageoptimierung ist jedoch ein kompliziertes Thema und erfordert oft einiges an Erfahrung mit der Abfrage-Sprache. Aus zeitlichen Gründen und fehlender Erfahrung war es daher nicht möglich, die Abfrage zu optimieren oder ein anderes Schema zu testen.

Ein weiterer möglicher Testfall wäre jener, der sich auf die Deck-Analyse beziehen, zum Beispiel die Verteilung der verschiedenen Karten-Typen oder Farben in einem Deck angeben. Des Weiteren könnte untersucht werden wie gut die beiden Datenbanken sind, wenn nach den meist verwendeten Karten *(gruppiert nach Farben)*, welche in Decks verwendet werden, gesucht wird. Bei diesem Testfall würden sowohl der Turnier als auch der Karten

Teil aus der Datenbank zusammen abgefragt werden, was bei den anderen Testfällen nicht der Fall war. Jedoch war nicht genügend Zeit, um diese und andere Testfälle zu untersuchen.

In dieser Arbeit wurde lediglich das TCG Magic: the Gathering getestet. MtG ist aber nicht das einzige TCG, sondern nur mit das älteste und daher umfangreichste. Bei anderen Kartenspielen können die Karten durchaus weniger komplex sein und damit weniger Verknüpfungen aufweisen. Folglich würde sich in einer späteren Arbeit anbietet zu untersuchen, ob die Ergebnisse auch mit anderen Kartenspielen wie Pokemon, Yu-Gi-Oh! oder Hearthstone reproduzierbar sind.

[1] Tony Agresta. *The Hype Around Graph Databases And Why It Matters*. 2016. URL: http://www.forbes.com/sites/ciocentral/2015/04/06/the-hype-around-graph-databases-and-why-it-matters.

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

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

[4] Wizards of the Coast. *Spielinfos/Gameplay Stelle Dein Eigenes Magic-Deck Zusammen*. 2016. URL: http://magic.wizards.com/de/gameplay/how-to-build-a-deck.

[5] Wizards of the Coast. *What are Swiss Pairings?* 2016. URL: http://magic.wizards.com/en/game-info/products/magic-online/swiss-pairings.

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

[7] Dustin Fink, Benjamin Pastel und Neil Sapra. *Predicting the Strength of Magic: The Gathering Cards From Card Mechanics*. CS 229 Machine Learning Final Projects. Stanford University, 2015.

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

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

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

[11] 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.

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

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

[14] Justin J Miller. "Graph Database Applications and Concepts with Neo4j". In: *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*. Bd. 2324. 2013.

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

[16] Matthew Pawlicki, Joseph Polin und Jesse Zhang. *Prediction of Price Increase for Magic: The Gathering Cards*. CS 229 Machine Learning Final Projects. Stanford University, 2014.

[17] Boris A Perkhounkov, Cooper Gates Frye und Emily Margaret Franklin. *Financial Magic*. CS 229 Machine Learning Final Projects. Stanford University, 2015.

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

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

[20] 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.

[21] 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.