### Лабораторная работа № 2

Основы работы с Neo4j в браузере

Терминология Neo4j и графовых баз данных в целом.

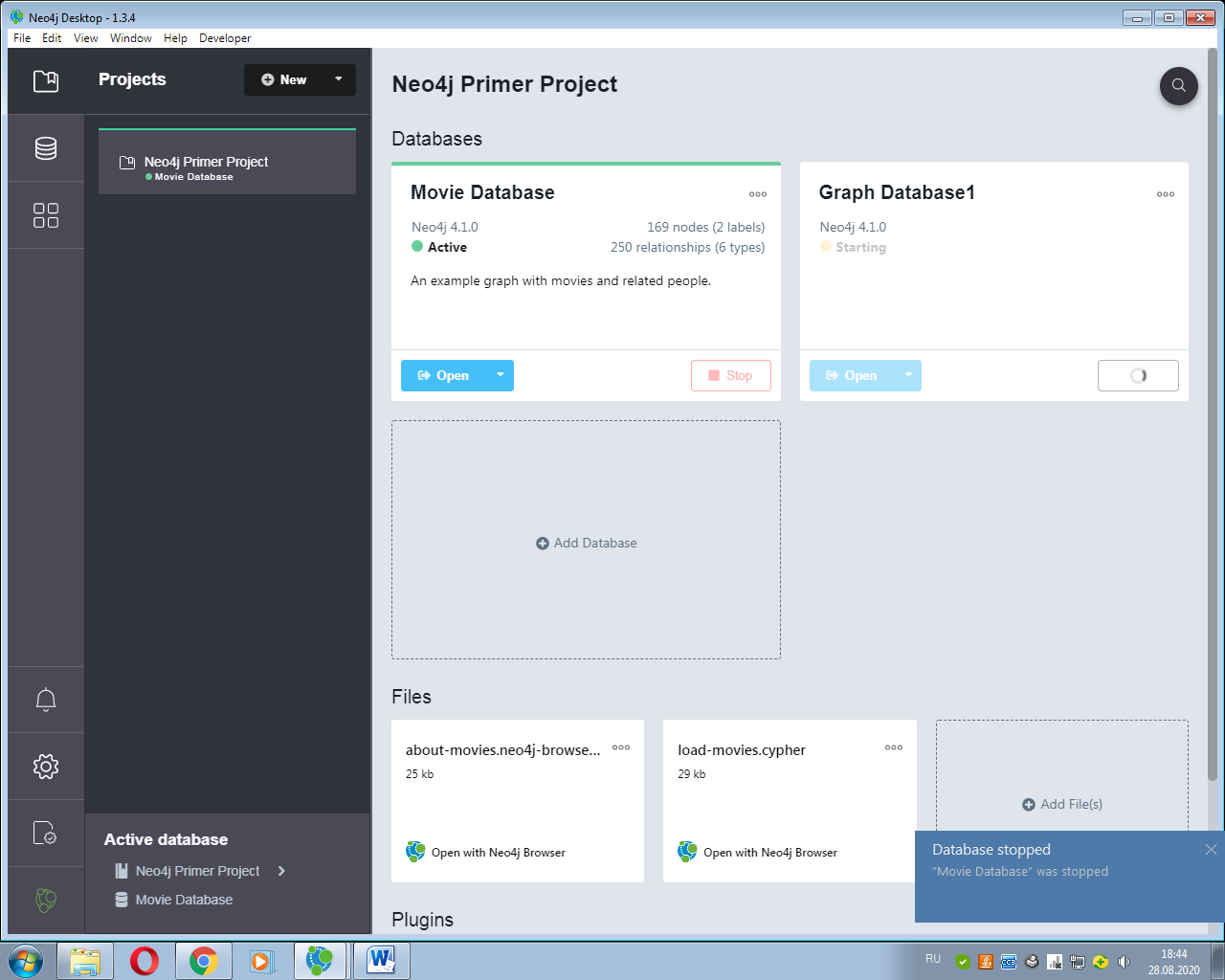
* **graph database**, **[графовая](http://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D1%84_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) база данных** — база данных построенная на графах — узлах и связях между ними
* [**Cypher**](http://www.neo4j.org/learn/cypher) — язык для написания запросов к базе данных Neo4j (примерно, как SQL в Oracle)
* **node**, **нода** — объект в базе данных, узел графа. Количество узлов ограниченно 2 в степени 35 ~ 34 биллиона
* **node label**, **метка ноды** — используется как условный «тип ноды». Например, ноды типа movie могут быть связанны с нодами типа actor. Метки нод **— регистрозависимые**, причем Cypher не выдает ошибок, если набрать не в том регистре название.
* **relation**, **связь** — связь между двумя нодами, ребро графа. Количество связей ограниченно 2 в степени 35 ~ 34 биллиона
* **relation identirfier**, **тип связи** — в Neo4j у связей. Максимальное количество типов связей 32767
* **properties**, **свойства ноды** — набор данных, которые можно назначить ноде. Например, если нода — это товар, то в свойствах ноды можно хранить id товара из базы Oracle
* **node ID**, **ID нода** — уникальный идентификатор ноды. По умолчанию, при просмотрах результата отображается именно этот ID. как его использовать в Cypher запросах
* Cypher - это декларативный язык запросов для neo4j

Основные принципы и возможности Cypher:

* Cypher сопоставляет шаблоны узлов и отношений на графике, чтобы извлечь информацию или изменить данные
* . Cypher имеет концепцию идентификаторов, которые обозначают именованные, связанные элементы и параметры.
* Cypher может создавать, обновлять и удалять узлы, отношения, метки и свойства.
* Cypher управляет индексами и ограничениями.

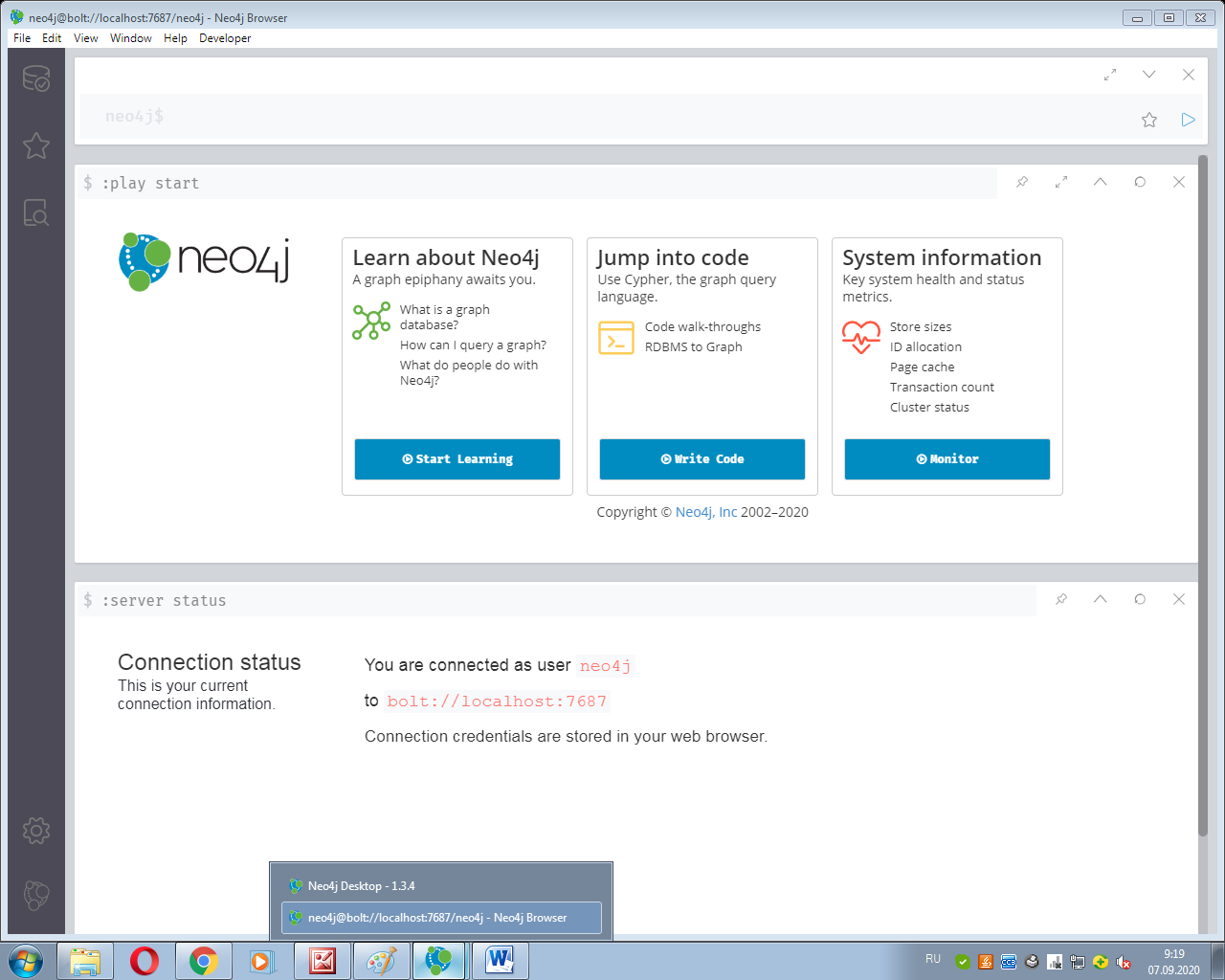
**Задание 1 – дайте команду «Start» БД «Movie Database» (шаблон - An example graph with movies and related people.)**

Познакомьтесь с меню Neo4j Destop



**Задание 2 – дайте команду «Open» БД «Movie Database»**

Познакомьтесь с меню Neo4j Browser



Вершины графа в Neo4j имеют свой тип, в БД **Movie Database** у нас 2 типа вершин:  
\* **Person** (name — имя актера, born — год рождения)  
**\* Movie** (title — наименование фильма, released — год выхода)

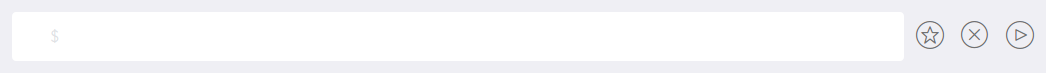
Если проводить аналогию с реляционными базами, то **Person**и **Movie —**это таблицы.

Для работы с БД используется язык запросов **Cypher.**

Cypher — декларативный язык запросов в виде графа, позволяющий получить выразительный и эффективный запрос данных. Язык представлен интуитивно понятным и простым в освоении.

**Редактор**

В верхней части окна Neo4j Browser располагается строка так называемого редактора:



Начиная набор команд с двоеточия, увидим список всех доступных команд с кратким описанием:

**Задание 3 В редакторе наберите команды**

Вызовем команду $ :help нажмите радиокнопку Run

Вызовем команду $ :sysinfo нажмите радиокнопку Run

Вызовем команду $ :play cypher нажмите радиокнопку Run

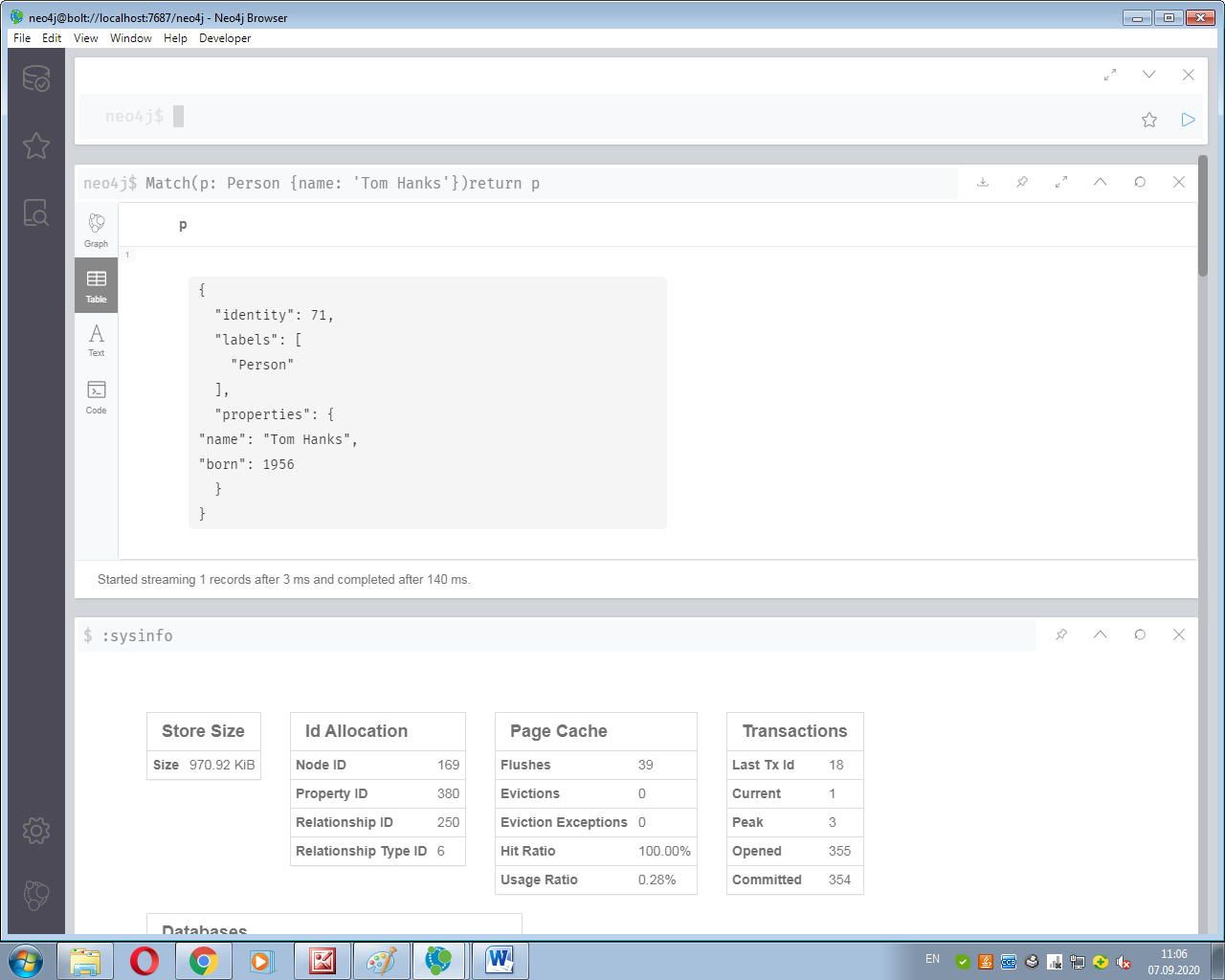
**Задание 4 В редакторе наберите команды (БД Movie Database)**

- Поиск: найдем актера с именем Том Хэнкс

Match(p: Person {name: 'Tom Hanks'})return p

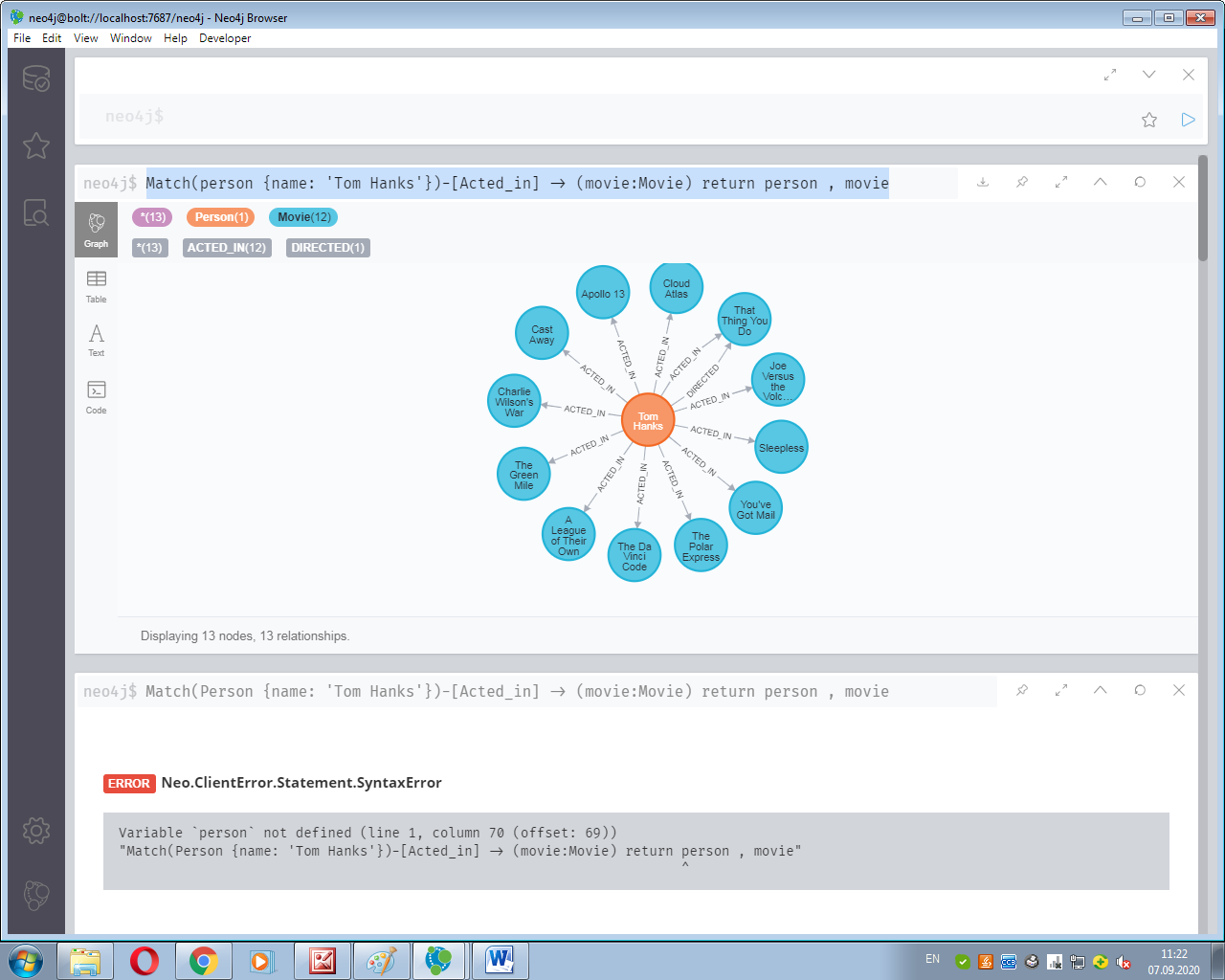
Осуществите навигацию по меню (просмотрите возможности)

* Чтобы указать узел, вы используете круглые скобки: ()
* Чтобы добавить свойства к узлу, используйте скобки: {} скобки



Найдем фильмы, в которых снимался Том Хэнкс:

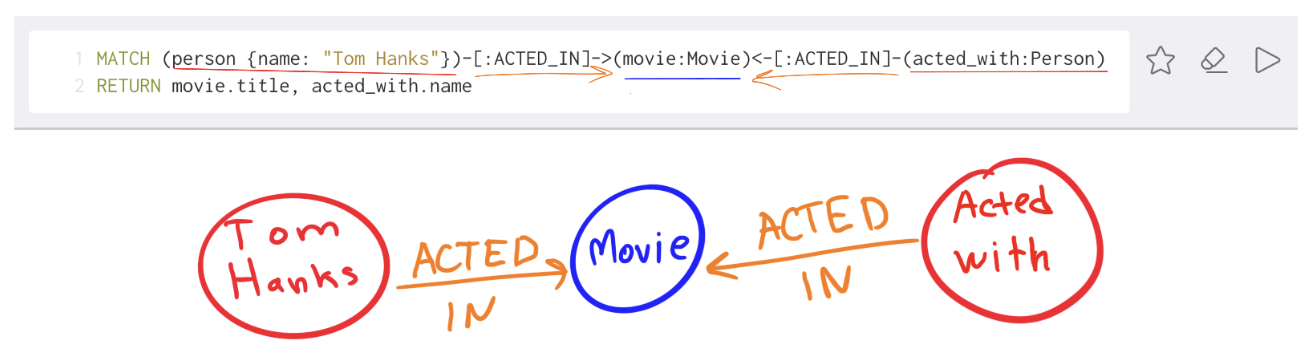
Match(person {name: 'Tom Hanks'})-[:ACTED\_IN] -> (movie:Movie) return person , movie



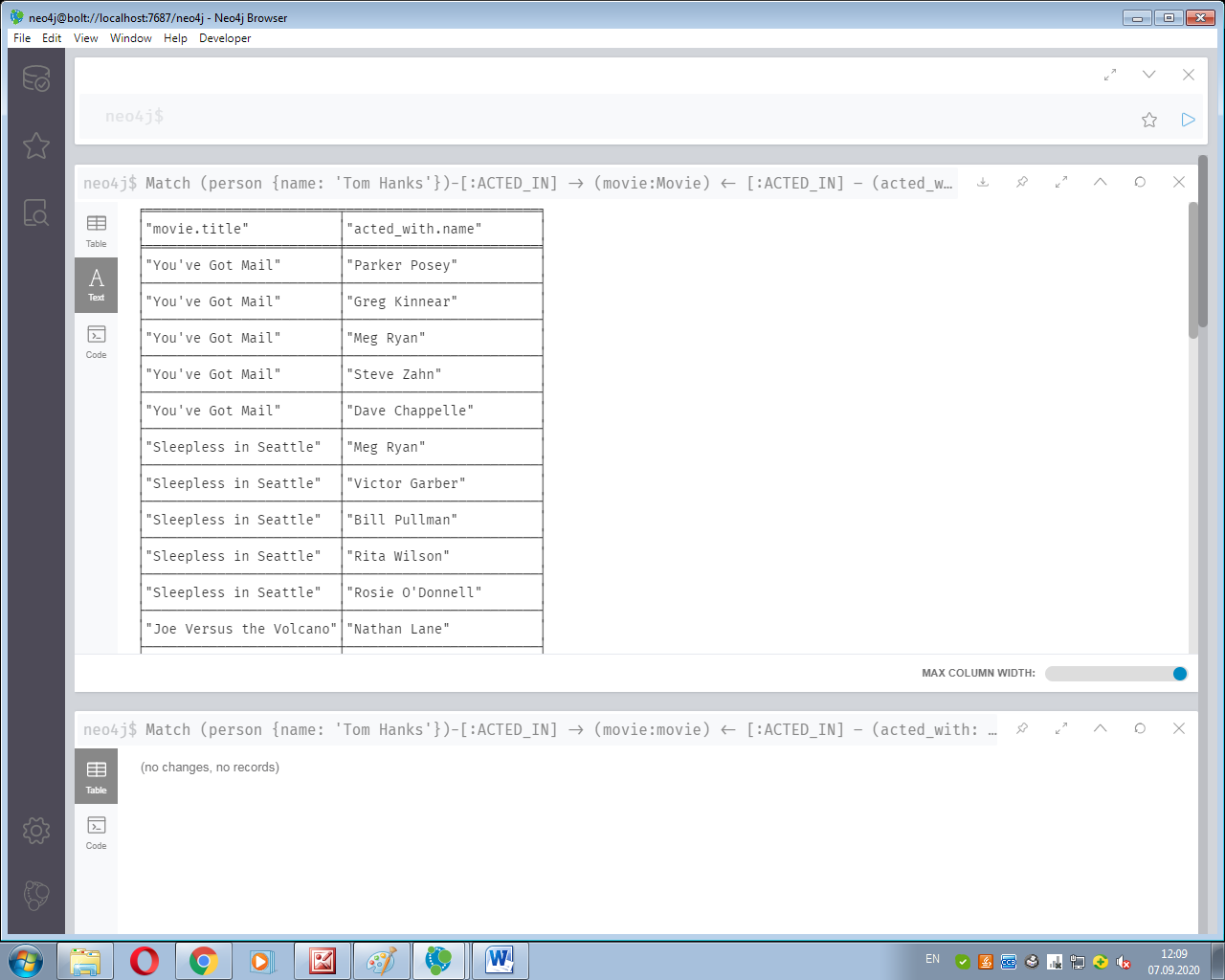
Теперь посмотрим, кто снимался с Томом на одной площадке, и в каком фильме:

Match (person {name: 'Tom Hanks'})-[:ACTED\_IN] -> (movie:Movie) <- [:ACTED\_IN] – (acted\_with: Person) return movie.title, acted\_with.name

Текст запроса приближен к текстовому отображению графа:



От вершины типа Person с именем Tom Hanks по ребру ACTED\_IN находятся все вершины Movie, в которых снимался актер, от вершин типа Movie по связи ACTED\_IN идет поиск всех актеров, снявшихся в каждом найденном фильме соответственно.



**Где и когда использовать графовые СУБД**

Графовые базы уже нашли применение в:

* Социальных сетях
* Системах рекомендаций (с этим товаром часто покупают…)
* Обработка пользовательских данных, корреляция данных из разных источников (информационный след в сети)

Если в вашем приложении планируется много сущностей и связей многие-ко-многим, то это один из признаков, что предпочтительней выбрать графовую СУБД *.* А если у вас уже есть приложение с множеством связей, и обход этих связей занимает много времени и ресурсов — стоит присмотреться в сторону графов, т.к. обход связей в них практически ничего не стоит.

**Выводы**

Графовые базы следует использовать для решения задач, которые решаются только графами.

Например, площадка **medium** использует Neo4j для хранения отношений между различными сущностями ([Стек, который позволил Medium обеспечить чтение на 2.6 тысячелетия](https://habr.com/ru/post/332860/)), а данные хранит в NoSQL БД **dynamoDB**.

Преимущества:

* Графы понятны на интуитивном уровне и являются частью школьной программы, поэтому их можно использовать при обсуждении задач с клиентом;
* Простой язык запросов и наглядный результат, поэтому может использоваться аналитиками;
* Гибкая структура данных, проще вносить изменения в случае изменения требований;
* Возможность создания приближенных к реальной жизни моделей, без низкоуровневых деталей.

Недостатки:

* БД занимает заметно больше места на диске, по отношению к реляционным СУБД;
* На простых запросах производительность ниже, чем в реляционных базах;
* Необходимо обучать разработчиков.

**Задание 5 Самостоятельно создайте дубликат БД** **Movie. Введите в нее свои данные. Выполните выше отработанные запросы.**

+ Add Database

CREATE…

**CREATE (n:МеткаN{ключ1:”значение1”})-[r:LOVES {since: }]->(m:МеткаМ)**

**return n,r,m**

В результате будет созданы вершины **n** и **m** соединенные направленным ребром **r**. Вершины **n** и **m** будут помечены метками **МеткаN** и **МеткаM** соответственно, a ребро **r** – меткой **LOVES**. Кроме того вершине n будет присвоено свойство с ключом – **ключ1** со значением **”значение1”**, а ребру **r** будет присвоено свойство с ключом **since** с пустым значением. После создание будут возвращены в точку вызова значения переменных **n**, **m** и **r**.

MATCH…

**MATCH (n) RETURN n**

Найти все вершины и вернуть их в точку вызова.

**MATCH (n:Person)-[:KNOWS]->(m:Person)**

**WHERE n.name = ”Alice”**

Найти все вершины **n** помеченные меткой **Person** соединенные с вершинами **m** помеченные меткой **Person** ребром с меткой **KNOWS**, причем у вершины **n** должно быть свойство с ключом **name** и со значением **”Alice”**.

DELETE…

**DELETE n, r**

Удалить вершины и ребра связанные с переменными n и r.

**DETACH DELETE n**

Удалить вершины, связанные с переменной n. Кроме того автоматически будут удалены ребра, ведущие в данную вершину.

**MATCH (n)**

**DETACH DELETE n**

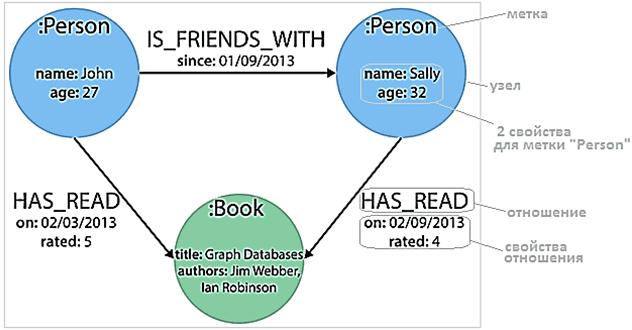
Полное удаление базы данных. Будут удалены все вершины и ребра с ними связанные.

Граф Neo4j состоит из четырех компонентов **[2]**:

* ***Узлы (Nodes)*** – вершины графа, которые представляют объект или сущности. Их можно промаркировать.
* ***Отношения (Relationships)*** – связи между любыми двумя узлами. Отношения имеют тип и направление. Типы обеспечивают общую категорию для каждого отношения. Отношения однонаправленны и специфичны: каждый узел может иметь много отношений с другими узлами, чтобы полностью описать контекст.
* ***Метки***или***ярлыки (Labels)*,**которые маркируют категорию узлов;
* ***Свойства (Properties)***– конкретная информация каждого узла и отношения.

В качестве примера рассмотрим граф из 3-х узлов, каждый из которых промаркирован меткой с 2-мя свойствами. 2 промаркированных отношения, связывающие эти 3 узла («IS\_FRINEDS\_WITH» и «HAS\_READ»), также имеют свойства, которые дают дополнительную информацию о контексте.

Пример графа в Neo4j



Cypher — это простой язык запросов, оптимизированный для графов, который определяет и использует отношения данных. Он исследует взаимосвязи во всех направлениях, чтобы обнаружить ранее невидимые отношения и кластеры. Хотя команды Cypher немного похожи на SQL , в некоторых случаях они отличаются. Например, инструкция **CREATE (n: Person)** создаст узел с меткой «Человек» (Person). Если нужно обратиться к этому узлу позже, можно сохранить его в переменной, назвав ее по своему усмотрению, и ссылаться на нее по мере необходимости.

В отличие от SQL-оператора SELECT, который используется для просмотра данных в реляционных СУБД, в Cypher применяется инструкция MATCH, которая сопоставит и вернет все узлы в базе данных. А RETURN указывает, какой результат мы хотим от нашего запроса. Это могут быть узлы, отношения или свойства. Так что это требуется только во время процедуры чтения. Обратите внимание, что здесь необходимо указать переменную, иначе запрос ничего не вернет.

Например, Cypher-код

MATCH (n: Person {name: “Sally”})

RETURN (n)

вернет все узлы с меткой «Человек» с именем Салли. А если вместо возвращения всего узла, нужны его отдельные свойства, их следует указать в инструкции RETURN, например, RETURN (n.name).

Также, подобно SQL, можно использовать условие WHERE для фильтрации запросов, например:

MATCH (n: Person)

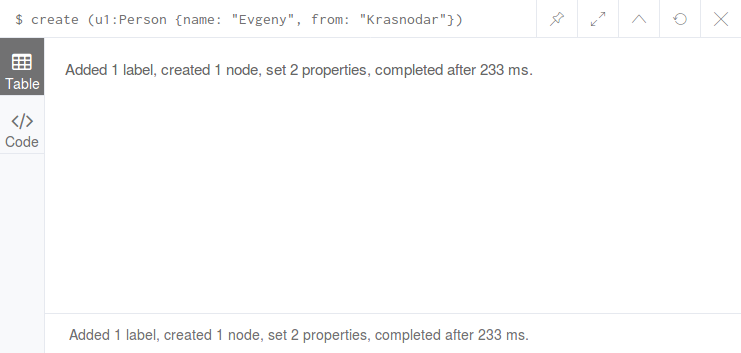
WHERE n.age>=18 AND n.age<=60

RETURN (n)

НАПРИМЕР: Создадим небольшой социальный граф. Перейдём в редактор и наберём первую команду на языке Cypher:

CREATE (u1:Person {name: "Evgeny", from: "Krasnodar"})

После выполнения команды Browser сообщит нам результат:



Добавлена 1 метка , создан 1 узел (), задано 2 свойства{}

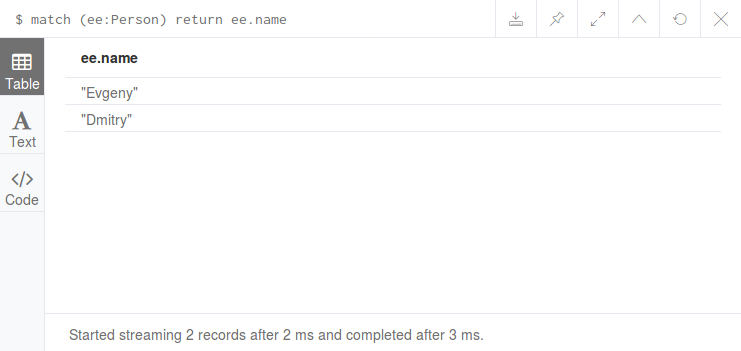
|  |  |
| --- | --- |
| Обозначение в языке Cypher | Описание |
| () | Узел. Сущность или объект. Изображается парой круглых скобок, внутри которых задаются тип вершины и/или значения свойств, например, (a:User) - вершина типа User. |
| [] | Рёбрo изображается парой квадратных скобок, внутри них могут задаваться условия. |
| : | Метка. Тип узла или ребра. Например, [:FRIEND] - связь типа FRIEND. |
| -> или <- или - | Направление связи. Например, ()-[]->(). Если можно не указывать ограничения на связи, опускаются прямые скобки и знаки направления () - -(). |
| {} | Свойство узла или ребра. Например, (mike{name:’Mike’}). |

Добавим ещё один узел:

CREATE (u2:Person {name: "Dmitry", from: "Tula"})

Теперь запросим все узлы типа Person и извлечём значения свойства name:

MATCH (ee:Person) RETURN ee.name



*Примечание.* Как и в SQL есть возможность упорядочить извлекаемые данные по какому-либо полю:

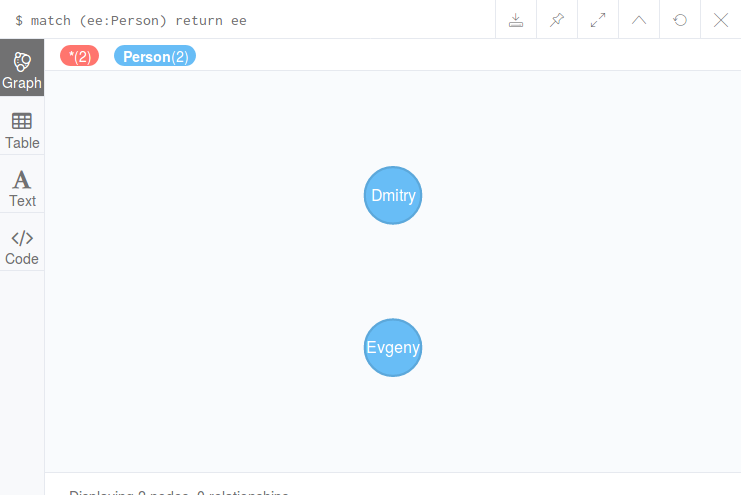
MATCH (ee:Person) RETURN ee.name ORDER BY ee.name

Далее можем запросить все узлы данного типа:

MATCH (ee:Person) RETURN ee



Обратите внимание на появившуюся кнопку **Graph**. Щёлкним на ней и увидим наши узлы в графическом виде:



!!!!! ВНИМАНИЕ Запрос MATCH (ee:person) RETURN ee

НИЧЕГО НЕ ВЫВЕДЕТ – чувствительность к регистру !!!!!

*Примечание.* В версии 3.4 по-умолчанию как-раз открывается графическое представление. Для получения табличного представления нужно щёлкнуть на кнопку с надписью "Table". Хотя, бывает и наоборот.

!!! запрос даст ошибку

MATCH (e:Person) WHERE e.name = "Evgeny"

!!! запрос не даст ошибки (учтена точка возврата)

MATCH (e:Person) WHERE e.name = "Evgeny" return e

!!! запрос не даст ошибки

MATCH (e:Person) WHERE e.name = "Evgeny"

MATCH (d:Person) WHERE d.name = "Dmitry"

Return e,d

Ошибочно введем Евгения второй раз

CREATE (u1:Person {name: "Evgeny", from: "Krasnodar"})

Посмотрим полученный граф

match (n) return n

Удалим дубликат (!!! Пока не созданы связи!!!)

MATCH (e:Person) WHERE e.name = "Evgeny"

Delete e

**Удаление определенного узла**

Чтобы удалить конкретный узел, вам необходимо указать детали узла вместо «n» в приведенном выше запросе.

**Синтаксис**

Ниже приведен синтаксис для удаления определенного узла из Neo4j с помощью предложения DELETE.

MATCH (node:label {properties . . . . . . . . . . })

DETACH DELETE node

**пример**

Прежде чем приступить к примеру, создайте узел «Ishant» в базе данных Neo4j, как показано ниже.

CREATE (Ishant:player {name: "Ishant Sharma", YOB: 1988, POB: "Delhi"})

Ниже приведен пример Cypher Query, который удаляет созданный выше узел с помощью предложения DELETE.

MATCH (Ishant:player {name: "Ishant Sharma", YOB: 1988, POB: "Delhi"})

DETACH DELETE Ishant

Посмотрим полученный граф (!!! Удалены все Евгенеии)

match (n) return n

Удалим все узлы (!!! Пока не созданы связи!!!)

MATCH (n) DELETE n

Посмотрим полученный граф

match (n) return n

ВНИМАНИЕ !!!! Полное удаление базы данных. Будут удалены все вершины и ребра с ними связанные.

MATCH (n)

DETACH DELETE n

Восстановим граф

CREATE (u1:Person {name: "Evgeny", from: "Krasnodar"})

CREATE (u2:Person {name: "Dmitry", from: "Tula"})

Добавим связь между узлами:

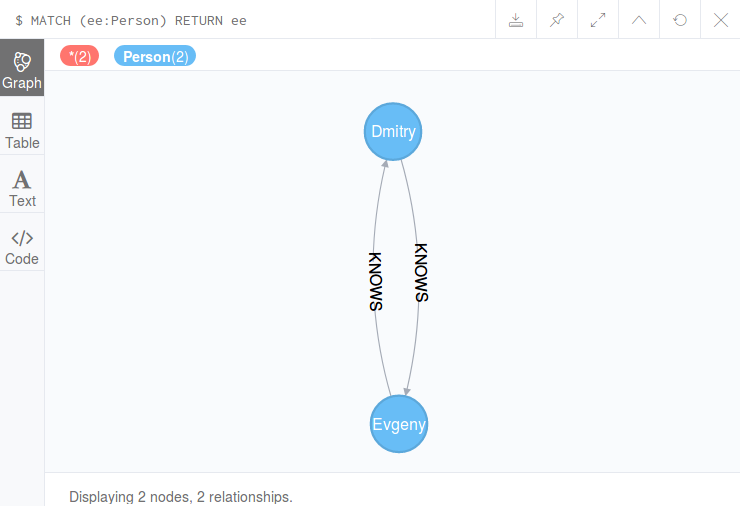
MATCH (e:Person) WHERE e.name = "Evgeny"

MATCH (d:Person) WHERE d.name = "Dmitry"

CREATE (e)-[:KNOWS]->(d),

(d)-[:KNOWS]->(e)

И вновь запросим наш граф:



Попробуем удалить все узлы (!!! Но созданные связи не дают этого сделать!!!)

MATCH (n) DELETE n

Очевидно при помощи DETACH удалить получиться!!!

MATCH (n)

DETACH DELETE n

Восстановим граф

CREATE (u1:Person {name: "Evgeny", from: "Krasnodar"})

CREATE (u2:Person {name: "Dmitry", from: "Tula"})

Восстановим связи

MATCH (e:Person[) WHERE e.name = "Evgeny"

MATCH (d:Person) WHERE d.name = "Dmitry"

CREATE (e)-[:KNOWS]->(d),

(d)-[:KNOWS]->(e)

Посмотрим полученный граф

match (n) return n

Удалим все связи

MATCH ()-[r:KNOWS]-()

DELETE r;

Восстановим связи

MATCH (e:Person) WHERE e.name = "Evgeny"

MATCH (d:Person) WHERE d.name = "Dmitry"

CREATE (e)-[:KNOWS]->(d),

(d)-[:KNOWS]->(e)

Удалим связи (др. подход)

MATCH (e:Person)-[r:KNOWS]-(f:Person) where e.name= "Dmitry"and f.name= "Evgeny"

DELETE r;

Создадим узлы и связи из шаблона (лаб. 1)



CREATE

(TomH)-[:ACTED\_IN {roles:['Zachry', 'Dr. Henry Goose', 'Isaac Sachs', 'Dermot Hoggins']}]->(CloudAtlas),

(Hugo)-[:ACTED\_IN {roles:['Bill Smoke', 'Haskell Moore', 'Tadeusz Kesselring', 'Nurse Noakes', 'Boardman Mephi', 'Old Georgie']}]->(CloudAtlas),

(HalleB)-[:ACTED\_IN {roles:['Luisa Rey', 'Jocasta Ayrs', 'Ovid', 'Meronym']}]->(CloudAtlas),

(JimB)-[:ACTED\_IN {roles:['Vyvyan Ayrs', 'Captain Molyneux', 'Timothy Cavendish']}]->(CloudAtlas),

(TomT)-[:DIRECTED]->(CloudAtlas),

(LillyW)-[:DIRECTED]->(CloudAtlas),

(LanaW)-[:DIRECTED]->(CloudAtlas),

(DavidMitchell)-[:WROTE]->(CloudAtlas),

(StefanArndt)-[:PRODUCED]->(CloudAtlas)

CREATE (TomH:Person {name:'Tom Hanks', born:1956})

CREATE (LanaW:Person {name:'Lana Wachowski', born:1965})

Посмотрим полученный граф

match (n) return n

УНИЧТОЖТЕ!!! ПОМЕНЯЙТЕ ПОСЛЕДОВАТЕЛЬНОСТЬ ВЫПОЛНЕНИЯ !!!

MATCH (n)

DETACH DELETE n

Восстановим связи

MATCH (e:Person) WHERE e.name = "Evgeny"

MATCH (d:Person) WHERE d.name = "Dmitry"

CREATE (e)-[:KNOWS]->(d),

(d)-[:KNOWS]->(e)

Удалим конкретную ОДНУ связь

MATCH (e:Person)-[r:KNOWS]->(f:Person) where e.name= "Dmitry"and f.name= "Evgeny"

DELETE r;

ЗАДАНИЕ: СОЗДАТЬ ГРАФ СВОИХ ДРУЗЕЙ и ЗНАКОМЫХ

С помощью Cypher можно также выполнять различные операции над графами, например, запрашивать смежные вершины, друзей друзей в социальном графе, удалять рёбра и вершины и многое другое.

Также можно настраивать Neo4j Browser на различный стиль отображения узлов и связей в зависимости от заданных им меток.

<https://coderlessons.com/tutorials/bazy-dannykh/uznaite-neo4j/neo4j-uchebnik>

LOAD CSV with headers FROM 'file:///export\_emp.csv' as row

MERGE (EMP:employees {e:row.ENAME,m:row.EMPNO,j:row.JOB,n:row.MGR,h:row.HIREDATE,s:row.SAL,c:row.COMM})

Написание запросов

Разберем простейший синтаксис написания запросов к Neo4j по CRUD.

Чтение данных

Для начала, надо вспомнить что в структуре данных у нас есть:

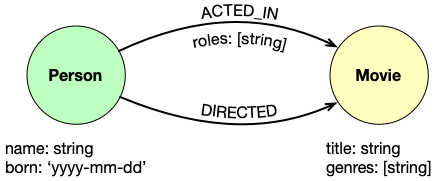
* Nodes — сущности;
* Labels — метки к сущностям;
* Relationships — однонаправленные отношения между сущностями;
* Properties — свойства сущностей или отношений.

Очень упрощенно, основной шаблон запроса для чтения записей можно описать следующим образом:

(<var>:<entity> [{<property>: <value>[, ...]}])<direction>[:<relationship>]<direction>(<var>:<entity>[{<property>: <value>[, ...]}])

Согласен, с примером будет понятнее :)

Вспомним нашу базу данных с фильмами и актерами, и попробуем сделать запрос к ней



// Найти все фильмы

MATCH (m:Movie)

RETURN m

Для получения какой-то выборки используется ключевое слово MATCH (точная копия SELECT из RDBMS).

m:Movie — тут мы присваиваем переменной m все найденные Movies по заданным условиям.

RETURN p — явно указываем *что* должен вернуть наш запрос.

Больше примеров:

// Найти всех актеров с именем Tom Hanks:

MATCH (p:Person {name: 'Tom Hanks'})

RETURN p

{name: 'Tom Hanks'} — условие фильтрации по сущности Person

Но, графовая БД не была бы таковой, если бы в ней не было главной особенности: связей между сущностями.

Давайте найдем все фильмы, в которых играл Tom Hanks:

MATCH (p:Person {name: 'Tom Hanks'})-->(m)

RETURN m.title

m — тут мы присваиваем переменной m все найденные зависимые сущности по заданным условиям.

Отлично! Но неправильно :) Потому что это вернет нам все его связи со всеми зависимыми сущностями по всей базе данных, если таковые будут. Нам это не нужно, мы хотим получить фильмы, в которых он играл. Добавим тип связи ACTED\_IN:

MATCH (p:Person {name: 'Tom Hanks'})-[:ACTED\_IN]->(m)

RETURN m.title

Уже лучше, но тоже неточно. Этот запрос вернет *все* сущности, в которых он играл, не только фильмы. Он же мог играть и в театральных постановках, правильно?

Эти примеры выше работают благодаря так называемомой способности *traverse*, если говорить в терминах Neo4j.

Нода, хоть как-то связанная с другой нодой (даже через другие N нод), соответственно, — *traversable*-нода.

Конкретно в нашей схеме данных других сущностей нет, поэтому получим только Movies. Но лучше не копить техдолг, а явно указать что мы ожидаем только фильмы:

MATCH (p:Person {name: 'Tom Hanks'})-[:ACTED\_IN]->(m:Movie)

RETURN m.title

Теперь отлично!

Больше примеров:

// Найти список всех актеров из "Матрицы"

MATCH (m:Movie {title: 'The Matrix'})<-[:ACTED\_IN]-(p:Person)

RETURN p

// Найти всех режиссеров, родившихся до 1970 года

// Обратите внимание: нам не важно ЧТО именно они режиссировали,

// поэтому тип ноды не указываем:

MATCH (p:Person)-[:DIRECTED]->()

WHERE p.born < 1970

RETURN p

Можно комбинировать связи:

// Найти все фильмы, в которых Tom Hanks

// был одновременно актером ИЛИ режиссеором:

MATCH (m:Movie)<-[:ACTED\_IN|:DIRECTED]-(p:Person {name: 'Tom Hanks'})

RETURN m.title, p.name

В общем случае, ключевое слово WHERE работает почти так-же, как и в RDBMS, поэтому на нем подробно останавливаться не буду. Однако есть интересные применения в контексте проверки связей:

// Найти всех актеров, которые играли в фильмах,

// но не являлись при этом режиссером фильма:

MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE NOT exists( (p)-[:DIRECTED]->(m) )

RETURN p.name, m.title

// Обратиться к атрибуту связи:

// найти имя актера, который играл Нео в Матрице

MATCH (p:Person)-[r:ACTED\_IN]->(m:Movie)

WHERE 'Neo' IN r.roles AND m.title='The Matrix'

RETURN p.name, r.roles

Запись данных

Для добавления данных в Neo4j используется слово MERGE, однако будет большой ошибкой проводить полную аналогию с ключевыми словами CREATE или INSERT из RDBMS.

Дело в том, что MERGE создает *паттерн* в базе данных.

Проще сразу с примерами:

// Создать ноду Person с именем Michael Cain

MERGE (p:Person {name: 'Michael Cain'})

// Создать две ноды (да, это один запрос на три строчки)

MERGE (p:Person {name: 'Katie Holmes'})

MERGE (m:Movie {title: 'The Dark Knight'})

RETURN p, m

Небольшое отступление: можно также использовать ключевое слово CREATE с той лишь разницей, что CREATE не будет проверять по первичному ключу существует ли уже такая нода в БД — это дает ему огромное преимущество в скорости. CREATE чаще всего используют для дампов, чтобы быстро можно было их развернуть на чистой базе. MERGE занимается вставкой данных более аккуратно — с проверкой всех условий, но и более медленно.

Создадим простой паттерн через MERGE:

MATCH (p:Person {name: 'Michael Cain'})

MATCH (m:Movie {title: 'The Dark Knight'})

MERGE (p)-[:ACTED\_IN]->(m)



Ничего не изменилось.

Дело в том, что в этом запросе MATCH говорит нам о том, что мы должны найти записи, а MERGE о том, что мы должны создать связь между ними. Однако, если записи не найдены, то связь и не создастся. Для гарантированного создания сущности или связи можно использовать MERGE для всех трех частей запроса:

MERGE (p:Person {name: 'Michael Cain'})

MERGE (m:Movie {title: 'The Dark Knight'})

MERGE (p)-[:ACTED\_IN]->(m)

// или в одну строку:

MERGE (p:Person {name: 'Michael Cain'})-[:ACTED\_IN]->(m:Movie {title: 'The Dark Knight'})

RETURN p, m



Что тут происходит по логике обработчика запроса:

1. Neo4j пытается найти сущность Person с именем Michael Cain;
2. Если сущность найдена, то Neo4j смотрит все ее связи ACTED\_IN;
3. Затем среди этих связей Neo4j ищет Movie с названием The Dark Knight.

Если хотя-бы на одном из этих этапов не было найдено объектов, то будет создан **весь паттерн**. То есть, если даже фильм The Dark Knight существует, а актер Michael Cain нет, то будет создан новый фильм с таким названием. Это важнейшая особенность, которую стоит учитывать, чтобы избегать дублирования данных.

Попробуем воспроизвести это:

// Выполним трижды запрос на добавление актеров в один и тот-же фильм

MERGE (p:Person {name: 'Yuri Nikulin'})-[:ACTED\_IN]->(m:Movie {title: 'The Diamond Arm'})

RETURN p, m

MERGE (p:Person {name: 'Nina Grebeshkova'})-[:ACTED\_IN]->(m:Movie {title: 'The Diamond Arm'})

RETURN p, m

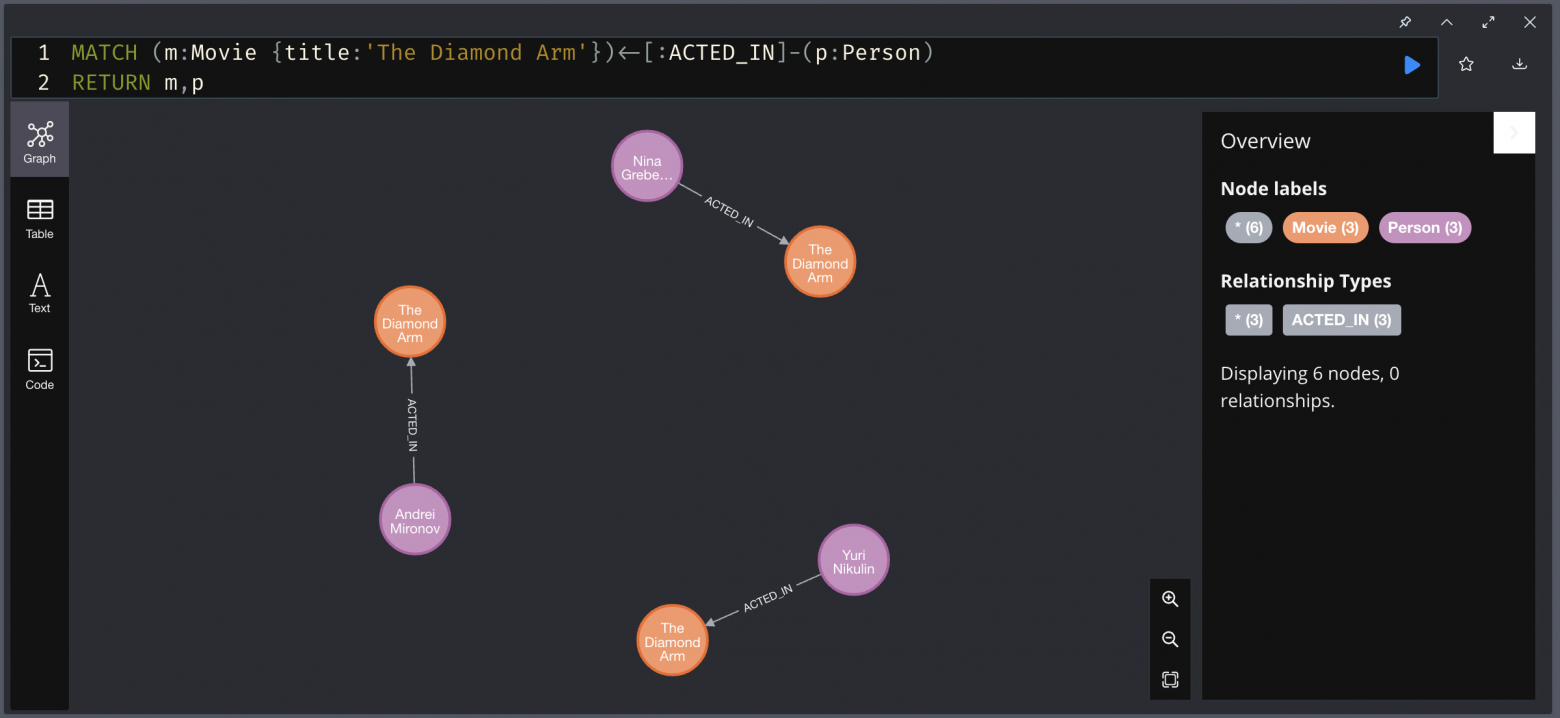
MERGE (p:Person {name: 'Andrei Mironov'})-[:ACTED\_IN]->(m:Movie {title: 'The Diamond Arm'})

RETURN p, m

// Найдем все записи с этим названием фильма

MATCH (m:Movie {title:'The Diamond Arm'})<-[:ACTED\_IN]-(p:Person)

RETURN m,p

Получилось явно не то, что мы хотели

Решением такого кейса может быть создание всех сущностей отдельно и затем отдельно связей между ними. Всё отдельными запросами.

Более сложный пример для создания паттернов:

// В нашей базе Oliver Stone и Rob Reiner никогда не работали вместе,

// то есть у них нет общего фильма.

//

// В этом запросе мы ищем обоих

// и создаем новый отличный безымянный фильм для них :)

MATCH

(oliver:Person {name: 'Oliver Stone'}),

(reiner:Person {name: 'Rob Reiner'})

MERGE (oliver)-[:DIRECTED]->(movie:Movie)<-[:ACTED\_IN]-(reiner)

RETURN movie

Также полезно знать, что для добавления и обновления атрибутов используется ключевое слово SET:

MATCH (p:Person)-[r:ACTED\_IN]->(m:Movie)

WHERE p.name = 'Michael Cain' AND m.title = 'The Dark Knight'

SET r.roles = ['Alfred Penny'], r.year = 2008

RETURN p, r, m

Удаление данных

Это, пожалуй, самый простой раздел, потому что ломать — не строить :)

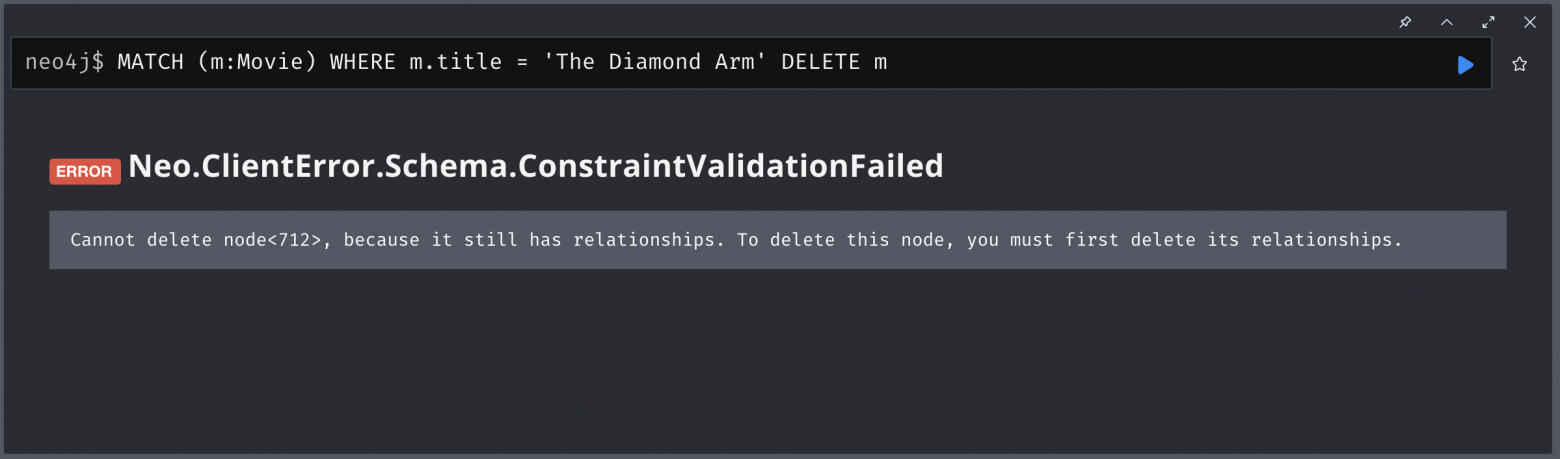
Самое важное знание тут — это то, что нельзя удалить ноду, пока у нее есть хоть одна связь.

Удалим все фильмы с названием The Diamond Arm, которые мы создали ранее:

MATCH (p:Person)

WHERE p.name = 'Jane Doe'

DELETE p

Oops!

Окей, нельзя удалять пока есть связи. Для быстрой починки этого у нас есть ключевая пара слов DETACH DELETE. Но если мы используем такое для удаления фильмов, то никогда потом не найдем актеров из них, которых тоже хотелось бы удалить.

Тогда удалим сначала актеров, потом сами дубликаты фильмов:

// Отцепим (DETACH) и удалим актеров фильмов с названием The Diamond Arm

MATCH (m:Movie {title:'The Diamond Arm'})<-[:ACTED\_IN]-(p:Person)

DETACH DELETE p

// Удалим сами дубликаты фильмов с названием The Diamond Arm.

// Отцеплять (DETACH) уже не нужно, т.к. актеров вместе со связями

// мы удалили только что

MATCH (m:Movie {title:'The Diamond Arm'})

DELETE m

Для удаления атрибутов можно просто присвоить им значение null или явно вызвать ключевое слово REMOVE:

// Вот так

MATCH (p:Person)

WHERE p.name = 'Gene Hackman'

SET p.born = null

RETURN p

// Или так

MATCH (p:Person)-[r:ACTED\_IN]->(m:Movie)

WHERE p.name = 'Michael Cain' AND m.title = 'The Dark Knight'

REMOVE r.roles

RETURN p, r, m

Заключение

Как видно из примеров выше, работать с графовыми БД довольно просто, но в то же время необычно для тех, кто много времени проводит с SQL, нужно немного перестроить мышление.