UNIVERSITÄT LEIPZIG

Graphdatenbanken: DEX und HypergraphDB

Seminararbeit

Leipzig, März 2012

vorgelegt von:

Dirk Albrecht

geb. am: 03.10.1985

Betreuer:

Dr. Michael Hartung

Inhaltsverzeichnis

1 Einleitung..... 1

2 DEX (Data Exploration)..... 1

2.1 Einführung..... 1

2.2 Datenmodell..... 2

2.3 Architektur..... 3

2.4 CRUD-Operationen..... 5

2.5 Module..... 6

2.6 Bewertung..... 6

3 HypergraphDB..... 7

3.1 Einführung..... 7

3.2 Datenmodell..... 7

3.3 Architektur..... 8

3.4 CRUD-Operationen..... 9

3.5 Peer-to-Peer..... 10

3.6 Bewertung..... 10

4 Benchmark..... 11

5 Zusammenfassung..... 12

1 Einleitung

Es findet eine zunehmende Vernetzung von mobilen Computern, Smartphones und Serverclustern durch komplexe datenintensive Anwendungen z.B. soziale Netzwerke, Geoinformationssystemen statt. Diese stellen höhere Anforderungen an die Flexibilität, Speicherung und Abfragen. Mit bisherigen Systemen ist es häufig nicht möglich das gesamte Potenzial der vorhandenen Daten, aufgrund z.B. mangelnder Vernetzung der Daten untereinander und fehlender Bedeutung der Daten, auszuschöpfen ([1]). Mit Hilfe von Graphdatenbanken sollen diese Probleme gelöst werden und so vernetzte Daten natürlicher durch Modelle repräsentiert werden.

In dieser Seminararbeit sollen die beiden Graphdatenbanken Data Exploration (DEX) und HypergraphDB näher vorgestellt werden. Neben einem Einblick in die grundlegende Funktionsweise und Architektur der Systeme, wird die Verwendung der beiden APIs an einem kleinem Beispielgraphen dargestellt. Im Abschnitt 4 wird ein Benchmark aufgezeigt, der die Performanz der Datenbanken vergleicht.

2 DEX (Data Exploration)

2.1 Einführung

Die, aus dem akademischen Umfeld entsprungene, Graphdatenbank DEX (Data Exploration) wird durch die Data Management Group (DAMA-UPC) an der polytechnischen Universität von Catalonia seit 2006 entwickelt. Zur besseren Weiterentwicklung und Pflege wurde für das Projekt 2010 das Unternehmen Sparsity Technologies gegründet ([2]). Das Universitäts-Spin-off übernimmt ebenfalls die Verwaltung der Lizenzen. DEX wird als kommerzielle und als Community-Version angeboten. Aktuell ist DEX in der Version 4.3 verfügbar, wodurch die API neben Java auch für .NET angeboten wird.

Die Entwickler bezeichnen die Graphdatenbank als schnellste im NoSQL-Umfeld ([3]). Dies wird unter anderem durch die hybride Architektur von C++-Kern und Java-Bindings erreicht. DEX zeichnet sich außerdem durch eine leichte Integration verschiedenster Datenquellen mit Hilfe der Abbildung auf virtuelle Graphen.

Neben der Traversierung von Graphen, findet sich auch eine Vielzahl an Modulen zur Beantwortung struktureller Fragestellungen ([4]). Dazu zählen unter anderem die Analyse von Kantenbeziehungen und der Relevanz von Knoten und Kanten, Mustererkennung, sowie Graph-Mining.

Anwendung findet DEX z.B. in der Webapplikation Bibex von Sparsity Technologies ([5]). Bibex ermöglicht das Erkunden von bibliographischen und dokumentarischen Archiven. In der zugänglichen Demo sind verschiedene wissenschaftliche Arbeiten mit ihren Autoren und Referenzen als Graph hinterlegt. Die Struktur des Graphen unterstützt schnelle Abfragen, um z.B. passende Vorschläge zur aktuell betrachteten Publikation zu finden. Die Datenbank von Bibex beinhaltet 2.752.948 Artikel, 1.003.211 Autoren und 237.348 Schlüsselworte. In Abbildung 1 wird beispielhaft die Vernetzung von Autoren untereinander anhand von wissenschaftlichen Arbeiten von Tim Berners-Lee als Graph dargestellt. Ebenso könnten Referenzen von Arbeiten untereinander auf gleiche Weise dargestellt werden.

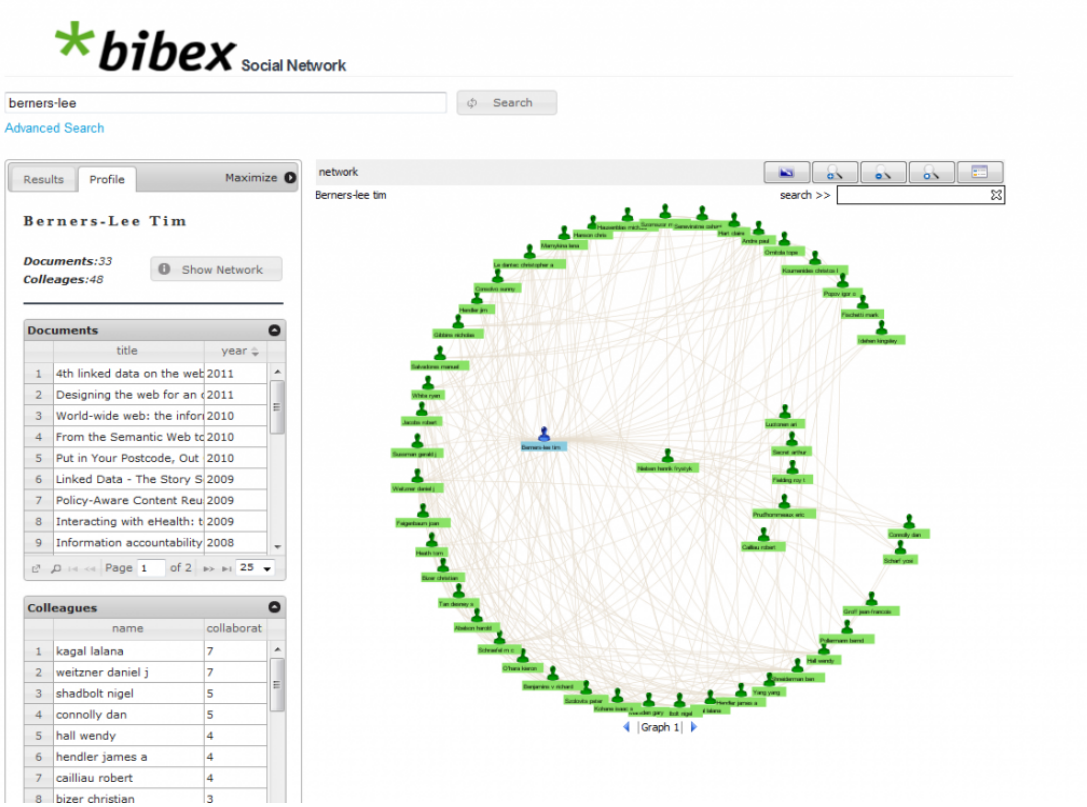

Abbildung 1: Darstellung der Co-Autoren von Tim Berners-Lee Arbeiten. (Quelle: [5])

Weitere Einsatzszenarien sehen die Entwickler im Bereich von Informationsnetzwerken (IMDB, Wikipedia), sozialen Netzwerken (Twitter, Facebook) und z.B. biologischen Netzwerken (Proteininteraktion). In einem Test konnte man 476M Twitter-Nachrichten und 1,2B Follower-Beziehungen in einem Graph abbilden ([2]).

2.2 Datenmodell

Bei DEX werden zwei grundlegende Graphstrukturen unterschieden, wie in Abbildung 2 dargestellt. Die DBGraphen sind die persistenten Graphen. Bei Anfragen werden die Zwischen- und Endergebnisse in den temporären RGraphen erzeugt. Die DBGraphen lassen sich weiter unterteilen in einen Schemasubgraphen und in einen Datensubgraphen. Letzterer ist für die Speicherung der eigentlich Daten zuständig. Im Schemasubgraphen werden die für die Interpretation benötigten Metadaten hinterlegt. Zu den Metadaten zählen die Datenquellen (CSV, XML, …), die Datensätze aus den Quellen (Zeilen, Spalten, XML-Elemente), sowie die Typen und Eigenschaften der Knoten und Kanten.

DEX bezeichnet sein Graphdatenmodell als typisierten und gerichteten Multigraph mit Attributen G = {T, N, E}. T steht für die Menge der Bezeichner, N die Menge der Knoten und E die Menge der gerichteten Kanten (siehe [4]). Im Vergleich zum Property-Graphmodell können ebenso ungerichtete Kanten oder Kanten zwischen verschiedenen Knotentypen dargestellt werden. Des Weiteren können Attribute mit gleichem Wert durch virtuelle Kanten miteinander verbunden werden. Kanten und Knoten kann eine variable Anzahl von Attributen hinzugefügt werden. Attribute entsprechen einem Wert, dem ein Name zugeordnet wird.

Bezeichner aus T bestehen aus einem eindeutigen Namenstyp und einer Menge von definierten Attributen. Jedes Attribut besitzt einen eindeutig Attributnamen und einen Wertetyp z.B. String, Integer. Jedem Knoten aus N ist sein Typ zugeordnet und die Tiefe in der Knotenhierarchie. Es kann einen eindeutigen Schlüssel geben, der die Datenquelle angibt. Außerdem besitzt er eine Menge V = {(a,v)} von Attributwerten entsprechend seinem Typs. D.h. Attribut a ist aus der Menge der Attribute

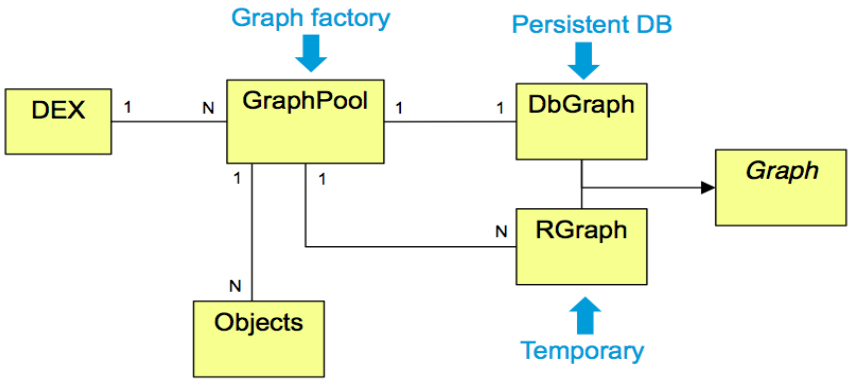

Abbildung 2: Klassendiagramm der DEX API. (Quelle: [6])

des Knotentyps und die Werte v aus dem Wertebereich des Attributes. Analog besitzt jede Kante aus E ebenfalls eine Menge V, einen Typ und eine Hierarchietiefe. Darüber hinaus sind noch Start- und Endknoten hinterlegt.

2.3 Architektur

Wie in Abbildung 3 dargestellt, verfügt DEX über mehrere Ebenen. Die unterste Ebene wird als Kern bezeichnet und umfasst neben dem Management auch das Durchsuchen der Graphstruktur. Die nächste Schicht beinhaltet die Java-API, die ein einfaches Programmierinterface zur Verfügung stellt. Darauf lässt sich dann eine Anwendungsebene aufbauen, die die Kernfunktionen erweitert und auch eine Visualisierung der Suchergebnisse ermöglicht. Nachfolgend wird auf die einzelnen Ebenen, wie in [4] beschrieben, näher eingegangen.

Zum, in C++ verfassten, Kern gehören die wichtigen Module *GraphPool*, *DbGraphManager* und *QueryEngine*. Der *GraphPool* verwaltet die Datenstruktur der Graphen, die Pufferung und die I/O Operationen. Des Weiteren ist er für die persistente Speicherung der *DbGraphen* zuständig und die Verwaltung der *RGraphen* im Cache. Der *DbGraphManager* verknüpft externe Datenmodule mit dem *DbGraph*, sofern die dafür notwendigen Dateiformate durch entsprechende Plugins unterstützt werden. Die *QueryEngine* ermöglicht ein schnelles Suchen und Ablaufen eines Graphes.
Mit Hilfe der Java-API wird ein Interface bereitgestellt, wodurch eine DEX-Instanz leicht gestartet, verwaltet und beendet werden kann. Auf wichtige Funktionen des C++-Kernes wird durch public Methoden Zugriff ermöglicht, um diese in der Applikationsebene zu nutzen.

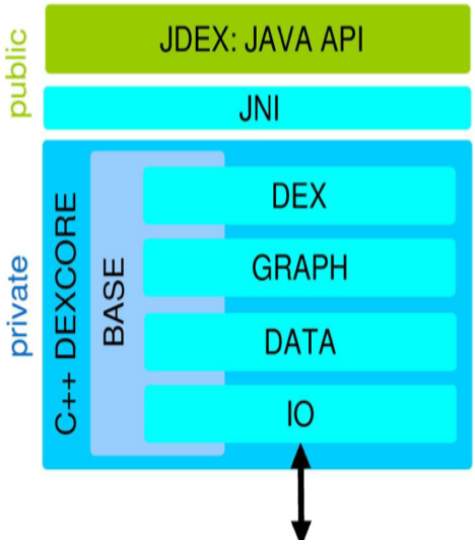

Abbildung 3: Der Ebenenaufbau von DEX. (Quelle: [7])

Um die Benutzerfreundlichkeit zu erhöhen, bietet sich auf der Anwendungsebene an, Werkzeuge für die Verwaltung der *DbGraphen*, die Optimierung von Suchanfragen, die leichte Erstellung von Suchanfragen und die Visualisierung von *RGraphen* zur Darstellung der Anfrageergebnisse und dem weiteren Erkunden in den Ergebnissen, bereit zu stellen.

In [4] wird ebenfalls ein kleiner Einblick in die interne DEX-Struktur gegeben, die in Abbildung 4 dargestellt ist. Objektidentifizierer *oids* sind eindeutige Integer-Werte, die zur Erstellung und Kompression der Strukturen in DEX verwendet werden. Weiter wurden die Mengen T={int, long, real, string, boolean, timestamp, O} und $S(T)=\{s_1, s_2, \ldots, s_n\}$ definiert, wobei T die Menge aller unterstützter Typen und S eine ungeordnete Menge von Werten aus dem Wertebereich von Typen aus T ist. Jeder Wert in S kommt höchstens einmal vor. Die Menge O der Objekte besteht aus allen Kanten und Knoten. Objektmengen werden mit Hilfe von Bitmaps gespeichert und ausgelesen. Diese Bitmaps sind komprimiert und bestehen aus Bit-Vektoren $B=\langle O \rangle$. Des Weiteren existieren Maps $M=\langle T_k, T_v \rangle=(key, value)$, die aus Mengen von Schlüsseln aus T_k und Werten aus T_v bestehen und durch Bäume gespeichert werden. Es besteht eine N:1 Beziehung zwischen Schlüssel und Werten. Die bijektive Map $R\langle T \rangle=M\langle T, O \rangle$ verknüpft einen Wert als *oid* mit einem Schlüssel. Durch die bijektive Map $v=R\langle T \rangle$ und die Map $o=M\langle V, B \rangle$ lassen sich Dictionaries $D\langle T \rangle=\{v, o\}$ anlegen. Durch diese Struktur ist es möglich beliebigen Wertetypen eine eindeutige ID zu geben um so den Speicherbedarf zu verringern und Operationen zu beschleunigen. Mit dieser Struktur lassen sich so Schlüsselwerte mit den entsprechend verknüpften *oids* in Verbindung setzen. Außerdem wurde eine Verknüpfung $L\langle T \rangle=\{v, d\}$ deklariert, wobei $v=R\langle O \rangle$ die Identifizierer der Werte der *oids* und $d=D\langle T \rangle$ die *oids* für jeden ID eines Wertes enthält.

Aus obigen Definitionen lässt sich ein Graph in DEX $G=\{n, e, o, t, h\}$ zusammensetzen. Dieser besteht aus $n=M\langle string, T \rangle$ allen Attributen eines Knotentyps bzw. $e=M\langle string, T \rangle$ allen Attributen eines Kantentyps. In $o=L\langle tid, nid \rangle (tid \in E \cup N, nid \in N)$ findet sich eine Liste aller Objekte eines Typs. Außerdem gibt es die beiden Kantenlisten $t=L\langle nid, eid \rangle (nid \in N, eid \in E)$ und $h=L\langle nid, eid \rangle (nid \in N, eid \in E)$ einmal für Endknoten und einmal für Startknoten einer Kante.

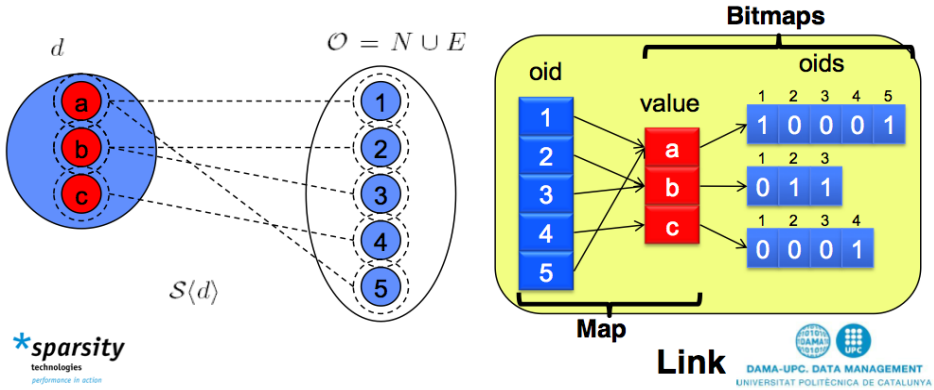

Abbildung 4: Interne Speicherdarstellung von DEX. (Quelle: [7])

2.4 CRUD-Operationen

In diesem Abschnitt werden einige grundlegende Funktionen und Befehle der API anhand eines Beispieles vorgestellt. Dazu wird auszugsweise auf den im Anhang A befindlichen Quelltext eingegangen. Abbildung 5 zeigt den resultierenden Graphen, der mit Hilfe der API erzeugt wird.

Zunächst wird eine Instanz von *DEX* durch den Aufruf *new DEX()* erstellt. Dadurch kann nun durch den Befehl *DEX.create()* ein *GraphPool* erzeugt werden. Dieser übernimmt die persistente Speicherung des *DbGraphen*, sowie die Verwaltung der temporären *RGraphen*.

Die Instanz des *DbGraphen* erhalten wir durch den Methodenaufruf von *getDbGraph()* des GraphPools. In diesem *DbGraphen* lassen sich nun die verschiedenen Knoten- und Attributtypen definieren, die später benötigt werden. Dies erfolgt über die Methoden *newNodeType(name)* für neue Knotentypen und über *newAttributeType(type, name)* für neue Attribute. Der Bezeichner wird durch *name* übergeben und bei *type* legt man den Typ z.B. *String* des Attributes fest.Die Methoden geben ein Integer für Knotentypen und ein Long-Integer für Attributtypen zurück, wodurch die Typen innerhalb der Datenbank identifiziert werden.

Nach der Definition der Typen können nun die Knoten in dem Graphen über *newNode(type)* mit Hilfe des *DbGraph* angelegt werden, wobei bei *type* der Integer-Wert des gewünschten Knotentyps übergeben wird.
Als Rückgabewert ist wie bei den Attributen ein Long-Integer zu erwarten. Mit Hilfe der Methode *setAttribute(oid, attr, value)* wird das gewünscht Attribute *attr* mit Inhalt *value* für den entsprechen Knoten *oid* gesetzt. Dabei wird jeweils der entsprechende Integer- bzw. Long-Integer-Wert als Referenz genutzt.
So erhalten wir die drei Personen John, Kelly und Mary als Knoten im Graphen aus Abbildung 5.

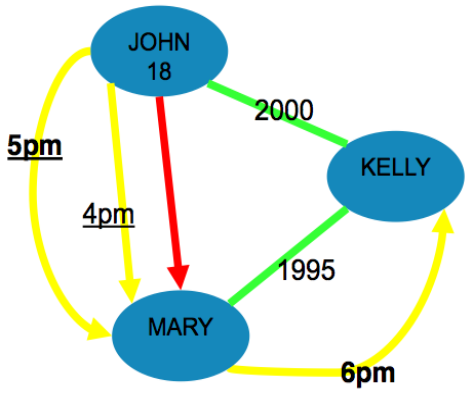

Abbildung 5: Beispiel für einen mit DEX erzeugten Graphen. (Quelle: [6])

Kanten werden wie Knoten durch einen Integer-Wert referenziert, der beim Erstellen des Kantentyps zurückgegeben wird. Kanten können entweder gerichtet durch *newEdgeType(name)* oder ungerichtet durch *newUndirectedEdgeType(name)* definiert werden. Auch Kanten können mittels *newAttribute(type, name)* Attribute hinzugefügt werden. Über *newEdge(oid1, oid2, type)* lassen sich zwei Knoten mit Hilfe ihrer ObjektID *oid* von einem Kantentyp *type* verbinden. Als Rückgabewert wird die ID der hinzugefügten Kanten zurückgegeben. Mit dieser ID kann über *setAttribute(oid, attr, value)* das Attribut *attr* für eine entsprechende Kante mit der ObjektID *oid* der Wert *value* gesetzt werden. Werden zu den Knoten John, Kelly und Mary nun die Kanten „befreundet seit“ (grün) und „liebt“ (rot), sowie „ruft an“ (gelb) hinzugefügt, ergibt sich der Graph wie in Abbildung 5.

Die DEX-API enthält auch Methoden um z.B. bestimmte Teile des Graphen zu selektieren. Durch den Aufruf von *select(type)* wird eine Menge von Objekten *Objects* mit der gleichen TypID *type* ausgewählt und diese mit Hilfe eines Iterators nacheinander durchlaufen. Ebenso lässt sich die Selektion durch weitere Einschränkungen präzisieren. *Objects* sind die sogenannten RGraphen, die nur temporär für die Bearbeitung von Anfragen erstellt werden.

2.5 Module

Wie in [6] aufgeführt, umfasst DEX weitere Java Packages, die eine Erweiterung um höhere Funktionen darstellen. Dazu zählen Module für Schema, Ein- und Ausgabe, Erkundung, Visualisierung und Graphoperationen.

Durch die Schema-Module lassen sich Graph-Schemata mittels Scripten definieren, sowie die Erzeugung und Wartung von Graphen vereinfachen. Bei den Eingaben ermöglichen es die entsprechenden Module unterstützte Datenformate wie z.B. CSV, XML, RDF Graphen direkt in einen DEX-Graphen umzuwandeln. Bei der Ausgabe lassen sich die DEX-Graphen in den Formaten GRAPHML oder Graphviz ablegen. Mit Hilfe der Visualisierungsmodule, wie Prefuse, lässt sich ein Graph direkt durch Java Swing Komponenten repräsentieren.

Andererseits erledigen die Module zur Erkundung des Graphen unter anderem nachfolgende Aufgaben. Es werden komplexe Anfragen ermöglicht, deren resultierende Antworten als Graph repräsentiert werden. Die Bearbeitung erfolgt nach einer festgelegten Reihenfolge. Passende Objekte aus dem *DbGraphen* werden selektiert und überflüssige Beziehungen entfernt. Aus den gefundenen Graphen werden jene ausgesucht, die bestimmten Kriterien entsprechen. Die Ergebnisgraphen werden dann graphisch dargestellt.

2.6 Bewertung

DEX ist die schnellste Graphdatenbank nach Messungen durch die Entwickler ([3]). Aufgrund der Architektur, die Daten und Schema in unterschiedlichen Teilgraphen speichert, lassen sich vernetzte Probleme gut lösen, besonders wenn die Integration verschiedener Quellen notwendig wird. Auf dem Graphen lassen sich nicht nur Graphtraversierungen, wie Breiten- oder Tiefensuche, sondern ebenso strukturelle Anfragen, wie z.B. die Relevanz von Knoten und Kanten durchführen. Negativ anzumerken ist, dass DEX noch keine Graphpartitionierung unterstützt, wodurch die Grenzen eines zu verarbeitenden Graphen durch den Server auferlegt werden. In den Abbildungen 6 und 7 wird ein Test dargestellt, bei dem die IMDB Datenbank als DEX Graph repräsentiert wird. Auf diesen werden verschiedene Anfragen gestellt und dies soll zeigen, dass DEX in bereits vorhandenen Netzwerken sinnvoll eingesetzt werden könnte.

	IMDB
DB	2.4 GB
Physical Mem	9 GB
Load time	21 min
Nodes	13 M
Edges	22 M
Values	48 M
Insertions per sec	65 K

Abbildung 6: IMDB Datenbank in DEX. (Quelle: [7])

	In-memory	128 MB
A	0.13 sec	0.13 sec
B	1.52 sec	1.79 sec
C	384 sec	385 sec

Abbildung 7: A: Auslesen eines kompletten Filmdatensatzes. B: Distanz zweier Darsteller C: Alle Filme mit einem bestimmten Muster (Quelle: [7])

3 HypergraphDB

3.1 Einführung

HypergraphDB wird seit 2010 von dem Unternehmen Kobrix Software entwickelt ([8]). Die Entwicklung nahm ihren Ursprung im Bereich der künstlichen Intelligenz, der Verarbeitung natürlicher Sprache und des Semantic Web.

Im Gegensatz zu anderen Graphdatenbanken verwendet HypergraphDB als Datenmodell die Hypergraphen, die beispielhaft in Abbildung 8 und 9 zu sehen sind. Diese ermöglichen das Verbinden von mehr als zwei Knoten durch eine Kante. Des Weiteren ist es möglich auch zwischen Kanten eine direkte Beziehung herzustellen. Als Einsatzszenarien lassen sich im Bereich der künstlichen Intelligenz finden (OpenCog AI), im Semantic Web und bei Mustererkennung in Netzwerken.

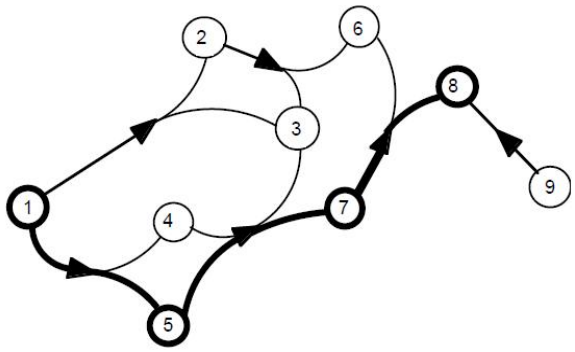

Abbildung 8: Gerichteter Hypergraph. (Quelle:[9])

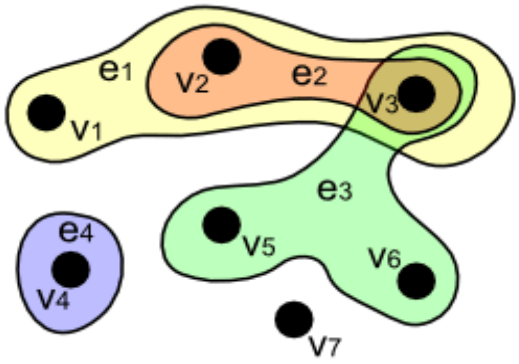

Abbildung 9: Ungerichteter Hypergraph. (Quelle: [9])

3.2 Datenmodell

Aufgrund des Ursprungs lassen sich ungewöhnliche Begriffe vorfinden. So bezeichnet das *Atom* die grundlegende Datenstruktur der Graphdatenbank. Das *Atom* stellt ein typisiertes Objekt dar und kann anwendungsspezifische Daten enthalten. Es bildet die Basis für alle anderen Objekte und kann Verbindungen mit einem oder mehreren anderen Atomen eingehen.

Kanten werden als Spezialisierung der Atome, die eine Kantenschnittstelle implementieren, modelliert. Dies ermöglicht auch direkte Beziehungen zwischen unterschiedlichen Kanten ([1]).

Wie in [10] von Iordanov beschrieben, bildet das *Atom* die Basiseinheit und besitzt eine sogenannte Zielmenge, die aus mit ihm verbundenen Tupel von *Atomen* besteht. Die Arität eines *Atoms* gibt die Anzahl der *Atome* in seiner Zielmenge an. Knoten haben eine Arität von Null. Dagegen werden *Atome* mit einer Arität > 0 als *Links* bezeichnet. Des Weiteren gibt es die Inzidenzmenge eines *Atoms x*, in der alle *Atome* enthalten sind, deren Zielmenge das *Atom x* enthält. Jedem *Atom* ist ein Wert zugeordnet, der einen beliebigen Typ, der wiederum ein *Atom* ist, haben kann.

3.3 Architektur

HypergraphDB verwendet zur Datenverwaltung zwei Ebenen, den Primitive Storage Layer und den Model Layer, wie in Abbildung 10 zu sehen ist. Als effizientes key-value Speichersystem setzt HypergraphDB die BerkeleyDB ein.

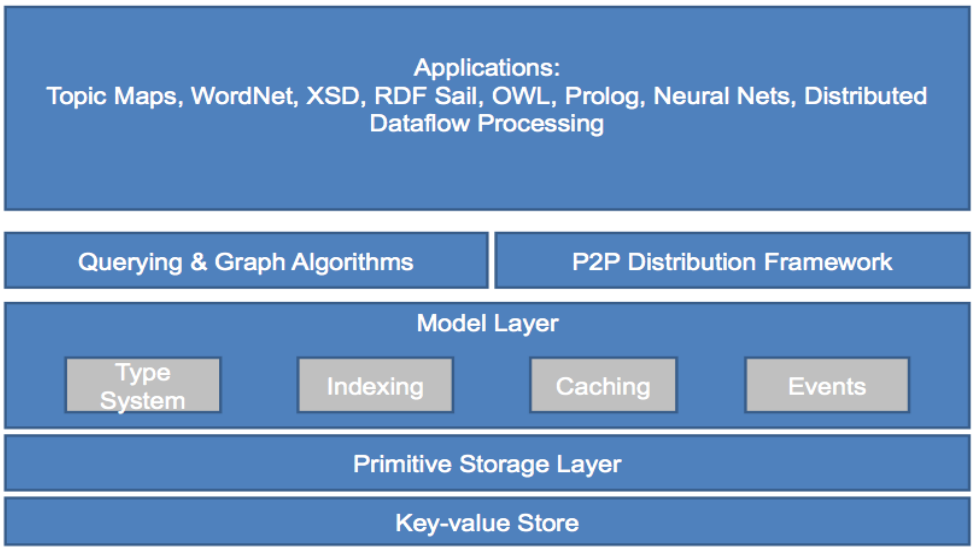

Abbildung 10: Architektur der HyperGraphDB. (Quelle: [11])

Im Primitive Storage Layer befinden sich folgende zwei Arrays: $LinkStore : ID \rightarrow List\langle ID \rangle$ und

$DataStore : ID \rightarrow List\langle byte \rangle$. Damit verweist jede ID entweder auf einen Datensatz oder eine Menge von IDs. Eine ID wird mit Hilfe der zufällig generierten type 4 UUID (Universally Unique Identifier) erzeugt. In verteilten Anwendungen kann so ausgenutzt werden, dass es sehr unwahrscheinlich ist, zwei gleiche IDs zu generieren. Es ist nicht nötig die IDs zentral zu verwalten.

Im Model Layer befindet sich das Typsystem, Caching, Indexierung und Anfragenverarbeitung. Weiter wurden folgende Definitionen festgelegt:

$$AtomID \rightarrow [TypeID, ValueID, TargetID, \ldots, TargetID]$$

$$TypeID \rightarrow AtomID$$

$$TargetID \rightarrow AtomID$$

$$ValueID \rightarrow List\langle ID \rangle | List\langle byte \rangle$$

Durch diese Festlegungen repräsentiert eine ID entweder ein *Atom* oder den Wert eines *Atoms*. Der rekursive Aufbau der *ValueID* ermöglicht es beliebig komplexe Strukturen in dem Graphen abzubilden. Weiter werden noch Indexe definiert. Der IncidenceIndex bildet ein Atom auf die Menge aller zu ihm zeigenden Links ab. Der TypeIndex bildet eine Atomtyp auf die Menge seiner Instanzierungen ab. Der ValueIndex verknüpft die IDs eines beliebig komplexen Wertes auf die Menge der Atome ab, die diesen Wert besitzen (Vgl. [10]).

Ein wichtiger Aspekt der HyperGraphDB ist die Typisierung. Durch *Atome* ist es möglich beliebige Datentypen zu speichern. Mit Hilfe des Interfaces *HGAtomType* können *Atome* spezialisiert werden und es wird dem System ermöglicht weiter Typen und Instanzen dieser zu verwalten. Ebenfalls können so Super-Subtypen Beziehungen konstruiert werden.

Mit Hilfe des Indexierungsmodul ist es möglich weitere Indexe anzulegen, indem das *HGIndexer* Interface implementiert wird. Dadurch wird der Index beim System registriert und gewährleistet, dass

zu jedem Atom auch ein Schlüssel erzeugt wird. Die Entwickler haben bereits einige Indexe vordefiniert (vgl. [10]).

Der *ByPartIndexer* indiziert Atome mit zusammengesetzten Typen. *Atome* können durch ein spezifisches Zielatom im Tupel durch den *ByTargetIndexer* verarbeitet werden. Beim *CompositeIndexer* wird eine Vorberechnung eines Joins durch die Verbindung von Indexen ermöglicht.

Die API ermöglicht das Erstellen von Anfragen aus Primitiven. Dazu gehört z.B. der Vergleich von *Atomen* durch *eq(x), eq(„name“,x)*, Feststellen des Atomtyps mit *type(TypeID)* oder Prüfen, ob ein *Atom* innerhalb der Zielmenge eines *Atoms* liegt *target(LinkID)*. Des Weiteren bietet die API Traversierungs-Algorithmen um nach Mustern oder benachbarten *Atomen* zu suchen. Die einzelnen Primitive lassen sich durch die gängigen and, or und not Operatoren miteinander verknüpfen, um komplexere Anfragen stellen zu können.

3.4 CRUD-Operationen

In diesem Abschnitt werden einige grundlegende Funktionen und Befehle der API anhand eines Beispieles vorgestellt. Dazu wird auszugsweise auf den im Anhang B befindlichen Quelltext eingegangen. Abbildung 11 zeigt den resultierenden Graphen, der mit Hilfe der API erzeugt wird.

Zunächst wird eine bestehende HyperGraphDB Datenbank durch den Aufruf *HGEnvironment.get(„c:/image_hgdb“)* geöffnet. Diese musste zuvor mit *new HyperGraph(databaseLocation)* angelegt werden und der Zugriff auf eine Instanz *Hypergraph graph* und seine Methoden ist mögich.
HyperGraphDB behandelt Kanten und Knoten auf gleiche Weise, wodurch nur ein Befehl zum Hinzufügen von Objekten vorhanden ist. Durch Reflection und Serialisierung werden die Java-Objekte verarbeitet. Durch *graph.add()* können nicht nur Basistypen wie *String* oder *Integer* hinzugefügt werden, sondern auch eigene definierte Klassen wie z.B. eine Klasse *Person*. Die Attribute eines Knoten werden somit direkt in der entsprechenden Knotenklasse gespeichert. Als Rückgabewert erhält man ein *HGHandle*, welches die Identität des Datenbankobjektes darstellt. Darüber ist es möglich Änderungen am Objekte vorzunehmen oder es ggf.
auch zu löschen.

Kanten werden ebenfalls mit Hilfe der *graph.add()* Methode zum Graphen hinzugefügt. Um es als Kante zu kennzeichnen und zu verwenden muss das *Atom* bzw. die Java-Klasse das *HGLink* Interface implementieren. Es stehen drei Standard-Implementierungen zur Verfügung. *HGPlainLink* stellt eine einfache Kante dar. *HGValueLink* ermöglicht das Anfügen eines zusätzlichen Java-Objektes, dass als Kantengewicht verwendet werden kann. *HGRel* stellt zusätzliche Bedingunger an zwei verbundene Knoten. Fügt man wie im Anhang B gezeigt, die Kanten und Knoten hinzu, ergibt sich der Graph aus Abbildung 11. Die Kanten können ebenfalls über den Rückgabewert *HGHandle* wieder angesprochen und verändert werden.

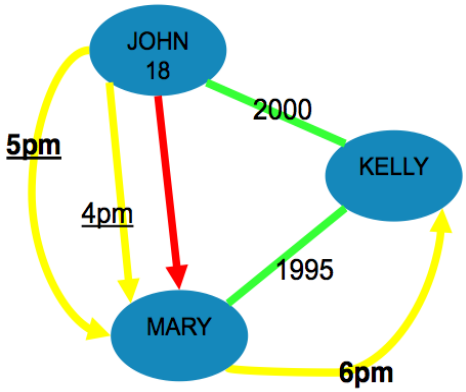

Abbildung 11: Beispiel für einen mit HyperGraphDB erzeugten Graphen. (Quelle: [6])

Mit Hilfe der in 3.3 angesprochenen Anfrage-Syntax lassen sich *HGQueryCondition* definieren, die eine Anfrage an die Datenbank darstellen. Diese Anfrage wird dem Suchalgorithmus übergeben und *graph.find(HGQueryCondition)* liefert als Ergebnis eine geordnete Menge an *HGHandles* ohne Wiederholungen zurück. Auf dieses *HGSearchResult* lässt mit einem Iterator zugreifen und die

einzelnen Ergebnisse durchlaufen. Um auf die entsprechenden Klassenmethoden zuzugreifen, muss ein Cast zu der gewünschten Klasse durch *(Klasse) graph.get(_currentHandle)* erfolgen.

3.5 Peer-to-Peer

Die HyperGraphDB unterstützt die Datenverteilung durch ein Peer-to-Peer Framework. Dieses ist im Model Layer angesiedelt und verwendet eine agenten-basierende Sprache ACL (Agent Communication Language) FIPA. Die Kommunikation wird durch einige Primitive wie *propose*, *accept*, *inform*, *request*, *query* gesteuert (Vgl. [10]). Die Nachrichtenübertragung erfolgt asynchron und die Nachrichten werden durch einen Scheduler und Threads abgearbeitet.

Die Idee an einer verteilten Wissensbasis für Iordanov et al. ist, dass es nicht möglich ist, alles Wissen überall konsistent zu halten. Dennoch bieten sie einen Algorithmus an, der Konsistenz auf Userebene herstellen könnte. Dazu wird zuerst von den Agenten ein Broadcast gesendet, in dem sie mitteilen an welchen *Atomen* sie interessiert wären. Sofern eine Änderung an *Atomen* erfolgt, die von anderen Agenten als interessant deklariert worden, sendet der entsprechende Agent ein *inform* an diese. Mit Hilfe einer Versionierung wird sichergestellt, dass alle interessierten Agenten die Änderung erhalten, damit ein konsistenter Zustand erreicht wird. Bei Erhalt der Änderungsnachricht ist es dem entsprechenden Agenten freigestellt die Änderung zu übernehmen oder nicht, aber in jedem Fall ist eine Benachrichtigung über den Erhalt zu übersenden, um den zusenden Agenten zu informieren.

3.6 Bewertung

Wie von Edlich et al. festgestellt ist die HyperGraphBD auch aufgrund ihres Ursprungs besonders für komplexe Anfragen aus dem Bereich der künstlichen Intelligenz oder des Semantic Web geeignet ([1]). Die leichte Einbindung von Java-Objekten durch die direkte Speicherung in Atomen und die Verwendung von Hyperkanten vereinfacht viele Anwendungsfälle. Durch die teilweise in 3.3 genannten Methodenaufrufe sind komplexe Anfragen und ihre effiziente Verarbeitung möglich. Die P2P-Architektur ermöglicht auf einfache Weise eine Skalierung der Datenbank.

Negativ auf die Performanz wirkt sich die Verwendung der BerkeleyDB zur Persistenzsicherung aus, da es hier bessere Alternativen gibt (Vgl. [1]). Durch fehlende Graphpartitionierungsalgorithmen kann aus der verteilten Datenbank noch kein Nutzen gezogen werden.

4 Benchmark

In diesem Abschnitt werden die Ergebnisse aus einem Benchmark von Dominquez-Sal et al. zusammenfassend dargestellt ([3]). Im Vordergrund stehen die Resultate der beiden in der Ausarbeitung betrachteten Graphdatenbanken HyperGraphDB und DEX.

Zur Erzeugung der Datenbank wurde der R-MAT Algorithmus verwendet ([12]). Für die Erzeugung unterschiedlicher Graphen wird nur der *scale* Parameter betrachtet. Damit werden Graphen erzeugt mit $N = 2^{scale}$ Knoten und $M = 8 \cdot N$ Kanten und einem positiven Kantengewicht von maximal 2^{scale} .

Zum Testen der System wurden vier Kerne konzipiert, wobei der erste zum Laden des Graphen dient und die restlichen Drei verschiedene Anfragen stellen an die Datenbanken. Wie bereits erwähnt dient Kernel 1 zum Laden der Knoten und Kanten des unter R-MAT erzeugten Graphen. In Kernel 2 wird nach allen Kanten gesucht, die das maximale Kantengewicht haben und gibt eine Liste der Kanten und damit verbundenen Knoten aus. Kernel 3 benutzt die aus Kernel 2 erstellte Liste von Knoten als Eingabe, um ausgehend von den Anfangsknoten Subgraphen aufzubauen. Dazu werden weitere Knoten zur Liste hinzugefügt die k Sprünge von den Ausgangsknoten entfernt sind. Als letztes wird in Kernel 4 für jeden Knoten berechnet, wie weit er vom Zentrum des Graphen entfernt ist. Dazu wird die kürzeste Entfernung von allen Knotenpaaren berechnet die den Knoten durchlaufen. Der Algorithmus zur Berechnung erreicht eine Komplexität von $O(k \cdot M)$ mit k als Anzahl der Stichproben und M der Anzahl der Kanten im Graph.

Dominquez-Sal et al. verwendeten als Testsystem zwei Quad Core Intel Xeon E5410 2,33 Ghz, 11 GB RAM und eine LFF 2,25 Tb Festplatte.

Durch die verwendeten Skalierungsfaktoren von 10, 15 und 20 wurden 1k, 32k bzw. 1M Knoten erzeugt. Diese bewirkten in den Graphen von ca. 10k bis 9,4M Objekte.

Table 1. Scale factor 10

Kernel	DEX	Neo4j	Jena	HypergraphDB
K1 Load (s)	**0.316**	19.30	7.895	376.9
K2 Scan edges (s)	**0.001**	0.131	0.090	0.052
K3 2-hops (s)	**0.003**	0.006	0.245	0.015
K4 BC (s)	0.512	**0.393**	5.064	1.242
Db size (MB)	2.1	**0.6**	6.6	26.0

Table 2. Scale factor 15

Kernel	DEX	Neo4j	Jena	HypergraphDB
K1 Load (s)	**7.44**	697	141	+24h
K2 Scan edges (s)	**0.001**	2.71	0.689	N/A
K3 2-hops (s)	**0.012**	0.026	0.443	N/A
K4 BC (s)	14.8	**8.24**	138	N/A
Db size (MB)	30	**17**	207	N/A

Table 3. Scale factor 20

Kernel	DEX	Neo4j	Jena	HypergraphDB
K1 Load (s)	**317**	32094	4560	+24h
K2 Scan edges (s)	**0.005**	751	18.60	N/A
K3 2-hops (s)	0.033	**0.023**	0.458	N/A
K4 BC (s)	**617**	7027	59512	N/A
Db size (MB)	893	**539**	6656	N/A

Abbildung 12: Ergebnisse eines Benchmark. (Quelle: [3])

Aus den Tabellen in Abbildung 12 ist zu erkennen, dass DEX im Gegensatz zu den anderen System mit steigender Größe der Datenbank gut skaliert. Ladezeiten halten sich gering. Besonders beim Vergleich von Tabelle 2 und 3 zeigt sich die gute Verarbeitung von großen Datenbanken bei DEX, da hier die Aufwärmphase weniger stark ins Gewicht fällt. Bei kleineren Graphen ist Neo4j im Vorteil. Leider konnten Dominquez-Sal et al. keine Aussagen über HyperGraphDB treffen, da die Datenbank eine zu große Ladezeit beanspruchte.

5 Zusammenfassung

Die beiden Graphdatenbanken HyperGraphDB und DEX haben gezeigt, wie unterschiedlich die Implementierungen und Herangehensweise sein können. Sie verwenden verschiedene Ansätze zur Verarbeitung von Graphen. DEX setzt auf die Verwendung von typisierter und gerichteten Multigraphen mit Attributen, aber verfügt durch die Identifizierung von Typen durch IDs über keine implizite Fehlererkennung durch den JAVA-Compiler. Im Gegensatz ist dies bei HyperGraphDB durch seine Kapselung der Knotenklassen in seine Grundklasse, das *Atom*, möglich. Die beiden API unterscheiden sich dahingehend, dass DEX noch einen C++-Kern besitzt und darauf die JAVA-Bindings aufsetzen und HyperGraphDB über eine reine JAVA-API verfügt. Des Weiteren verfügt HyperGraphDB durch die Verwendung Peer-to-Peer über eine Verteilung der Daten. Bei DEX ist keine Verteilung der Daten möglich. Beim Vergleich der beiden Datenbanken im Benchmark im Kapitel 4 zeigt sich, dass wesentlich besser mit großen Mengen an Daten umgehen kann und schneller Ergebnisse liefert. In Tabelle 1 wurden ein paar wesentlichen Punkten als abschließender Vergleich zwischen den Graphdatenbanken herausgegriffen und gegenüberstellt. Eine breite Anwendung können beide bis jetzt nicht vorzeigen, auch wenn DEX bereits eindrucksvoll seine Verwendung für Netzwerke am Beispiel von Bibex zeigt.

HyperGraphDB		**DEX**
Hypergraph - Knoten: Java-Objekte (POJO) - Kanten: Java-Objekte (Kanteninterface)	**Datenmodell**	Typisierter und gerichteter Multigraph mit Attributen (Property-Graph)
Traverser-API, HGQuery API	**Query-Methode**	Traverser-API
Java	**API**	Java
Berkeley DB	**Persistenz**	Eigenes Persistenzformat
ACI(D), Software-Transactional-Memory	**Transaktionen**	-
Peer-to-Peer	**Replikation/Skalierung**	-
LGPL	**Lizenz**	Kommerziell und Test-Lizenz
Künstliche Intelligenz, Verarbeitung natürlicher Sprache und Semantic Web	**Zielsetzung**	Hohe Performanz und Integration verteilter Informationsquellen

Tabelle 1: Vergleich der Graphdatenbanken HyperGraphDB und DEX anhand einiger Parameter.

Literaturverzeichnis

1: Edlich, S.; Friedland, A.; Hampe, J.; Brauer, B., NoSQL - Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken, 2010

2: DEX, DEX Website, , http://www.sparsity-technologies.com/dex

3: Dominquez-Sal, D.; Urbon-Bayes, P.; Giménez-Vanó, A., Survey of Graph Database Performance on the HPC Scalable Gaph Analysis Benchmark, 2010

4: Martinez-Bazan, N.; Muntes-Mulero, V.; Gomez-Villamor, S., DEX: High-Performance Exploration on large Graphs for Information Retrieval, 2007

5: Bibex, Bibex: Website, 2012, http://www.sparsity-technologies.com/bibex

6: DEX, DEX Seminar October 2010, 2010, http://www.slideshare.net/SparsityTechnologies/dex-seminar-tutorial

7: DEX, A High-Performance Graph Database Management System, 2012, http://www.slideshare.net/sakrsherif/dex-7656848

8: HypergraphDB, HypergraphDB Website, 2012, http://www.hypergraphdb.org/index

9: HypergrahDB, HypergraphDB: Motivation, Architecture and Applications, 2012, http://www.slideshare.net/borislav/hypergraphdb

10: Iordanov, B., HyperGraphDB: A Generalized Graph Database, 2010

11: HypergraphDB, HypergraphDB: Data Management for Complex Systems, 2012, http://www.hypergraphdb.org/docs/HyperGraphDB-Presentation.pdf

12: Chakrabarti, D.; Zhan, Y.; Faloutsos, C., R-mat: A recursive model for graph mining., 2004

Anhang

A: Nachfolgendes Code-Beispiel ist aus dem Seminartutorial von DEX vom Oktober 2010 übernommen ([6]).

```
DEX dex = new DEX();
GraphPool gpool = dex.create(“C:/image.dex”);
...
DbGraph dbg = gpool.getDbGraph();
int person = dbg.newNodeType(“PERSON”);
long name = dbg.newAttribute(person, “NAME”, STRING);
long age= dbg.newAttribute(person, “AGE”, INT);
long p1 = dbg.newNode(person);
dbg.setAttribute(p1, name, “JOHN”);
dbg.setAttribute(p1, age, 18);
long p2 = dbg.newNode(person);
dbg.setAttribute(p2, name, “KELLY”);
long p3 = dbg.newNode(person);
dbg.setAttribute(p3, name, “MARY”);
int friend = dbg.newUndirectedEdgeType(“FRIEND”);
int since = dbg.newAttribute(friend, “SINCE”, INT);
long e1 = dbg.newEdge(p1, p2, friend);
dbg.setAttribute(e1, since, 2000);
long e2 = dbg.newEdge(p2, p3, friend);
dbg.setAttribute(e2, since, 1995);
...
int loves = dbg.newEdgeType(“LOVES”);
long e3 = dbg.newEdge(p1, p3, loves);
int phones = dbg.newEdgeType(“PHONES”);
int when = dbg.newAttribute(phones, “WHEN”, TIMESTAMP);
long e4 = dbg.newEdge(p1, p3, phones);
```

```
dbg.setAttribute(e4, when, 4pm);
long e5 = dbg.newEdge(p1, p3, phones);
dbg.setAttribute(e5, when, 5pm);
long e6 = dbg.newEdge(p3, p2, phones);
dbg.setAttribute(e6, when, 6pm);
Objects persons = dbg.select(person);
Objects.Iterator it = persons.iterator();
While (it.hasNext()) {
       long p = it.next();
       String name = dbg.getAttribute(p, name);
}
it.close();
persons.close();
Objects objs1 = dbg.select(when, >=, 5pm); // objs1 = { e5, e6 }
Objects objs2 = dbg.explode(p1, phones, OUT); // objs2 = { e4, e5 }
Objects objs = objs1.intersection(objs2); // objs = { e5, e6 }    { e4, e5 } = { e5 } ...
objs.close();
objs1.close();
objs2.close();
...
gpool.close();
dex.close();
```

B: Das Code-Beispiel für HyperGraphDB wurde mit Hilfe der Beschreibung aus [1] und der Dokumentation auf [8] erstellt, sodass ein Vergleich zum Anhang A möglich wird.

```
HyperGraph graph = HGEnvironment.get(„c:/image_hgdb“);
graph.getTransactionManager().beginTransaction();
try
{
       HGHandle _John = graph.add(new Person(„John“, 18));
       HGHandle _Kelly = graph.add(new Person(„Kelly“));
       HGHandle _Mary = graph.add(new Person(„Mary“));

       HGHandle _John_Mary_loves = graph.add(new HGValueLink(„loves“, _John, _Mary ));
       HGHandle _John_Kelly_friend = graph.add(new HGValueLink(2000, _John, _Kelly ));
       HGHandle _Kelly_Mary_friend = graph.add(new HGValueLink(1995, _Kelly, _Mary ));

       HGHandle _Kelly_Mary_phones = graph.add(new HGValueLink(6pm, _Kelly, _Mary ));
       HGHandle _John_Mary_phones4 = graph.add(new HGValueLink(4pm, _John, _Mary ));
       HGHandle _John_Mary_phones5 = graph.add(new HGValueLink(5pm, _John, _Mary ));

       HGQueryCondition _AllPersons = new And(new AtomTypeCondition(Person.class));
       HGSearchResult _RS_Persons = graph.find(_AllPersons);
       try
       {
              while(_RS_Persons.hasNext())
              {
                     HGHandle _CurrentHandle = _RS_Persons.next();
                     Person _CurrentPerson = (Person) graph.get(_CurrentHandle);
              }
       }
       finally
       {
              _RS_Persons.close();
       }

       graph.getTransactionManager().commit();
}
catch (Throwable t)
{
       graph.getTransactionManager().abort();
}
```