(Neo4j) Graph-relational back-end for Minibase
Universität Konstanz – Winter 2016 / 2017

Michael BRENDLE

27. November 2016

1 Einleitung

Dieses Dokument gibt zunächst einen Einblick in die Struktur der Knoten, Beziehungen, ... von Neo4j und insbesondere wie sie im Speicher hinterlegt sind. Im Anschluss werden zwei Varianten diskutiert wie (solche) Graphen in Minibase gespeichert werden könnten, um sowohl die Vorteile von Neo4j als auch die Vorteile von Minibase zu kombinieren. Danach folgt ein kurzer Einblick in Indexe, bevor näher auf das Thema der Anfragesprache und Optimierung eingegangen wird.

2 File Storage in Neo4j

2.1 Node

- fixe Größe von 15 Bytes (früher 9 Bytes).
- besitzt 4 Byte Referenz zu Key-Value Properties als einfach verlinkte Liste dieses Knotens.
- besitzt 4 Byte Referenz zum Startpunkt der Beziehungen bzw. Kanten, welche als doppelt verlinkte Liste gespeichert werden.
- besitzt 4 Byte (in neueren Versionen) für bis zu 7 Label IDs bzw. Referenz in den Dynamic Store bei mehr als 7 Label IDs.

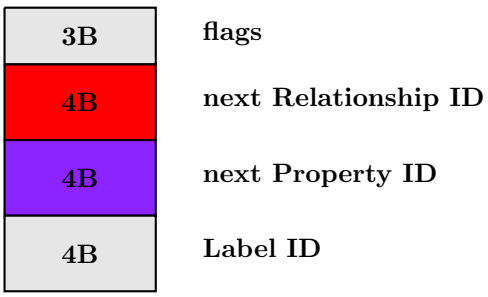

Abbildung 1: Aufbau und Struktur einer Node (in Anlehnung an [3] und [4])

2.2 Relationship

- fixe Größe von 33 Bytes.
- besitzt jeweils eine 4 Byte Referenz zum Start- und Endknoten der Beziehung bzw. Kante.
- besitzt eine 4 Byte Referenz zur ID des Typs dieser Referenz, wie *Freundschaft, ...*.
- besitzt vier 4 Byte Referenzen zur vorherigen und nächsten Beziehung bzw. Kante vom Start- und Endknoten als doppelt verlinkte Liste.
- besitzt eine 4 Byte Referenz zu Key-Value Properties als einfach verlinkte Liste dieser Beziehung bzw. Kante.

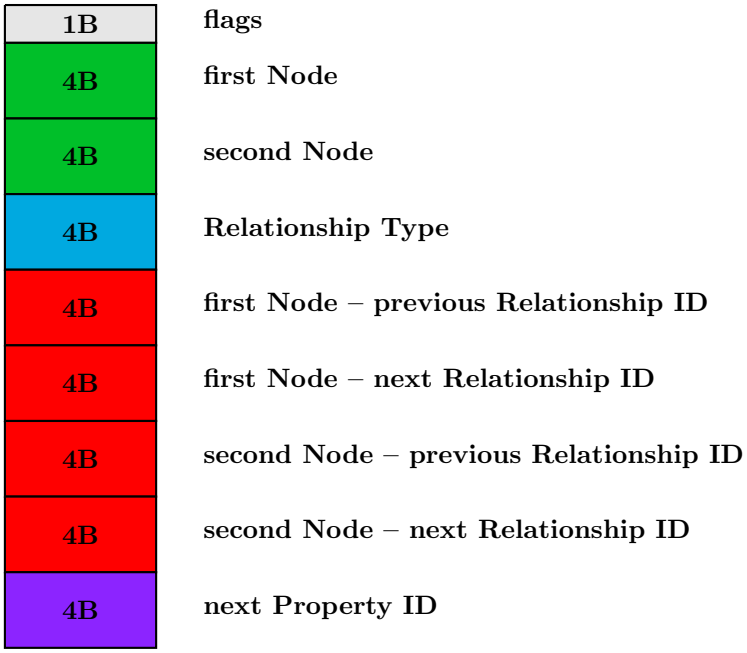

Abbildung 2: Aufbau und Struktur einer Relationship (in Anlehnung an [3] und [4])

2.3 Relationship Group

Sollte man nur an Beziehungen eines speziellen Relationship Typs interessiert sein, so muss im vorherigen Modell durch alle Beziehungen iteriert werden. Um dies zu umgehen, wird in neueren Versionen auch eine Relationship Group für dichte Knoten angeboten. Bei solchen dichten Knoten wird zunächst in einer verlinkten Liste durch die verschiedenen Relationship Types iteriert, welche aus einer Relationship Group bestehen. Solch eine Relationship Group besitzt dann den entsprechenden Relationship Typ und wieder verlinkte Listen zu allen Beziehungen dieses Typs.

- fixe Größe von 25 Bytes.
- besitzt eine 4 Byte Referenz zum eigentlichen Knoten.
- besitzt eine 4 Byte Referenz zur ID des Typs dieser Referenz, wie *Freundschaft, ...*.
- besitzt eine 4 Byte Referenz zur ersten Beziehung, in der der Knoten als Startknoten agiert.

- besitzt eine 4 Byte Referenz zur ersten Beziehung, in der der Knoten als Endknoten agiert.
- besitzt eine 4 Byte Referenz zur ersten Beziehung, in der der Knoten als Loop-Knoten agiert.
- besitzt eine 4 Byte Referenz zur nächsten Relationship Group.

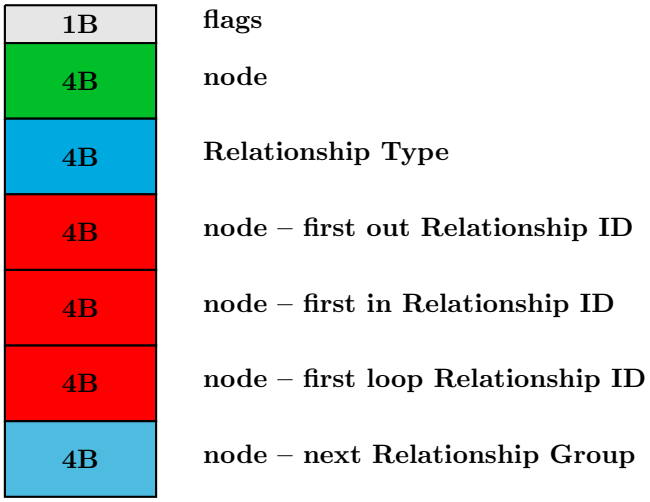

Abbildung 3: Aufbau und Struktur einer Relationship Group (in Anlehnung an [3])

2.4 Relationship Type

- fixe Größe von 5 Bytes.
- Eine Beziehung kann einen bestimmten Typ haben, wie *Freundschaft, ...*.
- Diese werden als Strings gespeichert. Da Strings eine variable Länge besitzen, gibt es hier eine 4 Byte Referenz zum entsprechenden String, welcher separat im Dynamic Store gespeichert wird.

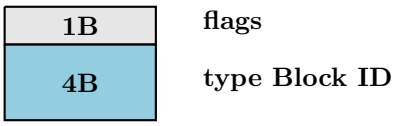

Abbildung 4: Aufbau und Struktur einer Relationship ID (in Anlehnung an [4])

2.5 Property

- fixe Größe von 41 Bytes.
- Da Knoten und Beziehungen i.d.R. mehr als eine Eigenschaften besitzen, werden in einer solchen 41 Bytes Property mehrere zusammengefasst (im Property Block).
- Solch ein Key-Value Paar besteht aus einem Property Index und dem entsprechenden Wert. Sollte der Wert eine fixe Größe besitzen, wie ein Integer, so wird dieser direkt im Property Block gespeichert.

- Sollte der Wert sehr viele Bytes oder eine variable Länge besitzen (Strings), so wird eine Referenz zum eigentlichen Wert wie beim Relationship Type gespeichert. Kurze Strings hingegen können auch direkt im Property Block gespeichert werden.
- Es werden zwar einige Key-Value Paare in diesem 41 Byte Block zusammengefasst, jedoch kann ein Knoten deutlich mehrere Key-Value Paare besitzen. Deshalb werden alle solche Property Blöcke eines Knotens als doppelt verlinke Liste mit jeweils 4 Byte Referenzen gespeichert.

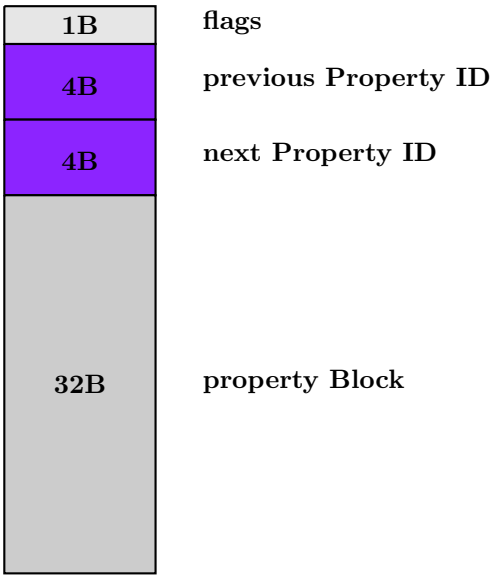

Abbildung 5: Aufbau und Struktur einer Property (in Anlehnung an [1], [3] und [4])

2.6 Property Index

- fixe Größe von 9 Bytes.
- Schlüssel werden als Strings gespeichert. Da Strings eine variable Länge besitzen, gibt es hier eine 4 Byte Referenz zum entsprechenden String, welcher separat gespeichert wird.

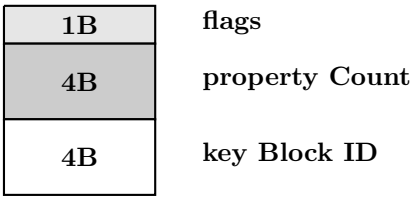

Abbildung 6: Aufbau und Struktur eines Property Index (in Anlehnung an [4])

2.7 Dynamic Store

- Variable Strings oder große Datentypen werden nicht im Property Block, sondern direkt in einem Dynamic Store gespeichert.
- Auch Label-Namen oder Namen des Reationship Typs werden im Dynamic Store gespeichert.

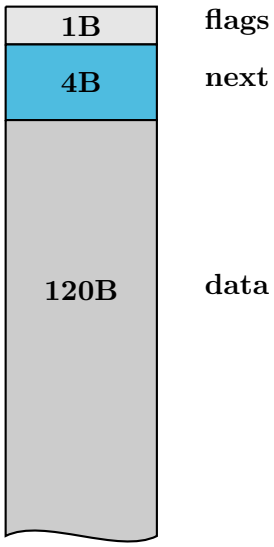

Abbildung 7: Aufbau und Struktur eines Dynamic Store (in Anlehnung an [3] und [4])

2.8 Kleines Beispiel

Um den Aufbau der einzelnen *Files* von Knoten, Beziehungen, ..., anschaulicher darzustellen, findet sich in Abbildung 8 ein kleiner Beispielgraph mit vier Knoten.

2.8.1 Knoten

- Knoten 1 besitzt zwei Eigenschaften mit Name = Alice und Age = 25, die als verlinkte Liste von Properties gespeichert werden. Außerdem besitzt sie zwei Beziehungen zu Knoten 2 und 3, zu der sie eine Referenz zur ersten Beziehung (zu Knoten 3) besitzt.
- Knoten 2 besitzt eine Eigenschaft mit Name = Bob, der als verlinkte Liste von Properties gespeichert wird. Außerdem besitzt er zwei Beziehungen zu Knoten 1 und 4, zu der er eine Referenz zur ersten Beziehung (zu Knoten 1) besitzt.
- Knoten 3 besitzt drei Eigenschaften mit Name = Peter, Club = TVL und Age = 21, die als verlinkte Liste von Properties gespeichert werden. Zusätzlich besitzt dieser Knoten ein Label mit dem Wert Student. Außerdem besitzt er zwei Beziehungen zu Knoten 1 und 4, zu der er eine Referenz zur ersten Beziehung (zu Knoten 1) besitzt.
- Knoten 4 besitzt eine Eigenschaft mit Name = Neo, der als verlinkte Liste von Properties gespeichert wird. Außerdem besitzt er zwei Beziehungen zu Knoten 2 und 3, zu der er eine Referenz zur ersten Beziehung (zu Knoten 2) besitzt.

2.8.2 Beziehungen

- Die Beziehung zwischen Knoten 1 und 2 ist vom Typ *KNOWS* und hat als Eigenschaft Since = 2006. Sie besitzt Referenzen zum Start- (Knoten 1) und Endknoten (Knoten 2). Die nächste Beziehung von Knoten 2 ist die Beziehung zwischen Knoten 2 und Knoten 4, während die vorherige Beziehung von Knoten 1 die Beziehung zwischen Knoten 1 und Knoten 3 ist.
- Die Beziehung zwischen Knoten 1 und 3 ist vom Typ *KNOWS*. Sie besitzt Referenzen zum Start- (Knoten 1) und Endknoten (Knoten 3). Die nächste Beziehung von Knoten 1 ist die Beziehung zwischen Knoten 1 und Knoten 2, während die nächste Beziehung von Knoten 3 die Beziehung zwischen Knoten 3 und Knoten 4 ist.
- Die Beziehung zwischen Knoten 2 und 4 ist vom Typ *KNOWS*. Sie besitzt Referenzen zum Start- (Knoten 2) und Endknoten (Knoten 4). Die vorherige Beziehung von Knoten 2 ist die Beziehung zwischen Knoten 1 und Knoten 2, während die nächste Beziehung von Knoten 4 die Beziehung zwischen Knoten 3 und Knoten 4 ist.
- Die Beziehung zwischen Knoten 3 und 4 ist vom Typ *KNOWS*. Sie besitzt Referenzen zum Start- (Knoten 3) und Endknoten (Knoten 4). Die vorherige Beziehung von Knoten 3 ist die Beziehung zwischen Knoten 1 und Knoten 3, während die vorherige Beziehung von Knoten 4 die Beziehung zwischen Knoten 2 und Knoten 4 ist.

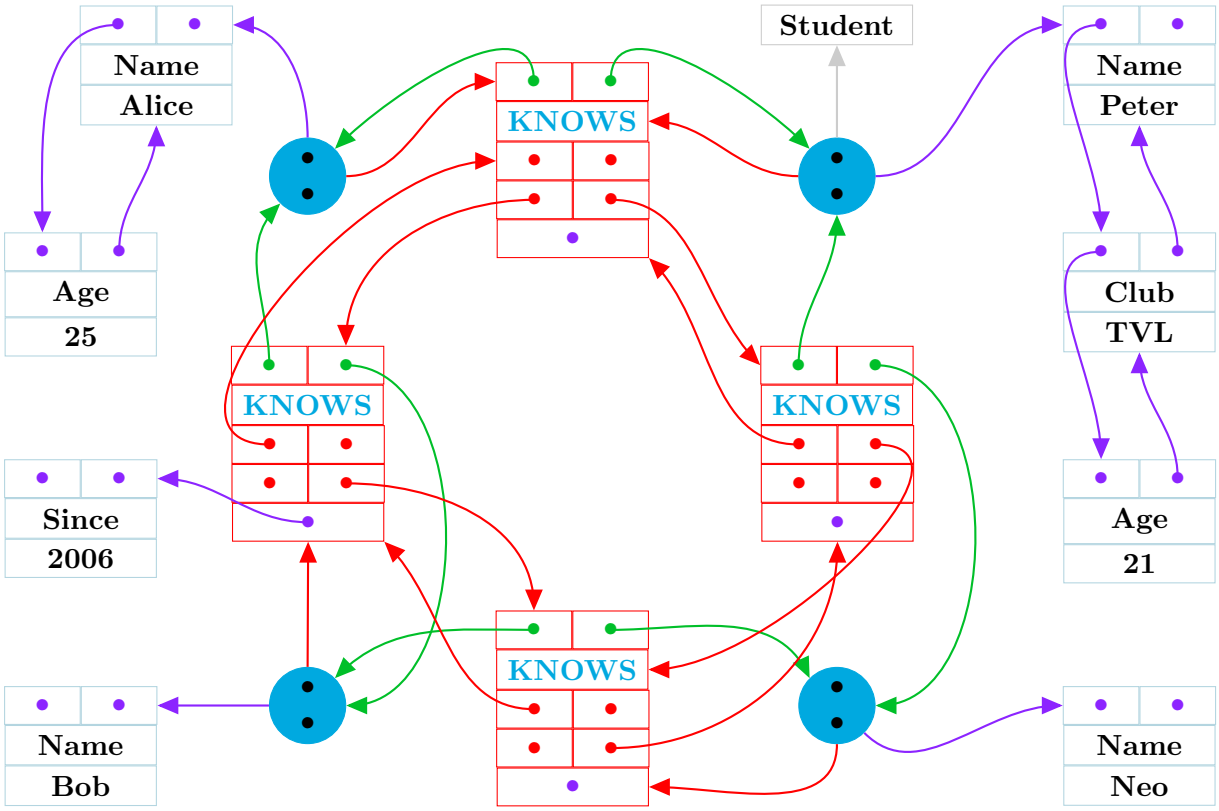

Abbildung 8: Beispiel eines kleinen Graphs (in Anlehnung an [4])

3 Mögliche Integration in Minibase

In diesem Abschnitt werden zwei Varianten vorgestellt um Graphen in Minibase zu integrieren. In der ersten Variante werden zwar die Tabellenstruktur von Minibase verwendet, jedoch werden (alle) Referenzen zwischen Kanten und Knoten direkt in der Tabelle gespeichert. Die zweite Variante verzichtet auf diesen *neuen Datentyp*, indem Referenzen separat in einer (neuen) Datenstruktur gespeichert und Tabellen wie bisher verwendet werden können.

Die Integration von Graphen sollte sich nicht zu stark an die Implementation von Neo4j orientieren, da auch diese Nachteile besitzt. Der entscheidende Aspekt sollte deshalb sein, wie können die Vorteile von RDBMS, wie es sie in Minibase gibt, verwendet werden um Graphen zu speichern. Eine weiterer Aspekt, in wie weit eine Struktur von Knoten eines bestimmten Labels und Kanten einer bestimmen Beziehung vorausgesetzt werden kann, sollte ebenfalls nicht vernachlässigt werden.

3.1 Variante 1 - Neo4j-orientiert

3.1.1 Neuer Datentyp

Zunächst müsste ein neuer Datentyp hinzugefügt werden, der die Referenz, RID, zu einem anderen Knoten oder zu einer Beziehung direkt in der Tabelle speichert (im Vergleich zu einem Fremdschlüssel). Dadurch würde das Traversieren im Graphen wie in Neo4j möglich.

- Erstelle neuen Datentyp RID, der eine Referenz zu einem Tupel in einer anderen Tabelle ist.

3.1.2 Node

Integration

- Speichere alle Knoten eines Labels in einer Tabelle. Ein Knoten wird durch eine Reihe bzw. Tupel repräsentiert.
- Sollte der Knoten kein Label besitzen, füge diesen einer Extra-Tabelle hinzu, in der alle Knoten ohne ein Label gespeichert werden.
- Füge eine *Zusatzspalte* mit dem Datentyp RID hinzu, in der jeder Knoten bzw. das neue Tupel auf die erste Beziehung in der entsprechenden Beziehungs-Tabelle zeigt (wie in Neo4j).

Vorteile

- Pro Knoten werden letztlich nur die Werte gespeichert. Das Speichern des Schlüssels bzw. der Schlüssel-ID erfolgt nur im Tabellen-Header (Vorteil der Tabellenstruktur).
- Effizientere Speicherverwendung bzw. weniger Overhead.
- Verwendung des Labels als Aufteilung der Knoten. Ist das Label eines Knoten bekannt, kann direkt in die entsprechende Tabelle gesprungen werden.
- Es können alle bisherigen Minibase-Features sowie Anfragen auch auf Knoten verwendet werden, da sie wie Tabellen gespeichert werden (Umgang mit RID ggfs. ignorieren).

Probleme und Einschränkungen

- Sollten Knoten des gleichen Labels eine zu unterschiedliche Struktur besitzen, müssen viele NULL-Werte gespeichert werden.
- Die Struktur von Knoten eines Labels ist zu Beginn nicht bekannt. Tabellen müsstenöfters erweitert werden.
- Wie soll die Tabellen-Struktur bei Knoten ohne Labels aussehen?
- In Neo4j können Knoten mehrere Labels besitzen.

  Mögliche Lösung: Speichere die RID eines Knotens als Vorwärtszeiger in allen Tabellen, die diesen Knoten theoretisch besitzen würden. Dadurch kann ohne großen Zusatzspeicher ein Knoten mehrere Labels besitzen. Nachteil: Es gibt keine direkte Verbindung vom Knoten zu allen Labels (Probleme beim Löschen).

  Weitere Möglichkeit: Beschränkung auf nur ein Label pro Knoten.
- In Neo4j sind Arrays als Datentypen möglich.
- Ist das Label eines Knoten nicht bekannt, muss durch alle (Neo4j)-Tabellen iteriert werden.

3.1.3 Relationship

Integration

- Speichere alle Beziehungen eines Typs in einer Tabelle. Eine Beziehung wird durch eine Reihe bzw. Tupel repräsentiert.
- Speichere die Key-Value Paare wie normale Spalten im RDBMS.
- Zusätzlich werden Start- und Endknoten mit einer RID gespeichert.
- Zusätzlich werden nächste und vorherige Beziehung vom Start- und Endknoten mittels RID (als verlinkte Liste in Spalten) gespeichert.

Vorteile

- Pro Beziehung werden letztlich nur die Werte gespeichert. Das Speichern des Schlüssels bzw. der Schlüssel-ID erfolgt nur im Tabellen-Header (Vorteil der Tabellenstruktur).
- Effizientere Speicherverwendung bzw. weniger Overhead.
- Verwendung des Labels als Aufteilung der Beziehungen, wie in Relationship Group. Ist der Beziehungs-Typ bekannt, kann direkt in die entsprechende Tabelle gesprungen werden.
- Es können (alle) bisherige(n) Minibase-Features sowie Anfragen auch auf Beziehungen verwendet werden, da sie wie Tabellen gespeichert werden (Umgang mit RID ggfs. ignorieren).

Probleme und Einschränkungen

- Sollten Beziehungen des gleichen Typs eine zu unterschiedliche Struktur besitzen, müssen ggfs. viele NULL-Werte gespeichert werden.
- Die Struktur von Knoten eines Labels ist zu Beginn nicht bekannt. Tabellen müssten öfters erweitert werden.
- In Neo4j sind Arrays als Datentypen möglich.

3.2 Variante 2 - Minibase-orientiert (favorisiert)

In dieser Variante werden die Knoten und Beziehungen ganz normal in Minibase mit allen bisherig verfügbaren Datentypen gespeichert. Der große Vorteil von Neo4j ist letztlich, dass kein JOIN zur Laufzeit generiert werden muss, da alle Beziehungen durch Zeiger bereits im Speicher vorhanden ist. Um diesen Vorteil auch in Minibase zu besitzen, werden diese Art von *JOINs* ebenfalls im Speicher hinterlegt.

3.2.1 Knoten

Integration

- Speichere alle Knoten eines Labels in einer Tabelle. Ein Knoten wird durch eine Reihe bzw. Tupel repräsentiert.
- Sollte der Knoten kein Label besitzen, füge dieser einer Extra-Tabelle hinzu, in der alle Knoten ohne ein Label gespeichert werden.

3.2.2 Relationship

Integration

- Speichere alle Beziehungen eines Typs in einer Tabelle. Eine Beziehung wird durch eine Reihe bzw. Tupel repräsentiert.
- Speichere die Key-Value Paare wie normale Spalten im RDBMS.
- Zusätzlich werden Start- und Endknoten als normaler Fremdschlüssel mit der Knoten ID gespeichert.

3.2.3 Neue interne Datenstruktur

- Erstelle einen B+ Baum für jeden Knoten, der alle Beziehungen (RID) dieses Knoten enthält (wie ein Index).
- Als Schlüssel sollen der Beziehungstyp (primär) und Zielknoten-Label (sekundär) verwendet werden.
- Als Werte soll die RID zur Beziehung verwendet werden.
- Damit die Schlüssel keine Strings sind, sollte ein Mapping in einer Extra-Tabelle zu einem Integer erfolgen.

- Dadurch sind in der Blattknoten Ebene des B+ Baums alle Beziehungen zuerst nach dem Beziehungstyp und dann nach dem Zielknoten-Label geordnet. Beim Iterieren durch die Beziehungen erfolgt somit kein (zufälliges) Springen durch verschiedene Tabellen. Diese werden der Reihenfolge nach iteriert.
- Sollten Knoten sehr wenige Beziehungen besitzen, kann solch eine Struktur auch auf allen Knoten eines Labels erstellt werden. Dann ist der Schlüssel eines Eintrags in diesem B+ Baum: Knoten ID, Beziehungstyp, Zielknoten-Label.

3.2.4 Analyse

Vorteile

- Da die Tabellen wie bisher in Minibase verwendet werden, können neben den neuen Anfragen auch alle SQL Anfragen ausgeführt werden.
- Keine Speicherung der verlinkte Liste innerhalb einer Tabelle.
- Pro Knoten und Beziehung werden letztlich nur die Werte gespeichert. Das Speichern des Schlüssels bzw. der Schlüssel-ID erfolgt nur im Tabellen-Header (Vorteil der Tabellenstruktur).
- Prefetching-Vorteil: Durch Gliederung der Beziehungen in Beziehungstyp und Endknoten-Label erfolgt der Zugriff auf die Beziehungen und späteren Endknoten der Reihe nach. Kein zufälliges Springen zwischen verschiedenen Tabellen und letztlich Pages.
- Da alle bisherigen Minibase-Features komplett unterstützt werden, könnte es auch Anfragen geben, die mit der bisherigen Engine schneller laufen (siehe auch weiter unten Abschnitt 5).
- Es könnten weitere Indexe erzeugt werden, die Eigenschaften einer Beziehung mit in den Schlüssel aufnehmen anstatt Beziehungstyp oder Endknoten-Label.

Probleme und Einschränkungen

- Es kann nicht direkt von einem Knoten zu einer Beziehung (sowie umgekehrt) gesprungen werden. Es muss zunächst über den *Index* gegangen werden. Je nach der Höhe des Baums können zusätzliche I/O Kosten entstehen, sofern diese nicht immer im Arbeitsspeicher gehalten werden können.
- Das Aktualisieren könnte mehr Kosten verursachen, da der B+ Baum aktualisiert werden muss. Auf der anderen Seite könnte es auch weniger Kosten verursachen, da nicht durch eine komplette Liste von Beziehungen iteriert werden muss.
- Bei kleinen Knoten ggfs. viel Overhead. *Mögliche Lösung:* Fasse alle Knoten eines Labels in einem B+ Baum zusammen.
- Sollten Knoten des gleichen Labels eine zu unterschiedliche Struktur besitzen, müssen viele NULL-Werte gespeichert werden.
- Die Struktur von Knoten eines Labels ist zu Beginn nicht bekannt. Tabellen müssten öfters erweitert werden.
- Wie soll die Tabellen-Struktur bei Knoten ohne Labels aussehen?

- In Neo4j können Knoten mehrere Labels besitzen.
- In Neo4j sind Arrays als Datentypen möglich.
- Ist das Label eines Knoten nicht bekannt, muss durch alle (Neo4j)-Tabellen iteriert werden.

4 Index

Da in beiden Varianten Knoten und Beziehungen in Tabellen gespeichert werden, können Indexe auch auf allen deren Datentypen erstellt werden, die durch Minibase unterstützt werden. Insbesondere in Variante 2 könnten solche Indexe im Query Optimizer herangezogen werden, sollten sie Eigenschaften der Beziehungen oder Knoten besitzen, die eine Vorselektierung vornimmt.

5 Query

5.1 Cypher

Um Anfragen auf Knoten und Kanten auszuführen, könnte die Cypher-Anfragen-Sprache wie in Neo4j unterstützt werden. Je nachdem welche Integrations-Variante gewählt würde, könnten einige Anfragen bzw. Teil-Anfragen (Selektion, Projektion, Insert, Update und Delete) wie bisher durchgeführt werden, da die Knoten und Beziehungen in Tabellen gespeichert werden. Durch das Speichern von Knoten mit demselben Label in einer Tabelle könnten außerdem Performance Vorteile erzielt werden. Das gleiche gilt für das Speichern von Beziehungen mit dem gleichen Beziehungstyp in einer Tabelle.

5.1.1 Match

Der wichtigste neue Operator, der hinzugefügt werden müsste, ist der Match-Operator, der das Traversieren von Knoten zu Beziehungen und umgekehrt vornimmt. Er übernimmt letztlich die Aufgabe eines JOINs. Der große Vorteil ist, dass alle Referenzen bereits zwischen den Tabellen vorliegen. Dafür müsste entsprechende Such-Algorithmen, die dieses Traversieren ermöglichen, hinzugefügt werden [2].

5.2 Erweitertes SQL

Insbesondere bei der zweiten Integrations-Variante könnte auch ein erweitertes SQL Abhilfe schaffen, da die Knoten und Beziehungen wie bisher gespeichert werden. Die komplette Bandbreite an SQL-Anfragen ist weiterhin möglich. Um das Traversieren von Knoten und Kanten ohne einen JOIN zu ermöglichen, könnte ein neues Schlüsselwort, wie MATCH, eingeführt werden. Wird MATCH zwischen zwei Tabellen verwendet, so findet eine Traversierung bzw. Suche mit Hilfe der neuen internen Datenstruktur statt. Durch Erweitern des Kostenmodells könnte auch eine Abschätzung getätigt werden, ob ein JOIN nicht doch die günstigere Variante wäre, und eine Entscheidung je nach Anfrage getroffen werden. Das gleiche könnte auch in der anderen Richtung erfolgen, wenn ein JOIN aufgerufen wird, obwohl ein MATCH schneller wäre, da es sich um zwei Tabellen handelt, die Knoten enthalten.

6 Query Optimizer

Erst nachdem die Integrationsvariante und die Anfragesprache gewählt sind, kann auf dieses Thema näher eingegangen werden.

Literatur

[1] Chris Gioran. The new neo4j property store, 2011. Verfügbar auf http://digitalstain.blogspot.de/2011/11/rooting-out-redundancy-new-neo4j.html.

[2] Neo4j. Neo4j manual, 2016. Verfügbar auf https://neo4j.com/docs/developer-manual/.

[3] Roman Leventov. Neo4j architecture, 2015. Verfügbar auf http://key-value-stories.blogspot.de/2015/02/neo4j-architecture.html.

[4] Tobias Lindaaker. An overview of neo4j internals, 2012. Verfügbar auf http://www.slideshare.net/thobe/an-overview-of-neo4j-internals and https://skillsmatter.com/skillscasts/2968-neo4j-internals.