Big Data Management and NoSQL Databases

Lecture 12. Graph Databases – Neo4j

PD Dr. Andreas Behrend
Neo4j

- Open source graph database
  - The most popular
- Initial release: 2007
- Written in Java
- OS: cross-platform
- Stores data in nodes connected by directed, typed relationships
  - With properties on both
  - Called property graph

http://www.neo4j.org/
Neo4j
Main Features (according to Authors)

- intuitive – a graph model for data representation
- reliable – with full ACID transactions
- durable and fast – disk-based, native storage engine
- massively scalable – up to several billions of nodes / relationships / properties
- highly-available – when distributed across multiple machines
- expressive – powerful, human readable graph query language
- fast – powerful traversal framework
- embeddable
- simple – accessible by REST interface / object-oriented Java API
RDBMS vs. Neo4j

- **RDBMS** is optimized for aggregated data
- Neo4j is optimized for highly connected data
Key-Value (Column Family) Store vs. Neo4j

- **Key-Value** model is for lookups of simple values or lists
  - Column family store can be considered as a step in evolution of key/value stores
    - The value contains a list of columns

- Neo4j lets you elaborate the simple data structures into more complex data
  - Interconnected
Document Store vs. Neo4j

- **Document database** accommodates data that can easily be represented as a tree
  - Schema-free
- References to other documents within the tree = more expressive representation
**Neo4j**

**Data Model – Node, Relationship, Property**

- **Fundamental units:** nodes + relationships
- **Both can contain properties**
  - Key-value pairs where the key is a string
  - Value can be primitive or an array of one primitive type
    - *e.g.*, `String`, `int`, `int[]`, ...
  - `null` is not a valid property value
    - nulls can be modelled by the absence of a key
- **Relationships**
  - Directed (incoming and outgoing edge)
    - Equally well traversed in either direction = no need to add both directions to increase performance
    - Direction can be ignored when not needed by applications
  - Always have start and end node
  - Can be recursive
<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Value range</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td></td>
<td>true/false</td>
</tr>
<tr>
<td>byte</td>
<td>8-bit integer</td>
<td>-128 to 127, inclusive</td>
</tr>
<tr>
<td>short</td>
<td>16-bit integer</td>
<td>-32768 to 32767, inclusive</td>
</tr>
<tr>
<td>int</td>
<td>32-bit integer</td>
<td>-2147483648 to 2147483647, inclusive</td>
</tr>
<tr>
<td>long</td>
<td>64-bit integer</td>
<td>-9223372036854775808 to 9223372036854775807, inclusive</td>
</tr>
<tr>
<td>float</td>
<td>32-bit IEEE 754 floating-point number</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>64-bit IEEE 754 floating-point number</td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>16-bit unsigned integers representing Unicode characters</td>
<td>u0000 to ufff (0 to 65535)</td>
</tr>
<tr>
<td>String</td>
<td>sequence of Unicode characters</td>
<td></td>
</tr>
<tr>
<td>What</td>
<td>How</td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>--------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>get who a person follows</td>
<td>outgoing follows relationships, depth one</td>
<td></td>
</tr>
<tr>
<td>get the followers of a person</td>
<td>incoming follows relationships, depth one</td>
<td></td>
</tr>
<tr>
<td>get who a person blocks</td>
<td>outgoing blocks relationships, depth one</td>
<td></td>
</tr>
<tr>
<td>get who a person is blocked by</td>
<td>incoming blocks relationships, depth one</td>
<td></td>
</tr>
<tr>
<td>get the full path of a file</td>
<td>incoming file relationships</td>
<td></td>
</tr>
<tr>
<td>get all paths for a file</td>
<td>incoming file and symbolic link relationships</td>
<td></td>
</tr>
<tr>
<td>get all files in a directory</td>
<td>outgoing file and symbolic link relationships, depth one</td>
<td></td>
</tr>
<tr>
<td>get all files in a directory, excluding symbolic links</td>
<td>outgoing file relationships, depth one</td>
<td></td>
</tr>
<tr>
<td>get all files in a directory, recursively</td>
<td>outgoing file and symbolic link relationships</td>
<td></td>
</tr>
</tbody>
</table>
Neo4j

“Hello World” Graph – Java API

// enum of types of relationships:
private static enum RelTypes implements RelationshipType
{
    KNOWS
};

GraphDatabaseService graphDb;
Node firstNode;
Node secondNode;
Relationship relationship;

// starting a database (directory is created if not exists):
graphDb = new
    GraphDatabaseFactory().newEmbeddedDatabase(DB_PATH);

// ...
Neo4j
“Hello World” Graph

```java
// create a small graph:
firstNode = graphDb.createNode();
firstNode.setProperty("message", "Hello, "
secondNode = graphDb.createNode();
secondNode.setProperty("message", "World!");

relationship = firstNode.createRelationshipTo
    (secondNode, RelTypes.KNOWS);
relationship.setProperty
    ("message", "brave Neo4j ");

// ...
```
Neo4j

“Hello World” Graph

// print the result:
System.out.print( firstNode.getProperty( "message" ) );
System.out.print( relationship.getProperty( "message" ) );
System.out.print( secondNode.getProperty( "message" ) );

// let's remove the data:
firstNode.getSINGLERelationship
   (RelTypes.KNOWS, Direction.OUTGOING).delete();
firstNode.delete();
secondNode.delete();

// shut down the database:
graphDb.shutdown();
Neo4j
“Hello World” Graph – Transactions

// all writes (creating, deleting and updating any data)
// have to be performed in a transaction,
// otherwise NotInTransactionException

Transaction tx = graphDb.beginTx();
try
{
    // updating operations go here
    tx.success(); // transaction is committed on close
}
catch (Exception e)
{
    tx.failure(); // transaction is rolled back on close
}
finally
{
    tx.close(); // or deprecated tx.finish()
}
Path = one or more nodes with connecting relationships
- Typically retrieved as a query or traversal result

Traversing a graph = visiting its nodes, following relationships according to some rules
- Mostly a subgraph is visited
- Neo4j: Traversal framework + Java API, Cypher, Gremlin
Neo4j
Traversal Framework

A traversal is influenced by

- **Expanders** – define what to traverse
  - i.e., relationship direction and type
- **Order** – depth-first / breadth-first
- **Uniqueness** – visit nodes (relationships, paths) only once
- **Evaluator** – what to return and whether to stop or continue traversal beyond a current position
- **Starting nodes** where the traversal will begin
Neo4j
Traversal Framework – Java API

- **TraversalDescription**
  - The main interface used for defining and initializing traversals
  - Not meant to be implemented by users
    - Just used
  - Can specify branch ordering
    - `breadthFirst()` / `depthFirst()`

- **Relationships**
  - Adds a relationship type to traverse
    - Empty (default) = traverse all relationships
    - At least one in the list = traverse the specified ones
  - Two methods: including / excluding direction
    - `Direction.BOTH`
    - `Direction.INCOMING`
    - `Direction.OUTGOING`
Neo4j
Traversal Framework – Java API

Evaluator

- Used for deciding at each position: should the traversal continue, and/or should the node be included in the result

- Actions:
  - Evaluation.INCLUDE_AND_CONTINUE: Include this node in the result and continue the traversal
  - Evaluation.INCLUDE_AND_PRUNE: Include this node in the result, but do not continue the traversal
  - Evaluation.EXCLUDE_AND_CONTINUE: Exclude this node from the result, but continue the traversal
  - Evaluation.EXCLUDE_AND_PRUNE: Exclude this node from the result and do not continue the traversal

- Pre-defined evaluators:
  - Evaluators.excludeStartPosition()
  - Evaluators.toDepth(int depth) / Evaluators.fromDepth(int depth)
  - ...

Neo4j
Traversal Framework – Java API

- **Uniqueness**
  - Can be supplied to the `TraversalDescription`
  - Indicates under what circumstances a traversal may revisit the same position in the graph
    - **NONE**: Any position in the graph may be revisited.
    - **NODE_GLOBAL**: No node in the graph may be re-visited (default)
    - ...

- **Traverser**
  - Traverser which is used to step through the results of a traversal
  - Steps can correspond to
    - **Path** (default)
    - **Node**
    - **Relationship**
Neo4j Example

- membership of a group
- group hierarchy
- top level group
Neo4j

Task 1. Get the Admins

Node admins = getNodeByName("Admins");
TraversalDescription traversalDescription = Traversal.description()
    .breadthFirst()
    .evaluator(Evaluators.excludeStartPosition())
    .relationships(RoleRels.PART_OF, Direction.INCOMING)
    .relationships(RoleRels.MEMBER_OF, Direction.INCOMING);
Traverser traverser = traversalDescription.traverse(admins);

String output = "";
for (Path path : traverser)
{
    Node node = path.endNode();
    output += "Found: "+ node.getProperty(NAME) + " at depth: "+ (path.length() - 1) + "\n";
}

Found: HelpDesk at depth: 0
Found: Ali at depth: 0
Found: Engin at depth: 1
Found: Demet at depth: 1
Node jale = getNodeByName( "Jale" );
traversalDescription = Traversal.description()
  .depthFirst()
  .evaluator( Evaluators.excludeStartPosition() )
  .relationships( RoleRels.MEMBER_OF, Direction.OUTGOING )
  .relationships( RoleRels.PART_OF, Direction.OUTGOING );
traverser = traversalDescription.traverse( jale );

Found: ABCTechnicians at depth: 0
Found: Technicians at depth: 1
Found: Users at depth: 2
Neo4j
Task 3. Get All Groups

Node referenceNode = getNodeByName( "Reference_Node" ) ;
traversalDescription = Traversal.description()
    .breadthFirst()
    .evaluator( Evaluators.excludeStartPosition() )
    .relationships( RoleRels.ROOT, Direction.INCOMING )
    .relationships( RoleRels.PART_OF, Direction.INCOMING );
traverser = traversalDescription.traverse( referenceNode );

Found: Admins at depth: 0
Found: Users at depth: 0
Found: HelpDesk at depth: 1
Found: Managers at depth: 1
Found: Technicians at depth: 1
Found: ABCTechnicians at depth: 2
Task 4. Get All Members of a Group

Node referenceNode = getNodeByName( "Reference_Node" ) ;
traversalDescription = Traversal.description()
    .breadthFirst()
    .evaluator(
        Evaluators.includeWhereLastRelationshipTypeIs
            ( RoleRels.MEMBER_OF ) );
traverser = traversalDescription.traverse( referenceNode );

<table>
<thead>
<tr>
<th>Found</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ali</td>
<td>1</td>
</tr>
<tr>
<td>Engin</td>
<td>1</td>
</tr>
<tr>
<td>Burcu</td>
<td>1</td>
</tr>
<tr>
<td>Can</td>
<td>1</td>
</tr>
<tr>
<td>Demet</td>
<td>2</td>
</tr>
<tr>
<td>Gul</td>
<td>2</td>
</tr>
<tr>
<td>Fuat</td>
<td>2</td>
</tr>
<tr>
<td>Hakan</td>
<td>2</td>
</tr>
<tr>
<td>Irmak</td>
<td>2</td>
</tr>
<tr>
<td>Jale</td>
<td>3</td>
</tr>
</tbody>
</table>
Cypher

- Neo4j graph query language
  - For querying and updating
- Still growing = syntax changes are probable
- Declarative – we describe what we want, not how to get it
  - Not necessary to express traversals
- Human-readable
  - Inspired by SQL and SPARQL

Cypher Clauses

- **START**: Starting points in the graph, obtained via index lookups or by element IDs.
- **MATCH**: The graph pattern to match, bound to the starting points in START.
- **WHERE**: Filtering criteria.
- **RETURN**: What to return.
- **CREATE**: Creates nodes and relationships.
- **DELETE**: Removes nodes, relationships and properties.
- **SET**: Set values to properties.
- **FOREACH**: Performs updating actions once per element in a list.
- **WITH**: Divides a query into multiple, distinct parts.
Cypher Examples
Creating Nodes

```
CREATE n

(empty result)
Nodes created: 1

CREATE (a {name: 'Andres'})
RETURN a

a
Node[2]{name:"Andres"}
1 row
Nodes created: 1
Properties set: 1

CREATE (n {name: 'Andres', title: 'Developer'})

(empty result)
Nodes created: 1
Properties set: 2
```
Cypher Examples
Creating Relationships

START a=node(1), b=node(2)
CREATE a-[r:RELTYPE]->b
RETURN r

r
:RELTYPE[1] {}
1 row
Relationships created: 1

START a=node(1), b=node(2)
CREATE a-[r:RELTYPE {name : a.name + '<->' + b.name }]->b
RETURN r

r
:RELTYPE[1] {name:"Andres<->Michael"}
1 row
Relationships created: 1
Properties set: 1
Cypher Examples
Creating Paths

CREATE p = (andres {name:'Andres'})-[[:WORKS_AT]->neo<-[:WORKS_AT]-(michael {name:'Michael'})]
RETURN p

P
[Node[4]{name:"Andres"},:WORKS_AT[2]{}
,Node[6]{name:"Michael"}]
1 row
Nodes created: 3
Relationships created: 2
Properties set: 2

all parts of the pattern not already in scope are created
Cypher Examples
Changing Properties

START n = node(2)
SET n.surname = 'Taylor'
RETURN n

n
Node[2]{name:"Andres",age:36,awesome:true,surname:"Taylor"}
1 row
Properties set: 1

START n = node(2)
SET n.name = null
RETURN n

n
Node[2]{age:36,awesome:true}
1 row
Properties set: 1
Cypher Examples

Delete

START n = node(4)
DELETE n

(empty result)
Nodes deleted: 1

START n = node(3)
MATCH n-[r]-()
DELETE n, r

(empty result)
Nodes deleted: 1
Relationships deleted: 2
Cypher Examples

Foreach

START begin = node(2), end = node(1)
MATCH p = begin -[*]-> end foreach(n in nodes(p) : 
SET n.marked = true)

(empty result)
Properties set: 4

can be combined with any update command
Cypher Examples

Querying

START john=node:node_auto_index(name = 'John')
MATCH john-[[:friend]]->()-[:friend]->fof
RETURN john, fof

<table>
<thead>
<tr>
<th>john</th>
<th>fof</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node[4]{name:&quot;John&quot;}</td>
<td>Node[2]{name:&quot;Maria&quot;}</td>
</tr>
<tr>
<td>Node[4]{name:&quot;John&quot;}</td>
<td>Node[3]{name:&quot;Steve&quot;}</td>
</tr>
</tbody>
</table>

neo4j.properties file:
...
node_auto_indexing=true
relationship_auto_indexing=true
node_keys_indexable=name,phone
relationship_keys_indexable=since
...
Cypher Examples

Querying

```
START user=node(5,4,1,2,3)
MATCH user-[friend]->follower
WHERE follower.name =~ 'S.*'
RETURN user, follower.name
```

<table>
<thead>
<tr>
<th>user</th>
<th>follower.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node[5]{name:&quot;Joe&quot;}</td>
<td>&quot;Steve&quot;</td>
</tr>
<tr>
<td>Node[4]{name:&quot;John&quot;}</td>
<td>&quot;Sara&quot;</td>
</tr>
</tbody>
</table>

List of users
Cypher Examples

Order by

START n=node(3,1,2)
RETURN n
ORDER BY n.name

We can use:
• multiple properties
• asc/desc

<table>
<thead>
<tr>
<th>n</th>
<th>Node[1]{name-&gt;&quot;A&quot;, age-&gt;34, length-&gt;170}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Node[2]{name-&gt;&quot;B&quot;, age-&gt;34}</td>
</tr>
<tr>
<td></td>
<td>Node[3]{name-&gt;&quot;C&quot;, age-&gt;32, length-&gt;185}</td>
</tr>
</tbody>
</table>
Cypher Examples

Count

START n=node(2)
MATCH (n)-->(x)
RETURN n, count(*)

START n=node(2)
MATCH (n)-[r]->()
RETURN type(r), count(*)

count the groups of relationship types
Cypher

- And there are many other features
  - Other aggregation functions
    - Count, sum, avg, max, min
  - LIMIT n - returns only subsets of the total result
    - SKIP n = trimmed from the top
    - Often combined with order by
  - Predicates ALL and ANY
  - Functions
    - LENGTH of a path, TYPE of a relationship, ID of node/relationship,
      NODES of a path, RELATIONSHIPS of a path, …
  - Operators
  - …
Gremlin

- Gremlin = graph traversal language for traversing property graphs
  - Maintained by TinkerPop
    - Open source software development group
    - Focuses on technologies related to graph databases
  - Implemented by most graph database vendors
  - Neo4j Gremlin Plugin

- Scripts are executed on the server database
- Results are returned as Neo4j Node and Relationship representations

http://gremlindocs.com/
Gremlin
Property Graph

http://www.slideshare.net/sakrsherif/gremlin
TinkerPop and Related Stuff

- **Blueprints** – interface for graph databases
  - Like ODBC (JDBC) for graph databases

- **Pipes** – dataflow framework for evaluating graph traversals

- **Groovy** – superset of Java used by Gremlin as a host language

Gremlin Examples

https://github.com/tinkerpop/gremlin/wiki/Basic-Graph-Traversals
Gremlin Examples

gremlin> g = new Neo4jGraph('I:\tmp\myDB.graphdb')
==> neo4jgraph[EmbeddedGraphDatabase[I:\tmp\myDB.graphdb]]
gremlin> v = g.v(1)
==>v[1]
gremlin> v.outE
==>e[7][1-knows->2]
==>e[9][1-created->3]
==>e[8][1-knows->4]
gremlin> v.outE.inV
==>v[2]
==>v[3]
==>v[4]
gremlin> v.outE.inV.outE.inV
==>v[5]
==>v[3]

Gremlin steps:
- adjacency: outE, inE, bothE, outV, inV, bothV
- to skip edges: out, in, and both
Gremlin

Examples

gremlin> v = g.v(1)
===> v[1]
gremlin> v.name
===> marko

gremlin> v.outE('knows').inV.filter{it.age > 30}.name
===> josh

gremlin> v.out('knows').filter{it.age > 21}.as('x').name.filter{it.matches('jo.{2}|JO.{2}')}.back('x').age
===> 32
Gremlin Examples

gremlin> g.v(1).note= "my friend" // set a property
==> my friend

gremlin> g.v(1).map // get property map
==> {name=marko, age=29, note=my friend}

gremlin> v1= g.addVertex([name: "irena"])
==> v[7]

gremlin> v2 = g.v(1)
==> v[1]

gremlin> g.addEdge(v1, v2, 'knows')
==> e[7][7-knows->1]
More on Internals
Neo4j
Transaction Management

- Support for ACID properties
- All write operations that work with the graph must be performed in a transaction
  - Can have nested transactions
  - Rollback of nested transaction ⇒ rollback of the whole transaction

- Required steps:
  1. Begin a transaction
  2. Operate on the graph performing write operations
  3. Mark the transaction as successful or not
  4. Finish the transaction
    - Memory + locks are released (= necessary step)
Neo4j
Transaction Example

// all writes (creating, deleting and updating any data)
// have to be performed in a transaction,
// otherwise NotInTransactionException

Transaction tx = graphDb.beginTx();
try {
    // updating operations go here
    tx.success(); // transaction is committed on close
} catch (Exception e) {
    tx.failure(); // transaction is rolled back on close
} finally {
    tx.close(); // or deprecated tx.finish()
}
Neo4j
Transaction Management – Read

- Default:
  - Read operation reads the last committed value
  - Reads do not block or take any locks
    - Non-repeatable reads can occur
      - A row is retrieved twice and the values within the row differ between reads

- Higher level of isolation: read locks can be acquired explicitly
Neo4j
Transaction Management – Write

- All modifications performed in a transaction are kept in memory
  - Very large updates have to be split
- Default locking:
  - Adding/changing/removing a property of a node/relationship ⇒ write lock on the node/relationship
  - Creating/deleting a node ⇒ write lock on the specific node
  - Creating/deleting a relationship ⇒ write lock on the relationship + its nodes
- Deadlocks:
  - Can occur
  - Are detected and an exception is thrown
- Node/relationship is deleted $\Rightarrow$ all properties are removed
- Deleted node can not have any attached relationships
  - Otherwise an exception is thrown
- Write operation on a node or relationship after it has been deleted (but not yet committed) $\Rightarrow$ exception
  - It is possible to acquire a reference to a deleted relationship / node that has not yet been committed
  - After commit, trying to acquire new / work with old reference to a deleted node / relationship $\Rightarrow$ exception
Neo4j

Indexing

- Index
  - Has a unique, user-specified name
  - Indexed entities = nodes / relationships
- Index = associating any number of key-value pairs with any number of entities
  - We can index a node / relationship with several key-value pairs that have the same key
    ⇒ An old value must be deleted to set new (otherwise we have both)
Neo4j
Indexing – Create / Delete Index

```java
graphDb = new
    GraphDatabaseFactory().newEmbeddedDatabase(DB_PATH);
IndexManager index = graphDb.index();

// check existence of an index
boolean indexExists = index.existsForNodes("actors");

// create three indexes
Index<Node> actors = index.forNodes("actors");
Index<Node> movies = index.forNodes("movies");
RelationshipIndex roles = index.forRelationships("roles");

// delete index
actors.delete();
```
Node reeves = graphDb.createNode();
reeves.setProperty("name", "Keanu Reeves");
actors.add(reeves, "name", reeves.getProperty("name"));

Node bellucci = graphDb.createNode();
bellucci.setProperty("name", "Monica Bellucci");

// multiple index values for a field
actors.add(bellucci, "name", bellucci.getProperty("name"));
actors.add(bellucci, "name", "La Bellucci");

Node matrix = graphDb.createNode();
matrix.setProperty("title", "The Matrix");
matrix.setProperty("year", 1999);
movies.add(matrix, "title", matrix.getProperty("title"));
movies.add(matrix, "year", matrix.getProperty("year"));
Relationship role1 =
    reeves.createRelationshipTo(matrix, ACTS_IN);
role1.setProperty("name", "Neo");
roles.add(role1, "name", role1.getProperty("name"));

// completely remove bellucci from actors index
actors.remove( bellucci );
// remove any "name" entry of bellucci from actors index
actors.remove( bellucci, "name" );
// remove the "name" -> "La Bellucci" entry of bellucci
actors.remove( bellucci, "name", "La Bellucci" );
Node fishburn = graphDb.createNode();
fishburn.setProperty("name", "Fishburn");

// add to index
actors.add(fishburn, "name", fishburn.getProperty("name"));

// update the index entry when the property value changes
actors.remove(fishburn, "name", fishburn.getProperty("name"));
fishburn.setProperty("name", "Laurence Fishburn");
actors.add(fishburn, "name", fishburn.getProperty("name"));
Neo4j
Indexing – Search using `get()`

```java
// get single exact match
IndexHits<Node> hits = actors.get("name", "Keanu Reeves");
Node reeves = hits.getSingle();

Relationship persephone =
    roles.get("name", "Persephone").getSingle();
Node actor = persephone.getStartNode();
Node movie = persephone.getEndNode();

// iterate over all exact matches from index
for ( Relationship role : roles.get("name", "Neo") )
{
    // this will give us Reeves e.g. twice
    Node reeves = role.getStartNode();
}
```
Neo4j
Indexing – Search using `query()`

```java
for ( Node a : actors.query("name", "*e*"))
{
    // This will return Reeves and Bellucci
}

for (Node m : movies.query("title:*Matrix* AND year:1999"))
{
    // This will return "The Matrix" from 1999 only
}
```
Neo4j
Automatic Indexing

- One automatic index for nodes and one for relationships
  - Follow property values
  - By default off

- We can specify properties of nodes / edges which are automatically indexed
  - We do not need to add them explicitly

- The index can be queried as any other index
# Neo4j Data Size

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>$2^{35}$ (~ 34 billion)</td>
</tr>
<tr>
<td>Relationships</td>
<td>$2^{35}$ (~ 34 billion)</td>
</tr>
<tr>
<td>Properties</td>
<td>$2^{36}$ to $2^{38}$ depending on property types (maximum ~ 274 billion, always at least ~ 68 billion)</td>
</tr>
<tr>
<td>Relationship Types</td>
<td>$2^{15}$ (~ 32 000)</td>
</tr>
</tbody>
</table>
Neo4j
High Availability (HA)

- Provides the following features:
  - Enables a fault-tolerant database architecture
    - Several Neo4j slave databases can be configured to be exact replicas of a single Neo4j master database
  - Enables a horizontally scaling read-mostly architecture
    - Enables the system to handle more read load than a single Neo4j database instance can handle

- Transactions are still atomic, consistent and durable, but eventually propagated out to other slaves
Neo4j
High Availability

- Transition from single machine to multi machine operation is simple
  - No need to change existing applications
  - Switch from `GraphDatabaseFactory` to `HighlyAvailableGraphDatabaseFactory`
    - Both implement the same interface
- Always one master and zero or more slaves
  - Write on master: eventually propagated to slaves
    - All other ACID properties remain the same
  - Write on slave: (immediate) synchronization with master
    - Slave has to be up-to-date with master
    - Operation must be performed on both
Neo4j
High Availability

- Each database instance contains the logic needed in order to coordinate with other members
- On startup Neo4j HA database instance will try to connect to an existing cluster specified by configuration
  - If the cluster exists, it becomes a slave
  - Otherwise, it becomes a master

- Failure:
  - Slave – other nodes recognize it
  - Master – a slave is elected as a new master

- Recovery:
  - Slave – synchronizes with the cluster
  - Old master – becomes a slave
References

- Neo4j Download [http://www.neo4j.org/download](http://www.neo4j.org/download)