Moderne Datenbanken
Einführung verschiedener Datenbanksysteme

Dokumentation der Gruppenaufgaben

Gruppe 1

Hochschule: IT-Center Dortmund

Prüfungsteilnehmer:

Nils Thorben Konopka
Wittener Straße 77
44149 Dortmund

Lukas Meier
Vormholzer Ring 50
58456 Witten

Ivan Rodrigo Quiroz Galarza
Ernst-Waldschmidt-Straße 7
44538 Lünen

Deutschland

05.04.2024 - 22.04.2024

Inhaltsverzeichnis

1 Redis

Redis ist eine In-Memory-Datenbank, welche Daten in Form von von Key-Value-Paaren ablegt. Unterstützt werden dabei unter Anderem die Datentypen Strings, Listen, Sets, Hashes und sortierte Sets.
Als In-Memory-Datenbank hält Redis die Daten im Arbeitsspeicher und persistiert nur in gewissen Zeitabständen, oder bei einem gezielten Aufruf der enstprechenden Funktion, auf die Festplatte. Aufgrund dieses Vorgehens verfügt Redis über eine hohe Geschwindigkeit beim Datenzugriff. Redis kann damit besonders dort glänzen, wo schnelle Lese- und Schreiboperationen vonnöten sind. Außerdem ist Redis durch die Key-Value-Architektur auch schemafrei. Allerdings benötigt Redis dementsprechend ausreichend Arbeitsspeicher und bei Ausfällen besteht ein hohes Risiko des Datenverlusts. Zudem müssen Dinge wie z.B. die Sicherstellung von Datenkonsistenz von einer übergeordneten Anwendung übernommen werden.

1.1 Aufgabenstellung

Im Rahmen der Veranstaltung sollte ein Teilbereich, z.B. ein View, der im vorrigen Semester entworfenen relationalen Datenbank zur Verwaltung von Stundenplänen stattdessen in Redis umgesetzt werden. Darüber hinaus sollte die bestehende Lösung eine Erweiterung erhalten, welche sich zur Veranschaulichung der Vorteile von Redis eignet.

1.2 Umsetzung des Dozenten-Views

Zur Umsetzung in Redis wurde der Dozenten-View aus der relationalen Datenbank gewählt. Die Auswahl erfolgte, weil er semantisch gut abgrenzbar ist.
Somit musste ein Weg gefunden werden, die in dieser SQL-Abfrage enthaltenen Daten in Redis abzubilden:

```
USE ‘Stundenplan‘;
CREATE VIEW dozentView AS
SELECT modul.name AS Modul, veranstaltung.veranstaltungid,
    veranstaltung.typ,
dozent.name AS Dozent, termin.datum, termin.beginn AS Start, termin
   .ende AS Ende FROM Termin termin
JOIN Veranstaltung veranstaltung ON veranstaltung.veranstaltungId
    = termin.veranstaltungId
JOIN Dozent dozent ON dozent.dozentId = veranstaltung.dozentId
JOIN Modul modul ON modul.modulId = veranstaltung.modulId;
```

Dieser View enthält alle Informationen, die mit einem Termin zusammenhängen. Die Termin-Entität ist die zentrale Entität der relationalen Datenbank, enthält jedoch selbst zunächst kaum semantische Informationen. Über die Joins werden die Tupel so zusammengesetzt, dass für jeden Termin ein gesammter Datensatz entsteht.

Da Redis auf Key-Value-Paaren aufbaut und keine komplexen Abfragen unterstützt und zudem Redundanzen im Gegensatz zu relationalen Datenmodellen hier kein Problem darstellen, wurde entschieden, dass der komplette View als Menge von Hashes in Redis dargestellt werden kann. Als Identifikator erhält jeder Hash eine Kombination aus einem Modulkürzel, dem Jahrgang und dem Datum, z.B. `iba23-011023` für den Termin zu Internetbasierte Anwendungen des Jahrgangs 23 am 1. Oktober 2023. Ein ganzer Datensatz würde dann so eingefügt werden:

```
HSET iba23-011023
id "iba23-011023"
dozentName "Anna␣Müller"
veranstaltungTyp "Vorlesungen"
semester "WS2023/2024"
modulName "Internetbasierte␣Anwendungen"
datum "01.10.2023"
beginn "08:00"
ende "10:00"
teilnehmer "iba23T"
jahrgang 23
```

Damit sind alle Informationen des Views in einem Hash abgebildet. Der Key `teilnehmer` ist hierbei dafür da, um über den ursprünglichen View hinausgehend auch noch die Teilnehmerliste nachzuhalten. Dies wurde in der relationalen Datenbank von einer stored function erledigt. Der Value gibt den Bezeichner des Hashes an, in welchem die Teilnehmerliste für diese Veranstaltung hinterlegt ist und besteht aus dem Modulkürzel, an welches ein T angehangen wird. So sieht eine solche Teilnehmerliste aus:

```
HSET iba23T
id "iba23T"
"Bubi␣Blauschuh" "Krank"
"Thomas␣Koenigsmann" "Krank"
"Maria␣Mandarina" "Entschuldigt"
"Katrin␣Kleeblatt" "unentschuldigt"
```

So kann nicht nur der View umgesetzt, sondern auch Informationen nachgehalten werden, die ursprünlich eng mit diesem verwoben waren.

Zur besseren Handhabung dieser neuen Lösung wurde ein Hash mit den Metadaten eingebaut, welcher das Kürzelschema zu der Datenbank enthält, falls man es nachschlagen muss:

```
HSET meta
uebung "<modulkuerzel>U<datum>"
teilnehmerliste "<modulkuerzel>T"
abfrage "<modulkuerzel>Abfrage"
```

Hier sieht man zum Beispiel, dass die Teilnehmerliste zu einer Veanstaltung immer in einem Hash finden kann, welcher aus dem jeweiligen Modulkürzel gefolgt von einem T besteht.

Die Modulkürzel wiederum sind in der Modulkürzel-Map gespeichert:

```
HSET modulkuerzel
"Internetbasierte␣Anwendungen" "iba"
```

1.3 Erweiterung des Lösung

Als Teil der Aufgabenstellung sollten auch Erweiterungen zu der ursprünglichen Lösung eingebaut werden. Hier wurden zwei Erweiterungen vorgenommen. Beide basieren auf dem Umstand, dass in vielen Modulen des ITC die Möglichkeit besteht an Übungen teilzunehmen.

Als erstes wurde ein Hash angelegt, in welchem eingetragen wird, ob ein Student an dem jeweiligen Termin seine Übungsaufgaben abgegeben hat:

```
HSET iba23U-011023
id "iba23"
"Bubi␣Blauschuh" "Nicht␣abgegeben"
"Thomas␣Koenigsmann" 10
"Maria␣Mandarina" "Nicht␣abgegeben"
"Katrin␣Kleeblatt" 5
```

Manche Dozenten am ITC gehen bei den Übungen nicht anhand von Meldungen, sondern anhand einer Liste vor, wer am längsten nicht aufgerufen wurde. Hierzu eignet sich Redis besonders gut, da ein solcher Listen-Datentyp bereits besteht. Zum Beispiel könnte man nun eine Liste erstellen, zu der man alle Studenten einer Veranstaltung hinzufügt:

```
LPUSH iba23Abfrage "Bubi␣Blauschuh" "Thomas␣Koenigsmann" "Maria␣
    Mandarina" "Katrin␣Kleeblatt"
```

Wenn man nun wissen will, welcher Student als nächstes an der Reihe ist, kann man dies mit `RPOP iba23Abfrage` herausfinden. In diesem Beispiel wäre das "Bubi Blauschuh". Anschließend wird dieser Student mit `LPUSH iba23Abfrage "Bubi Blauschuh"` wieder angehangen.

2 Cassandra

Cassandra ist eine NoSQL, linear-skalierende, ausfallsichere, spaltenorientierte Datenbank, die schnell und zuverlässig funktioniert, wenn das Datenmodell richtig entworfen wurde.

Im Rahmen des Moduls Moderne Datenbanken haben wir Gruppenaufgaben zu Cassandra bekommen. Für diese werden die Lösungsideen und Aspekte dieser betrachtet und beschrieben.

2.1 Aufgabenstellung

Jeder Gruppe wurden für Cassandra sechs Aufgaben gegeben. Diese sind den Folien zu entnehmen. Besagte Aufgaben wurden von Prof. Königsmann genauer spezifiziert und abgeändert. Wir haben die Aufgaben wie folgend verstanden:

1. Ein paar Anwendungsfälle wählen, für die sich Cassandra gut eignet.
2. Die Tabellen auf eine geeignete[1] Art und Weise designen. Diese Aufgabe wurde in zwei Teilaufgaben aufteilen.
    - Geeignete Partitionen und Cluster Columns wählen
    - Daten mehrmals schreiben um die Lesegeschwindigkeit zu erhöhen. (Redundanz einbauen)
3. Eine geeignete Methode wählen um Daten konsistent zu halten. Auf der Folie wird geschrieben, dass ein BATCH benutzt werden soll.
4. Ein Einsatzszenario für TTL überlegen.
5. Die Performance von der Relationalen Datenbank zu Cassandra in den Ausgewählten Anwendungsfällen vergleichen.

[1] Wie eine gute Tabelle für Cassandra aussieht wird von Tyler Hobbs und Sebastian Sibl gut beschrieben. Die Beiträge:
https://www.datastax.com/blog/basic-rules-cassandra-data-modeling und
https://www.freecodecamp.org/news/the-apache-cassandra-beginner-tutorial/

2.2 Anwendungsfälle

Bevor Tabellen in Cassandra erzeugt werden können, muss bestimmt werden, welche Daten, in welchem Kontext abgefragt werden. Das ist nötig, weil sich die Tabellen in Cassandra auf einzelne Queries bzw. Anwendungsfälle beziehen und für diesen Anwendungsfall sehr performant sind. Wenn man In Cassandra versucht mit einer Tabelle mehrere Anwendungsfälle abzudecken, wird man recht wahrscheinlich performance Probleme bekommen. Es wurden zwei Anwendungsfälle gefunden, bei denen davon ausgegangen wird, dass sich Cassandra besonders gut eignet.

2.2.1 Termine von Studenten

Ein Student möchte all seine Termine einsehen können. Dafür sollen die folgenden Daten angezeigt werden. Datum, Beginn, Ende, Modul-Bezeichnung, VorlesungsTyp(Übung etc...), Dozent und Teilnahmestatus.

2.2.2 Termine von Dozenten

Ein Dozent möchte all seine Termine einsehen. Dafür sollen die folgenden Daten angezeigt werden. Datum, Beginn, Ende, Modul-Bezeichnung und den Vorlesungs-Typ(Übung etc...). Der einfachhalt halber wird die Vertretung oder der Ausfall von Terminen nicht berücksichtigt. Dozent haben in diesem Projekt immer Zeit und sind unverwundbar, weshalb alle Termine immer stattfinden und auch niemals vertreten werden.

2.3 Tabellen

Beim designen von Cassandra Tabellen gibt es zwei Ziele, die auf jede Fall erfüllt sein sollen:

- Die Daten müssen gleichmäßig über alle Knoten verteilt sein.
- Beim Lesen muss von so wenig Partitionen, wie es geht gelesen werden.

Um diese beiden Ziele zu erfüllen ist es notwendig den PartionKey für die Tabelle richtig zu wählen.

2.3.1 Termine von Studenten

Im ersten Anwendungsfall wird der Student als PartitionKey und der Termin als ClusterColumn benutzt. Wir gehe davon aus, dass ein Student ungefähr 60 Termine pro Semester hat. Nach zehn Semestern wären das ungefähr 600 Termine, die auf einem Knoten liegen würden. Das halten wir in Bezug auf das erste Ziel (die Datenverteilung) für akzeptable. Wenn auffallen sollte, dass die Daten nicht gleichmäßig verteilt werden, weil die Partitionen zu groß sind, könnte das Semester noch als PartionKey hinzugefügt werden. Dadurch müsste das System aber für jedes Semester eine andere Patition abfragen, was voraussichtlich die Performance verringert.

2.3.2 Termine von Dozenten

Im zweiten Anwendungsfall wird davon ausgegangen, dass nur die Termine für zwei Semester pro Dozent in Cassandra persistiert werden. Der Dozent wird als PartitionKey und der Termin als ClusterColumn benutzt. Wir gehen davon aus, dass ein Dozent 300 Termine pro Semester hat. Bei zwei Semestern wären das 600 Termine, was wir in Bezug auf die Datenverteilung ebenfalls für akzeptable halten.

2.3.3 Datenmodel

Das Datenmodel der beiden Tabellen kann den folgenden Create-Statements entnommen werden.

```
CREATE TABLE student_termine (student_id  int, termin_id int, datum
    date, beginn time, ende time, bezeichnung text, typ text,
   dozent text, teilnahmestatus text, PRIMARY KEY((student_id),
   termin_id));
```

```
CREATE TABLE dozent_termine (dozent_id  int, termin_id int, datum
   date, beginn time, ende time, bezeichnung text, typ text,
   PRIMARY KEY((dozent_id), termin_id));
```

2.4 Schreib- und Leseoperationen

In Cassandra sind Schreib- und Leseoperationen ein wichtiges Thema und werden bei der Modellierung von Tabellen bedacht. Eine wichtige Rolle spielen Redundanzen, das Consistencylevel und die Konsistenz der Persistierten Daten.

2.4.1 Redundanz

Diese beiden Tabellen sind redundant, weil im Kern die gleichen Termine persistiert werden. Diese Redundanz wird für deutlich schnelleres Lesen der Termine für Dozenten und Studenten in kauf genommen.

Durch die Redundanz müssen neue Termine in Cassandra an mehreren Stellen hinzugefügt werden. Die Anzahl der Schreiboperationen auf unterschiedliche Partitionen beträgt 1 (dozent) + Anzahl der Studenten in dem Termin. Wenn das Modul Moderne DB von 30 Studenten belegt wird, werden pro Termin 30 + 1 Schreiboperationen auf unterschiedlichen Partitionen durchgeführt.

Die Redundanz ist ein gewünschter Handel, bei dem mehr Schreiboperationen für weniger Leseoperationen eingetauscht werden. Die Schreiboperationen sind in Cassandra deutlich schneller als Leseoperationen, weshalb Redundanzen in der Regel gewollt sind.

2.4.2 Consistencylevel

Das Consistencylevel bei Cassandra beschreibt weniger eine Konsistenz, die man aus Relationalen Datenbanken kennt, sondern eher, wie sicher die Daten geschrieben/gelesen werden. Wenn ein Datensatz mit dem Consistencylevel ONE geschrieben wird, bedeutet das, dass der Datensatz auf min. einem Knoten hinzugefügt wurde. Das heißt aber nicht, dass wirklich nur auf einem Knoten geschrieben wurde. Das Consistencylevel gibt nur die Anzahl der Knoten an, die beim Lesen bzw. Schreiben mit einbezogen werden müssen. Die Daten werden trotzdem auf mehrere Knoten geschrieben und letztendlich auf allen Replikationsknoten verteilt. Beim Lesen werden basierend auf dem Consistencylevel eine unterschiedliche Anzahl von Knoten abgefragt. Bei ungleichen Daten wird der neueste Datenstand benutzt.

Wichtig wird das Consistencylevel erst in extrem fällen, wenn z.B. während des Schreibvorgangs Knoten ausfallen, oder direkt gelesen wird, nachdem geschrieben wurde. Wenn gelesen wurde, bevor die Knoten genug Zeit hatten, die Daten an alle Replikationsknoten zu verteilen, kann es sein, dass ein Knoten Gelesen wird, der die neuen Daten noch gar nicht besitzt. In so einem Fall kann ein Höheres Consistencylevel wie z.B.: Quorum dafür sorgen, dass mit einer deutlich höheren Wahrscheinlichkeit, die richtigen Daten gelesen werden. Das sicherere Schreiben und Lesen kostet aber Performance (siehe CAP).

Wenn genug Zeit zwischen den Schreib- und Leseoperationen liegt, reicht ein geringes Con-

sistencylevel aus. Bei unseren Anwendungsfällen gehen wir davon aus, dass einige Zeit (min. eine Nacht) zwischen dem Schreiben und dem Lesen der Termine besteht. Deshalb wird das Consistencylevel ONE sowohl für Schreib als auch für Leseoperationen ausreichen. Zudem werden die wirklich wichtigen Termine z.B.: Klausuren zusätzlich per E-Mail versendet.

2.4.3 Konsistenz

Bei redundanten Daten ist es notwendig darauf zu achten, dass die Daten konsistent persistiert werden. Dafür konnten wir zwei Ansätze identifizieren. Die einfachste Lösung ist ein BATCH. Ein BATCH ist eine Sammlung von Statements und Ähnelt einer Transaktion. Bei einem BATCH werden entweder alle Statements ausgeführt oder gar keins. Dadurch kann die Konsistenz der Daten sicher gestellt werden.

Es ist vorgesehen, dass Logged BATCH benutzt werden, damit die redundant gespeicherten Daten konsistent bleiben. Sollte es bei der Ausführung von dem BATCH zu einem Fehler kommen, muss eine Strategie angewendet werden, damit die Termine trotzdem persistiert werden oder damit auf den Fehler aufmerksam gemacht werden kann. Dabei gibt es zwei Ansätze:

1. Der Termin wird in Redis gespeichert und später wird versucht den Termin erneut hinzuzufügen. Das ist im größeren Kontext nur dann möglich, wenn der Termin nur dann in Redis gespeichert wird, wenn er nicht persistiert werden konnte. Das Ziel soll es nicht sein, alle Termine zu cachen und im Laufe der Zeit zu persistieren.
2. Es wird versucht den Termin erneut zu persistieren. Hierbei sollte die Anzahl der Versuche begrenzt werden. Sollte der Termin nicht persistiert werden können, ist es notwendig den Dozenten zu benachrichtigen.

Die Lösung für die wir uns entschieden haben sieht eine Mischung aus beiden Ansätzen vor: Sollte ein Termin nicht persistiert werden können, wird er in Redis abgelegt. In Regelmäßigen Zeitabständen (z.B.: 30 min,) wird versucht den Termin zu persistieren. Es gibt drei Retries und wenn der Termin nicht persistiert werden kann, wird der Dozent und die Verwaltung darüber benachrichtigt.

Das BatchStatement für das Hinzufügen von Daten folgt dem folgenden Schema.

```
BEGIN BATCH
INSERT INTO stundenplan.dozent_termine(dozent_id, termin_id, datum,
    beginn, ende, bezeichnung, typ)
VALUES(1, 1, '2024-04-15', '08:00:00', '15:30:00', 'Moderne DB', '
    Vorlesung');

INSERT INTO stundenplan.student_termine(student_id, termin_id,
    datum, beginn, ende, bezeichnung, typ, dozent)
VALUES(1, 1, '2024-04-15', '08:00:00', '15:30:00', 'Moderne DB', '
    Vorlesung', 'Königsmann');

INSERT INTO stundenplan.student_termine(student_id, termin_id,
    datum, beginn, ende, bezeichnung, typ, dozent)
VALUES(2, 1, '2024-04-15', '08:00:00', '15:30:00', 'Moderne DB', '
    Vorlesung', 'Königsmann');

INSERT INTO stundenplan.student_termine(student_id, termin_id,
    datum, beginn, ende, bezeichnung, typ, dozent)
VALUES(3, 1, '2024-04-15', '08:00:00', '15:30:00', 'Moderne DB', '
    Vorlesung', 'Königsmann');

APPLY BATCH;
```

2.5 Löschen von Daten

Bei dem Löschvorgang wird von Cassandra ein Tombstone gesetzt. Dieser makiert einen Zeitpunkt in der zukunft, an dem etwas entfernt wird. Das ist notwendig, damit die von dem Tombstone makierten Daten von allen Knoten gleichzeitig gelöscht und nicht wieder zurück auf andere Knoten repliziert werden. Deshalb wird das makierte Objekt erst später gelöscht. Die standard Zeit dafür beträgt 864000 s = 14400 min = 240 h = 10 d. In dieser Zeit kann der Tombstone an alle Knoten, die betroffene Daten enthalten weiter gegeben werden. Obwohl die makierten Daten theoretisch noch da sind, verhindert der Tombstone, dass betroffene Daten abgefragt werden können. Siehe https://cassandra.apache.org/doc/latest/cassandra/-managing/operating/compaction/tombstones.html

Ein möglicher Anwendungsfall für das Löschen von Daten über TTL wäre bei uns das zwei Semester Limit bei den Terminen des Dozenten. Ein angelegter Termin könnte eine vordefinierte Lebenszeit von ungefähr einem Jahr haben.

2.6 Performance

Die Performance von Cassandra und unsere Relationalen Datenbank wird in zwei Kategorien getestet. Beim initialen befüllen der Datenbanken mit Testdaten und bei der Ausführung der Anwendungsfälle.

2.6.1 Testdaten

Es werden 10 Jahrgänge mit jeweils 30 Studenten erzeugt. Es gibt insgesamt 7 Module. Für jedes Modul gibt es 10 Termine pro Jahrgang. Die Termine Unterteilen sich in die drei Veranstaltungen **Übungen**, **Verlesungen** und **Praktika** auf. Zudem wurden 3 Dozenten erzeugt.

Bei der Persistierung von den Testdaten war es wichtig, dass die Testdaten realitätsnah gespeichert wurden. Das bedeutet, dass Für Cassandra das bereits Vorgestellte Batch benutzt wird um Termine hinzuzufügen. Bei MySQL wird darauf geachtet, dass die einzelnen Termine ebenfalls in einer eignen Transaktion in der Datenbank persistiert werden.

Es gibt insgesamt 300 Studenten, die jeweils 70 Termine haben. In Cassandra werden in der Studenten-Tabelle also 21000 Datensätze geschrieben. In die Dozenten Tabelle werden 700 Termine geschrieben.

In der MySQL Datenbank werden 22174 Datensätze geschrieben. Das Problem hier ist die referentielle Integrität. Es müssen mehr Informationen angelegt werden, als für die Anwendungsfälle benötigt werden.

Ergebnisse

Cassandra	**MySQL**
2500 - 6000ms (meistens 3500 - 3800)	3500 - 3850ms

MySQL ist in der Schreibgeschwindigkeit sehr konsistent. Es ist davon auszugehen, dass Cassandra wegen den ganzen batches langsamer ist, als wenn man die Statements ohne Batch ausführen würde. Als Alternative zu dem

2.6.2 Anwendungsfälle

Bei den Anwendungsfällen wird auf beiden Datenbanken ein Select ausgeführt, dass die zuvor beschriebenen Ergebnisse in einer Tabelle zurückgibt. Die Statements sehen wie Folgend aus.

Cassandra

```
SELECT datum, beginn, ende, bezeichnung, typ, dozent,
    teilnahmestatus FROM stundenplan.student_termine WHERE
    student_id = <id>
```

```
SELECT datum, beginn, ende, bezeichnung, typ FROM stundenplan.
    dozent_termine WHERE dozent_id = <id>
```

MySQL

```
SELECT Termin.datum, Termin.beginn, Termin.ende, Modul.name,
    Veranstaltung.typ
FROM Termin
JOIN Veranstaltung ON Termin.veranstaltungId = Veranstaltung.
    veranstaltungId
JOIN Modul ON Veranstaltung.modulId = Modul.modulId
WHERE Veranstaltung.dozentId = <dozentId>;
```

```
SELECT DISTINCT Termin.datum, Termin.beginn, Termin.ende, Modul.
    name, Anwesenheit.fehlgrund, Veranstaltung.typ, Dozent.name FROM
     Termin JOIN Veranstaltung ON Termin.veranstaltungId =
    Veranstaltung.veranstaltungId JOIN Modul ON Veranstaltung.
    modulId = Modul.modulId JOIN BelegteVeranstaltung ON
    Veranstaltung.veranstaltungId = BelegteVeranstaltung.
    veranstaltungId JOIN Dozent ON Veranstaltung.dozentId = Dozent.
    dozentId LEFT JOIN Anwesenheit ON BelegteVeranstaltung.
    matrikelnummer = Anwesenheit.matrikelnummer AND Termin.terminId
    = Anwesenheit.terminId WHERE BelegteVeranstaltung.matrikelnummer
     = <matrikelnummer>;
```

Ergebnisse

Cassandra	MySQL
Student: 3-5ms	5-15ms
Dozent: 5-6ms	3-6ms

Besonderheit

Für die Performance-Tests wurden die Datenbank Instanzen als Container über Docker ausgeführt. Dabei wurden 3 Cassandra Instanzen benutzt. Im Treiber wurden alle drei Instanzen als Kontaktpunkte hinzugefügt, sodass der Treiber die möglichkeit hatte, die Last über alle drei Knoten gleichmäßig zu verteilen. Dafür musste lediglich der Port für den Container angepasst werden, weil nicht alle drei Container den gleichen Port belegen können.

3 CouchDB

CouchDB ist eine dokumentenorientierte NoSQL-Datenbank, die Daten im JSON-Format speichert. Sie bietet eine flexible Datenstrukturierung, was besonders für Anwendungen mit variablen Datenschemata geeignet ist. CouchDB kommuniziert über HTTP/REST, wodurch es einfach wird, Webdienste direkt mit der Datenbank zu verbinden.
Als dokumentenorientierte Datenbank ermöglicht CouchDB den Nutzern, komplexe Datenstrukturen in einem einzigen Dokument zu speichern.
Die Indexierung und Abfrage von Daten erfolgt über sogenannte „Views“, die in JavaScript geschrieben werden. Dieses Vorgehen macht CouchDB besonders gut für Webanwendungen und Anwendungen,
die eine hohe Menge an unstrukturierten Daten verarbeiten und/oder allgemein ein schemafreies Vorgehen benötigen.
Die Datenkonsistenz wird dabei durch die Verwendung von MVCC (Multi-Version Concurrency Control) sichergestellt, was Konflikte bei gleichzeitigen Datenänderungen effektiv verwaltet.
Allerdings sind bei CouchDB ausreichende Ressourcen für die Speicherung und Verwaltung der Daten notwendig, und es kann bei sehr großen Datenmengen zu Performance-Einbußen kommen. CouchDB ist somit gut für Projekte, bei denen Schemafreiheit steht.

3.1 Aufgabenstellung

Für diese Datenbank sollte das ursprüngliche relationale Modell in Dokumenten abgebildet werden. Zudem sollten drei Erweiterungen gefunden werden, welche sich für schemafreies Vorgehen eignen. Dazu sollten sechs sinnvolle Views und drei sinnvolle Reduce-Funktionen gefunden werden.

3.2 Übertragung der relationalen Datenbank in Dokumente

Auch für CouchDB gilt, dass dort im Gegensatz zu relationalen Datenbanken Redundanzen nichts Schlimmes sind. Insofern konnten viele Strukturen, welche in den ursprünglichen Entwurf eingebaut wurden, um Redundanzen zu vermeiden und die Konsistenz zu erhalten, einfach plattgeklopft”werden. Bei der Übertragung galten zudem die drei Prämissen:

1. Redundanzen sind nicht schlimm
2. Konsistenzen werden nicht über das Modell abgefangen
3. Da es keine Joins gibt, müssen die Daten entsprechend strukturiert sein

In ein JSON-Dokument abgebildet wurden jeweils die Tupel:

- der drei Tabellen `BelegteVeranstaltung`, `D_Jahrgang` und `Student` zu einem Studenten-Dokument
- der sechs Tabellen `D_Typ`, `Dozent`, `VertretenderDozent`, `Veranstaltung`, `Modul` und `D_Semester` zu einem Semester-Dokument
- aller Tabellen zu einem semantisch aussagekräftigem Veranstaltung-Dokument
- der Tabelle Modul zu **einem** Modul-Dokument
- der beiden Tabellen Dozent und Veranstaltung zu einem Dozenten-Dokument

3.3 Erweiterung der Lösung

Als sinnvolle Erweiterungen wurden identifiziert:

1. Nachhalten der Social Points und der zugehörigen Messe, Werbeaktionen, usw.
2. Raumnutzungsplan
3. Erweiterung der Termininformationen, z.B. Termine für besondere Inhalte vormerken ((Probe)Klausur, Wiederholungstermine,...)

3.4 Views und Reduce-Funktionen

Folgende Views und Reduce-Funktionen wurden umgesetzt:

1. Alle Studenten der Gruppe A-F
2. Alle ausgefallenen Termine (zum Nachholen) mit Reduce-Funktion zur Anzahl der noch nachzuholenden Termine
3. Alle bislang nicht entschuldigten Fehlzeiten mit Reduce-Funktion zur Anzahl der nicht entschuldigten Fehlzeiten
4. Alle Studenten mit Reduce-Funktion zur Anzahl der Studenten
5. Alle Termine nach Dozenten
6. Alle Veranstaltungen zu IBA

3.5 Performance

Mithilfe der Ektorp-Persistence-API wurden Performance-Tests durchhgeführt und mit der ursprüglichen, auf MySQL basierenden Lösung verglichen. Für die CRUD-Operationen ergaben sich:

- 127ms für das Anlegen eines Dokuments
- 61ms für das Lesen eines Dokuments
- 38ms Dokument bearbeiten
- 20ms Dokument löschen

Im Vergleich ergaben sich folgende Werte (MySQL ggü. CouchDB)

- Für das Anlegen von 100 Studenten einzeln: 444ms ggü. 912ms
- Für das Löschen von 100 Studenten einzeln: 600ms ggü. 1223ms
- Für das Anlegen von 100 Studenten in einem Statement: 20ms ggü 28ms
- Für das Löschen von 100 Studenten in einem Statement: 10ms ggü. 17ms

3.5.1 Beobachtungen

1. In den Einzeloperationen dauert das Erstellen mit Abstand am längsten
2. Bei der Massenoperation dauert das Löschen mit Abstand am längsten
3. Im Allgemeinen ist MySQL bzw. dessen Java-Schnittstelle performanter
4. Die Schemagebundenheit von MySQL verträgt sich gut mit Javas Klassengebundenheit
5. Die Nutzung ist stark Use-Case abhängig, man muss sich anhand der Operationen orientieren
6. CouchDB löscht nicht, es markiert als gelöscht

4 Neo4J

Neo4J ist eine in Java implementierte Graphdatenbank, die sich gut dazu eignet um unstrukturierte Daten zu speichern und zu verwalten.

Im Rahmen des Moduls Moderne Datenbanken haben wir Gruppenaufgaben zu Neo4J bekommen. Für diese werden die Lösungsideen und Aspekte dieser betrachtet und beschrieben.

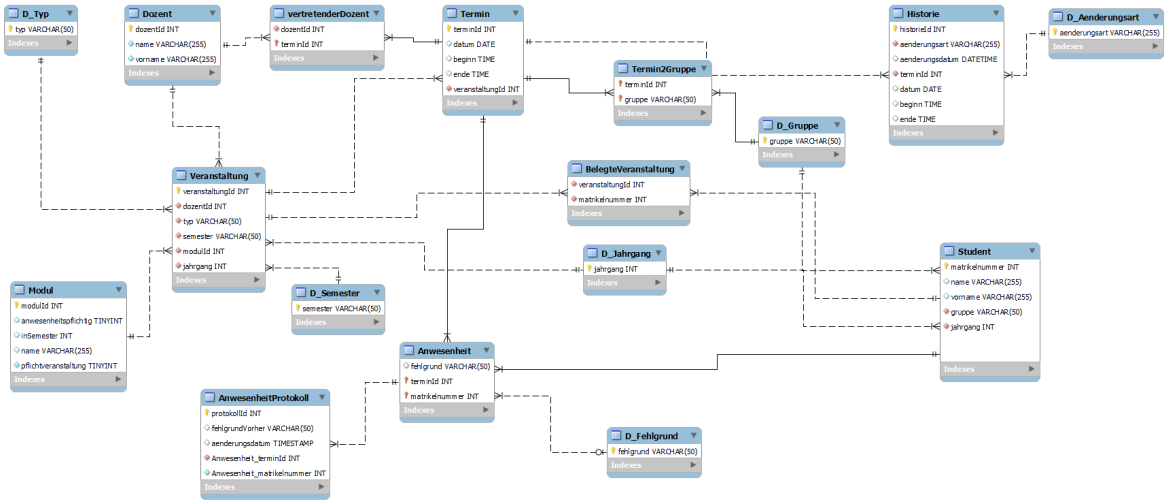
4.1 Aufgabenstellung

Jeder Gruppe wurden für Neo4J sechs Aufgaben gegeben. Diese sind den Folien zu entnehmen. Besagte Aufgaben wurden von Prof. Königsmann genauer spezifiziert und abgeändert. Wir haben die Aufgaben wie folgend verstanden:

1. Unsere Datenbank soll auf Neo4J umgestellt werden. Dabei sollen in Neo4J die gleichen Informationen abrufbar sein, wie vorher.
2. Wir sollen mindestens drei sinnvolle Erweiterungen für unser Datenbank-Schema identifizieren. Diese zu implementieren ist nicht vorgesehen.
3. Es müssen Daten persistiert werden.
4. Es soll die Performance von Create, Read, Update und Delete-Operationen getestet werden.
5. Die existierenden Relationalen-abfragen für die Datenbank sollen in Neo4J abgebildet werden.
6. Wir sollen mindestens drei Beispiele finden, in denen sich Neo4J besonders gut eignet.

4.2 Datenbankschema

Das aktuelle Datenbankschema und die Tabellen sind dem folgenden Bild zu entnehmen.



Die Tabellen des jetzigen Datenbankschemas lassen sich in die drei Kategorien **Entity**, **Relation** und **Domain** unterteilen.

4.2.1 Entity

Eine Tabelle, die Fachliche Objekte bzw. Informationen Enthält, kann und wird in Neo4J durch Knoten dargestellt. Also wird jedes Tupel aus einer Entitäts-Tabelle zu einem Label. Die folgenden Tabellen wurden dieser Kategorie zugeteilt. 17

- Dozent
- Termin
- Veranstaltung
- Modul
- Student
- Historie (Wird nicht weiter betrachtet, weil sich unserer Meinung nach neo4J schlecht dafür eignet eine Historie von Änderungen darzustellen)

4.2.2 Releation

Relationstabellen sind Matching Tabellen. Diese werden sich in Beziehungen zwischen den knoten auflösen. Die folgenden Tabellen wurden dieser Kategorie zugeteilt.

- vertretenderDozent
- Termin2Gruppe (Wird nicht weiter mit einbezogen, weil diese Tabelle keine Rolle für Funktionsfähigkeit des Projektes spielt.)

- BelegteVeranstaltung
- Anwesenheit

4.2.3 Domain

Eine Domain Tabelle ist eine Tabelle, die in einer Relationalen Datenbank für die Konsistenz von Daten sorgt. So eine Tabelle besteht häufig nur aus einer Spalte. Dadurch soll sichergestellt werden, dass nur vordefinierte Werde benutzt werden können. Diese Tabellen werden beider Konvertierung zu Neo4J wegfallen. Die folgenden Tabellen wurden dieser Kategorie zugeteilt.

- D_Typ
- D_Gruppe
- D_Aenderungsart
- D_Semester
- D_Jahrgang
- D_Fehlgrund

4.2.4 Ergebnis

Wir haben die folgenden Labels und Beziehungen ausgearbeitet.

Labels

- Dozent
- Termin
- Modul
- Veranstaltung
- Student

Beziehungen

- VERTRITT_IN
- BELEGT_VERANSTALTUNG
- NICHT_ANWESEND
- HAT
- LEITET

4.3 Erweiterungen

Wir sollen drei sinnvolle Erweiterungen ausarbeiten. Einmal soll nachvollziehbar sein, welche Module aufeinander aufbauen. Zum anderen sollen Studenten, die das Modul gerade machen, Studenten finden können, die das Modul bereits abgeschlossen haben und um Hilfe bitten.

4.3.1 Aufeinander aufbauende Module

Es gibt Module, die aufeinander aufbauen. Das wird im Jetzigen Zustand nicht von unserem Datenbankschema abgebildet. Es gibt bereits Modul-Knoten. Hier wird nur eine weitere Beziehung hinzugefügt, die **BASIERT_AUF** heißt und darstellt, dass in dem Modul davon ausgegangen wird, dass andere Module bereits absolviert wurden.

4.3.2 Vertiefungen

Es gibt Vertiefungen. Eine Vertiefung ist eine Gruppierung von Modulen, die Logisch zusammen gehören und einzelne Themen eines Großen Themas behandeln. Im Augenblick gibt es keine Möglichkeit, die Gruppierung von Modulen darzustellen. Dafür muss ein neuer Knotentyp und eine neue Beziehung hinzugefügt werden. Der Knotentyp heißt **Vertiefung**. Die neue Beziehung heißt **BESTEHT_AUS**. Eine Vertiefung hat einen Namen.

4.3.3 Hilfe unter Studenten

Zum anderen sollen Studenten, die das Modul gerade machen, Studenten finden können, die das Modul bereits abgeschlossen haben und um Hilfe bitten. Dafür müssen keine Erweiterungen durchgeführt werden.

4.4 Daten

Nachdem bestimmt wurde, wie die Daten gespeichert werden, sollen sinnvolle Daten gespeichert werden.