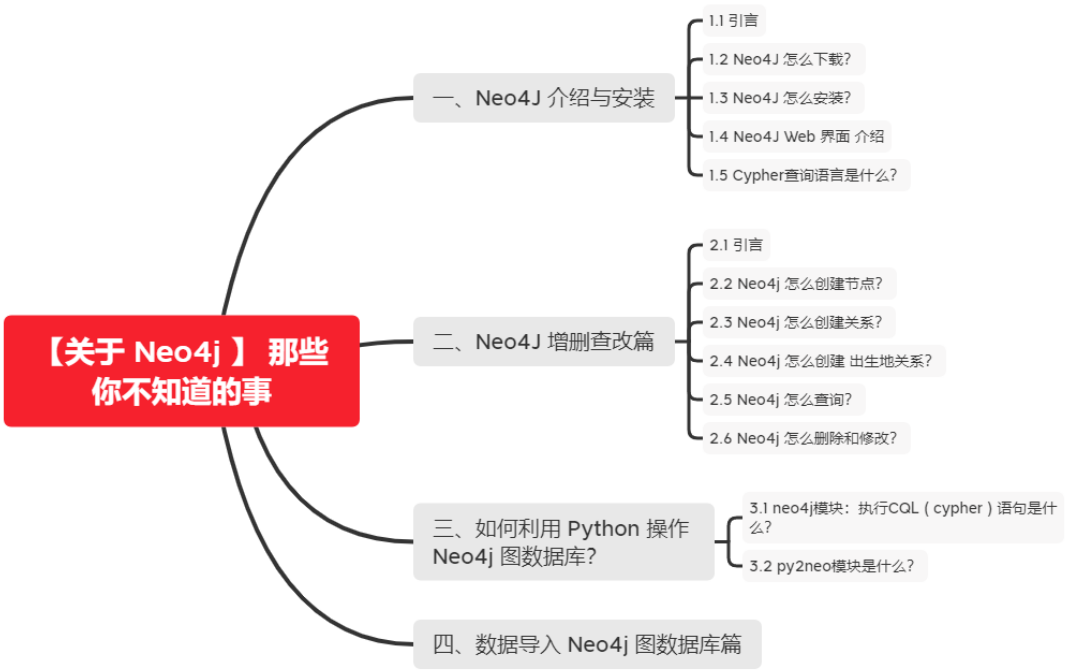
【关于 Neo4j 】 那些你不知道的事



一、Neo4J 介绍与安装

1.1 引言

“工欲善其事，必先利其器”，知识图谱作为一种特殊的图结构，自然需要专门的图数据库进行存储。

知识图谱由于其数据包含实体、属性、关系等，常见的关系型数据库诸如MySQL之类不能很好的体现数据的这些特点，因此知识图谱数据的存储一般是采用图数据库（Graph Databases）。而Neo4j是其中最为常见的图数据库。

1.2 Neo4J 怎么下载?

首先在 Neo4j官网 下载 Neo4J。

- Neo4j分为社区版和企业版：
  - 企业版：收费，在横向扩展、权限控制、运行性能、HA等方面都比社区版好，适合正式的生产环境；
  - 社区版：**免费**，普通的学习和开发采用免费社区版就好。

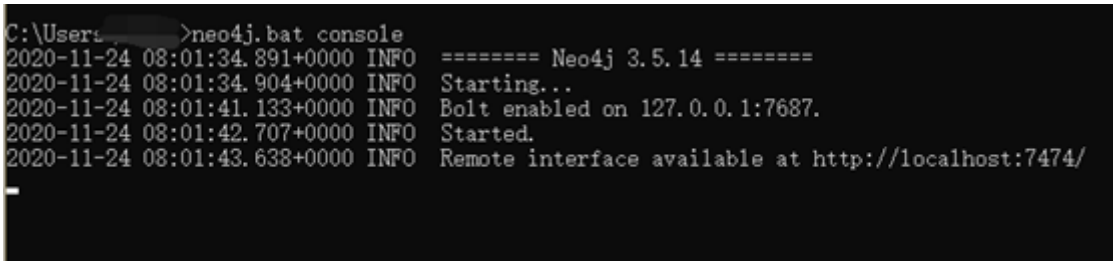
1.3 Neo4J 怎么安装?

- 在Mac或者Linux中，安装好jdk后，直接解压下载好的Neo4j包，运行命令

```
bin/neo4j start
```

- windows系统下载好neo4j和jdk 1.8.0后，输入以下命令启动后neo4j

```
neo4j.bat console
```

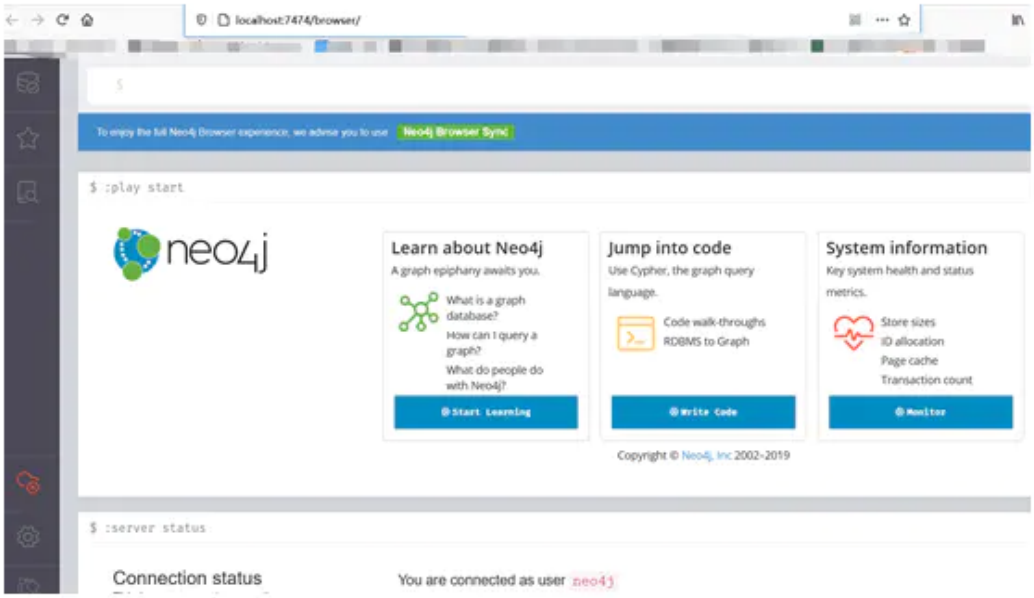


> 图 12 Neo4j 运行结果

1.4 Neo4J Web 界面 介绍

Neo4j 提供了一个用户友好的 Web 界面，可以进行各项配置、写入、查询等操作，并且提供了可视化功能。类似ElasticSearch一样，我个人非常喜欢这种开箱即用的设计。

打开浏览器，输入http://127.0.0.1:7474/browser/，如下图 13 所示，界面最上方就是交互的输入框。



> 图 13 Neo4J Web界面

1.5 Cypher查询语言是什么?

- Cypher：
  - 介绍：是Neo4j的声明式图形查询语言，允许用户不必编写图形结构的遍历代码，就可以对图形数据进行高效的查询。
  - 设计目的：类似SQL，适合于开发者以及在数据库上做点对点模式（ad-hoc）查询的专业操作人员。
  - 其具备的能力包括：
    - 创建、更新、删除节点和关系
    - 通过模式匹配来查询和修改节点和关系 - 管理索引和约束等

二、Neo4J 增删查改篇

2.1 引言

这个案例的节点主要包括人物和城市两类，人物和人物之间有朋友、夫妻等关系，人物和城市之间有出生地的关系。

- Person-Friends-PERSON
- Person-Married-PERSON
- Person-Born_in-Location

2.2 Neo4j 怎么创建节点?

1. 删除数据库中以往的图，确保一个空白的环境进行操作【注：慎用，如果库内有重要信息的话】：

> 图 14 Neo4j 删库操作

```
MATCH (n) DETACH DELETE n
```

这里，MATCH是匹配操作，而小括号()代表一个节点node（可理解为括号类似一个圆形），括号里面的n为标识符。

1. 创建一个人物节点：

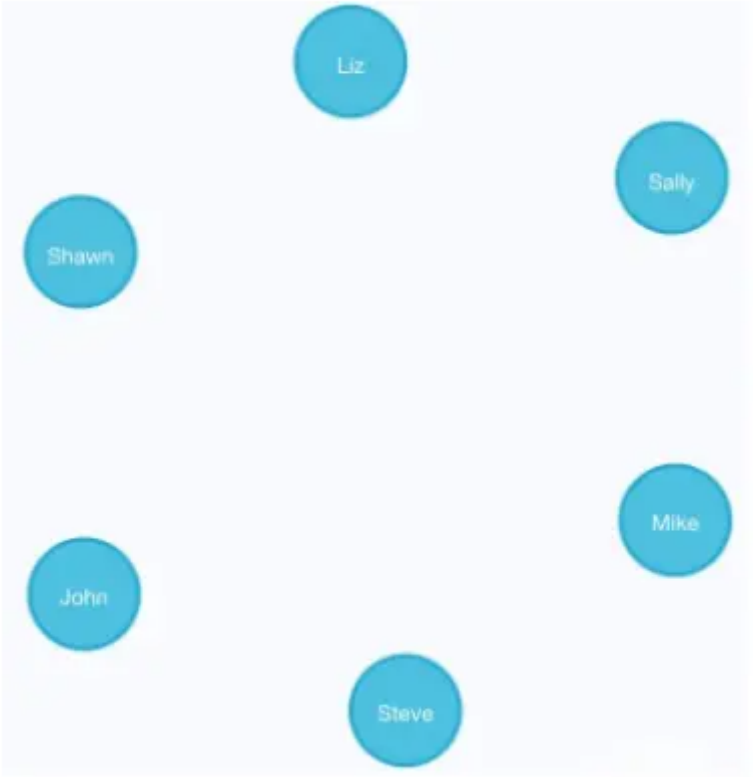
```
CREATE (n:Person {name:'John'}) RETURN n
```

> 注：
>
> CREATE是创建操作，Person是标签，代表节点的类型。
>
> 花括号{}代表节点的属性，属性类似Python的字典。
>
> 这条语句的含义就是创建一个标签为Person的节点，该节点具有一个name属性，属性值是John。

3. 创建更多的人物节点，并分别命名：

```
CREATE (n:Person {name:'Sally'}) RETURN n
CREATE (n:Person {name:'Steve'}) RETURN n
CREATE (n:Person {name:'Mike'}) RETURN n
CREATE (n:Person {name:'Liz'}) RETURN n
CREATE (n:Person {name:'Shawn'}) RETURN n
```

如图 15 所示，6个人物节点创建成功



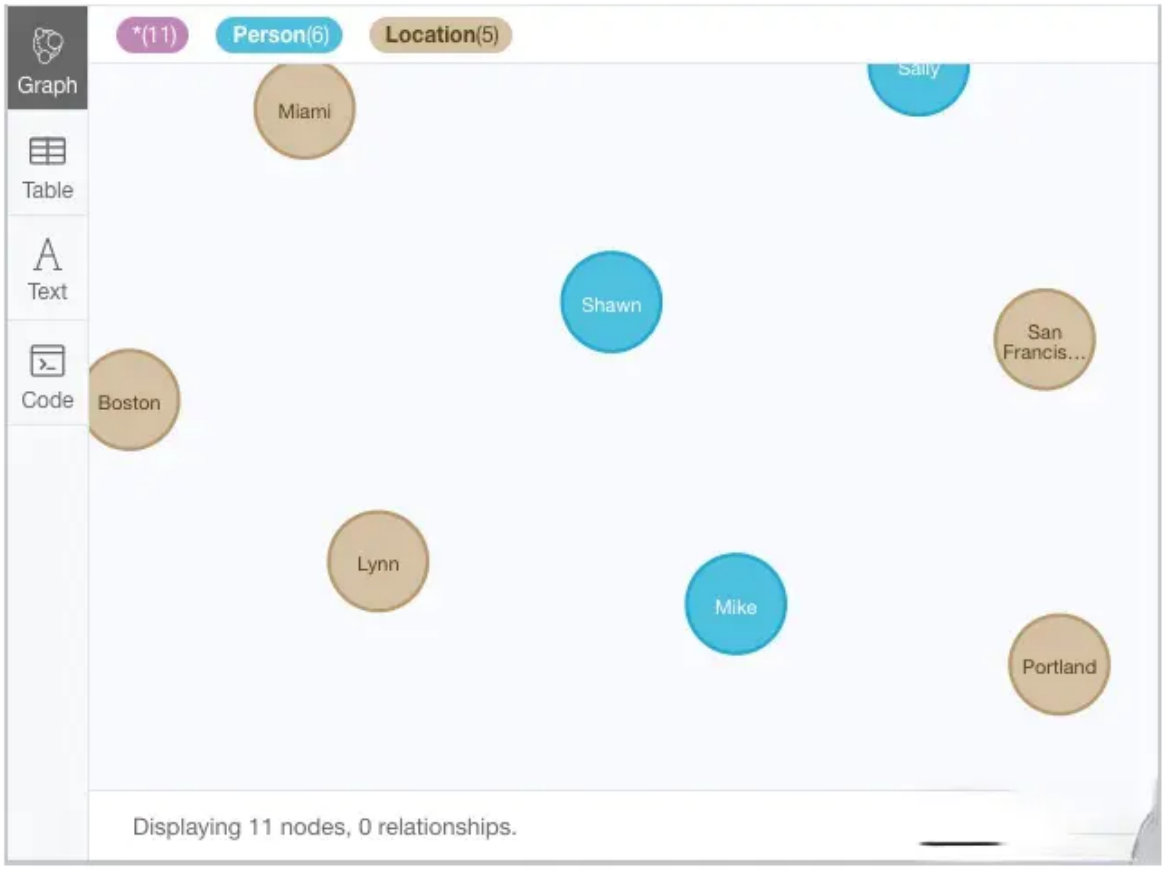
> 图 15 创建 人物节点

1. 创建地区节点

```
CREATE (n:Location {city:'Miami', state:'FL'})
CREATE (n:Location {city:'Boston', state:'MA'})
CREATE (n:Location {city:'Lynn', state:'MA'})
CREATE (n:Location {city:'Portland', state:'ME'})
CREATE (n:Location {city:'San Francisco', state:'CA'})
```

可以看到，节点类型为Location，属性包括city和state。

如图 16 所示，共有6个人物节点、5个地区节点，Neo4j贴心地使用不用的颜色来表示不同类型的节点。



> 图 16 创建地区节点

2.3 Neo4j 怎么创建关系?

1. 朋友关系

```
MATCH (a:Person {name:'Liz'}),
      (b:Person {name:'Mike'})
MERGE (a)-[:FRIENDS]->(b)
```

> 注:
>
> 方括号[]即为关系，FRIENDS为关系的类型。
>
> 注意这里的箭头-->是有方向的，表示是从a到b的关系。 这样，Liz和Mike之间建立了FRIENDS关系。

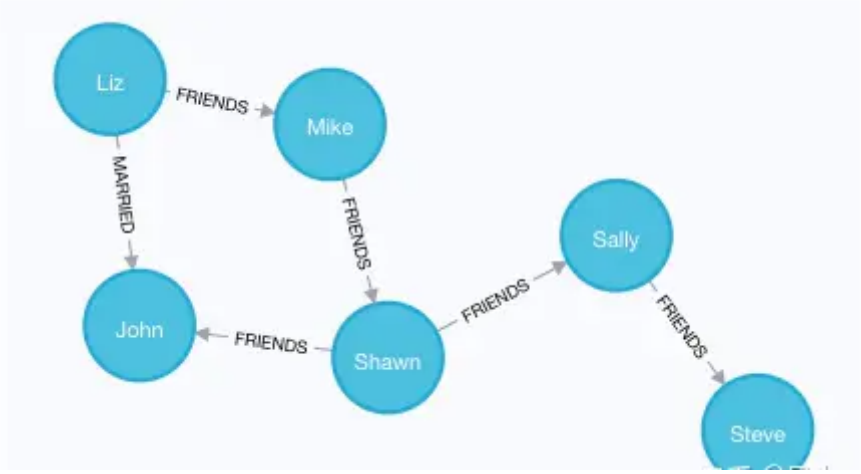
2. 关系增加属性

```
MATCH (a:Person {name:'Shawn'}),
      (b:Person {name:'Sally'})
MERGE (a)-[:FRIENDS {since:2001}]->(b)
```

3. 增加更多的朋友关系:

```
  MATCH (a:Person {name:'Shawn'}), (b:Person {name:'John'}) MERGE (a)-[:FRIENDS
{since:2012}]->(b)
  MATCH (a:Person {name:'Mike'}), (b:Person {name:'Shawn'}) MERGE (a)-[:FRIENDS
{since:2006}]->(b)
  MATCH (a:Person {name:'Sally'}), (b:Person {name:'Steve'}) MERGE (a)-[:FRIENDS
{since:2006}]->(b)
  MATCH (a:Person {name:'Liz'}), (b:Person {name:'John'}) MERGE (a)-[:MARRIED
{since:1998}]->(b)
```

这样，图谱就已经建立好了：



> 图 17 图谱

2.4 Neo4j 怎么创建 出生地关系?

1. 建立不同类型节点之间的关系-人物和地点的关系

```
  MATCH (a:Person {name:'John'}), (b:Location {city:'Boston'}) MERGE (a)-
[:BORN_IN {year:1978}]->(b)
  MATCH (a:Person {name:'Liz'}), (b:Location {city:'Boston'}) MERGE (a)-[:BORN_IN
{year:1981}]->(b)
  MATCH (a:Person {name:'Mike'}), (b:Location {city:'San Francisco'}) MERGE (a)-
[:BORN_IN {year:1960}]->(b)
  MATCH (a:Person {name:'Shawn'}), (b:Location {city:'Miami'}) MERGE (a)-
[:BORN_IN {year:1960}]->(b)
  MATCH (a:Person {name:'Steve'}), (b:Location {city:'Lynn'}) MERGE (a)-[:BORN_IN
{year:1970}]->(b)
```

这里的关系是BORN_IN，表示出生地，同样有一个属性，表示出生年份。

如图 18 ，在人物节点和地区节点之间，人物出生地关系已建立好。

2. 创建节点的时候就建好关系

```
  CREATE (a:Person {name:'Todd'})-[r:FRIENDS]->(b:Person {name:'Carlos'})
```

最终该图谱如下图所示：

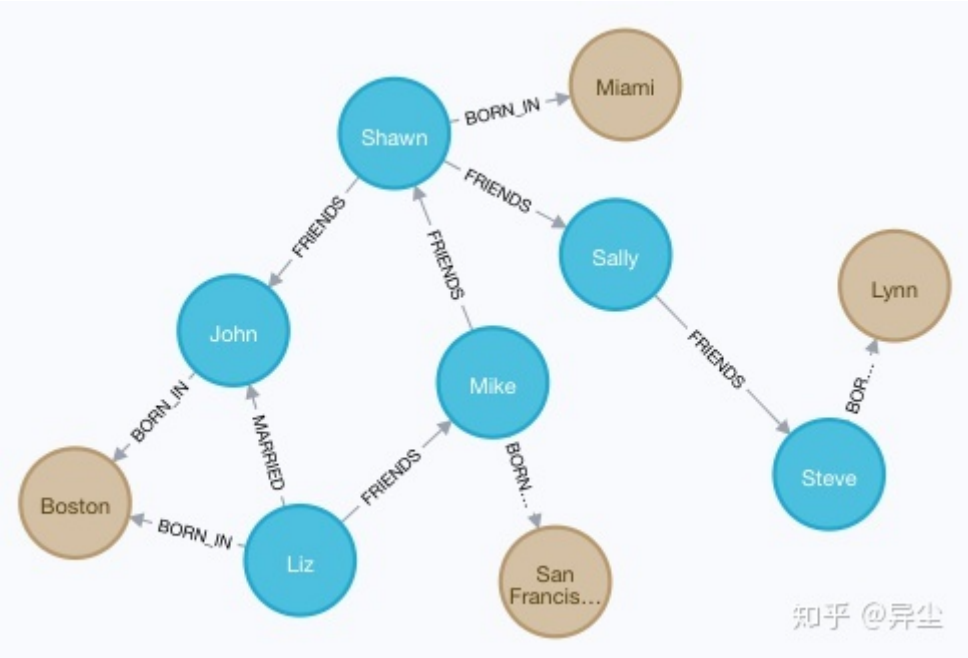


图 18 图谱

2.5 Neo4j 怎么查询?

1. 查询下所有在Boston出生的人物

```
MATCH (a:Person)-[:BORN_IN]->(b:Location {city:'Boston'}) RETURN a,b
```

结果如图 19：

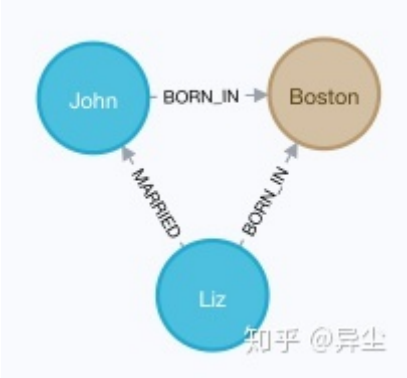


图 19 查询下所有在Boston出生的人物

1. 查询所有对外有关系的节点

```
MATCH (a)--() RETURN a
```

结果如图 20：

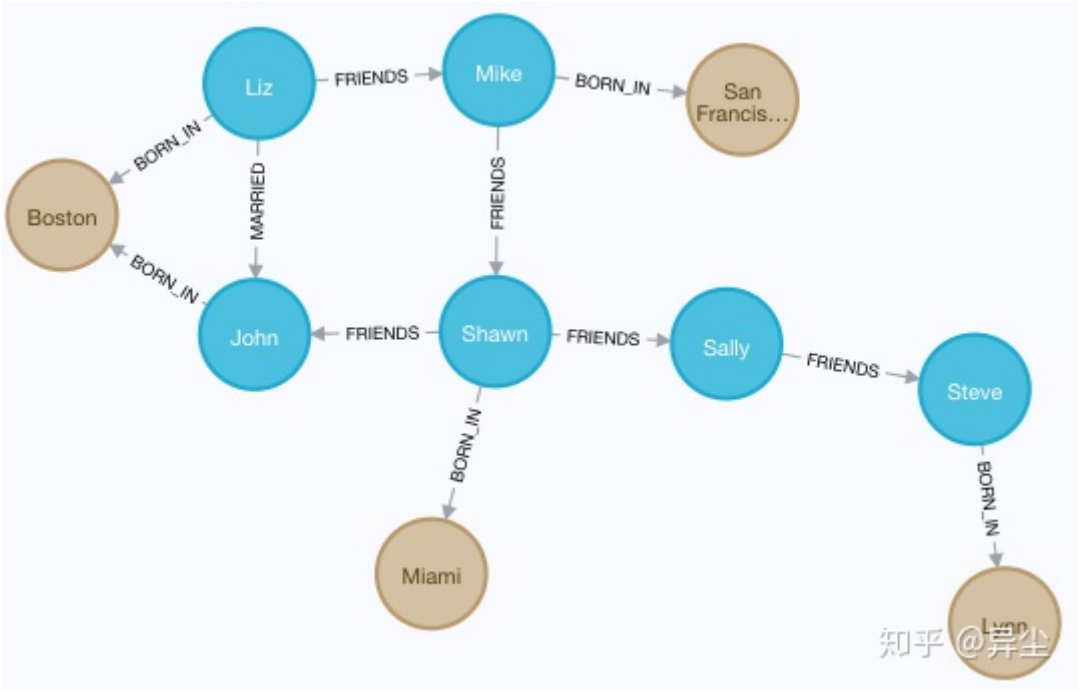


图 20 查询所有对外有关系的节点

1. 查询所有有关系的节点

```
MATCH (a)-[r]->() RETURN a.name, type(r)
```

结果如图21：

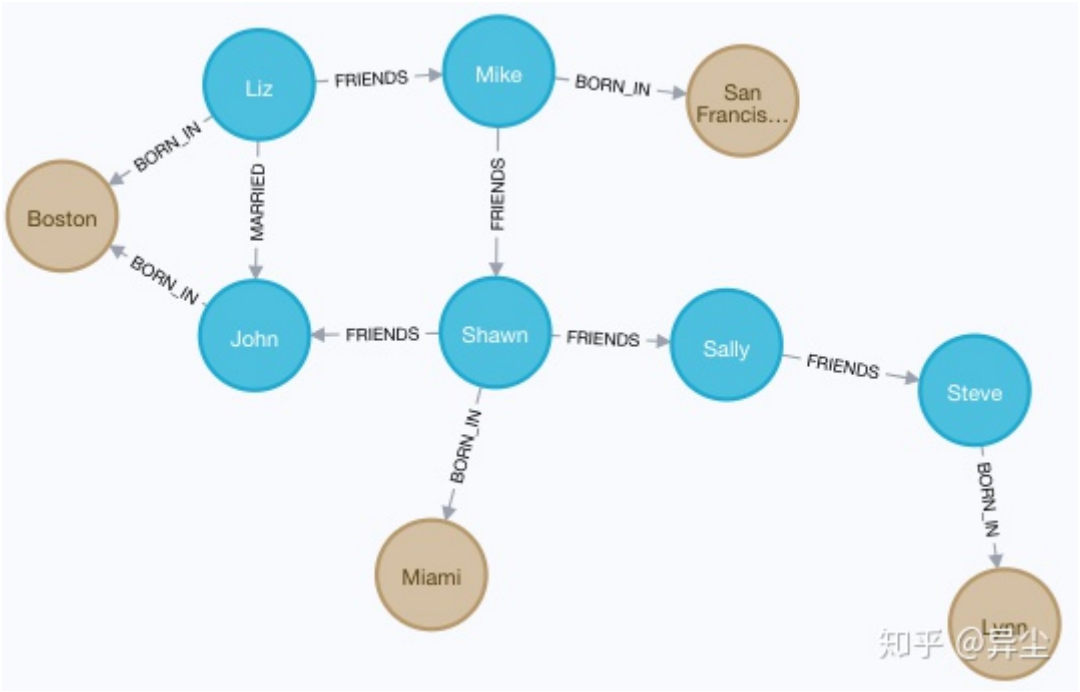


图 21 查询所有有关系的节点

1. 查询所有对外有关系的节点，以及关系类型

```
MATCH (a)-[r]->() RETURN a.name, type(r)
```

结果如图22：

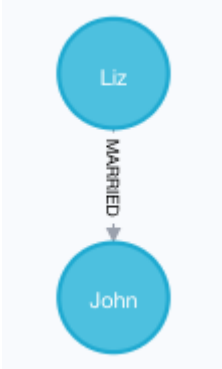
a.name	type(r)
"Steve"	"BORN_IN"
"Mike"	"BORN_IN"
"Mike"	"FRIENDS"
"Sally"	"FRIENDS"
"Liz"	"BORN_IN"
"Liz"	"MARRIED"
"Liz"	"FRIENDS"
"Shawn"	"BORN_IN"
"Shawn"	"FRIENDS"
"Shawn"	"FRIENDS"
"John"	[illegible]

> 图 22 查询所有对外有关系的节点，以及关系类型

1. 查询所有有结婚关系的节点

```
MATCH (n)-[:MARRIED]-() RETURN n
```

结果如图 23：



> 图 23 查询所有有结婚关系的节点

1. 查找某人的朋友的朋友

```
MATCH (a:Person {name:'Mike'})-[r1:FRIENDS]-()-[r2:FRIENDS]-
(friend_of_a_friend) RETURN friend_of_a_friend.name AS fofName
```

返回Mike的朋友的朋友，结果如图 24：

fofName
"John"
"Sally"

> 图 24 查找某人的朋友的朋友

2.6 Neo4j 怎么删除和修改?

1. 增加/修改节点的属性

```
MATCH (a:Person {name:'Liz'}) SET a.age=34
MATCH (a:Person {name:'Shawn'}) SET a.age=32
MATCH (a:Person {name:'John'}) SET a.age=44
MATCH (a:Person {name:'Mike'}) SET a.age=25
```

这里，SET表示修改操作

2. 删除节点的属性

```
MATCH (a:Person {name:'Mike'}) SET a.test='test'
MATCH (a:Person {name:'Mike'}) REMOVE a.test
```

删除属性操作主要通过REMOVE

3. 删除节点

```
MATCH (a:Location {city:'Portland'}) DELETE a
```

删除节点操作是DELETE

4. 删除有关系的节点

```
MATCH (a:Person {name:'Todd'})-[rel]-(b:Person) DELETE a,b,rel
```

三、如何利用 Python 操作 Neo4j 图数据库?

3.1 neo4j模块：执行CQL (cypher) 语句是什么?

```
# step 1: 导入 Neo4j 驱动包
from neo4j import GraphDatabase
# step 2: 连接 Neo4j 图数据库
driver = GraphDatabase.driver("bolt://localhost:7687", auth=("neo4j",
"password"))
# 添加 关系 函数
def add_friend(tx, name, friend_name):
    tx.run("MERGE (a:Person {name: $name}) "
           "MERGE (a)-[:KNOWS]->(friend:Person {name: $friend_name})",
           name=name, friend_name=friend_name)
# 定义 关系函数
def print_friends(tx, name):
    for record in tx.run("MATCH (a:Person)-[:KNOWS]->(friend) WHERE a.name =
$name "
                         "RETURN friend.name ORDER BY friend.name", name=name):
        print(record["friend.name"])
# step 3: 运行
with driver.session() as session:
    session.write_transaction(add_friend, "Arthur", "Guinevere")
    session.write_transaction(add_friend, "Arthur", "Lancelot")
    session.write_transaction(add_friend, "Arthur", "Merlin")
```

```
        session.read_transaction(print_friends, "Arthur")
```

上述程序的核心部分，抽象一下就是：

```
  neo4j.GraphDatabase.driver(xxxx).session().write_transaction(函数(含tx.run(CQL语句)))
```

或者

```
  neo4j.GraphDatabase.driver(xxxx).session().begin_transaction.run(CQL语句)
```

3.2 py2neo模块是什么?

- 介绍：通过操作python变量，达到操作neo4j的目的

```
  # step 1：导包
  from py2neo import Graph, Node, Relationship
  # step 2：构建图
  g = Graph()
  # step 3：创建节点
  tx = g.begin()
  a = Node("Person", name="Alice")
  tx.create(a)
  b = Node("Person", name="Bob")
  # step 4：创建边
  ab = Relationship(a, "KNOWS", b)
  # step 5：运行
  tx.create(ab)
  tx.commit()
```

py2neo模块符合python的习惯，写着感觉顺畅，其实可以完全不会CQL也能写

四、数据导入 Neo4j 图数据库篇

- 动机：前面学习的是单个创建节点，不适合大批量导入。这里我们使用neo4j-admin import命令导入，其他导入方法也可以参考Neo4j之导入数据

csv分为两个nodes.csv和relations.csv，注意关系里的起始节点必须是在nodes.csv里能找到的：

```
  # nodes.csv需要指定唯一ID和nam,
  headers = [
  'unique_id:ID', # 图数据库中节点存储的唯一标识
  'name', # 节点展示的名称
  'node_type:LABEL', # 节点的类型，比如Person和Location
  'property' # 节点的其他属性
  ]
```

```
  # relations.csv
  headers = [
  'unique_id', # 图数据库中关系存储的唯一标识
  'begin_node_id:START_ID', # begin_node和end_node的值来自于nodes.csv中节点
  'end_node_id:END_ID',
  'begin_node_name',
  'end_node_name',
  'begin_node_type',
  'end_node_type',
  'relation_type:TYPE', # 关系的类型，比如Friends和Married
  'property' # 关系的其他属性
  ]
```

制作出两个csv后，通过以下步骤导入neo4j:

1. 两个文件nodes.csv ， relas.csv放在

```
  neo4j安装绝对路径/import
```

2. 导入到图数据库mygraph.db

```
  neo4j bin/neo4j-admin import --nodes=/var/lib/neo4j/import/nodes.csv --
relationships=/var/lib/neo4j/import/relas.csv   --delimiter=^ --
database=xinfang*.db
```

delimiter=^ 指的是csv的分隔符

3. 指定neo4j使用哪个数据库

```
  修改 /root/neo4j/conf/neo4j.conf 文件中的 dbms.default_database=mygraph.db
```

4. 重启neo4j就可以看到数据已经导入成功了

参考资料

1. 干货 | 从零到一学习知识图谱的技术与应用
2. 手把手教你快速入门知识图谱 - Neo4J教程
3. python操作图数据库neo4j的两种方式
4. Neo4j之导入数据
5. schema 介绍
6. 知识图谱Schema
7. 美团大脑：知识图谱的建模方法及其应用
8. 肖仰华. *知识图谱：概念与技术*. 北京：电子工业出版社, 2020．2 - 39．