FACHHOCHSCHULE SÜDWESTFALEN

Modellierung und Performancevergleich eines To-do Datenmodells in MariaDB, MongoDB und Neo4j

Schriftliche Ausarbeitung

für die Vorlesung 'NoSQL-Datenbanken' (SS 2022)

Ablieferungstermin: 25. April 2022

Inhaltsverzeichnis

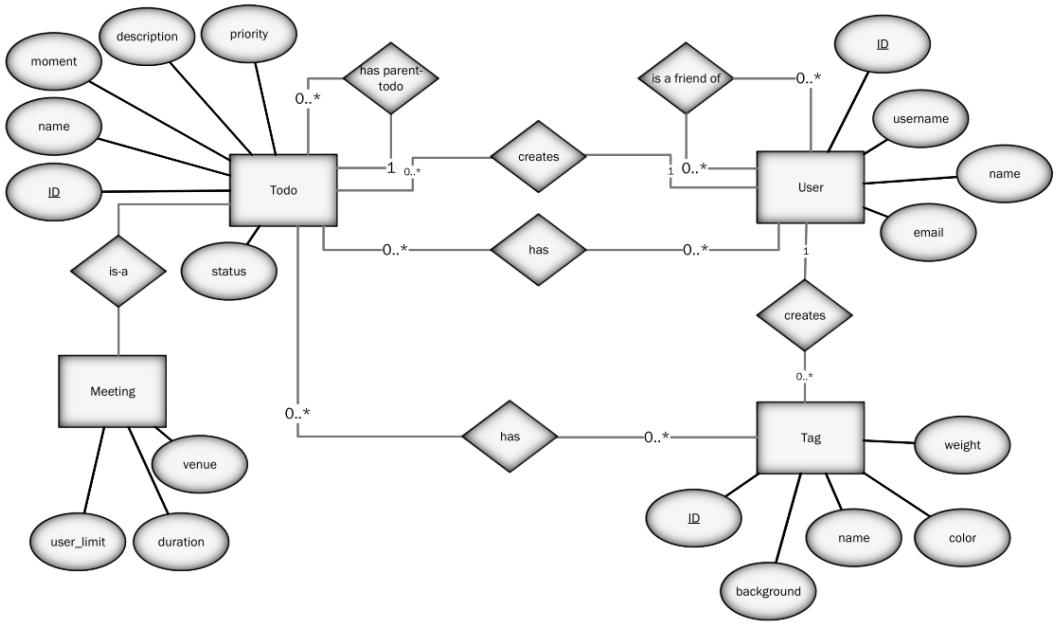

Abbildung 1: ER-Modell

1 Anwendungsszenario

Als Anwendungsszenario wurde eine To-do-Anwendung gewählt. Die Grundidee war es ein Datenmodell zu entwerfen dass sowohl mehrere Benutzer als auch das Verschachteln von To-dos unterstützt. Das Ergebnis, zu sehen in Abbildung 1, soll im Folgenden kurz erläutert werden.

Den Ausgangspunkt bildet der Benutzer (*User*). Der Kern des Datenmodells besteht darin, dass ein Benutzer die Möglichkeit hat, beliebig viele To-dos (*Todo*) zu erstellen. Diese können bei Bedarf unterhalb eines bereits existierenden To-dos eingefügt werden, so dass es möglich ist, eine hierarchische Struktur zu erzeugen. Zu den Eigenschaften eines To-dos gehört unter anderem *moment*, eine optionale Deadline, sowie *checked*, durch die ein To-do als erledigt gekennzeichnet werden kann. Außerdem beinhaltet das Datenmodell eine Spezialisierung für To-dos, das *Meeting*. Dieses besitzt zusätzliche Eigenschaften wie eine veranschlagte Dauer (*duration*).

Zusätzlich zu den To-dos hat der Benutzer die Möglichkeit, eine beliebige Anzahl von *Tags* zu erstellen, und durch deren Zuordnung zu seinen To-dos diese flexibel zu

kategorisieren. Die Eigenschaften der Tags beschreiben dabei eine spätere Darstellungsform.

Der Benutzer ist außerdem Teil von zwei weiteren Beziehungen. Über die erste kann ein Benutzer einen anderen Benutzer als Freund hinzufügen. Diese bildet die Grundlage für die zweite Beziehung, über die ein Benutzer Freunde zu seinen To-dos hinzufügen kann, beispielsweise zu einem von ihm erstellten Meeting.

2 Umsetzung

Bei der Auswahl der beiden NoSQL-Datenbanken wurde das im ersten Schritt festgelegte Datenmodell als Entscheidungsgrundlage genommen. Für die Umsetzung boten sich dokumentenorientierte Datenbanken sowie Graph-Datenbanken an, während spaltenorientierte Datenbanken oder Key-Value-Datenbanken nicht wirklich sinnvoll bzw. maximal nur ergänzend einsetzbar gewesen wären. So wurde von den zwei geeigneteren Datenbanktypen jeweils eine Datenbank ausgewählt. Neben *MariaDB* als SQL-basierte Vergleichsgrundlage fiel die Wahl auf **MongoDB** und **Neo4j**.

Diese Wahl basierte unter anderem auf der aktuellen Popularität [1] sowie der Tatsache, dass beide Datenbanken sowohl Open-Source als auch sehr gut dokumentiert sind.

Um im späteren Verlauf die Performance der Anwendungsfälle sinnvoll vergleichen zu können müssen die Datenbanken zuvor mit den gleichen Daten gefüllt werden. Dieses ist durch eine eigens entwickelte Anwendung [2], basierend auf *Node.js* und in *Typescript* geschrieben, komfortabel möglich. Für den Datenbankzugriff werden weit verbreitete Bibliotheken verwendet: *sequelize* für MariaDB, *mongoose* für MongoDB und *neo4j-javascript-driver* für Neo4j.

Die Anwendung ermöglicht es im ersten Schritt, einen vollständigen Datensatz zufällig zu erzeugen und in einer MariaDB-Datenbank zu speichern. Hier ist zu erwähnen dass die Verwendung von Wahrscheinlichkeiten dafür sorgt, dass der erzeugte Datensatz ein großes Maß an Variationen beinhaltet. So kann beispielsweise ein Benutzer keine To-dos haben, während einem anderer Benutzer 100 verschachtelte To-dos zugeordnet sind. Diese Wahrscheinlichkeiten können vom Anwender belie-

[1] https://db-engines.com/en/ranking
[2] https://github.com/JohannesW-DE/todo-seeder

big modifiziert werden, genauso wie der Wert für die Anzahl der Benutzer für die ein Datensatz generiert werden soll.

In zwei weiteren Schritten kann der generierte Datensatz dann aus der MariaDB-Datenbank ausgelesen werden und in einer MongoDB- oder Neo4j-Datenbank, entsprechend aufbereitet, abgespeichert werden.

3 Modellierung

3.1 MariaDB

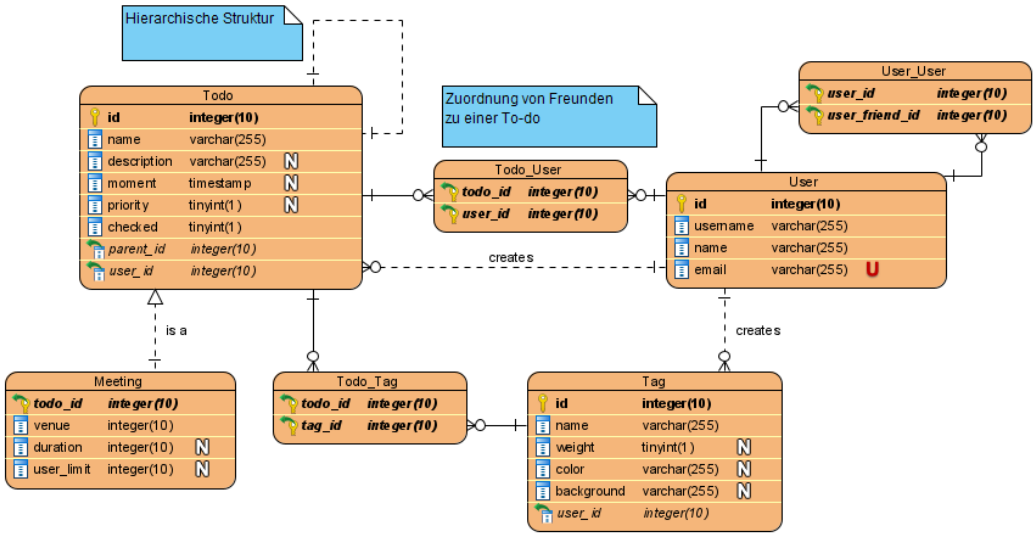

Abbildung 2: ER-Modell für MariaDB

3.2 MongoDB

Die Modellierung in MongoDB wurde durch zwei *Collections* realisiert, eine für die Benutzer (*User*), eine für die To-dos (*Todo*). Tags, bei denen davon auszugehen ist dass sich ihre Anzahl pro Benutzer in Grenzen halten wird, werden als eingebettete Dokumente in dem jeweiligen Benutzerdokument gespeichert. Das Konzept, bei dem es sich bei To-dos um Meetings handeln kann, ist so umgesetzt dass Todo-Dokumente einfach die zusätzlichen Eigenschaften eines Meetings als Felder besitzen können; dadurch sind diese gleichzeitig als Meeting identifizierbar. Alle anderen Beziehungen

werden über Felder in den Dokumenten der beiden Collections gespeichert. Dabei handelt es sich entweder direkt um *ObjectIds* (z.B. der Ersteller eines To-do's) oder Arrays von *ObjectIds* (z.B. die Freunde eines Benutzers).

3.3 Neo4j

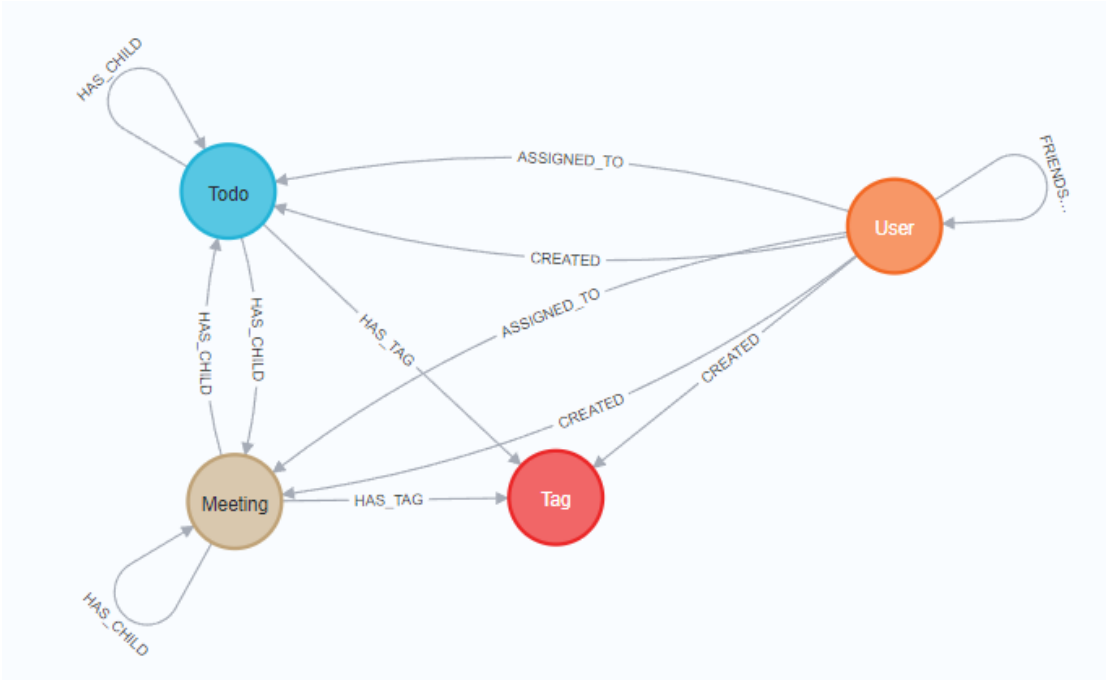

Abbildung 3: Neo4j - Schema

Das Schema, welches in Abbildung 3 zu stehen ist, ähnelt mit den vier *Node labels* und den fünf *Relationship types* (*CREATED* ist die Bezeichnung von drei verschiedenen Beziehungstypen) dem Diagramm aus Abbildung 1. Es gibt hierbei keine Knoten die ausschliesslich als *Meeting* gelabelt sind, dieses Label wird nur bei Bedarf bereits existierenden Knoten mit dem Label *Todo* hinzugefügt.

4 Anwendungsfälle & Performance-Tests

Bei der Definition der Anwendungsfälle wurde darauf geachtet alle vier *CRUD* Operationen abzudecken, wobei der Fokus deutlich auf dem Lesen von Daten liegen sollte. Dementsprechend wurden hierfür vier Anwendungsfälle formuliert, während die drei anderen Operationstypen mit je nur einem Anwendungsfall bedacht wurden.

Für das Testen der Performance der vier lesenden Anwendungsfälle wurden zunächst mittels Docker für alle Datenbanken je zwei Container auf Basis der aktuellen offiziellen Images erzeugt. Diese wurden, wie in Kapitel 2 beschrieben, mit zwei unterschiedlich umfangreichen Datensätzen gefüllt. Der kleinere Datensatz beinhaltet 50 Benutzer und 7382 To-dos, der größere Datensatz hingegen 500 Benutzer sowie 76653 To-dos. Die Verwendung von zwei unterschiedlich großen Datensätzen sollte dazu dienen um zu erkennen, ob und wenn ja wie stark der Umfang der Daten die Dauer der Abfragen beeinflusst. An den Datenbanken wurden keinerlei Änderungen oder Optimierungen vorgenommen, sie wurden im Ausgangszustand belassen.

Die Durchführung der Tests erfolgte ebenfalls durch die in Kapitel 2 vorgestellte Anwendung, bei der es für jeden der sieben Anwendungsfälle ein eigenes Skript gibt. Der Ablauf ist jeweils insofern gleich als dass die drei jeweiligen Queries bei jedem Aufruf des Skripts so oft wie möglich - ohne dass sich die Werte für die Parameter wiederholen - und so direkt wie möglich, also ohne Nutzung der ORM Funktionalitäten der Bibliotheken, abgesetzt werden. Die Dauer aller Queries wird für jede Datenbank summiert und zusammen mit der Anzahl der Queries in einem eigenen Logfile gespeichert. Die Laufzeiten der Tests, durchgeführt auf einem aktuellen handelsüblichen PC, werden mit Hilfe des *process* Moduls von *Node.js* ermittelt.

4.1 Wie viel Prozent der To-dos in einer To-do-Hierarchie sind bereits erledigt?

Tabelle 1: Durchschnittliche Dauer pro Query in ms

Benutzeranzahl	MariaDB	Neo4j	MongoDB
50	1,1	11	15,2
500	1,1	27,4	138,1

Für diesen Anwendungsfall wurde so vorgegangen, dass für alle To-dos, die die Wurzel einer To-do-Hierarchie bilden, ein Query abgesetzt wird was nach bereits erledigten To-dos innerhalb dieser Hierarchie sucht. Auffällig ist, dass die Queries auf MongoDB und Neo4j bei der kleinen Datenbank ähnlich schnell sind, bei der großen Datenbank sich jedoch ein signifikanter Unterschied um einen Faktor von fast 5 auftut. Dieser übermäßige Anstieg der Dauer wird auf Seiten von MongoDB durch den *$graphLookup*-Schritt in der *Aggregation Pipeline* verursacht, was allerdings der empfohlene Weg ist um rekursive Beziehungen aufzulösen.

4.2 Welche To-dos innerhalb der nächsten 7 Tage sind mit einem bestimmten Tag versehen?

Tabelle 2: Durchschnittliche Dauer pro Query in ms

Benutzeranzahl	MariaDB	Neo4j	MongoDB
50	0,8	7,3	4
500	0,8	7,2	30,7

Dieser Anwendungsfall wurde bewusst einfach gehalten. Hierbei wurden die Queries für jedes Tag ausgeführt. Aus Perspektive von MariaDB handelt es sich hier um einfach einfaches *SELECT* mit einem *INNER JOIN* und einer relativ einfachen *WHERE*-Klausel. Da solche Abfragen typisch für SQL-Datenbanken sind ist ist es hier am wenigsten überraschend dass MariaDB so gut abschneidet. Auffällig hingegen ist der extreme Anstieg der Dauer der Queries bei der MongoDB-Datenbank. Es ist überraschend dass bei dem einzigen Schritt der Aggregation Pipeline, einem *$match*, die Anzahl der Dokumente einen solch großen Einfluss zu haben scheint.

4.3 Welche Freunde hat der Freund eines Benutzers die er selbst nicht als Freund hat?

Tabelle 3: Durchschnittliche Dauer pro Query in ms

Benutzeranzahl	MariaDB	Neo4j	MongoDB
50	0,8	5,5	1,3
500	0,8	5,9	1

Bei diesem Anwendungsfall geht es darum, die Freunde eines Freundes zu finden, die der Benutzer selbst noch nicht als Freund hat, ein relativ typischer Anwendungsfall für Szenarien mit mehreren Benutzern. Dafür werden alle Freundschaftsbeziehungen mit einem Query überprüft. Hier liegen die drei Datenbanken bei Betrachtung beider Datensätze am nächsten beisammen. Es fällt auf, dass die Größe der Datenbank keinen signifikanten Einfluss auf die Dauer hat; die Differenzen bei MongoDB und Neo4j, je einmal in jede Richtung, können durchaus zufallsbedingt zu sein. Bei der Betrachtung der Queries fällt auf, dass das Neo4j-Query sehr ausdrucksstark und einfach verständlich ist, die Aggregation Pipeline von MongoDB hingegen mit 11 Schritten deutlich komplexer ist.

4.4 Mit welchem anderen Benutzer wurde ein Benutzer am häufigsten zu einer To-do hinzugefügt?

Tabelle 4: Durchschnittliche Dauer pro Query in ms

Benutzeranzahl	MariaDB	Neo4j	MongoDBj
50	0,9	5,4	4,3
500	0,9	7,5	30,6

Für diesen Anwendungsfall werden die Queries für alle Benutzer abgesetzt. Das Ergebnis ähnelt hier sehr stark dem Anwendungsfall in 4.2, wo besonders der Anstieg der Dauer bei der Kombination aus dem großen Datensatz und der MongoDB-Datenbank auffällt. Im Rahmen der Aggregation Pipeline findet sowohl ein *$match* auf der User-Collection als auch ein *$lookup* in der Todo-Collection statt, eine Kombination die als Erklärung für den Anstieg dienen könnte.

4.5 Einfügen, Aktualisieren und Löschen von Daten

Tabelle 5: Durchschnittliche Dauer pro Query in ms

Operation	MariaDB	Neo4j	MongoDB
Einfügen (pro To-do)	2,1	89,5	2,6
Aktualisieren (pro Benutzer)	1,3	288	1,2
Löschen (pro Benutzer)	4,3	11,7	105

Für die drei hierbei betrachteten Anwendungsfälle lag der Fokus auf einem Ausgangsdatensatz mit 500 Benutzern und 76177 To-dos. Der Test für das Einfügen von Daten wurden so gestaltet, dass komplette To-do-Hierarchien mit durchschnittlich 82 To-dos generiert wurden, und diese dann dem zuletzt einfügten Benutzer zugeordnet in die Datenbanken eingefügt wurden.

Bei dem Aktualisieren von Daten wurde der Anwendungsfall so definiert, dass bei einem Benutzer die 10 zuerst eingefügten und noch nicht abgeschlossen To-dos als erledigt markiert werden sollten. Dieses wurde bei einem kompletten Testdurchlauf für jeweils alle Benutzer durchgeführt.

Für das Löschen von Daten wurde untersucht, wie lange ein vollständiges Löschen eines Benutzers samt aller vorhandenen Beziehungen bzw. Abhängigkeiten benötigt. Dies umfasst beispielsweise das Löschen von betroffenen Beziehungen in Neo4j wie

auch das Entfernen von *ObjectIds* aus Arrays in MongoDB. Hier bot es sich für das Testen an, die Datenbanken Benutzer für Benutzer der Reihe nach zu leeren.

Bei Betrachtung der Resultate fallen zuerst die extremen Performanceprobleme von Neo4j beim Einfügen und Aktualisieren von Daten auf. Dieses ist allerdings nicht überraschend, sind die Stärken von Neo4j von Natur aus ehr bei lesenden Operationen zu erwarten. Des weiteren ist hervorzuheben, dass sich MongoDB bei diesen beiden Anwendungsfällen nahezu auf Augenhöhe mit MariaDB bewegt. Beim Löschen ergibt sich hingegen ein anderes Bild. Dieses lässt sich dadurch erklären dass das Löschen in MongoDB drei einzelne Queries erfordert: das Löschen des Benutzer-Dokuments, das Löschen der zugehörigen Todo-Dokumente und vor allem das Entfernen der *ObjectId* des Benutzers aus den Arrays, die einer Todo andere Benutzer zuordnen.

5 Fazit

Die Ergebnisse bzw. Erkenntnisse die im Rahmen dieser Ausarbeitung gewonnen wurden, sind differenziert zu betrachten. Auf die reine Performance bezogen konnte MariaDB durchgehend überzeugen und waren in allen Anwendungsfällen am schnellsten.

Dieses galt überraschenderweise zum Beispiel auch für Anwendungsfälle wie 4.1, die extra im Hinblick auf eine stark rekursive Komponente formuliert wurden. Gerade dort war zu vermuten, dass Neo4j aufgrund der Natur vom Graph-Datenbanken besser abschneiden würde. Mögliche Erklärungen für solche Überraschungen sind zum einen die Tatsache, dass man Queries mit einem nur grundlegenden Wissen zwar erfolgreich, aber nicht sonderlich effizient konstruieren kann. So konnten diverse Möglichkeiten und Konstrukte zur Optimierung im Rahmen dieser Ausarbeitung nicht weiterführend betrachtet werden. Außerdem ist es vorstellbar, dass beispielsweise noch deutlich größere Datensätze die Performance zu Gunsten von Neo4j verschieben würden.

MongoDB hatte auf die reine Performance bezogen, ebenso wie Neo4j, ebenfalls teils deutliche Performance-Nachteile gegenüber MariaDB. Allerdings ist hier hervorzuheben, dass MongoDB mit der Aggregation Pipeline ein sehr mächtiges Werkzeug mitbringt um auch sehr komplexe Queries Schritt für Schritt nachvollziehbar und erfolgreich umzusetzen. Jedoch gilt auch hier, dass es eine extreme Vielzahl an Möglichkeiten gibt, die mit ziemlicher Sicherheit eine weitere Optimierung der Performance

der verschiedenen Queries zulassen würden.

Abschließend lässt sich sagen, dass zumindest in dem Bereich der untersuchten Datenbankgrößen und dem Kontext dieses nicht performance-kritischen Anwendungsszenarios alle drei Datenbanken problemlos einzusetzen wären.