Neo4j

Mimanshu Maheshwari

October 4, 2024

Contents

List of Figures

Listings

0.1 Setting up Neo4j with docker

Using official docker setup link for neo4j.

```
docker pull neo4j
```

Listing 1: get neo4j image

```
docker run \
   --publish=7474:7474 --publish=7687:7687 \
   --volume=$HOME/neo4j/data:/data \
   neo4j
```

Listing 2: Starting neo4j container

0.2 Creating Nodes

```
CREATE( var: label{key1:value1, key2:value2, key3:value3,...,keyn:valuen})
```

Listing 3: Create Node

- Properties of nodes present between { and }
- Node present between *and*
  1. var is the variable name such as John
  2. label is the entity type such as Customer
  3. key:value are the attributes of the properties such as `contact_num: 1234567890`

Example:

```
CREATE (b:Bank{name:"Oziku"})
```

Listing 4: Create Node Example

0.3 Creating Relationships

```
CREATE (node1) -[var:Rel_type{key1:value1, key2:value2, ...keyn:valuen}] ->(node2)
```

Listing 5: Create Relationships

Example:

```
CREATE (BG:Customer {cust_id: 675489, cust_name: "BlackGreyTechnologies",
    ↪ contact_num: 1298764592 })-
[:Owns]->(AcBG:Account {acc_num: 3498761, cust_id: 675489, type: "checking",
    ↪ balance:958990 })
```

Listing 6: Create Relation Example

Nodes, relationships can be created with/without label, properties Nodes or relationships can have single or multiple labels

0.4 Defining a constraint

Create a unique constraint on `cust_id` property of the Customer node as shown below:

```
CREATE CONSTRAINT ON (c:Customer) ASSERT c.cust_id IS UNIQUE
```

Listing 7: constraint

0.5 MATCH

Is used to search for a pattern. To return all the nodes created:

```
MATCH (n) RETURN n
```

Listing 8: Match syntax

```
MATCH (n:Bank)
RETURN n
```

Listing 9: Match example

```
MATCH (Customer {cust_name: 'BlackGreyTechnologies' })--(Account)
RETURN Account.acc_num, Account.balance
```

Listing 10: Match example

```
MATCH (city:City {name:"Raleigh"})
MERGE (state:State{name:"North Carolina"})
MERGE (city)-[:LOCATED_IN]->(state)
RETURN city,state
```

Listing 11: Match example

0.6 Optional Match

```
MATCH (c:Customer {cust_id: 675489 })
OPTIONAL MATCH (c)-[r:having] -()
RETURN c.cust_name, r.name
```

0.7 WHERE

```
MATCH (c:Customer), (a:Account)
WHERE a.type='checking'
RETURN c.cust_name,a.acc_num,a.balance
```

0.8 Merge

```
MERGE (c:Customer {cust_name: 'Charlie Sheen' })
RETURN c
```

0.9 UNION, UNION ALL

Union combines the results of multiple queries and removes duplicates whereas Union All performs the same operation but retains duplicates.

```
MATCH (c:Customer), (a:Account)
WHERE a.type='checking'
RETURN c.cust_name,a.acc_num,a.balance
UNION
MATCH (c:Customer {cust_name: 'BlackGreyTechnologies' })--(a:Account)
RETURN c.cust_name, a.acc_num, a.balance
```

NOTE: All sub queries must have the same column names.

0.10 ORDER BY, SKIP, LIMIT, DISTINCT

ORDER BY - specifies how the output of RETURN or WITH should be sorted. SKIP - defines from which record to start including the records in the output. LIMIT - constraints the number of output records. DISTINCT - retrieves only unique rows. The below query sorts the Customer node in the ascending order of its customer names, skips the first record and limits the output to only one record:

```
ORDER BY, SKIP, LIMIT, DISTINCT
```

To retrieve the unique relationships present in the database:

```
MATCH (n)-[r]-()
RETURN distinct type(r)
```

0.11 SET

It is used to update node labels and properties of nodes and relationships. For example, adding new properties such as email and country to the existing Customer BlackGreyTechnologies:

```
MATCH (c:Customer {cust_name: 'BlackGreyTechnologies'})
SET c.email= 'BGT@blackgrey.com', c.country ='France'
RETURN c
```

Listing 12: Match example

0.12 FOREACH

It updates data within a list which can be components of a path* or result of an aggregation Path is a directed sequence of nodes and relationships. Assume you want to track funds transfer between two accounts suspected to be laundering funds illegally between intermediary accounts.

```
CREATE p =(:Account {acc_num:65178})-[:Funds_transfer]->(:Account
{acc_num:98567})-[:Fundstransfer]->(:Account{acc_num:46378})-[:Fundstransfer]->(:
    ↪ Account
{acc_num:95648})-[:Fundstransfer]->(:Account{acc_num:46897})
RETURN p

MATCH p =(:Account{acc_num:65178})-[*]->(:Account{acc_num:46897})
FOREACH (n IN nodes(p)| SET n.marked ="flaggedFraud")
```

Note: In the above code, [*] is used to define any number of intermediary relationships `acc_num=65178 to acc_num=46897` In the output shown below, you can observe that a new property `marked:"flaggedFraud"` has been added:

0.13 REMOVE

Is used to remove labels and properties of nodes and relationships.

To remove the email property from Customer having `cust_name=BlackGreyTechnologies`:

```
MATCH (c:Customer {cust_name: 'BlackGreyTechnologies'})
REMOVE c.email
RETURN c
```

0.14 DELETE

Is used to delete nodes, relationships or paths. Node cannot be deleted without deleting its associated relationships. Either delete relationships explicitly or use DETACH DELETE that is discussed next.

```
MATCH (n {acc_num: 65178 })-[r:Funds_transfer]->()
DELETE r
```

0.15 DETACH DELETE

Is used to delete nodes along with their relationships.

To delete Customer node with `name:"BlackGreyTechnologies"` and all its associated links:

```
MATCH (a {cust_name: "BlackGreyTechnologies"})
DETACH DELETE a
```

Delete all the nodes and relationships from the database:

```
MATCH (n)
OPTIONAL MATCH (n)-[r]-()
DELETE n,r
```

0.16 Bulk Load of Data

Step 1: To load data from csv file, the below configuration property needs to be added to the neo4j.conf configuration file: `dbms.security.allow_csv_import_from_file_urls=true`

Step 2: Create Bank node, Customer node and ensure that the customer ID is unique as discussed previously.

Step 3: Load data from Customer.csv file to Customer node

```
load csv with headers from 'file:///C:/Users/Sahana_Basavaraja/Desktop/Customer.csv
    ↪ ' as cust
merge (c:Customer{cust_id:toInteger(cust.cid),cust_name:cust.cname,contact_num:
    ↪ toInteger(cust.phnum)})
```

Note: By default, the row retrieved from the file is always string, for example cust. You need to explicitly convert to the appropriate datatypes if required, for example `toInteger()` as shown above. Establish the relationship between Customer and Bank as `Customer_of`:

```
MATCH (c:Customer),(b:Bank) merge (c)-[b1:Customer_of]->(b)
MATCH (n) RETURN n
CREATE CONSTRAINT ON (a:Account) ASSERT a.acc_num IS UNIQUE

// load data present in account csv
load csv with headers from 'file:///C:/Users/Sahana_Basavaraja/Desktop/Account.csv'
    ↪ as acc
merge(a:Account{acc_num:toInteger(acc.acc_num)})
set
a.cust_id=toInteger(acc.cid),
a.balance=toInteger(acc.balance)
```

Note: Use SET to map columns of the file to properties of the node as illustrated above. Establish the relationship between Customer and Account as Owns:

```
load csv with headers from 'file:///C:/Users/Sahana_Basavaraja/Desktop/Account.csv'
    ↪ as acc
```

```
match(a:Account{acc_num:toInteger(acc.acc_num)}),(c:Customer{cust_id:toInteger(acc.cid)})
merge (a)<-[o:Owns]-(c)
```

```
load csv with headers from 'file:///C:/Users/Sahana_Basavaraja/Desktop/txn.csv' as txn
match (a:Account), (b:Account)
where a.acc_num=toInteger(txn.from_acc) AND b.acc_num=toInteger(txn.to_acc) AND toInteger(a.balance)>=toInteger(txn.amount_acc)
create (a)-[:Funds_transfer{txn_id:toInteger(txn.tid),from_acc:toInteger(txn.from_acc),to_acc:toInteger(txn.to_acc),amount:toInteger
(txn.amount_acc),txntime:txn.txntime}]->(b)
```

0.17 Functions

0.17.1 toLower()

```
MATCH (c:Customer) RETURN toLower(c.cust_name) LIMIT 1
```

0.17.2 toUpper()

```
MATCH (c:Customer) RETURN toUpper(c.cust_name) LIMIT 1
```

0.17.3 substring()

```
MATCH (c:Customer) RETURN SUBSTRING(c.cust_name,0,5) LIMIT 1
```

0.17.4 replase()

```
MATCH (c:Customer{cust_name:"BlackGreyTechnologies"}) RETURN replace("BlackGreyTechnologies","Grey","Gradient")
```

0.17.5 reverse()

```
MATCH (c:Customer) RETURN reverse(c.cust_name) LIMIT 1
```

0.17.6 Aggregate function

It is similar to GROUP BY clause in SQL. It takes multiple values as arguments, performs computation on it and returns the computed value.

- sum() - returns the sum of a set of values
- avg() - returns the average of a set of values

- count() - returns number of rows or values
- max() - returns maximum value in set of values
- min() - returns minimum value in set of values

The frauds do not transfer amount in one single transaction. Multiple transactions would take place in order to hide the transit of illegally obtained money.

The below query retrieves the details of the account number having highest `total_amount` transferred in a series that is greater than 20K:

```
MATCH (a)-[r:Funds_transfer]->(b)
WITH r.to_acc AS account_number,
count(r.to_acc) as total_txn_to_acc,
sum(r.amount) as total_amount
WHERE total_amount >20000
RETURN account_number,total_amount,total_txn_to_acc
ORDER BY total_amount DESC
LIMIT 1
```

0.17.7 List functions

It returns a list of nodes and relationships in a path, labels, keys.

- keys() - returns a list of property names of a node, relationship or map
- labels() - returns a list of all the labels of a node
- nodes() - returns a list of all nodes in a path
- relationships() - returns a list of all relationships in a path

To retrieve the list of distinct labels and its respective property keys:

```
MATCH (n)
RETURN labels(n), keys(n)
```

0.18 User defined functions(UDF) in Neo4J

Neo4J has many built-in functions. In order to extend the functionalities of Neo4J, you can create your own UDFs in Java.

0.18.1 UDFs in Neo4J

- are read-only and always returns a single-value
- is annotated with `@UserFunction`
- valid input types and output types are `string, long, double, boolean, node, relationship, path, object, map<K(string),V>, list`
- is called as `package-name.method-name`

From the Account node, retrieve the average balance amount.

```java
package com.infy.bda;
import java.util.List;
import org.neo4j.procedure.Description;
import org.neo4j.procedure.Name;
import org.neo4j.procedure.UserFunction;
public class AverageBalance {
   @UserFunction
   @Description("com.infy.bda.AverageBalance([0.5,1,2.3]) returns the average of the given list
       ↪ of values")
   public double avg(@Name("numbers") List<Number> list) {
      double avg = 0;
      for (Number number : list) {
         avg += number.doubleValue();
      }
      return (avg/(double)list.size());
   }
}
```

0.18.2 Deploying UDF

UDFs are packaged in a jar file. You have to copy the jar file into the plugins directory of your Neo4J server and restart.

0.18.3 Calling a UDF

Before calling the UDF, to see the list of UDFs deployed in your server, use the following command:

```
CALL dbms.functions()
```

User-defined functions are called in the same way as any other Cypher function. The function name must be fully qualified, that is, a function named avg defined in the package `com.infy.bda` is called using:

```
MATCH(a:Account)
RETURN com.infy.bda.avg(collect(a.balance))
```

0.19 Optional Schema Indexes and constraints

0.19.1 Analyze query performance using EXPLAIN

Query performance can be analysed using the EXPLAIN clause. It provides the execution plan for the Cypher query. Consider the below query that retrieves the Customer node having `cust_name:"George Clooney"`:

```
EXPLAIN
MATCH(c:Customer{cust_name:"George Clooney"})
RETURN c
```

Few properties generated from the output of EXPLAIN command are explained below:

- NodeByLabelScan: Scanning the nodes based on a specific label
- Filter: Conditional expression

0.19.2 Improve performance using indexes

Create Index:

```
CREATE INDEX ON :Label(property) //Index on single property
CREATE INDEX ON :Label(property1, property2, ...,propertyn) //Index on multiple
    ↪ properties
```

Example:

```
CREATE INDEX ON :Customer(cust_name)
```

View Index:

```
CALL db.indexes()
```

Drop Index:

```
DROP INDEX ON :Label(property)
DROP INDEX ON :Label(property1, property2, .....propertyn)
```

Example:

```
DROP INDEX ON :Customer(cust_name)
```

0.20 Working with constraints

Data integrity in Neo4J can be enforced using constraints. Two types of constraints can be created:

- Unique node property constraint
- Property and relationship existence constraint[1]

0.20.1 Unique node property constraint

It ensures that property values for all nodes corresponding to a particular label are unique.

Creation of unique node constraint

Earlier in the course, you have created a unique constraint on the `cust_id` property of the Customer node as shown below:

```
CREATE CONSTRAINT ON (c:Customer) ASSERT c.cust_id IS UNIQUE
```

NOTE:

- Nodes without the property used in the constraint will have no effect
- Creation of unique property constraint will also create index on that property

[1] Available only in the Neo4J Enterprise Edition.

Drop unique constraint:

Use DROP CONSTRAINT clause to remove a constraint from the database as shown below:

```
DROP CONSTRAINT ON (c:Customer) ASSERT c.cust_id IS UNIQUE
```

Note: Dropping the constraint will also remove the index created.