Graph Processing with Neo4j

Lecture BigData Analytics

Julian M. Kunkel

julian.kunkel@googlemail.com

University of Hamburg / German Climate Computing Center (DKRZ)

2017-12-08

Disclaimer: Big Data software is constantly updated, code samples may be outdated.

Outline

Neo4j [31, 32]

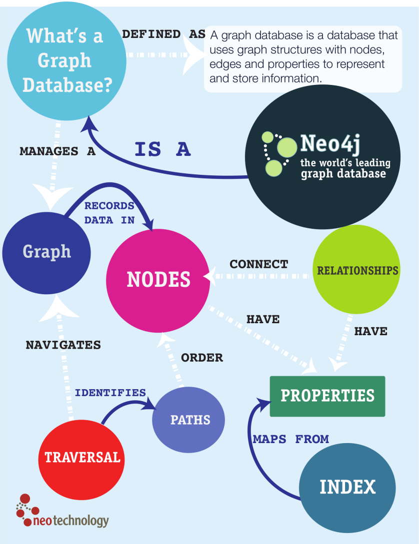
- Graph database written in Java
- Supports ACID transaction semantics
- One server scales to billions of nodes/relationships
  - Performance: Millions of node traversals/s
- High availability (and performance) through clustering
- Declarative query language Cypher (instead of SQL)
- Note: Very loose connection to Hadoop ecosystem
  - E.g., Prepare data in HBASE for batch import in Neo4j
  - Suboptimal import of Millions of nodes can take days
- Schema-optional: You can use a schema
  - To gain performance
  - To improve modeling, e.g., via constraints
- Many interfaces to the graph database

Graph Data Model (Slight Changes for Neo4j)

- Nodes: Entity
- Edges: Relationship between two nodes
  - They have a direction and type
- Property: (key, value)
  - Attributes describe relationships/nodes
  - Key is string, the value has a datatype
- Label: Organize nodes into groups

Definitions for queries

- Path: One or more nodes with connecting relationships
- Traversal: Navigates through a graph to find paths

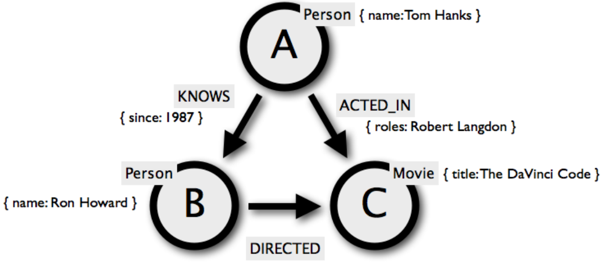


Source: What's a Graph Database [31]

Example Graph Use-Cases

Movie and actors data [31]

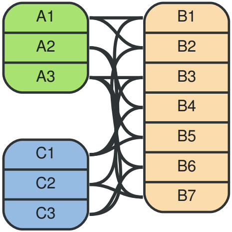
- Movies: label, title, released date, tagline
- People: label, name, born (optional date)
- Relationships
  - ACTED_IN from actor to movie, roles (list of played chars)
  - DIRECTED from director to movie



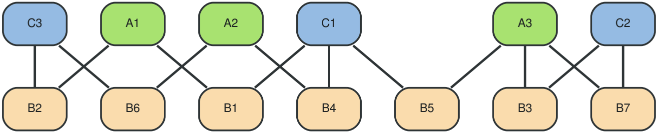
Source: Online Course: Introduction to Graph Databases and Neo4j [31]

Converting RDBMS to Graphs

- Consider three tables A,B,C
- Relations between rows (foreign keys) become edges

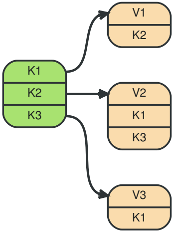


Source: RDBMS. The Neo4j Manual v2.2.5 [33]

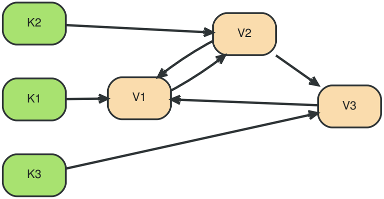


Source: Graph Database as RDBMS. The Neo4j Manual v2.2.5 [33]

Converting Key-Value Store Models to Graphs

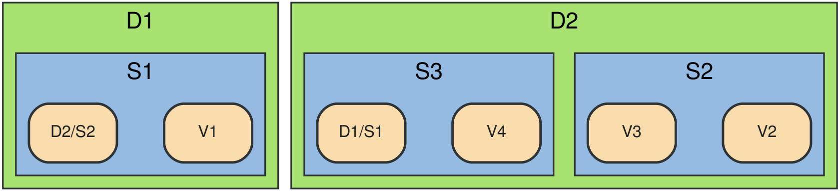


Source: Key-Value Store. The Neo4j Manual v2.2.5 [33]



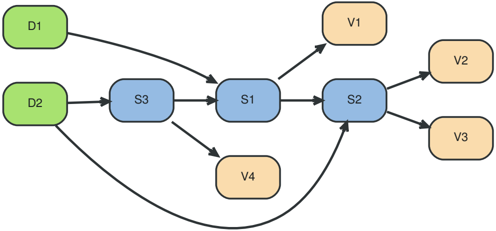
Source: Graph Database as Key-Value Store. The Neo4j Manual [33]

Converting the Document Store Model to Graphs



Source: Document Store. The Neo4j Manual v2.2.5 [33]

D=Document, S=Subdocument, V=Value, X/Y=reference to a subdocument in another document



Source: Graph Database as Document Store. The Neo4j Manual v2.2.5 [33]

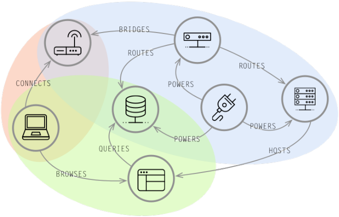
Neo4j Case Success Studies [31]

For the logistics company Accenture

- Use case: Dynamic parcel routing (5 million parcels/day)
- With Neo4j: Routing of packets online, i.e., where to load a parcel

For the communication company SFR

- Use case: Prioritize hardware replacement to minimize downtime
  - Run automated "what if" analysis to ensure resilience
- With Neo4j: Loading data from > 30 systems works; easier analysis model



Source: [36]

Cypher Query Language Basics [31]

- Declarative query language for formulating graph queries
  - Define matches based on structural patterns
- Allows query and/or update of the graph
  - Each part of a query must be read-only or write-only
  - A query consists of multiple clauses
- Transactions can span multiple queries
- Supports: variables, expressions[1], operators, comments
- Supports collections (list, dictionary)
- Provides functions for aggregation, collections, strings, math

[1] Handling missing values with NULL is possible, see
http://neo4j.com/docs/stable/cypher-working-with-null.html

Cypher Query Language [33]

Syntax: specifying graph structures via patterns

- Node
  - Anonymous node: ()
  - Named node: (x), the variable x is used to refer to it
  - Node with a specific label/class: $(x : label)$
- Relationship
  - Named relationship: $-[r]->$
  - Typed relationship: $-[r : t]->$
  - Two nodes with a relationship: $(a) - [r]-> (b)$
- Properties can be specified in {}, e.g., (x {name:"Hans"})
- A pattern combines several nodes/relations

Cypher Query Language Read Clauses [33]

- LOAD CSV: read data from a CSV file, can be used for importing
- MATCH: search for something (returns a relational table)
  - DISTINCT keyword: Avoid replicates (e.g., returning a node twice)
  - OPTIONAL MATCH: optional relationship like SQL outer join
- WHERE: Filtering based on properties and pattern
  - Supports regex matching of strings
  - Pattern predicates restrict the graph's shape
- Aggregation functions
  - **Automatic grouping on all non-aggregated columns**
  - sum, avg, percentileDisc, count
    - e.g., count(*), count(DISTINCT X)
  - collect(x): creates a list of all values

Cypher Query Language Write Clauses [33]

- CREATE: an element or relation
- MERGE: Create or lookup (CREATE + MATCH)
- SET: Modify/Add data/labels
- REMOVE: remove labels and properties
- DELETE: remove graph elements

Cypher Query Language: Interactive Session

```
# Create a star graph
$ CREATE (c) FOREACH (x IN range(1,6)| CREATE (l),(c)-[:X]->(l))  RETURN id(c);
 id
  0
Updated the graph - created 7 nodes and 6 relationships

# Count the number of nodes
$ MATCH (n) RETURN count(n); # since we have not defined any restriction, all nodes
 count(n)
    7

# Count relationships based on their relationship type
$ MATCH ()-[r]->() RETURN type(r), count(*);
 type(r) count(*)
    X    6

# Set the center node's name property to CENTER
$ MATCH (n) WHERE id(n) = 184 SET n.name = "CENTER";

# Clean the database
$ MATCH (n) OPTIONAL MATCH (n)-[r]-() DELETE n, r;
```

Cypher Query Language General Clauses [33]

- FOREACH($< col >$|$< op >$): iterates through a collection, apply op
- RETURN: return the subgraph/table
  - Usually you can convert those into a response table
- AS x: rename column to x
- ORDER BY x (ASC|DESC): sorting
- SKIP, LIMIT X: paginate – organize output
- UNION: compose statements
- WITH: a barrier for a pipeline of multiple statements
  - Example: retrieve the top entries by a criteria and join it with other data
  - Allows also to combine read-only and write-only parts
  - Aggregated results must pass through a WITH clause
- UNWIND: expand a collection into a sequence of rows
- USING: instruction to use/avoid indexes

Cypher Query Language [33]: Selection of Functions

- id(): the node id
- timestamp(): a timestamp
- label(): the node label
- upper(), lower(): change case
- range(l,u): return a collection with numbers from l to u
- length(x): size of a collection
- keys(x): keys of a dictionary
- coalesce(x, y): use property x if available, else y
- nodes(path), rels(path), length(path)

Cypher Query Language: Examples [33]

```
# Return a collection
$ RETURN [1, 2, 3]

# Return a string with a row name of X
$ RETURN "BigData" as X

# Return a dictionary
$ RETURN {key1 : 2, key2 : "test"}

# Return a list of x^3 where x is an even number
$ RETURN [x IN range(1,10) WHERE x % 2 = 0 | x^3] AS result

# populate a table
$ CREATE (matrix1:Movie { title : 'The Matrix', year : '1999-03-31' })
$ CREATE (keanu:Actor { name:'Keanu Reeves' })
$ CREATE (keanu)-[:ACTS_IN { role : 'Neo' }]->(matrix1)

# Create actor keanu if he does not exist
$ MERGE (keanu:Actor { name:'Keanu Reeves' })

# Eliminate duplicates from a collection
$ WITH [1,1,2,2] AS coll UNWIND coll AS x WITH DISTINCT x RETURN collect(x) AS SET
# [1,2]
```

Cypher Query Language: Examples [33]

```
# Read a table from a (large) CSV
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM 'http://neo4j.com/docs/2.2.5/csv/artists-with-headers.csv' AS line
CREATE (:Artist { name: line.Name, year: toInt(line.Year)})

MATCH (a:Movie { title: 'Wall Street' })
OPTIONAL MATCH (a)-->(x)
RETURN x

# return a movie with all its properties
MATCH (movie:Movie { title: 'The Matrix' })
RETURN movie;

# return certain attributes
MATCH (movie:Movie { title: 'The Matrix' })
RETURN movie.title, movie.year;

# show all actors sorted by name
MATCH (actor:Actor)
RETURN actor ORDER BY actor.name;

# all actors whose name end with s
MATCH (actor:Actor)
WHERE actor.name =~ ".*s$"
RETURN actor.name;
```

Cypher Query Language: Examples [33]

```
# List all nodes together with their relationsships
MATCH (n)-[r]->(m)  RETURN n AS from, r AS ‘->‘, m AS to;

# Return number of movies for actors acting in "The Matrix"
MATCH (:Movie { title: "The Matrix" })<-[:ACTS_IN]-(actor)-[:ACTS_IN]->(movie)
RETURN movie.title, collect(actor.name), count(*) AS count
ORDER BY count DESC;

# Filtering
MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
WHERE p.name =~ "K.+" OR m.released > 2000 OR "Neo" IN r.roles
RETURN p,r,m;

# Filtering based on graph structure
# Here: Search for people that are actors in any movie but never directed any movie
MATCH (p:Person)-[:ACTED_IN]->(m)
WHERE NOT (p)-[:DIRECTED]->()
RETURN p,m;

# Identify how often actors and directors worked together
MATCH (actor:Person)-[:ACTED_IN]->(movie:Movie)<-[:DIRECTED]-(director:Person)
RETURN actor,director,count(*) AS collaborations;
```

Cypher Query Language: Examples [33]

```
# Use UNION to combine results
MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
RETURN p,type(r) AS rel,m
UNION
MATCH (p:Person)-[r:DIRECTED]->(m:Movie)
RETURN p,type(r) AS rel,m

# Return five actors of each movie
MATCH (m:Movie)<-[:ACTED_IN]-(a:Person)
RETURN m.title AS movie, collect(a.name)[0..5] AS five_of_cast

# Use list predicates to restrict set further, here all relations must be acted in
MATCH path =(:Person)-->(:Movie)<--(:Person)
WHERE ALL (r IN rels(path) WHERE type(r)= ’ACTED_IN’) AND ANY (n IN nodes(path) WHERE
      ↪ n.name = ’Clint Eastwood’)
RETURN path
# Update values based on a query
MATCH (n {name: ’John’})-[:FRIEND]-(friend)
WITH n, count(friend) as friendsCount
WHERE friendsCount > 3
SET n.friendCount = friendsCount
RETURN n, friendsCount

# Update all nodes on possible paths between two nodes
MATCH p =(begin)-[*]->(end)
WHERE begin.name=’A’ AND end.name=’D’
FOREACH (n IN nodes(p)| SET n.marked = TRUE )
```

Schemas [33]

- Neo4j offers a few schema options to influence graph setup
- Simple constraints can be created using CREATE

```
CREATE CONSTRAINT ON (p:Person) ASSERT p.name IS UNIQUE
# And removed with DROP
DROP CONSTRAINT ON (p:Person) ASSERT p.name IS UNIQUE
```

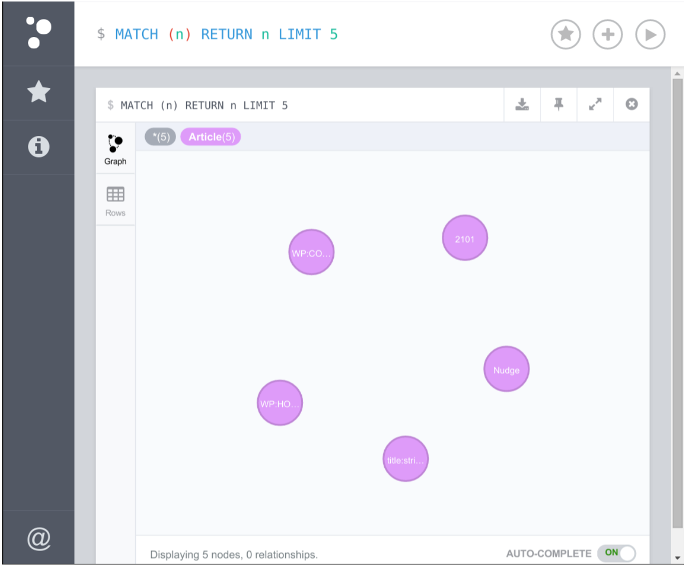
- Indexes for lookup

```
CREATE INDEX ON :Person(name)
DROP INDEX ON :Person(name)
```

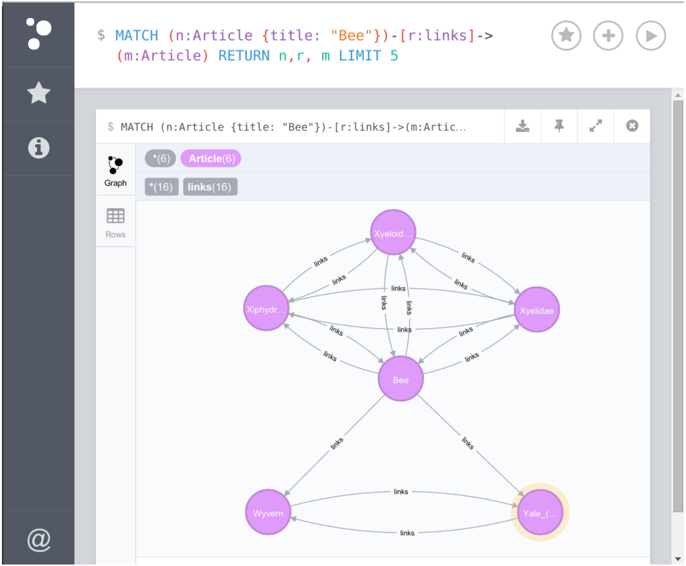
Overview of the Interfaces

- Neo4j shell [38]
  - Create, import, export, execute Cypher
  - Present results as ASCII tables
- Web interface
  - Provides a shell for Cypher
  - Visualizes query results
  - Allows (performance) monitoring of Neo4j
  - Ships with Examples/Tutorials!
  - HTTPS support
- Java API
  - Core Java API offers graph algorithms & is faster than CQL
  - JCypher: DSL for higher abstraction level
  - Automatic object-graph mapping via annotations
- Relational mapping with JDBC driver
- REST, Python, ...

Web Interface: Example Queries

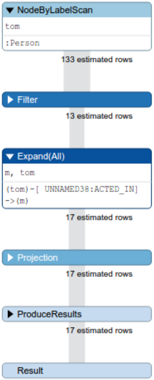


Web Interface: Example Queries

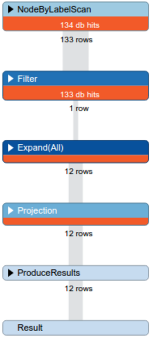


Clauses for Debugging of Queries

- EXPLAIN: shows the execution plan
- PROFILE: runs the statement and shows where time is spend



EXPLAIN ...



Cypher version: CYPHER 2.3, planner: COST. 292 total db hits in 78 ms.

PROFILE ...

MATCH (tom:Person name:"Tom Hanks")-[:ACTED_IN]->(m) RETURN m.name

Java API: Example for our Student Table. See [37]

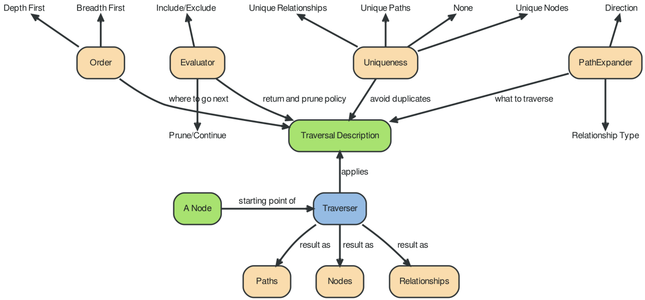
```
private static enum MyRelationTypes implements RelationshipType
{ ATTENDS } // we can use enums for relation types

public static void main(String [ ] args){
  GraphDatabaseService graphDb; // start database server
  graphDb = new GraphDatabaseFactory().newEmbeddedDatabaseBuilder(File("x"));
  registerShutdownHook( graphDb );

  Node student;   Node lecture;  Relationship attends;
  // encapsulate operations into a transaction
  try ( Transaction tx = graphDb.beginTx() ){
    student = graphDb.createNode();
    student.setProperty( "Name", "Hans" );
    lecture = graphDb.createNode();
    lecture.setProperty( "Lecture", "Big Data Analytics" );
    attends = student.createRelationshipTo( lecture, RelTypes.ATTENDS );
    attends.setProperty( "Semester", "1718" );
    tx.success();
  }
  graphDb.shutdown(); // shutdown application server
}
```

Evaluation of Cypher expressions [33]

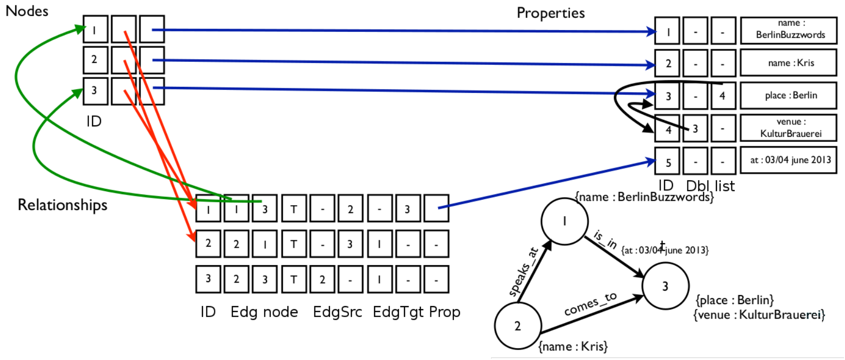
- An execution planner transforms a query into a plan
  - Rule-based planner uses indexes
  - Cost-based planner uses statistical information
- Use indices if available
- Order (DFS or BFS)
- Uniqueness: avoid duplicates
- Evaluator: decide what to return and when to stop
- Recursive matching with backtracking



Source: The Neo4j Manual 2.2.5 (36.1. Main Concepts) [33]

Neo4j Architecture: On-Disk Format [32]

- Physically, multiple “store files” are used
- Data is stored as double linked lists of records
- Storage for nodes, relationships and properties
  - Long values are persisted in separate array and string stores



Source: K. Geusebroek. I MapReduced a Neo store [34] (modified)

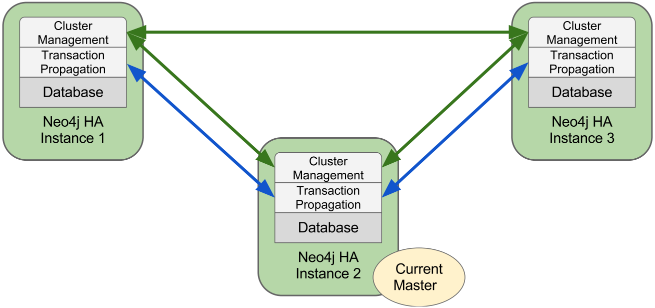
Neo4j Consistency [32]

- ACID transaction support
  - Isolation of concurrent operations until transaction is completed
  - All write operations are sorted (before stored/communicated) to ensure predictable update order
  - Write changes are applied in sorted order to the transaction log
  - Apply the changes to the store files
  - Implemented via locking of Nodes/Relationships during transaction
- Upon completion of transaction, changes are persisted
- Recovery: re-applies the transaction log

Neo4j High-Availability [32, 33, 35]

- Neo4j clustering replicates the database across servers
- One master multiple slaves provides
  - Data redundancy
  - Service fault tolerance
- A master election protocol is used
- A quorum (majority) of servers must be up to serve writes
- Transactions are first commited to master
  - Creating an incrementing transaction id (txid)
  - Eventually applied to slaves sending streams
  - Update interval defines delay
- Applying transactions to a slave
  - The master coordinates locking
  - After applying transaction on master
  - The slave uses the same txid

Neo4j High-Availability Architecture [33]



Source: The Neo4j Manual 2.2.5 (25.1. Architecture) [33]

Neo4j Performance Aspects [32]

- Remember: Data is completely replicated across servers
- Clustered Neo4j allows horizontal scaling of reads
- Writes are always coordinated by the master
  - Transactions can be speed up with batch inserts and periodic commits
  - The file format is optimzed for graph-local operations
  - Indexing and caching speed up access
- Fine lock granularity (on node/relationship level)
- Consistency: Nodes/Relationships have an unique ID
  - Blocks for IDs are pre-allocated from the master
  - Creation of nodes/relationships does not require a lock

Performance Aspects [32]

Indexing

- Index: Node labels, relation type and property values
- Eventually available, populated in background
- Handled via Apache Lucene search library
- Automatic indexing possible

Caches

- Filesystem cache: for blocks of store files
  - LFU eviction policy
  - Uses *mmap*() to map data blocks into memory
- Node/Relationship cache

Summary

- Neo4j is a powerful graph database
- ACID transaction semantics
- Other data models can be converted to graphs
- Many interfaces for accessing graph
- CypherQL is the SQL for the Neo4j graph DB
- Interactive web interface processes CQL
- Simple file format with linked lists
- Clustering increases read scalability

Bibliography

10 Wikipedia

31 Interactive Online Course
http://neo4j.com/graphacademy/online-training/

32 http://de.slideshare.net/thobe/an-overview-of-neo4j-internals

33 The Neo4j Manual v2.2.5. http://neo4j.com/docs/stable/

34 I MapReduced a Neo store. Kris Geusebroek.
http://2013.berlinbuzzwords.de/sites/2013.berlinbuzzwords.de/files/slides/CreatingLargeNeo4jDatabasesWithHaddoop.pdf

35 D. Montag. Understanding Neo4j Scalability.

36 http://neo4j.com/use-cases/

37 http://neo4j.com/docs/stable/tutorials-java-embedded-hello-world.html

38 http://neo4j.com/docs/stable/tools.html