COMS W4111: Introduction to Databases
Spring 2024, Sections 002/V02

Homework 4

Introduction

- This notebook contains HW4. **Both Programming and Nonprogramming tracks should complete this homework.**
- You will submit **PDF and ZIP files** for this assignment. Gradescope will have two separate assignments for these.
- For the PDF:
  - The most reliable way to save as PDF is to go to your browser's menu bar and click `File -> Print`. Switch the orientation to landscape mode, and hit save.
  - **MAKE SURE ALL YOUR WORK (CODE AND SCREENSHOTS) IS VISIBLE ON THE PDF. YOU WILL NOT GET CREDIT IF ANYTHING IS CUT OFF.** Reach out for troubleshooting.
  - **MAKE SURE YOU DON'T SUBMIT A SINGLE PAGE PDF.** Your PDF should have multiple pages.
- For the ZIP:
  - Zip a folder containing this notebook and any screenshots.
  - You may delete any unnecessary files, such as caches.

Setup

In [1]:
```
%load_ext sql
%sql mysql+pymysql://root:dbuserdbuser@localhost
```

In [2]:
```
import sys

!{sys.executable} -m pip install --upgrade pymongo
!{sys.executable} -m pip install --upgrade neo4j
```

```
Requirement already satisfied: pymongo in /Users/sparshbinjrajka/anaconda3/lib/python3.11/site-p
ackages (4.6.3)
Requirement already satisfied: dnspython<3.0.0,>=1.16.0 in /Users/sparshbinjrajka/anaconda3/lib/
python3.11/site-packages (from pymongo) (2.6.1)
Requirement already satisfied: neo4j in /Users/sparshbinjrajka/anaconda3/lib/python3.11/site-pac
kages (5.19.0)
Requirement already satisfied: pytz in /Users/sparshbinjrajka/anaconda3/lib/python3.11/site-pack
ages (from neo4j) (2023.3.post1)
```

- If you get warnings below, try restarting your kernel

In [3]:
```
import neo4j
import pandas
import pymongo

# TODO: Fill in with your Mongo URL
mongo_url = "mongodb+srv://dbuser:dbuserdbuser@w4111.9h6xmk2.mongodb.net/?retryWrites=true&w=major
mongo_client = pymongo.MongoClient(mongo_url)

# TODO: Fill in with your Neo4j credentials
neo4j_url = "neo4j+s://dfb8d6ea.databases.neo4j.io"
neo4j_password = "gQO3s059io_lyaqpYRQ8nRHuVB8ynsDernycxfcdEKI"
# username is always "neo4j"
graph = neo4j.GraphDatabase.driver(neo4j_url, auth=("neo4j", neo4j_password))
graph.verify_connectivity()
```

Written Questions

- As usual, do not bloviate

W1

Explain the following concepts:

1. Clustering index
2. Nonclustering index
3. Sparse index
4. Dense index

Answer

Ref: https://www.geeksforgeeks.org/indexing-in-databases-set-1/# (https://www.geeksforgeeks.org/indexing-in-databases-set-1/#)

1. Clustering index sorts and stores table based on key-value pairs of the index. There can be at most one clustering index per table since the rows can only be sorted in one way. It has very fast lookup time but insertion/deletion might be costly due to reordering.
2. Nonclustering index has a separate structure that orders the data. Index entries point to the actual data rows, which are stored separately. There can be many nonclustering indices per table but it has slower query time compared to clustering index.
3. Sparse index contains records for some of the rows in the table where each record corresponds to a range of rows. It requires less space but lookup time is more compared to dense indices.
4. Dense index contains a record for each row in the table. Hence, it takes up more space than sparse indices but offers fastest lookups.

W2

Explain why nonclustering indexes must be dense.

Answer

Since the actual data is not stored with the index but separately and not in sorted order, nonclustering index must be dense to ensure all the records are "identifiable" so any query for any row is successful.

W3

Suppose that, in a table containing information about Columbia classes, the columns `class_code` , `semester` , and `year` are queried frequently **individually**. Would putting a composite index on `(class_code, semester, year)` be a good idea? Why or why not?

Answer

That would not be a good idea since the queries are for individual columns and not for the composite. Queries to such an index would only benefit the first column of the composite but not the others and as a result this index would be ineffective. Having separate indices for each of the columns is better even though it will take up more space.

W4

Explain the following concepts:

1. Hash index
2. B+ tree index

Answer

Ref: https://www.sqlpipe.com/blog/b-tree-vs-hash-index-and-when-to-use-them#:~:text=B%2B%20trees%20are%20the%20default,an%20extremely%20performance%20sensitive%20table (https://www.sqlpipe.com/blog/b-tree-vs-hash-index-and-when-to-use-them#:~:text=B%2B%20trees%20are%20the%20default,an%20extremely%20performance%20sensitive%20table).

1. In a Hash index, the "buckets" store entries with pointers to records. It essentially hashes to the location of the row in a table and thus lookup time is O(1). Ideal for exact searching but not suited for range queries/maintaining order since it only supports equality operation.
2. In a B+ Tree index, we use a the B+ tree structure to index the table. It supports a range of operations and is optimal for all sorts of queries while also offering good lookup/insert query time.

W5

Give one advantage and one disadvantage of hash indexes compared to B+ tree indexes.

Answer

1. Adv: O(1) lookup time due to exact match criteria and hash function usage.
2. Disadv: Worse than B+ tree for range-type queries since they don't maintain any order.

W6

Explain the role of the buffer in a DBMS. Why doesn't the DBMS simply load the entire database in its buffer?

Answer

Accessing data in disk is an expensive operation so a buffer holds a chunk of data in the main/primary memory temporarily to speed up read/write operations and reduce disk access. Buffer has a limited space so we cannot load the entire database into it (plus that would inefficient as it is the same as accessing the disk directly). Since most ops do not need the entire memory all the time, buffer loads in specific chunks while they are in use and when they are no longer in use, it copies out the changes into disk and brings in another chunk of memory.

W7

Explain the following concepts as they relate to buffer replacement policies:

1. Clairvoyant algorithm
2. Least recently used strategy
3. Most recently used strategy
4. Clock algorithm

Answer

Ref: [https://www.geeksforgeeks.org/second-chance-or-clock-page-replacement-policy/ (https://www.geeksforgeeks.org/second-chance-or-clock-page-replacement-policy/)](https://www.geeksforgeeks.org/second-chance-or-clock-page-replacement-policy/)

1. A hypothetical algorithm that can look into the future and make the best possible set of decisions. Used as a benchmark for performance evaluation.
2. This strategy replaces the least recently used block of memory in buffer with the assumption that more recently used blocks will be used again.
3. Opposite of LRU strategy where it replaces the most-recently used block under the assumption that the one used recently wont be used again.

4. Also called second-chance algorithm where blocks of memory are treated in a round-robin manner and when it is considered for replacement, it is given a second chance. If the next time it is considered for replacement and it has not yet been looked up, it is replaced. If it was looked up, it is not replaced and will be given another second chance when it is considered for replacement the next time.

W8

NoSQL databases have become increasingly popular for applications. List 3 benefits of using NoSQL databases over SQL ones.

Answer

1. They are flexible since they can work with structured, unstructured, or semi-structured data but SQL only works with a rigid schema (hence structured).
2. They can be scaled out and so are best suited for dynamic traffic handling and distributed architectures but SQL is best for scaling up.
3. Allows for faster and complex queries for semi-structured and unstructured data but SQL needs to first structure the data and then run queries on it which takes more time.

W9

Explain the concept between impedance mismatch and how it relates to SQL vs. NoSQL databases.

Answer

Ref: [https://www.geeksforgeeks.org/impedance-mismatch-in-dbms/ (https://www.geeksforgeeks.org/impedance-mismatch-in-dbms/)](https://www.geeksforgeeks.org/impedance-mismatch-in-dbms/)

Impedance mismatch in general refers to communication problem between two components when they each follow a different structure/schema/model. In DBMS context, it refers to the conflict between the OOP-model used in applications and relational model used in SQL. This is because there is no fixed way to map objects in the program to tables in a database or vice-versa so we need to perform the mapping ourselves (eg: using ORM tools). But, for NoSQL, the flexible structure (eg: document-based, graph structure) easily allow for OOP programs to be mapped to NoSQL databases and vice-versa. This reduces the impedance mismatch in NoSQL as compared to SQL.

W10

The relationship between students and courses is many-to-many. Due to its emphasis on atomicity, modeling this relationship in a relational database would require an associative entity. Explain how this relationship could be modeled in

1. A document database, such as MongoDB
2. A graph database, such as Neo4j

Answer

1. In MongoDB, we can use embedded documents to model many-to-many relationships. That is, for each Student document we can associate it with an array of courses (eg: array stores course IDs) and each Course document can have an array of students (eg: array stores student IDs)
2. In Neo4j, we can use the graph structure directly to model many-to-many relationships. We can have nodes for students and courses and for each student S that takes a course C, we can add an edge from S to C. We can add relationship properties to the edge if required since Neo4j permits that.

MongoDB

- The cell below creates a database `w4111` , then a collection `episodes` inside `w4111` . It then inserts GoT episode data into the collection.

In [4]:
```python
import json

with open("episodes.json") as f:
    data = json.load(f)

episodes = mongo_client["w4111"]["episodes"]
episodes.drop()
episodes.insert_many(data)
print("Successfully inserted episode data")
```

```
Successfully inserted episode data
```

M1

- Write and execute a query that shows episodes and the number of scenes they contain
- Your aggregation should have the following attributes:
  - `episodeTitle`
  - `seasonNum`
  - `episodeNum`
  - `numScenes` , which is the length of the episode's `scenes` array
- Order your output on `numScenes` descending, and only keep episodes with more than 100 scenes

In [5]:
```
res = episodes.aggregate(
 [
    {
        '$project': {
            '_id': 0,
            'episodeTitle': 1,
            'seasonNum': 1,
            'episodeNum': 1,
            'numScenes': {
                '$size': '$scenes'
            }
        }
    }, {
        '$match': {
            'numScenes': {
                '$gt': 100
            }
        }
    }, {
        '$sort': {
            'numScenes': -1
        }
    }, {
        '$project': {
            '_id': 0,
            'episodeTitle': '$episodeTitle',
            'seasonNum': '$seasonNum',
            'episodeNum': '$episodeNum',
            'numScenes': '$numScenes'
        }
    }
]
)

pandas.DataFrame(list(res))
```

Out[5]:

	episodeTitle	seasonNum	episodeNum	numScenes
0	The Long Night	8	3	292
1	The Bells	8	5	220
2	Blackwater	2	9	133
3	The Last of the Starks	8	4	113
4	The Dragon and the Wolf	7	7	104

M2

- Write and execute a query that shows the first three episodes for each season
- Your aggregation should have the following attributes:
  - `seasonNum`
  - `firstThreeEpisodes` , which is an array that contains the titles of the first, second, and third episodes (in that order) of the season
- Order your output on `seasonNum` ascending
  - It's okay if the `firstThreeEpisodes` column is a bit truncated by the dataframe

In [6]:
```python
res = episodes.aggregate(
    [
    {
        '$project': {
            '_id': 0,
            'seasonNum': 1,
            'episodeNum': 1,
            'episodeTitle': 1
        }
    }, {
        '$match': {
            'episodeNum': {
                '$lt': 4
            }
        }
    }, {
        '$group': {
            '_id': '$seasonNum',
            'firstThreeEpisodes': {
                '$push': '$episodeTitle'
            }
        }
    }, {
        '$sort': {
            '_id': 1
        }
    }, {
        '$project': {
            '_id': 0,
            'seasonNum': '$_id',
            'firstThreeEpisodes': 1
        }
    }
]
)

pandas.DataFrame(list(res))
```

Out[6]:

	firstThreeEpisodes	seasonNum
0	[Winter Is Coming, The Kingsroad, Lord Snow]	1
1	[The North Remembers, The Night Lands, What Is...	2
2	[Valar Dohaeris, Dark Wings, Dark Words, Walk ...	3
3	[Two Swords, The Lion and the Rose, Breaker of...	4
4	[The Wars to Come, The House of Black and Whit...	5
5	[The Red Woman, Home, Oathbreaker]	6
6	[Dragonstone, Stormborn, The Queen's Justice]	7
7	[Winterfell, A Knight of the Seven Kingdoms, T...	8

M3

- Write and execute a query that shows statistics about each season
- Your aggregation should have the following attributes:
    - `seasonNum`
    - `numEpisodes` , which is the number of episodes in the season
    - `startDate` , which is the earliest air date associated with an episode in the season
    - `endDate` , which is the latest air date associated with an episode in the season
    - `shortestEpisodeLength`
    - `longestEpisodeLength`
        - The length of an episode is the greatest `sceneEnd` value in the episode's `scenes` array
- Order your output on `seasonNum` ascending

```
In [7]: res = episodes.aggregate(
    [
     {
         '$unwind': {
             'path': '$scenes'
         }
     }, {
         '$addFields': {
             'sceneLength': {
                 '$sum': [
                     {
                         '$multiply': [
                             {
                                 '$toInt': {
                                     '$arrayElemAt': [
                                         {
                                             '$split': [
                                                 '$scenes.sceneEnd', ':'
                                             ]
                                         }, 0
                                     ]
                                 }
                             }, 3600
                         ]
                     }, {
                         '$multiply': [
                             {
                                 '$toInt': {
                                     '$arrayElemAt': [
                                         {
                                             '$split': [
                                                 '$scenes.sceneEnd', ':'
                                             ]
                                         }, 1
                                     ]
                                 }
                             }, 60
                         ]
                     }, {
                         '$toInt': {
                             '$arrayElemAt': [
                                 {
                                     '$split': [
```

```
                                        '$scenes.sceneEnd', ':'
                                    ]
                                }, 2
                            ]
                        }
                    }
                ]
            }
        }
    }, {
        '$group': {
            '_id': {
                'seasonNum': '$seasonNum',
                'episodeNum': '$episodeNum',
                'episodeAirDate': '$episodeAirDate'
            },
            'episodeLength': {
                '$max': '$sceneLength'
            }
        }
    }, {
        '$group': {
            '_id': '$_id.seasonNum',
            'numEpisodes': {
                '$sum': 1
            },
            'startDate': {
                '$min': '$_id.episodeAirDate'
            },
            'endDate': {
                '$max': '$_id.episodeAirDate'
            },
            'shortE': {
                '$min': '$episodeLength'
            },
            'longE': {
                '$max': '$episodeLength'
            }
        }
    }, {
        '$addFields': {
            'shortHr': {
                '$floor': {
```

```
                '$divide': [
                    '$shortE', 3600
                ]
            }
        },
        'shortMin': {
            '$floor': {
                '$divide': [
                    {
                        '$mod': [
                            '$shortE', 3600
                        ]
                    }, 60
                ]
            }
        },
        'shortSec': {
            '$mod': [
                '$shortE', 60
            ]
        },
        'longHr': {
            '$floor': {
                '$divide': [
                    '$longE', 3600
                ]
            }
        },
        'longMin': {
            '$floor': {
                '$divide': [
                    {
                        '$mod': [
                            '$longE', 3600
                        ]
                    }, 60
                ]
            }
        },
        'longSec': {
            '$mod': [
                '$longE', 60
            ]
```

```
            }
        }
    }, {
        '$sort': {
            '_id': 1
        }
    }, {
        '$project': {
            '_id': 0,
            'seasonNum': '$_id',
            'numEpisodes': '$numEpisodes',
            'startDate': '$startDate',
            'endDate': '$endDate',
            'shortestEpisodeLength': {
                '$concat': [
                    {
                        '$toString': '$shortHr'
                    }, ':', {
                        '$toString': '$shortMin'
                    }, ':', {
                        '$toString': '$shortSec'
                    }
                ]
            },
            'longestEpisodeLength': {
                '$concat': [
                    {
                        '$toString': '$longHr'
                    }, ':', {
                        '$toString': '$longMin'
                    }, ':', {
                        '$toString': '$longSec'
                    }
                ]
            }
        }
    }
]
)

pandas.DataFrame(list(res))
```

Out[7]:

	seasonNum	numEpisodes	startDate	endDate	shortestEpisodeLength	longestEpisodeLength
0	1	10	2011-04-17	2011-06-19	0:51:30	1:0:57
1	2	10	2012-04-01	2012-06-03	0:49:18	1:2:4
2	3	10	2013-03-31	2013-06-09	0:49:25	1:1:20
3	4	10	2014-04-06	2014-06-15	0:49:19	1:4:49
4	5	10	2015-03-29	2015-06-14	0:50:32	1:2:1
5	6	10	2016-04-24	2016-06-26	0:51:52	1:10:14
6	7	7	2017-07-16	2017-08-27	0:50:5	1:21:10
7	8	6	2019-04-14	2019-05-19	0:54:29	1:21:37

M4

- Write and execute a query that shows sublocations and the scenes they appear in
- Your aggregation should have the following attributes:
    - `subLocation`
    - `totalScenes` , which is the number of scenes that are set in the sublocation
    - `firstSeasonNum`
    - `firstEpisodeNum`
        - `(firstSeasonNum, firstEpisodeNum)` identifies the first episode that the sublocation appears in
    - `lastSeasonNum`
    - `lastEpisodeNum`
        - `(lastSeasonNum, lastEpisodeNum)` identifies the last episode that the sublocation appears in
- Order your output on `totalScenes` descending, and only keep the sublocations with more than 50 scenes

In [8]:
```
res = episodes.aggregate(
    [
    {
        '$unwind': '$scenes'
    }, {
        '$group': {
            '_id': '$scenes.subLocation',
            'totalScenes': {
                '$sum': 1
            },
            'firstSeasonNum': {
                '$first': '$seasonNum'
            },
            'firstEpisodeNum': {
                '$first': '$episodeNum'
            },
            'lastSeasonNum': {
                '$last': '$seasonNum'
            },
            'lastEpisodeNum': {
                '$last': '$episodeNum'
            }
        }
    }, {
        '$match': {
            'totalScenes': {
                '$gt': 50
            },
            '_id': {
                '$ne': None
            }
        }
    }, {
        '$sort': {
            'totalScenes': -1
        }
    }, {
        '$project': {
            '_id': 0,
            'subLocation': '$_id',
            'totalScenes': 1,
            'firstSeasonNum': 1,
            'firstEpisodeNum': 1,
```

```
            'lastSeasonNum': 1,
            'lastEpisodeNum': 1
        }
    }
]
)

pandas.DataFrame(list(res))
```

Out[8]:

	totalScenes	firstSeasonNum	firstEpisodeNum	lastSeasonNum	lastEpisodeNum	subLocation
0	1094	1	1	8	6	King's Landing
1	734	1	1	8	6	Winterfell
2	267	1	1	8	6	Castle Black
3	142	2	1	8	5	Dragonstone
4	77	1	1	8	6	The Haunted Forest
5	69	1	1	8	4	Outside Winterfell
6	66	2	1	4	5	Craster's Keep
7	60	2	10	8	6	The Wall
8	57	1	9	7	1	The Twins
9	56	7	4	7	5	Blackwater Rush
10	53	2	8	8	6	Blackwater Bay

Neo4j

- The cell below creates nodes and relationships that model movies and the people involved in them

TA TIPS

1. format: Match *[Req]* ... with ... where ... return *[Req]* ... order by ...
2.

In [9]:
```
with open("movies.txt") as f:
    queries = str(f.read())

graph.execute_query("match (p:Person), (m:Movie) detach delete p, m")
graph.execute_query(queries)
print("Successfully inserted movie data")
```

```
Successfully inserted movie data
```

N1

- Write and execute a cypher that shows actors and the number of movies they appear in
  - You should focus only on the `ACTED_IN` relationship, no other relationship
- Your output should have the following attributes:
  - `name` , which is the name of the actor
  - `num_movies`
- Order your output on `num_movies` descending, and only keep actors who have acted in 4 or more movies

In [10]:
```python
res = graph.execute_query("""
    match (p:Person)-[:ACTED_IN]->(m:Movie)
    with p.name AS name, count(m) AS num_movies
    where num_movies > 3
    return name, num_movies
    order by num_movies desc
""")

pandas.DataFrame([dict(r) for r in res.records])
```

Out[10]:

	name	num_movies
0	Tom Hanks	12
1	Keanu Reeves	7
2	Hugo Weaving	5
3	Jack Nicholson	5
4	Meg Ryan	5
5	Cuba Gooding Jr.	4

N2

- Write and execute a cypher that shows people and movies they either acted in or directed
- Your output should have the following attributes:
  - `name` , which is the name of the person
  - `directed_movies` , which is an array of titles of movies that the person directed
  - `acted_in_movies` , which is an array of titles of movies that the person acted in
- Order your output on `name` ascending, and only keep people that have directed at least one movie **and** acted in at least one movie (i.e., there should be no empty arrays. Arrays with one element are fine.)

In [11]:
```python
res = graph.execute_query("""
    match (p:Person)-[:ACTED_IN]->(m1:Movie), (p)-[:DIRECTED]->(m2:Movie)
WITH p.name AS name, collect(DISTINCT m1.title) as acted_in_movies, collect(DISTINCT m2.title) as
where size(acted_in_movies) > 0 and size(directed_movies) > 0
return name, directed_movies, acted_in_movies
order by name
""")

pandas.DataFrame([dict(r) for r in res.records])
```

Out[11]:

	name	directed_movies	acted_in_movies
0	Clint Eastwood	[Unforgiven]	[Unforgiven]
1	Danny DeVito	[Hoffa]	[Hoffa, One Flew Over the Cuckoo's Nest]
2	James Marshall	[V for Vendetta, Ninja Assassin]	[A Few Good Men]
3	Tom Hanks	[That Thing You Do]	[Cloud Atlas, The Da Vinci Code, The Green Mil...
4	Werner Herzog	[RescueDawn]	[What Dreams May Come]

N3

- Write and execute a cypher that shows people and movies they both acted in and directed
- Your output should have the following attributes:
  - `name` , which is the name of the person
  - `acted_in_and_directed_movies` , which is an array of titles of movies that the person both acted in and directed
- Order your output on `name` ascending, and only keep people that have acted in at least one movie that they directed (i.e., there should be no empty arrays. Arrays with one element are fine.)

In [12]:
```
res = graph.execute_query("""
    match (p:Person)-[:ACTED_IN]->(m:Movie), (p)-[:DIRECTED]->(m)
WITH p.name AS name, collect(DISTINCT m.title) as acted_in_and_directed_movies
where size(acted_in_and_directed_movies) > 0
return name, acted_in_and_directed_movies
order by name
""")

pandas.DataFrame([dict(r) for r in res.records])
```

Out[12]:

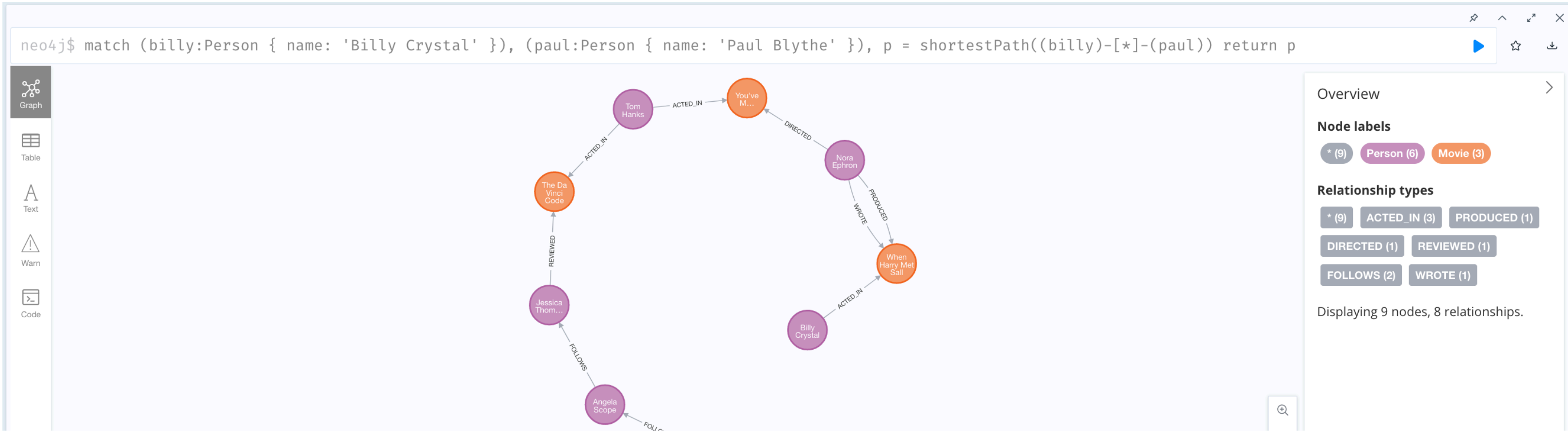
	name	acted_in_and_directed_movies
0	Clint Eastwood	[Unforgiven]
1	Danny DeVito	[Hoffa]
2	Tom Hanks	[That Thing You Do]

N4

- Write and execute a cypher that shows pairs of people and how closely connected they are
- Your output should have the following attributes:
  - `person_1_name` , which is the name of the first person in the pair
  - `person_2_name` , which is the name of the second person in the pair
  - `num_people_between` , which is the number of people (including the pair itself) separating the pair. You should use the `shortestPath` function to compute this.
- To prevent duplicates in your output, you should only keep rows where `person_1_name < person_2_name`
- Order your output on `(person_1_name, person_2_name)` , and only keep rows where `num_people_between > 5`
- As an example, you should get the following row in your output:

person_1_name	person_2_name	num_people_between
Billy Crystal	Paul Blythe	6

- The shortest path between Billy Crystal and Paul Blythe is shown below
  - `num_people_between` is 6 because there are 6 nodes marked as `Person` (including Billy's and Paul's nodes)


neo4j$ match (billy:Person { name: 'Billy Crystal' }), (paul:Person { name: 'Paul Blythe' }), p = shortestPath((billy)-[*]-(paul)) return p
Graph
Table
Text
Warn
Code
Tom Hanks
ACTED_IN
You've M...
DIRECTED
Nora Ephron
The Da Vinci Code
PRODUCED
WROTE
When Harry Met Sall
REVIEWED
Jessica Thom...
ACTED_IN
Billy Crystal
FOLLOWS
Angela Scope
Overview
Node labels
* (9)
Person (6)
Movie (3)
Relationship types
* (9)
ACTED_IN (3)
PRODUCED (1)
DIRECTED (1)
REVIEWED (1)
FOLLOWS (2)
WROTE (1)
Displaying 9 nodes, 8 relationships.

```python
In [13]: res = graph.execute_query("""
    MATCH path = shortestPath((p1:Person)-[*]-(p2:Person))
where p1.name < p2.name
WITH p1.name AS person_1_name, p2.name AS person_2_name, size([n in nodes(path) where n:Person]) a
WHERE num_people_between > 5
RETURN person_1_name, person_2_name, num_people_between
ORDER BY person_1_name, person_2_name;
""")

pandas.DataFrame([dict(r) for r in res.records])
```

Out[13]:

	person_1_name	person_2_name	num_people_between
0	Billy Crystal	Paul Blythe	6
1	Bruno Kirby	Paul Blythe	6
2	Carrie Fisher	Paul Blythe	6
3	Christian Bale	Dina Meyer	6
4	Christian Bale	Ice-T	6
5	Christian Bale	Paul Blythe	6
6	Christian Bale	Robert Longo	6
7	Christian Bale	Takeshi Kitano	6
8	Ethan Hawke	Paul Blythe	6
9	Jan de Bont	Paul Blythe	6
10	Paul Blythe	Scott Hicks	6
11	Paul Blythe	Zach Grenier	6

SQL To NoSQL

- You will move relational data to document and graph databases
  - **You will do your modeling in Python. You shouldn't be writing any SQL.**
- You will be using the `classicmodels` database for this section. You may want to drop the database and re-run the SQL script (included in the directory) to ensure you have the right data.
  - You will be modeling customers and the products they ordered

MongoDB: Customers

- For the document database, you will create two collections: `customers` and `products`
- `customers` will contain customer information as well as all the orders they've placed
- You will use `customer_orders_all_df` to create your `customers` collection

In [14]:
```
%%sql

customer_orders_all <<

select
    c.customerNumber, c.customerName, c.country, o.orderNumber,
    o.orderDate, od.productCode, od.quantityOrdered, od.priceEach
from classicmodels.orders o
    join classicmodels.customers c using (customerNumber)
    join classicmodels.orderdetails od using (orderNumber);
```

```
 * mysql+pymysql://root:***@localhost
2996 rows affected.
Returning data to local variable customer_orders_all
```

In [15]:
```python
customer_orders_all_df = customer_orders_all.DataFrame()
customer_orders_all_df.head(10)
```

Out[15]:

	customerNumber	customerName	country	orderNumber	orderDate	productCode	quantityOrdered	priceEach
0	363	Online Diecast Creations Co.	USA	10100	2003-01-06	S18_1749	30	136.00
1	363	Online Diecast Creations Co.	USA	10100	2003-01-06	S18_2248	50	55.09
2	363	Online Diecast Creations Co.	USA	10100	2003-01-06	S18_4409	22	75.46
3	363	Online Diecast Creations Co.	USA	10100	2003-01-06	S24_3969	49	35.29
4	128	Blauer See Auto, Co.	Germany	10101	2003-01-09	S18_2325	25	108.06
5	128	Blauer See Auto, Co.	Germany	10101	2003-01-09	S18_2795	26	167.06
6	128	Blauer See Auto, Co.	Germany	10101	2003-01-09	S24_1937	45	32.53
7	128	Blauer See Auto, Co.	Germany	10101	2003-01-09	S24_2022	46	44.35
8	181	Vitachrome Inc.	USA	10102	2003-01-10	S18_1342	39	95.55
9	181	Vitachrome Inc.	USA	10102	2003-01-10	S18_1367	41	43.13

- Below is an example of how a customer and their orders are stored in MySQL, and how the document should look like in MongoDB
- The document should have the following attributes:
  - `customerNumber`
  - `customerName`
  - `country`
  - `orders` , which is a list of objects. Each object represents one order
    - `orderNumber`
    - `orderDate`
    - `orderContents` , which is a list of objects. Each object represents one product in the order
      - `productCode`
      - `quantityOrdered`
      - `priceEach`

MySQL relation:

	customerNumber	customerName	country	orderNumber	orderDate	productCode	quantityOrdered	priceEach
0	103	Atelier graphique	France	10123	2003-05-20	S18_1589	26	120.71
1	103	Atelier graphique	France	10123	2003-05-20	S18_2870	46	114.84
2	103	Atelier graphique	France	10123	2003-05-20	S18_3685	34	117.26
3	103	Atelier graphique	France	10123	2003-05-20	S24_1628	50	43.27
4	103	Atelier graphique	France	10298	2004-09-27	S10_2016	39	105.86
5	103	Atelier graphique	France	10298	2004-09-27	S18_2625	32	60.57
6	103	Atelier graphique	France	10345	2004-11-25	S24_2022	43	38.98

MongoDB document:

```
{
    customerNumber: 103
    customerName: "Atelier graphique",
    country: "France",
    orders: [
        {
            orderNumber: 10123,
            orderDate: "2003-05-20",
            orderContents: [
                {
                    productCode: "S18_1589",
                    quantityOrdered: 26,
                    priceEach: "120.71"
                },
                {
                    productCode: "S18_2870",
                    quantityOrdered: 46,
                    priceEach: "114.84"
                },
                {
                    productCode: "S18_3685",
                    quantityOrdered: 34,
                    priceEach: "117.26"
                },
                {
                    productCode: "S24_1628",
                    quantityOrdered: 50,
                    priceEach: "43.27"
                }
            ]
        },
        {
            orderNumber: 10298,
            orderDate: "2004-09-27",
            orderContents: [
                {
```

In [16]:
```python
# TODO: Create a list of dicts. Each dict represents one customer.

"""
Tips:

    To iterate through dataframe:

        for _, r in customer_orders_all_df.iterrows():
            r = dict(r)
            Access fields like r['customerName'], r['country'], ...


    The orderDate and priceEach fields are stored as datetime.date and Decimal
    objects in the dataframe. These types are not compatible with the pymongo API.
    You can convert them to strings by calling str(r['orderDate']) and str(r['priceEach']).
    Alternatively, you can look into the datetime.datetime and bson.decimal128.Decimal128
    objects, which are supported by pymongo.

"""
customers = []
customer_group_df = customer_orders_all_df.groupby(['customerNumber', 'customerName', 'country'])
for group_name, customer_df in customer_group_df:
#     print("\n-- Group with {} rows(s)".format(len(df_group)))
#     print('CREATE TABLE {}('.format(group_name))
#     print(type(group_name))
    customer_data = {}
    customer_data['customerNumber'] = int(group_name[0])
    customer_data['customerName'] = group_name[1]
    customer_data['country'] = group_name[2]
    customer_data['orders'] = []

    order_group_df = customer_df.groupby(['orderNumber', 'orderDate'])
    for order_details, order_df in order_group_df:
        order_data = {}
        order_data['orderNumber'] = int(order_details[0])
#         print(order_details[1].strftime("%Y-%m-%d"))
#         break
        order_data['orderDate'] = str(order_details[1])
        order_data['orderContents'] = []
        for index, row in order_df.iterrows():
            order_data['orderContents'].append({'productCode': row[5],
                    'quantityOrdered': row[6],
                    'priceEach': str(row[7])})
```

```
        customer_data['orders'].append(order_data)
    customers.append(customer_data)
```

In [17]:
```
def insert_customers(d):
    mongo_client['w4111']['customers'].drop()
    mongo_client['w4111']['customers'].insert_many(d)


# TODO: Put the name of your list of dicts below
insert_customers(customers)
print("Successfully inserted customer data")
```

```
Successfully inserted customer data
```

MongoDB: Products

- To create the `products` collection, you will use `products_all_df`
- A document in `products` simply contains product information, as shown below

```
{
    productCode: "S10_1678",
    productName: "1969 Harley Davidson Ultimate Chopper",
    productVendor: "Min Lin Diecast"
}
```

In [18]:
```
%%sql

products_all <<

select productCode, productName, productVendor
from classicmodels.products;
```

```
 * mysql+pymysql://root:***@localhost
110 rows affected.
Returning data to local variable products_all
```

In [19]:
```
products_all_df = products_all.DataFrame()
products_all_df.head(10)
```

Out[19]:

	productCode	productName	productVendor
0	S10_1678	1969 Harley Davidson Ultimate Chopper	Min Lin Diecast
1	S10_1949	1952 Alpine Renault 1300	Classic Metal Creations
2	S10_2016	1996 Moto Guzzi 1100i	Highway 66 Mini Classics
3	S10_4698	2003 Harley-Davidson Eagle Drag Bike	Red Start Diecast
4	S10_4757	1972 Alfa Romeo GTA	Motor City Art Classics
5	S10_4962	1962 LanciaA Delta 16V	Second Gear Diecast
6	S12_1099	1968 Ford Mustang	Autoart Studio Design
7	S12_1108	2001 Ferrari Enzo	Second Gear Diecast
8	S12_1666	1958 Setra Bus	Welly Diecast Productions
9	S12_2823	2002 Suzuki XREO	Unimax Art Galleries

In [20]:
```
# TODO: Create a list of dicts. Each dict represents one product.

"""
Tips:

    To iterate through dataframe:

        for _, r in products_all_df.iterrows():
            r = dict(r)
            Access fields like r['productName'], r['productVendor'], ...

"""

products = []
for _, r in products_all_df.iterrows():
    product_data = {}
    product_data['productCode'] = r['productCode']
    product_data['productName'] = r['productName']
    product_data['productVendor'] = r['productVendor']
    products.append(product_data)
#     print(r)
#     break
```

In [21]:
```
def insert_products(d):
    mongo_client['w4111']['products'].drop()
    mongo_client['w4111']['products'].insert_many(d)


# TODO: Put the name of your list of dicts below
insert_products(products)
print("Successfully inserted product data")
```

```
Successfully inserted product data
```

MongoDB: Testing

- Run through the following cells
- **Make sure the outputs are completely visible. You shouldn't need to scroll to see the entire output.**

- You may need to click on the blank section immediately to the left of your output to toggle between scrolling and unscrolling

In [22]:
```
import json

def prepr(doc):
    try:
        del doc['_id']
    except KeyError:
        pass

    def convert_str(d):
        if isinstance(d, dict):
            for k, v in d.items():
                d[k] = convert_str(v)
            return d
        elif isinstance(d, list):
            for i, v in enumerate(d):
                d[i] = convert_str(v)
            return d
        else:
            return str(d)

    convert_str(doc)
    return json.dumps(doc, indent=2)
```

In [23]:

```
res = mongo_client['w4111']['customers'].aggregate([
    {
        '$match': {
            'customerNumber': 219
        }
    }
])

print(prepr(list(res)[0]))
```

```json
{
  "customerNumber": "219",
  "customerName": "Boards & Toys Co.",
  "country": "USA",
  "orders": [
    {
      "orderNumber": "10154",
      "orderDate": "2003-10-02",
      "orderContents": [
        {
          "productCode": "S24_3151",
          "quantityOrdered": "31",
          "priceEach": "75.23"
        },
        {
          "productCode": "S700_2610",
          "quantityOrdered": "36",
          "priceEach": "59.27"
        }
      ]
    },
    {
      "orderNumber": "10376",
      "orderDate": "2005-02-08",
      "orderContents": [
        {
          "productCode": "S12_3380",
          "quantityOrdered": "35",
          "priceEach": "98.65"
        }
      ]
    }
  ]
}
```

In [24]:
```
res = mongo_client['w4111']['customers'].aggregate([
    {
        '$match': {
            'customerNumber': 103
        }
    }
])

print(prepr(list(res)[0]))
```

```
{
  "customerNumber": "103",
  "customerName": "Atelier graphique",
  "country": "France",
  "orders": [
    {
      "orderNumber": "10123",
      "orderDate": "2003-05-20",
      "orderContents": [
        {
          "productCode": "S18_1589",
          "quantityOrdered": "26",
          "priceEach": "120.71"
        },
        {
          "productCode": "S18_2870",
          "quantityOrdered": "46",
          "priceEach": "114.84"
        },
        {
          "productCode": "S18_3685",
          "quantityOrdered": "34",
          "priceEach": "117.26"
        },
        {
          "productCode": "S24_1628",
          "quantityOrdered": "50",
          "priceEach": "43.27"
        }
      ]
    },
    {
      "orderNumber": "10298",
      "orderDate": "2004-09-27",
      "orderContents": [
        {
          "productCode": "S10_2016",
          "quantityOrdered": "39",
          "priceEach": "105.86"
        },
        {
          "productCode": "S18_2625",
          "quantityOrdered": "32",
```

```
            "priceEach": "60.57"
          }
        ]
      },
      {
        "orderNumber": "10345",
        "orderDate": "2004-11-25",
        "orderContents": [
          {
            "productCode": "S24_2022",
            "quantityOrdered": "43",
            "priceEach": "38.98"
          }
        ]
      }
    ]
}
```

In [25]:

```python
res = mongo_client['w4111']['products'].aggregate([
    {
        '$match': {
            'productCode': 'S18_1889'
        }
    }
])

print(prepr(list(res)[0]))
```

```
{
  "productCode": "S18_1889",
  "productName": "1948 Porsche 356-A Roadster",
  "productVendor": "Gearbox Collectibles"
}
```

Neo4j: All Data

- For the graph database, you will have two types of nodes: `Customer` and `Product`
  - **Make sure to use these exact names**

- An order is represented as a relationship from the `Customer` node to the `Product` node. The type of the relationship should be `ORDERED`.

In [26]:
```
%%sql

customer_orders_limit <<

select
    c.customerNumber, c.customerName, c.country, o.orderNumber,
    o.orderDate, od.productCode, p.productName, p.productVendor,
    od.quantityOrdered, od.priceEach
from classicmodels.orders o
    join classicmodels.customers c using (customerNumber)
    join classicmodels.orderdetails od using (orderNumber)
    join classicmodels.products p using (productCode)
where od.quantityOrdered > 49;
```

```
 * mysql+pymysql://root:***@localhost
139 rows affected.
Returning data to local variable customer_orders_limit
```

In [27]:
```python
customer_orders_limit_df = customer_orders_limit.DataFrame()
customer_orders_limit_df.head(10)
# customer_orders_limit_df.loc[customer_orders_limit_df['quantityOrdered'] == 60]
```

Out[27]:

	customerNumber	customerName	country	orderNumber	orderDate	productCode	productName	productVendor	quantityOrdere
0	363	Online Diecast Creations Co.	USA	10100	2003-01-06	S18_2248	1911 Ford Town Car	Motor City Art Classics	5
1	145	Danish Wholesale Imports	Denmark	10105	2003-02-11	S10_4757	1972 Alfa Romeo GTA	Motor City Art Classics	5
2	145	Danish Wholesale Imports	Denmark	10105	2003-02-11	S24_3816	1940 Ford Delivery Sedan	Carousel DieCast Legends	5
3	278	Rovelli Gifts	Italy	10106	2003-02-17	S24_3949	Corsair F4U (Bird Cage)	Second Gear Diecast	5
4	124	Mini Gifts Distributors Ltd.	USA	10113	2003-03-26	S18_4668	1939 Cadillac Limousine	Studio M Art Models	5
5	148	Dragon Souveniers, Ltd.	Singapore	10117	2003-04-16	S72_3212	Pont Yacht	Unimax Art Galleries	5
6	353	Reims Collectables	France	10121	2003-05-07	S12_2823	2002 Suzuki XREO	Unimax Art Galleries	5
7	103	Atelier graphique	France	10123	2003-05-20	S24_1628	1966 Shelby Cobra 427 S/C	Carousel DieCast Legends	5
8	458	Corrida Auto Replicas, Ltd	Spain	10126	2003-05-28	S18_4600	1940s Ford truck	Motor City Art Classics	5
9	324	Stylish Desk Decors, Co.	UK	10129	2003-06-12	S24_3816	1940 Ford Delivery Sedan	Carousel DieCast Legends	5

- Below is an example of how a customer and their orders are stored in MySQL, and how the graph should look like in Neo4j
  - Note that the same order may be represented as many relationships since one order could contain many products
- The `Customer` nodes should have the following attributes:
  - `customerNumber`
  - `customerName`

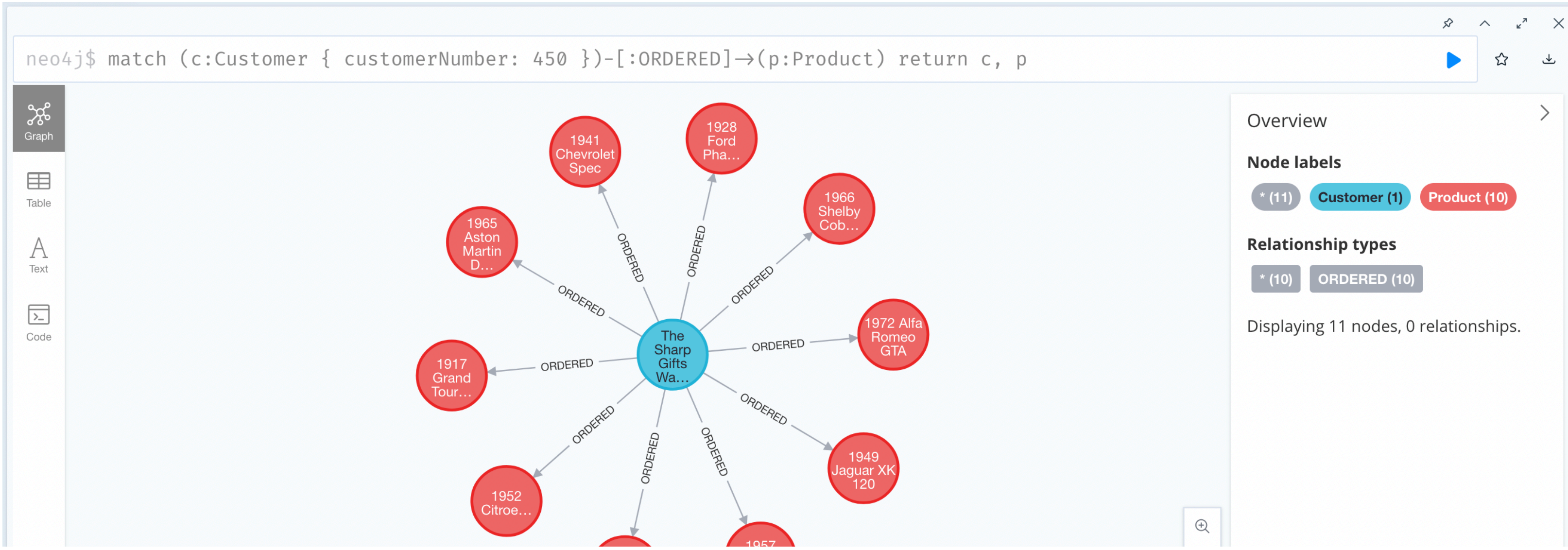
    - `country`
- The `Product` nodes should have the following attributes:
    - `productCode`
    - `productName`
    - `productVendor`
- The `ORDERED` relationships should have the following attributes:
    - `orderNumber`
    - `orderDate`
    - `quantityOrdered`
    - `priceEach`

MySQL relation:

	customerNumber	customerName	country	orderNumber	orderDate	productCode	productName	productVendor	quantityOrdered
0	450	The Sharp Gifts Warehouse	USA	10250	2004-05-11	S32_4289	1928 Ford Phaeton Deluxe	Highway 66 Mini Classics	50
1	450	The Sharp Gifts Warehouse	USA	10257	2004-06-14	S18_2949	1913 Ford Model T Speedster	Carousel DieCast Legends	50
2	450	The Sharp Gifts Warehouse	USA	10400	2005-04-01	S10_4757	1972 Alfa Romeo GTA	Motor City Art Classics	64
3	450	The Sharp Gifts Warehouse	USA	10400	2005-04-01	S18_3856	1941 Chevrolet Special Deluxe Cabriolet	Exoto Designs	58
4	450	The Sharp Gifts Warehouse	USA	10407	2005-04-22	S18_1589	1965 Aston Martin DB5	Classic Metal Creations	59
5	450	The Sharp Gifts Warehouse	USA	10407	2005-04-22	S18_1749	1917 Grand Touring Sedan	Welly Diecast Productions	76
6	450	The Sharp Gifts Warehouse	USA	10407	2005-04-22	S18_4933	1957 Ford Thunderbird	Studio M Art Models	66
7	450	The Sharp Gifts Warehouse	USA	10407	2005-04-22	S24_1628	1966 Shelby Cobra 427 S/C	Carousel DieCast Legends	64
8	450	The Sharp Gifts Warehouse	USA	10407	2005-04-22	S24_2766	1949 Jaguar XK 120	Classic Metal Creations	76

	customerNumber	customerName	country	orderNumber	orderDate	productCode	productName	productVendor	quantityOrdered
9	450	The Sharp Gifts Warehouse	USA	10407	2005-04-22	S24_2887	1952 Citroen-15CV	Exoto Designs	59

Neo4j graph:



In [28]:
```python
# Deletes all customers and products (and their relationships).
# Feel free to run this as many times as you want to reset your data.
_ = graph.execute_query("""
    match (c:Customer), (p:Product)
    detach delete c, p
""")
```

In [29]:
```python
df = customer_orders_limit_df.groupby(['productCode', 'productName', 'productVendor'])
# len(df.groups)
len(df.groups.keys())
```

Out[29]: 84

In [30]:
```
# TODO: Write and execute queries to create nodes and relationships

"""
Tips:

    To iterate through dataframe:

        for _, r in customer_orders_limit_df.iterrows():
            r = dict(r)
            Access fields like r['customerName'], r['country'], ...


    The priceEach field are stored as a Decimal object in the dataframe. This type is not
    compatible with the neo4j API. You can convert it to a string by calling str(r['priceEach']).


    You should call graph.execute_query to execute your queries. This method takes in a second
    optional argument, a dict. This allows you to do query parameters. For instance, to execute
    the query in the screenshot above, you could run

        graph.execute_query(
            "match (c:Customer { customerNumber: $custNum })-[:ORDERED]->(p:Product) return c, p"
            { "custNum": 450 }
        )

"""
customer_groups = customer_orders_limit_df.groupby(['customerNumber', 'customerName', 'country'])
for groupName, _ in customer_groups:
    graph.execute_query(
        """
        MERGE (c:Customer {customerNumber: $customerNumber, customerName: $customerName, country:
        """,
        {
            "customerNumber": groupName[0],
            "customerName": groupName[1],
            "country": groupName[2]
        }
    )

product_groups = customer_orders_limit_df.groupby(['productCode', 'productName', 'productVendor']
for groupName, _ in product_groups:
    graph.execute_query(
        """
```

```
        MERGE (p:Product {productCode: $productCode})
        SET p.productName = $productName,
            p.productVendor = $productVendor
        """,
        {
            "productCode": groupName[0],
            "productName": groupName[1],
            "productVendor": groupName[2]
        }
    )

for _, r in customer_orders_limit_df.iterrows():
    r = dict(r)
    graph.execute_query(
            """
            MATCH (c:Customer {customerNumber: $customerNumber})
            MATCH (p:Product {productCode: $productCode})
            MERGE (c)-[o:ORDERED {orderNumber: $orderNumber}]->(p)
            SET o.orderDate = $orderDate,
                o.quantityOrdered = $quantityOrdered,
                o.priceEach = $priceEach
            """,
            {
                "customerNumber": r['customerNumber'],
                "productCode": r['productCode'],
                "orderNumber": r['orderNumber'],
                "orderDate": r['orderDate'],
                "quantityOrdered": r['quantityOrdered'],
                "priceEach": str(r['priceEach'])
            }
    )
```

Neo4j: Testing

- Run through the following cells
- **Make sure the outputs are fully visible**

In [31]:
```python
res = []

for r in graph.execute_query("""
    match (c:Customer { customerNumber: 412 })-[o:ORDERED]->(p:Product)
    return c, o, p
""").records:
    res.append(dict(r['c']) | dict(r['o']) | dict(r['p']))

pandas.DataFrame(res)
```

Out[31]:

	country	customerNumber	customerName	orderNumber	quantityOrdered	orderDate	priceEach	productCode	productName	pro
0	New Zealand	412	Extreme Desk Decorations, Ltd	10418	52	2005-05-16	64.41	S24_2360	1982 Ducati 900 Monster	I
1	New Zealand	412	Extreme Desk Decorations, Ltd	10418	50	2005-05-16	100.01	S32_4485	1974 Ducati 350 Mk3 Desmo	S
2	New Zealand	412	Extreme Desk Decorations, Ltd	10234	50	2004-03-30	146.65	S18_1662	1980s Black Hawk Helicopter	
3	New Zealand	412	Extreme Desk Decorations, Ltd	10268	50	2004-07-12	124.59	S18_2325	1932 Model A Ford J-Coupe	Au

In [32]:
```
res = []

for r in graph.execute_query("""
    match (c:Customer)-[o:ORDERED]->(p:Product { productCode: 'S12_2823' })
    return c, o, p
""").records:
    res.append(dict(r['c']) | dict(r['o']) | dict(r['p']))

pandas.DataFrame(res)
```

Out[32]:

	country	customerNumber	customerName	orderNumber	quantityOrdered	orderDate	priceEach	productCode	productName	pro
0	UK	201	UK Collectables, Ltd.	10403	66	2005-04-08	122.00	S12_2823	2002 Suzuki XREO	
1	France	353	Reims Collectables	10121	50	2003-05-07	126.52	S12_2823	2002 Suzuki XREO	
2	Austria	382	Salzburg Collectables	10341	55	2004-11-24	120.50	S12_2823	2002 Suzuki XREO	

In [33]:
```
res = []

for r in graph.execute_query("""
    match (c:Customer)-[o:ORDERED { quantityOrdered: 60 }]->(p:Product)
    return c, o, p
""").records:
    res.append(dict(r['c']) | dict(r['o']) | dict(r['p']))

pandas.DataFrame(res)
```

Out[33]:

	country	customerNumber	customerName	orderNumber	quantityOrdered	orderDate	priceEach	productCode	productName	p
0	Spain	141	Euro+ Shopping Channel	10412	60	2005-05-03	157.49	S18_3232	1992 Ferrari 360 Spider red	
1	Australia	282	Souveniers And Things Co.	10420	60	2005-05-29	60.26	S24_1046	1970 Chevy Chevelle SS 454	
2	USA	362	Gifts4AllAges.com	10414	60	2005-05-06	72.58	S24_3151	1912 Ford Model T Delivery Wagon	I

In [34]:

```
res = []

for r in graph.execute_query("""
    match (n:Customer | Product)
    return labels(n) as type, count(*) as count
    union
    match ()-[r:ORDERED]->()
    return type(r) as type, count(*) as count
""").records:
    res.append(dict(r))

pandas.DataFrame(res)
```

Out[34]:

	type	count
0	[Product]	84
1	[Customer]	57
2	ORDERED	139