InfiniteGraph, die verteilte Graphendatenbank

Oliver Koch
Universität Siegen
oliver.koch@student.uni-siegen.de

ABSTRACT

Relationale Datenbanken stehen immer häufiger vor einem Leistungsproblem: Sie sind nicht für große und sich schnell verändernde Datenmengen, wie zum Beispiel soziale Netzwerke sie nutzen, geeignet. Um dieses Problem zu lösen entstanden nicht-relationale Datenbanken.
Dieser Artikel befasst sich mit der NoSQL Graphendatenbank InfiniteGraph, welche von dem Unternehmen Objectivity Inc. entwickelt wird und auf der Objektdatenbank Objectivity/DB basiert. Es wird die grundlegene Architektur und Funktionsweise von InfiniteGraph erläutert, um zu ermitteln, in welchen Bereichen es sinnvoll ist diese Datenbank zu nutzen.

1. EINLEITUNG

Big Data ist heutzutage einer der wichtigsten Begriffe im Bereich der Datenspeicherung. Laut einer Studie der IDC (International Data Corporation) aus dem Jahr 2014 verdoppelt sich die Speichergröße der Daten alle zwei Jahre. Während im Jahr 2013 noch 4,4 Zettabyte digitale Daten gespeichert wurden, sollen es im Jahr 2020 bis zu 44 ZettaByte (44 Billion GB) werden. Schon jetzt haben relationale Datenbanken Probleme, diese Daten zu verarbeiten und nutzen, da es zu viele sind und sie sich zu schnell verändern. Im Zuge dessen wurden viele neue Datenbanken entwickelt, die einen anderen Ansatz verfolgen. [9, 12]
InfiniteGraph ist eine dieser Datenbanken und lässt sich in die Kategorie der Graphendatenbanken einordnen. Version 1.0 erschien im Jahr 2010, seitdem wird sie von dem Entwickler Objectivity Inc. stetig erweitert und gepflegt. InfiniteGraphs Infrakstruktur baut auf der Objektdatenbank Objectivity/DB auf, welche ebenfalls von Objectivity Inc. entwickelt wird. Dadurch erbt InfiniteGraph bestimmte Funktionen, wie zum Beispiel die Replikation oder SQL Anfragen. Ziel der Arbeit ist es, sowohl einen Einblick in die technischen Aspekte, als auch in die Funktionsweise von InfiniteGraph zu gewähren, um am Ende feststellen zu können, ob diese Graphendatenbank dazu geeignet ist, mit den immer stetig steigenden Datenmengen umzugehen und in welchen Bereichen sie eingesetzt werden kann. [11]
Im folgenden Kapitel wird der Aufbau von Graphendatenbanken erklärt. Im Anschluss erläutert Abschnitt drei *Technischer Aufbau* die Technik hinter InfiniteGraph. Kapitel vier *Nutzung von InfiniteGraph* befasst sich mit der Konfiguration und dem Benutzen der Datenbank. Am Ende zeigt Abschnitt fünf *Geschwindigkeit* wie schnell die Datenbank arbeitet.

2. AUFBAU GRAPHEN DATENBANKEN

Eine graphenorientierte Datenbank benutzt Graphen, um Informationen zu speichern und darzustellen. Diese Graphen bestehen aus Knoten und Kanten, wobei Kanten die Verbindungen/Beziehungen zwischen den Knoten beschreiben. Man unterscheidet grundsätzlich zwischen zwei Arten von Graph-Modellen: Das *einfache Graph-Modell* und das *Property-Graph-Modell*.
Ein Graph im *einfachen Graph-Modell* wird durch zwei Mengen beschrieben, und zwar der Knotenmenge V und der Kantenmenge E. Bei den Kanten wird zwischen gerichtet (einseitige Beziehung) und ungerichtet (beidseitige Beziehung) unterschieden. Ein Graph besteht aus einer der beiden Arten, eine Kombination ist nicht möglich. Weiterhin ist es möglich numerische Werte in Kanten zu speichern, man spricht dann von einem gewichteten Graphen.

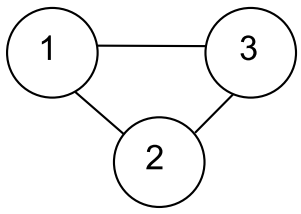

Figure 1: Einfaches Graphmodell

Das *Property-Graph-Modell* ist eine Erweiterung von dem einfachen Modell. Auch hier ist es möglich, gerichtete und ungerichtete Kanten zu erstellen. Es wird hier jedoch nicht nur ein Wert in den Knoten oder Kanten gespeichert, sondern Key/Value-Beziehungen, wie zum Beispiel „Vorname: Oliver“. Zusätzlich bekommt jeder Knoten eine ID zur Bestimmung einer eindeutigen Identität, welche in den Knoten gespeichert und zur Identifikation benutzt wird. [1]

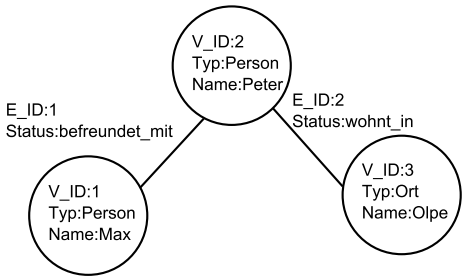

Figure 2: Property Graphmodell

2.1 Traversierung

Neben der Angabe einer ID können Graphendatenbanken auch auf die Traversierung eines Graphen zurückgreifen, um von einem gegebenen Startknoten einen weiteren Zielknoten zu suchen oder um die Verbindungen zwischen den Objekten darzustellen. Es gibt mehrere Algorithmen zur Traversierung, die folgenden zwei sind für diese Arbeit jedoch am wichtigsten.

2.1.1 Breitensuche

Die Breitensuche durchsucht als erstes alle Nachbarknoten des Startknotens. Im Anschluss werden von diesen Nachbarknoten die Nachbarn durchsucht. Ein besuchter Knoten wird markiert. Falls ein Knoten schon durch einen kürzeren Pfad markiert wurde, wird dieser nicht mehr neu besucht. Dieser Vorgang wiederholt sich, bis der gewünschte Zielknoten gefunden oder, falls kein Zielknoten angegeben, der komplette Graph durchlaufen wurde. [1]

2.1.2 Tiefensuche

Die Tiefensuche verfolgt, anders als die Breitensuche, einen Pfad solange, bis sie an einen Knoten kommt, der keine weiteren ausgehenden Kanten mehr besitzt. Im Anschluss wird dieser Pfad nun rückwärts verfolgt, bis wieder ein Knoten erreicht wurde, der noch andere nicht benutzte ausgehende Kanten besitzt. Wie bei der Breitensuche werden bereits besuchte Knoten markiert. Dort wird die Suche dann fortgesetzt. [1]

3. TECHNISCHER AUFBAU

Die folgenden Abschnitte befassen sich mit der Architektur von InfiniteGraph, die sich auf mehrere Dateien verteilt. Weiterhin werden wichtige Themen wie die Datenspeicherung und der Zugriff auf selbige erläutert.

3.1 Datenmodell

Das Datenmodell von InfiniteGraph entspricht einem Property-Graphen. Die Knoten und Kanten werden durch objektorientierte Klassen dargestellt. Diese erben von Basisklassen aus der API Schnittstelle. Die Basisklassen geben ein statisches Grundschema vor, zum Beispiel eine eindeutige ID. Attribute der Klassen können weitere Eigenschaften hinzufügen. Kanten besitzen in InfiniteGraph eine Gewichtung, wodurch bei Analyseverfahren eine reduzierte Betrachtung von Kanten entsteht, was sehr hilfreich ist, da dort häufig ein großer Teil der betrachteten Kanten unwichtig ist. Für die Indizierung nutzt InfiniteGraph eigene Indextypen, jedoch auch Alternativen wie sie das Open Source Project Lucene anbietet. Um einen schnelleren Zugriff auf Daten zu ermöglichen, können Knoten Namen gegeben werden. [2, 11]

3.2 Architektur

Der Aufbau einer Datenbank in Infinitegraph verteilt sich auf mehrere Dateien, wobei der größte Teil in der Speicherung von Daten liegt. Um eine Datenbank zu erstellen, muss mindestes ein Name angegeben werden. Dies reicht aus, damit InfiniteGraph einen Initial Graphen und die restlichen benötigten Dateien erstellt. Zusätzlich kann auch noch eine Konfigurationsdatei (engl. property file) angefügt werden. In dieser können der Standort der Datenbank auf der Festplatte festgelegt werden, aber auch die Anpassung von speziellen Eigenschaften wie die Spezifikation der Speichereinheit (engl. Storage Unit) oder die Behandlung der *MROW* (Multiple Readers, One Writer) Funktionalität sind möglich. [2, 11]

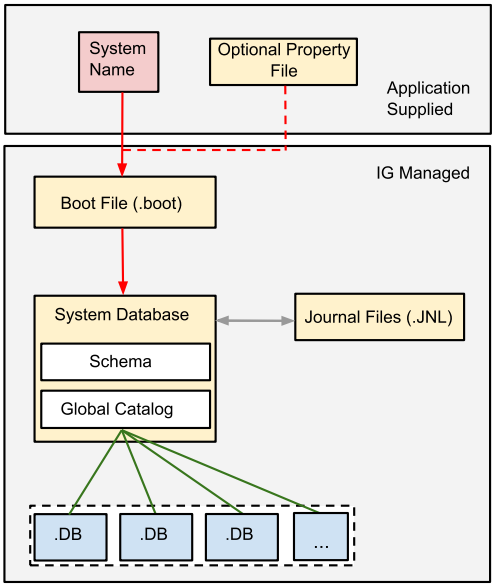

Figure 3: IG Archtitektur

3.2.1 Bootdatei

In der Bootdatei befinden sich alle Informationen, die eine Applikation benötigt, um die Graphendatenbank zu öffnen. Wenn also ein Programm auf die Datenbank zugreifen möchte, stellt es die Verbindung durch die Bootdatei her. Hier werden auch Informationen aus der Konfigurationsdatei eingebunden. Dazu zählen unter anderem der verwendete Lockserver sowie der Name der Datenbank. Außerdem enthält sie den Pfad zur Systemdatenbankdatei. [2, 11]

3.2.2 Systemdatenbankdatei

Die Systemdatenbankdatei (engl. System Database file) speichert Informationen über die Graphen. Bei der Erstellung der Datenbank werden hier anfangs nur die Informationen zu den Standorten der vom Graphen genutzen Daten gespeichert (Global Catalog). Bei der ersten Speicherung eines Objektes in der Datenbankt erzeugt InfiniteGraph das zugehörige Schema (ähnlich zu Java Klasse), dieses wird ebenfalls in der Systemdatenbankdatei gespeichert. [2, 11]

3.2.3 Journaldatei

Sobald Programme Änderungen an der Datenbank vornehmen, erzeugt InfiniteGraph Journaldatein. Falls eine Transaktion abbricht bzw. fehlschlägt werden diese genutzt, um den letzten konsistenten Datenbankzustand wieder herzustellen. [2, 11]

3.2.4 Speichereinheit

Die Speicherung von Daten findet in der Speichereinheit (engl. Storage Unit) statt. Diese besteht aus drei oder mehr Datenbankdateien, welche sich in drei Arten aufteilen: Knoten, Kanten und Connectoren(beschreiben welche Kante zu welchen Knoten gehört). Jede dieser Datenbankdateien kann einen oder mehrere Container enthalten. Sie sind die technische Umsetzung der Speicherung von Daten auf der Festplatte. Sie werden nochmal unterteilt in storage pages, die die minimale Größe von Containern besitzen und die Daten von und zu der Festplatte übertragen. Die storage pages können eine beliebige Menge von Daten enthalten. Durch das Speichern der Daten in Storage Units, ist es besonders einfach diese auf mehrere Systeme zu verteilen. [2, 11]

3.3 Speicherung von Daten

InfiniteGraph nutzt für die Speicherung von Daten ein flexibles Platzierungssystem namens *managed placement*. Dieses nutzt eine modellbasierte Technik, um neue Daten in der Graphendatenbank zu speichern. Jedes mal, wenn ein Programm Daten erzeugt, schaut managed placement im Platzierungsmodell (engl. placement model, beinhaltet Regeln für Auswahl des Speicherortes) nach, wo die Daten gespeichert werden sollen.
Eine neu erstellte Datenbank besitzt ein standart placement model, welches für die meisten Programme ausreichend ist. Es legt automatisch fest, wann und wo neue storage pages, container und Datenbankdateien erstellt werden müssen, um neue Daten unterbringen zu können. Objekte der Knoten und Kanten befinden sich in den Datenbankendateien *VertexGroup* und *EdgeGroup*. Informationen zu Kanten stehen in der dritten Datenbankdatei *ConnectorGroup*.
Falls Datenbankdateien eine feste Größe erreicht haben oder die maximale Anzahl an Cotainer besitzen, wird eine neue Datenbankdatei erstellt. Standardmäßig besitzt eine Datei anfangs einen Container, der die Größe von 3200KB erreichen kann, falls benötigt, werden neue erzeugt. Ein Container startet mit 100 storage pages, die jeweils 16KB groß sind. Somit ist ein maximum von 200 storage pages pro Container möglich. [2]

3.4 Festlegen der Speicherstandorte

Der standardmäßige Speicherort von Daten befindet sich, falls nicht anders festgelegt, im selben Verzeichnis wie die Systemdatenbankdatei. Bei größeren Graphendatenbanken, auf die von vielen Programmen mit unterschiedlichen Standorten zugegriffen wird, macht es jedoch Sinn, die Daten verteilt zu speichern. Zum Beispiel möchte man Storage Units an Standorten einrichten, die geografisch näher an den Programmen sind die sie benutzen. Dies kann mit dem Registrieren von Speicherstandorten durch das administrative Tool *AddStorageLocation* erreicht werden. Im Anschluss müssen nur noch die Einstellungen der Applikation auf den neuen Standort geändert werden. Jeder Standort der registriert wird ist verfügbar für jedes Programm, das Zugriff auf die Graphendatenbank hat. Eine Zusammenfassung der Speicherstandorte legt InfiniteGraph in der main Storage group (MSG) ab. [2]

3.5 Lockserver

Ein Lockserver hat die Aufgabe, den Zugriff auf die Datenbank durch Programme oder Nutzer zu steuern, um die Daten konsistent zu halten. Anfragenden Tranksaktionen werden Locks gegeben oder entzogen. Greift eine Transaktion nun auf einen Container zu, verhindert der Lockserver weitere Anfragen, indem er die benutzten Datein lockt. Möchte nun eine zweite Transaktion ebenfalls auf den Container zugreifen, so wird der Zugriff nur erlaubt, wenn die beiden Locks kompatibel sind. Handelt es sich bei beiden Locks um Leseanfragen, so sind parallele Zugriffe erlaubt. Auch eine Kombination von lesen und schreiben ist möglich, mehr als ein Schreibzugriff ist jedoch unzulässig. Da InfiniteGraph MROW (Multiple Readers, One Writer) erlaubt, ist es möglich, stale Data (veraltete Daten) zu erhalten. Zwar lässt sich diese Eigenschaft abschalten, führt jedoch zu Performance Verlust. [2, 11]

3.6 Zugriff auf Daten

Möchte ein Programm auf Daten der Graphendatenbank zugreifen, so muss zwischen lokalen (Datenbank befindet sich auf demselben Computer wie das Programm) und verteilten (Datenbank und Programm auf verschiedenen Systemen) Zugriffen unterschieden werden.

3.6.1 Lokaler Datenzugriff

Um Zugriff auf Dateien zu bekommen, wird hier das lokale Dateisystem genutzt, nachdem es die Erlaubnis des Lockservers erhalten hat; bei einer standardmäßigen Installation ist dieser als Service dabei. Das Programm verbindet sich im Anschluss mit der Graphendatenbank durch die Bootdatei, diese verweist auf die Systemdatenbankdatei, in der sich die Standorte der vom Programm gesuchten Knoten und Kanten befinden. Während des Zugriffes auf die Daten wird ein Journal File angelegt. [2]

3.6.2 Verteilter Datenzugriff

Will ein Programm nun auf eine Graphendatenbank mit verteilten Speichereinheiten zugreifen, nutzt InfiniteGraph das Feature *Advanced Multithreaded Server* (AMS) für den Datenzugriff. AMS ist eine Alternative zu nativen Dateiservern wie NFS oder Microsoft Windows Network. Bei der Installation wird AMS standardmäßig als Service installiert, startet jedoch nicht automatisch; eine manuelle Aktivierung ist nötig. Wie auch beim lokalen Zugriff muss der Lockserver die Transkation genehmigen. Ebenso verbindet sich die Applikation mit der Graphendatenbank, um an die Standorte der gesuchten Daten zu gelangen. Der Zugriff auf diese erfolgt jedoch über AMS. [2]

3.7 Replikation

Um Datenreplikation in InfiniteGraph zu erreichen, steht Objectivity/DRO (Data Replication Option) zur Verfügung. Dieses Feature ist Teil der Grundlage von Objectivity/DB und InfiniteGraph, es greift dabei auf mehrere Datenbankimages zurück und gewährleistet so synchrone Replikation.
Synchrone Replikation bedeutet, dass bei einer Veränderung

eines Datenbankimages diese Änderung auch bei allen anderen Images vorgenommen wird. Ein Vorteil ist die ständige Konsistenz der Daten. Jedoch ist der Mechanismus anfällig für Ausfälle von Servern, die ein Image enthalten. Eine synchrone Replikation ist dann nicht mehr gewährleistet.

Objectivity/DRO verändert den Mechanismus. Es müssen nicht mehr alle Datenbankimages verfügbar sein, sondern nur eine Mindestanzahl, um einen konsistenten Datenbankzustand darstellen zu können. Um dies zu gewährleisten, werden die Datenbankimages in autonome Partitionen unterteilt. Diese sind unabhängig von Ausfällen anderer Partitionen und besitzen einen eigenen Lockserver. Jede dieser autonomen Partitionen besitzt eine komplette Datenbankarchitektur; Verbindungen zwischen Partitionen speichert das Feature als Referenzen in den Systemdatenbankdateien.
Führt ein Programm eine Transaktion durch, stellt es eine Verbindung mit einem der Lockserver her, dieser kontaktiert nun auch die Lockserver aller anderen Partitionen. Antwortet eine festgelegte Anzahl an Mindestservern, wird der Datenzugriff gewährt. Die Durchführung der Transaktion findet nun auf allen erreichbaren Servern synchron statt. Währenddessen ist es nicht möglich, weitere Transaktionen durchzuführen, die auf dieselben Container zugreifen, da ansonsten die synchrone Replikation nicht möglich ist.
Sobald ein ausgefallenes Datenbankimage wieder verfügbar ist, bringt es sich automatisch auf den aktuellen Stand. Dieser wird durch die Mehrzahl der Datenbankimages repräsentiert. Dieses Verfahren ist für READ und WRITE Transaktionen möglich. Es ist jedoch auch möglich, und in den meisten Fällen sinnvoll, das Verfahren für READ Zugriffe zu deaktivieren, da diese die Daten nicht verändern. [11]

4. NUTZUNG VON INFINITEGRAPH

Programme steuern InfiniteGraph über die Java-API an. Die folgenden Abschnitte erläutern, wie Programme und Entwickler Daten einfügen, verwalten und nutzen können.

4.1 Erstellung einer Datenbank

Die Erstellung und Verwaltung einer Datenbank sowie die Kommunikation zwischen Applikation und InfiniteGraph erfolgt über die Java-API. Um eine voll funktionsfähige Datenbank zu erstellen, sind, neben dem Einfügen von Bibliotheken, nur wenige Zeilen Code nötig.

```
GraphFactory.create("DbName");
GraphDatabase GraphDB = GraphFactory.
open("DbName");
```

Der *.create* Befehl erstellt alle benötigten Dateien. Die Verbindung zwischen Programm und Datenbank stellt die *.open* Anweisung her. [3]

4.2 Kommunikation mit der Datenbank

Es gibt vier Basisoperationen, um mit den Daten einer Datenbank zu arbeiten: Create, Read, Update und Delete. Diese Operationen werden mittels Transaktionen an die Datenbank übermittelt.

4.2.1 Transaktionen

Die Kommunikation mit einer InfiniteGraph Datenbank findet durch Transaktionen statt. Diese sorgen dafür, dass die ACID Eigenschaften (Atomicity, Consistency, Isolation, Durability) erreicht werden. Möchte eine Applikation eine Transaktion durchführen, so sperrt der Lockserver den Container mit den benötigten Daten, um keine unzulässigen Transaktionen zuzulassen. Erst nach dem Commit einer Transaktion hebt der Lockserver die Sperrung auf. InfiniteGraph unterscheidet zwischen zwei Zugriffsmöglichkeiten: READ, welches nur das Laden und Suchen erlaubt, und READ_WRITE, welches neben dem Laden auch noch die Manipulation von Daten ermöglicht. Das folgende Beispiel zeigt den Rahmen einer Transaktion, um Leseoperationen ausführen zu können. [3, 5, 11]

```
tx = GraphDB.beginTransaction(AccessMode.
READ);
tx.commit();
```

4.2.2 Create

Nach der Erstellung einer Datenbank und dem Aufbau einer Verbindung zu ihr, ist es möglich, Knoten und Kanten hinzuzufügen. Um dies zu erreichen müssen Klassen gebildet werden, die von der Basisklasse der Knoten ($BaseVertex$) oder Kanten ($BaseEdge$) erben. Hat man nun ein oder mehrere Objekte erstellt, so lassen sie sich mit einem add Befehl während einer Transaktion der Datenbank hinzufügen. Folgender Code zeigt das Hinzufügen eines Knoten. [3, 8, 11]

```
GraphDB.addVertex(objekt);
```

4.2.3 Read

Um Knoten wieder aus der Datenbank zu laden gibt es zwei Möglichkeiten: Die PQL (Predicate Query Language) oder Traversierung.

4.2.4 PQL

Die Predicate Query Language kommt ursprünglich von Objectivity/DB und ist deshalb auch für InfiniteGraph verfügbar. PQL unterstützt das Abfragen von Daten, jedoch nicht die Funktionen Create, Update und Delete. Die Syntax von PQL erinnert dabei sehr an SQL. Um beispielweise alle Personen mit dem Vornamen Hans und Nachnamen Meier mit Verwendung eines Index zu erhalten, reicht folgender Befehl. [7, 11]

```
IndexIterable<Person> indexItr =
personGraphIndex.query(new PredicateQuery(
"vorname='Hans' && nachname ='Meier'"));
```

4.2.5 Traversierung

Die zweite Möglichkeit zur Datensuche ist die Traversierung. InfiniteGraph unterstützt das Breiten- und Tiefensucheverfahren. Die Suche kann von einem beliebigen Knoten im Graphen gestartet werden. Zur Steigerung der Effizienz von Suchverfahren, können Entwickler die vorgehensweise manipulieren. Um dies zu erreichen, bietet InfiniteGraph *Path-* und *Result Qualifier* an. Der Path Qualifier entscheidet bei jedem Knoten, ob dieser Pfad noch weiter verfolgt wird oder nicht. Der Result Qualifier legt fest, welche Pfade für das Ergebnis relevant sind. Für beide Qualifier definiert der Entwickler Regeln, beispielsweise lassen sich einzelne Knoten oder sogar ganze Gruppen von der Suche ausschließen. Für

die Verwertung der Ergebnisse sind *Result Handler* nötig. Diese verwerten Pfade, die durch die Qualifier als qualifiziert gelten. Zum Beispiel kann der Entwickler sich nur die Pfade ausgeben lassen, die zwischen Start- und Zielknoten liegen. [4, 11]

4.2.6 Update

Um Datensätze zu ändern, müssen sie erst aus der Datenbank geladen werden. Da Knoten und Kanten als normale Objekte von Java Klassen definiert sind, reichen die Getter und Setter Methoden zum Verändern der Attribute. Wichtig hierbei ist jedoch, dass Setter Methoden die Funktion $markModified()$ aufrufen. Diese ist nötig damit die Datenbank Änderungen an den Knoten erkennt. Getter Methoden benötigen die Funktion $fetch()$, welche für ein Nachladen der Daten aus der Datenbank sorgt. [6, 11]

```
public String getName() {
        fetch();
        return this.name;}
```

4.2.7 Delete

Um Knoten oder Kanten zu löschen, muss entweder deren ID bekannt sein oder ein laden der Daten ist vorher nötig. Ist eine Vorraussetzung erfüllt, lassen sich die Objekte mit einem der folgenden zwei Befehle aus der Datenbank entfernen (Beispiel mit Knoten). [11]

```
GraphDB.removeVertex(objekt);
GraphDB.removeVertexById(id);
```

4.3 Visualisierung

Da InfiniteGraph eine Graphendatenbank ist, lassen sich die Daten und Suchergebnisse mit dem Programm IG Visualizer als Graphen ausgeben. Dieses Programm bietet einige Möglichkeiten, die Datenbank zu nutzen. Beispielsweise bietet der Visualizer mehrere Möglichkeiten an, Daten zu durchsuchen. Gibt man den Namen eines Knoten oder seine ID an, stellt der IG Visualizer automatisch die Verbindungen zwischen ihm und anderen Knoten dar. Weiterhin besteht die Möglichkeit einen Zielknoten anzugeben, so lässt sich die Beziehung zwischen zwei Objekten zeigen. Dies ist zum Beispiel nützlich, um herauszufinden, wie zwei Personen, die sich nicht kennen, über eine Kette von anderen Personen trotzdem in Verbindung stehen. Da die Knoten und Kanten im Prinzip Objekte mit Attributen sind, kann der Nutzer selber entscheiden, welche Daten angezeigt werden. [3, 4]

5. GESCHWINDIGKEIT

Ein wichtiger Bestandteil der neuen Datenbanken ist ihre Geschwindigkeit. Objectivity führte dazu einen umfangreichen Test durch, der das Einfügen von Knoten und Kanten, die Graphennavigierung (Traversierung durch Tiefen- und Breitensuche) und das Lesen der Attribute sowie Beziehungen von Knoten aus der Datenbank beinhaltet. Als Testdaten dienen automatisch indizierte Knoten, die eine Eigenschaft besitzen und Kanten, welche die Beziehungen zwischen den Knoten darstellen.
Beim Einfügen von Knoten ist InfiniteGraph in der Lage, mehr als 70.000 Objekte pro Sekunde der Datenbank hinzuzufügen, dies ist vor allem möglich durch das Management der Locks. Bei der Travisierung ist InfiniteGraph in der Lage, bis zu 60.000 Kanten pro Sekunde zu durchlaufen. Das Lesen der Attribute und Beziehungen von Knoten geht noch schneller, dort wurden bis zu 145.000 Objekte pro Sekunde überprüft. Durch diese sehr hohen Zugriffswerte eignet sich InfiniteGraph sehr für Systeme mit vielen Objekten. [10]

6. FAZIT

Ziel der vorliegenden Arbeit war es, Einblicke in InfiniteGraph zu gewähren und zu zeigen, in welchen Bereichen es sinnvoll ist, diese Graphendatenbank zu nutzen. Zu diesem Zweck wurden die technischen Einzelteile der Architektur erläutert sowie die wichtigsten Themen zum Konfigurieren und Nutzen erklärt. Dabei ergab sich, dass sich InfiniteGraph, dank der Darstellung von Daten als Knoten und Kanten, vor allem dazu eignet, Beziehungen zwischen zwei oder mehreren Objekten darzustellen. Was die Big Data Problematik betrifft, so konnten anhand des Benchmark und der leichten Verteilung von Daten auf mehrere Speicherorte gezeigt werden, dass sich die Graphendatenbank besonders für verteilte Systeme mit viel Datenverkehr eignet. Für kleinere Systeme, die nicht auf die Datenverteilung und Geschwindigkeit angewiesen sind, ist InfiniteGraph jedoch nicht geeignet, da der Aufwand zur Entwicklung, Einrichtung und Wartung groß ist.

7. REFERENCES

[1] Prof. Dr. rer. nat. Peter Tittman: "Graphentheorie", 2003

[2] System Übersicht, Dezember 2015 http://wiki.infinitegraph.com/3.4/w/index.php?title=InfiniteGraph_System_Overview

[3] Einstieg in InfiniteGraph, Dezember 2015 http://wiki.infinitegraph.com/3.4/w/index.php?title=Tutorial:_Hello_Graph!

[4] Graph Navigierung, Dezember 2015 http://wiki.infinitegraph.com/3.4/w/index.php?title=Tutorial:_Navigating_Graph_Data

[5] Transkationen, Dezember 2015 http://wiki.infinitegraph.com/3.4/w/index.php?title=Using_Transactions

[6] Persistente Elemente, Dezember 2015 http://wiki.infinitegraph.com/3.4/w/index.php?title=Defining_Persistent_Elements

[7] PQL Einführung, Dezember 2015 http://wiki.infinitegraph.com/3.4/w/index.php?title=PQL/Getting_Started_with_PQL

[8] Datengenerierung, Dezember 2015 http://wiki.infinitegraph.com/3.4/w/index.php?title=Tutorial:_Creating_Graph_Data

[9] EMC Studie zu Big Data, Dezember 2015 http://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm

[10] InfiniteGraph Benchmark, Dezember 2015 http://www.objectivity.com/wp-content/uploads/Objectivity_WP_IG_Distr_Benchmark.pdf

[11] Fachhochschule Köln zu InfiniteGraph, Dezember 2015 http://lwibs01.gm.fh−koeln.de/wikis/wiki_db/index.php?n=Datenbanken.InfiniteGraph

[12] Relationale Datenbanken vs NoSQL, Dezember 2015 http://www.bigdata-insider.de/relationale-datenbanken-sind-nicht-immer-ideal-a-472678/