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 [3]:
```
%load_ext sql
%sql mysql+pymysql://root:dbuserbdbuser@localhost
```

```
The sql extension is already loaded. To reload it, use:
  %reload_ext sql
```

In [4]:
```
import sys

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

```
Requirement already satisfied: pymongo in /Users/lidongzhou/anaconda3/lib/python3.9/site-packages (4.6.3)
Requirement already satisfied: dnspython<3.0.0,>=1.16.0 in /Users/lidongzhou/anaconda3/lib/python3.9/site-
packages (from pymongo) (2.6.1)
Requirement already satisfied: neo4j in /Users/lidongzhou/anaconda3/lib/python3.9/site-packages (5.19.0)
Requirement already satisfied: pytz in /Users/lidongzhou/anaconda3/lib/python3.9/site-packages (from neo4
j) (2022.1)
```

- If you get warnings below, try restarting your kernel

In [5]:
```
import neo4j
import pandas
import pymongo

# TODO: Fill in with your Mongo URL
mongo_url = "mongodb+srv://dl3648:JiawS6iPvQgyLQEZ@w4111.rjse3ck.mongodb.net/?retryWrites=true&w=majority&
mongo_client = pymongo.MongoClient(mongo_url)

# TODO: Fill in with your Neo4j credentials
neo4j_url = "neo4j+s://e013ae70.databases.neo4j.io"
neo4j_password = "Uirl46odiy1JaOXvcmrXN4x9Bde5CTQiQlD-iV0-MJI"

# 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

Clustering index: A clustering index is used in databases where records are physically stored on the disk in the order of the index. This means that the index aligns with the way data is stored, which can enhance performance for range queries that fetch multiple records.

Nonclustering index: A nonclustering index, unlike a clustering index, does not dictate the physical ordering of the data records. Instead, it creates a separate structure to point to where the data lives. This allows more flexibility in indexing but might require more time to access specific records since the data might be scattered across the storage.

Sparse index: In a sparse index, index entries are not created for every record but rather for blocks of records. This means that each index entry points to a block where the actual data begins, and searching within the block is needed to find a specific record. Sparse indexes use less space and are quicker to scan.

Dense index: A dense index has an index entry for every single record in the database. This type of index provides faster direct access to data as it can directly point to every record's location but requires more storage space compared to a sparse index.

W2

Explain why nonclustering indexes must be dense.

Nonclustering indexes must be dense because they do not affect the physical order of the records in the database. Each entry in a nonclustering index points directly to a specific record, regardless of its physical location. This design ensures that every record can be efficiently located and accessed without needing to sequentially search through other records, thereby enhancing the speed and accuracy of queries that rely on these indexes. Nonclustering indexes must be dense because they do not affect the physical order of the records in the database. Each entry in a nonclustering index points directly to a specific record, regardless of its physical location. This design ensures that every record can be efficiently located and accessed without needing to sequentially search through other records, thereby enhancing the speed and accuracy of queries that rely on these indexes.

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?

It is not a good idea if these columns are frequently queried individually rather than in combination. A composite index is most effective when the indexed columns are used together in queries, in the order they are indexed. If class_code, semester, or year are often queried separately, individual indexes on each column would better optimize query performance, as each index could be directly and efficiently utilized by the database engine. Separate indexes provide greater flexibility and efficiency for queries targeting only one of these columns at a time.

W4

Explain the following concepts:

1. Hash index
2. B+ tree index

Hash index: A hash index uses a hash function to compute the location of data in a database. It's extremely efficient for point queries, where you retrieve data by its exact key, as it provides direct access to the data. However, it's less effective for range queries because the hash function does not preserve any ordering of keys. B+ tree index: A B+ tree index is a type of sorted tree structure that maintains data in a hierarchical order. It allows for efficient insertion, deletion, and lookup of data, and is particularly effective for range queries in addition to exact lookups. The leaves of the B+ tree, which contain pointers to the actual data records, are linked, facilitating efficient traversal of ordered records.

W5

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

Advantage: Hash indexes offer very fast data retrieval for exact match queries because they directly map keys to their data locations using a hash function, resulting in almost constant time complexity.

Disadvantage: Unlike B+ tree indexes, hash indexes do not support efficient range queries. Since hash functions do not preserve the natural order of keys, retrieving a range of data based on their order is inefficient compared to B+ tree indexes, which maintain keys in a sorted order conducive to sequential access and range searches.

W6

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

The buffer in a DBMS temporarily stores frequently or recently accessed data to minimize costly disk I/O operations, making data retrieval faster. A DBMS does not load the entire database into its buffer due to size constraints, as the database often exceeds available memory, and memory needs to be efficiently managed to accommodate various system processes and applications.

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

Clairvoyant algorithm: it is a theoretical approach used for academic purposes rather than practical implementations. This algorithm evicts the page that will not be needed for the longest period in the future, minimizing the number of page faults. It requires future knowledge of requests, which is typically not possible in real-world scenarios.

Least recently used strategy: This common replacement strategy evicts the page that has been used the least recently. LRU assumes that pages that have not been used recently will not be needed immediately, thus it keeps recently used data in the buffer. This is effective in many practical scenarios but can be expensive to implement precisely due to the need to track the order of access for all pages in the buffer.

Most recently used strategy: Contrary to LRU, the MRU strategy removes the most recently used page from the buffer. The rationale is that the most recently accessed page might not be needed soon again, especially in scenarios where once data is processed, it is less likely to be needed immediately. This approach can be useful in certain specific contexts where the most recent accesses are less likely to be repeated soon.

Clock algorithm: The clock algorithm is a practical approximation of LRU with lower overhead. It organizes the pages in a circular list and uses a pointer that sweeps around the clock, marking pages for replacement. Each page has a use bit that is

set when the page is accessed and cleared by the clock pointer when it considers the page for eviction. If the use bit is cleared, the page is evicted; otherwise, the bit is cleared and the pointer moves on, effectively giving the page a second chance before it can be evicted.

W8

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

1. Scalability: NoSQL databases are designed to scale out by using distributed architecture, allowing for horizontal scaling across many servers. This is particularly effective for handling large volumes of unstructured data or rapid growth.
2. Flexibility: NoSQL databases typically allow for a flexible schema that can evolve over time without requiring downtime for migrations. This makes it easier to adjust to changing data structures in applications.
3. Performance: NoSQL databases are optimized for specific data models and access patterns, which can provide performance advantages for certain types of queries and workloads, such as key-value lookups or real-time data processing.

W9

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

Impedance mismatch refers to the conflict that arises when transferring data between relational databases (SQL) and object-oriented programming structures. SQL databases use tables that don't align naturally with the object-oriented models used in programming, requiring cumbersome mapping between objects and database tables. NoSQL databases alleviate this by using data structures like documents or key-value stores, which are more akin to programming data types, reducing the need for complex mapping and streamlining the development process.

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
3. MongoDB: The many-to-many relationship between students and courses can be modeled by embedding lists of references within each document. For instance, each student document could contain a list of course IDs that the student is enrolled in, and similarly, each course document could contain a list of student IDs of those enrolled in the course. Alternatively, you could maintain a separate collection for enrollments that store references (IDs) to both students and courses.
4. Neo4j: This relationship is naturally suited to a graph database, where students and courses can be modeled as nodes. The relationship between these nodes can be directly represented using edges. For instance, an edge labeled "enrolled" could connect a student node to a course node, effectively modeling the many-to-many relationship. This allows for efficient queries on the network of connections, such as quickly finding all students in a particular course or all courses a student is enrolled in.

MongoDB

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

In [6]:
```
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 [7]:
```
res = episodes.aggregate(
    # TODO: Put your query here
[
    {
        '$addFields': {
            'numScenes': {
                '$size': '$scenes'
            }
        }
    }, {
        '$match': {
            'numScenes': {
                '$gte': 100
            }
        }
    }, {
        '$project': {
            '_id': False,
            'episodeTitle': '$episodeTitle',
            'seasonNum': '$seasonNum',
            'episodeNum': '$episodeNum',
            'numScenes': '$numScenes'
        }
    }, {
        '$sort': {
            'numScenes': -1
        }
    }
])

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

Out[7]:

	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 [8]:
```
res = episodes.aggregate(
    # TODO: Put your query here
[
    {
        '$sort': {
            'seasonNum': 1,
            'episodeNum': 1
        }
    }, {
        '$group': {
            '_id': {
                'seasonNum': '$seasonNum'
            },
            'firstThreeEpisodes': {
                '$firstN': {
                    'input': '$episodeTitle',
                    'n': 3
                }
            }
        }
    }, {
        '$project': {
            '_id': False,
            'seasonNum': '$_id.seasonNum',
            'firstThreeEpisodes': '$firstThreeEpisodes'
        }
    }, {
        '$sort': {
            'seasonNum': 1
        }
    }
])

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

Out[8]:

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

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 [9]:
```
res = episodes.aggregate(
    # TODO: Put your query here
    [
    {
        '$addFields': {
            'episodeLength': {
                '$max': '$scenes.sceneEnd'
            }
        }
    }, {
        '$group': {
            '_id': '$seasonNum',
            'shortestEpisodeLength': {
                '$min': '$episodeLength'
            },
            'longestEpisodeLength': {
                '$max': '$episodeLength'
            },
            'numEpisodes': {
                '$count': {}
            },
            'startDate': {
                '$min': '$episodeAirDate'
            },
            'endDate': {
                '$max': '$episodeAirDate'
            },
        }
    }, {
        '$project': {
            '_id': False,
            'seasonNum': '$_id',
            'numEpisodes': '$numEpisodes',
            'startDate': '$startDate',
            'endDate': '$endDate',
            'shortestEpisodeLength': '$shortestEpisodeLength',
            'longestEpisodeLength': '$longestEpisodeLength'
        }
    }, {
        '$sort': {
            'seasonNum': 1
        }
    }
])

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

Out[9]:

	seasonNum	numEpisodes	startDate	endDate	shortestEpisodeLength	longestEpisodeLength
0	1	10	2011-04-17	2011-06-19	0:51:30	1:00:57
1	2	10	2012-04-01	2012-06-03	0:49:18	1:02:04
2	3	10	2013-03-31	2013-06-09	0:49:25	1:01:20
3	4	10	2014-04-06	2014-06-15	0:49:19	1:04:49
4	5	10	2015-03-29	2015-06-14	0:50:32	1:02:01
5	6	10	2016-04-24	2016-06-26	0:51:52	1:10:14
6	7	7	2017-07-16	2017-08-27	0:50:05	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 [10]:
```
res = episodes.aggregate(
    # TODO: Put your query here
    [
    {
        '$unwind': {
            'path': '$scenes',
            'preserveNullAndEmptyArrays': False
        }
    }, {
        '$sort': {
            'seasonNum': 1,
            'episodeNum': 1
        }
    }, {
        '$group': {
            '_id': '$scenes.subLocation',
            'totalScenes': {
                '$count': {}
            },
            'firstSeasonNum': {
                '$first': '$seasonNum'
            },
            'firstEpisodeNum': {
                '$first': '$episodeNum'
            },
            'lastSeasonNum': {
                '$last': '$seasonNum'
            },
            'lastEpisodeNum': {
                '$last': '$episodeNum'
            }
        }
    }, {
        '$match': {
            'totalScenes': {
                '$gt': 50
            },
            '_id': {
                '$ne': None
            }
        }
    }, {
        '$project': {
            '_id': False,
            'subLocation': '$_id',
             'totalScenes': '$totalScenes',
            'firstSeasonNum': '$firstSeasonNum',
            'firstEpisodeNum': '$firstEpisodeNum',
            'lastSeasonNum': '$lastSeasonNum',
            'lastEpisodeNum': '$lastEpisodeNum'
        }
    }, {
        '$sort': {
            'totalScenes': -1
        }
    }
])

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

Out[10]:

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

Neo4j

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

In [11]:
```
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 [12]:
```
res = graph.execute_query("""
    match (p:Person), (p)-[:ACTED_IN]->(m:Movie)
    with p.name as name, count(m) as num_movies
    where num_movies >= 4
    return name, num_movies
    order by num_movies desc
""")

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

Out[12]:

	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 [13]:
```
res = graph.execute_query("""
    match (p:Person), (p)-[:DIRECTED]->(m1:Movie), (p)-[:ACTED_IN]->(m2:Movie)
    with p.name as name, collect(distinct m1.title) as directed_movies,
    collect(distinct m2.title) as acted_in_movies
    return name, directed_movies, acted_in_movies
    order by name
""")

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

Out[13]:

	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]	[You've Got Mail, Sleepless in Seattle, Joe Ve...
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 [14]:
```
res = graph.execute_query("""
    match (p:Person), (m:Movie), (p)-[:DIRECTED]->(m), (p)-[:ACTED_IN]->(m)
    with p.name as name, collect(distinct m.title) as acted_in_and_directed_movies
    return name, acted_in_and_directed_movies
    order by name
""")

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

Out[14]:

	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)

In [15]:
```
res = graph.execute_query("""
    match path = shortestPath((p1:Person)-[*]-(p2:Person))
    where p1 <> p2
    with p1.name as person_1_name, p2.name as person_2_name,
    size([n in nodes(path) where n:Person]) as num_people_between
    where person_1_name < person_2_name and 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[15]:

	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 [17]:
```
%%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 [18]:
```
customer_orders_all_df = customer_orders_all.DataFrame()
customer_orders_all_df.head(10)
```

Out[18]:

	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

	customerNumber	customerName	country	orderNumber	orderDate	productCode	quantityOrdered	priceEach
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: [
                {
                    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 [26]:

```
# 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.

"""

from datetime import datetime
from bson.decimal128 import Decimal128

customer_to_info = {}

for _, r in customer_orders_all_df.iterrows():
    r = dict(r)
    if r['customerNumber'] not in customer_to_info:
        customer_to_info[r['customerNumber']] = dict(
            customerNumber=r['customerNumber'],
            customerName=r['customerName'],
            country=r['country'],
            orders={}
        )
    cust_info = customer_to_info[r['customerNumber']]
    if r['orderNumber'] not in cust_info['orders']:
        cust_info['orders'][r['orderNumber']] = dict(
            orderNumber=r['orderNumber'],
            orderDate=datetime.combine(r['orderDate'], datetime.min.time()),
            orderContents=[]
        )
    cust_info['orders'][r['orderNumber']]['orderContents'].append(dict(
        productCode=r['productCode'],
        quantityOrdered=r['quantityOrdered'],
        priceEach=Decimal128(r['priceEach'])
    ))

customers_docs = []

for ci in customer_to_info.values():
    ords = []
    for o in ci['orders'].values():
        ords.append(o)
    customers_docs.append(dict(
        customerNumber=ci['customerNumber'],
        customerName=ci['customerName'],
        country=ci['country'],
        orders=ords
    ))

customers_docs[0]
```

Out[26]:

```
{'customerNumber': 363,
 'customerName': 'Online Diecast Creations Co.',
 'country': 'USA',
 'orders': [{'orderNumber': 10100,
   'orderDate': datetime.datetime(2003, 1, 6, 0, 0),
   'orderContents': [{'productCode': 'S18_1749',
     'quantityOrdered': 30,
     'priceEach': Decimal128('136.00')},
    {'productCode': 'S18_2248',
     'quantityOrdered': 50,
     'priceEach': Decimal128('55.09')},
    {'productCode': 'S18_4409',
     'quantityOrdered': 22,
     'priceEach': Decimal128('75.46')},
    {'productCode': 'S24_3969',
     'quantityOrdered': 49,
     'priceEach': Decimal128('35.29')}]},
  {'orderNumber': 10192,
   'orderDate': datetime.datetime(2003, 11, 20, 0, 0),
   'orderContents': [{'productCode': 'S12_4675',
     'quantityOrdered': 27,
     'priceEach': Decimal128('99.04')},
    {'productCode': 'S18_1129',
     'quantityOrdered': 22,
     'priceEach': Decimal128('140.12')},
    {'productCode': 'S18_1589',
     'quantityOrdered': 29,
     'priceEach': Decimal128('100.80')},
    {'productCode': 'S18_1889',
     'quantityOrdered': 45,
     'priceEach': Decimal128('70.84')},
    {'productCode': 'S18_1984',
     'quantityOrdered': 47,
     'priceEach': Decimal128('128.03')},
    {'productCode': 'S18_2870',
     'quantityOrdered': 38,
     'priceEach': Decimal128('110.88')},
    {'productCode': 'S18_3232',
     'quantityOrdered': 26,
     'priceEach': Decimal128('137.17')},
    {'productCode': 'S18_3685',
     'quantityOrdered': 45,
     'priceEach': Decimal128('125.74')},
    {'productCode': 'S24_1046',
     'quantityOrdered': 37,
     'priceEach': Decimal128('72.02')},
    {'productCode': 'S24_1628',
     'quantityOrdered': 47,
     'priceEach': Decimal128('49.30')},
    {'productCode': 'S24_2766',
     'quantityOrdered': 46,
     'priceEach': Decimal128('86.33')},
    {'productCode': 'S24_2887',
     'quantityOrdered': 23,
     'priceEach': Decimal128('112.74')},
    {'productCode': 'S24_2972',
     'quantityOrdered': 30,
     'priceEach': Decimal128('33.23')},
    {'productCode': 'S24_3191',
     'quantityOrdered': 32,
     'priceEach': Decimal128('69.34')},
    {'productCode': 'S24_3432',
     'quantityOrdered': 46,
     'priceEach': Decimal128('93.16')},
    {'productCode': 'S24_3856',
     'quantityOrdered': 45,
     'priceEach': Decimal128('112.34')}]},
  {'orderNumber': 10322,
   'orderDate': datetime.datetime(2004, 11, 4, 0, 0),
   'orderContents': [{'productCode': 'S10_1949',
     'quantityOrdered': 40,
     'priceEach': Decimal128('180.01')},
    {'productCode': 'S10_4962',
     'quantityOrdered': 46,
     'priceEach': Decimal128('141.83')},
    {'productCode': 'S12_1666',
```

```
     'quantityOrdered': 27,
     'priceEach': Decimal128('136.67')},
    {'productCode': 'S18_1097',
     'quantityOrdered': 22,
     'priceEach': Decimal128('101.50')},
    {'productCode': 'S18_1342',
     'quantityOrdered': 43,
     'priceEach': Decimal128('92.47')},
    {'productCode': 'S18_1367',
     'quantityOrdered': 41,
     'priceEach': Decimal128('44.21')},
    {'productCode': 'S18_2325',
     'quantityOrdered': 50,
     'priceEach': Decimal128('120.77')},
    {'productCode': 'S18_2432',
     'quantityOrdered': 35,
     'priceEach': Decimal128('57.12')},
    {'productCode': 'S18_2795',
     'quantityOrdered': 36,
     'priceEach': Decimal128('158.63')},
    {'productCode': 'S18_2949',
     'quantityOrdered': 33,
     'priceEach': Decimal128('100.30')},
    {'productCode': 'S18_2957',
     'quantityOrdered': 41,
     'priceEach': Decimal128('54.34')},
    {'productCode': 'S18_3136',
     'quantityOrdered': 48,
     'priceEach': Decimal128('90.06')},
    {'productCode': 'S24_1937',
     'quantityOrdered': 20,
     'priceEach': Decimal128('26.55')},
    {'productCode': 'S24_2022',
     'quantityOrdered': 30,
     'priceEach': Decimal128('40.77')}]}]}
```

In [30]:

```
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_docs)
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 [31]:

```
%%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 [32]:

```
products_all_df = products_all.DataFrame()
products_all_df.head(10)
```

Out[32]:

	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 [34]:
```
# 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'], ...

"""

product_docs = [dict(r) for _, r in products_all_df.iterrows()]
product_docs[0]
```

Out[34]:
```
{'productCode': 'S10_1678',
 'productName': '1969 Harley Davidson Ultimate Chopper',
 'productVendor': 'Min Lin Diecast'}
```

In [35]:
```
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(product_docs)
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 [36]:
```
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 [37]:
```
res = mongo_client['w4111']['customers'].aggregate([
    {
        '$match': {
            'customerNumber': 219
        }
    }
])

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


```
{
  "customerNumber": "219",
  "customerName": "Boards & Toys Co.",
  "country": "USA",
  "orders": [
    {
      "orderNumber": "10154",
      "orderDate": "2003-10-02 00:00:00",
      "orderContents": [
        {
          "productCode": "S24_3151",
          "quantityOrdered": "31",
          "priceEach": "75.23"
        },
        {
          "productCode": "S700_2610",
          "quantityOrdered": "36",
          "priceEach": "59.27"
        }
      ]
    },
    {
      "orderNumber": "10376",
      "orderDate": "2005-02-08 00:00:00",
      "orderContents": [
        {
          "productCode": "S12_3380",
          "quantityOrdered": "35",
          "priceEach": "98.65"
        }
      ]
    }
  ]
}
```


In [38]:
```
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 00:00:00",
      "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 00:00:00",
      "orderContents": [
        {
          "productCode": "S10_2016",
          "quantityOrdered": "39",
          "priceEach": "105.86"
        },
        {
          "productCode": "S18_2625",
          "quantityOrdered": "32",
          "priceEach": "60.57"
        }
      ]
    },
    {
      "orderNumber": "10345",
      "orderDate": "2004-11-25 00:00:00",
      "orderContents": [
        {
          "productCode": "S24_2022",
          "quantityOrdered": "43",
          "priceEach": "38.98"
        }
      ]
    }
  ]
}
```


In [54]:

```
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`.
- You will use `customer_orders_limit_df` to create your graph

In [55]:
```
%%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 [56]:
```
customer_orders_limit_df = customer_orders_limit.DataFrame()
customer_orders_limit_df.head(10)
```

Out[56]:

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

- 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	quantityOrd
0	450	The Sharp Gifts Warehouse	USA	10250	2004-05-11	S32_4289	1928 Ford Phaeton Deluxe	Highway 66 Mini Classics	
1	450	The Sharp Gifts Warehouse	USA	10257	2004-06-14	S18_2949	1913 Ford Model T Speedster	Carousel DieCast Legends	
2	450	The Sharp Gifts Warehouse	USA	10400	2005-04-01	S10_4757	1972 Alfa Romeo GTA	Motor City Art Classics	
3	450	The Sharp Gifts Warehouse	USA	10400	2005-04-01	S18_3856	1941 Chevrolet Special Deluxe Cabriolet	Exoto Designs	
4	450	The Sharp Gifts Warehouse	USA	10407	2005-04-22	S18_1589	1965 Aston Martin DB5	Classic Metal Creations	
5	450	The Sharp Gifts Warehouse	USA	10407	2005-04-22	S18_1749	1917 Grand Touring Sedan	Welly Diecast Productions	
6	450	The Sharp Gifts Warehouse	USA	10407	2005-04-22	S18_4933	1957 Ford Thunderbird	Studio M Art Models	
7	450	The Sharp Gifts Warehouse	USA	10407	2005-04-22	S24_1628	1966 Shelby Cobra 427 S/C	Carousel DieCast Legends	
8	450	The Sharp Gifts Warehouse	USA	10407	2005-04-22	S24_2766	1949 Jaguar XK 120	Classic Metal Creations	
9	450	The Sharp Gifts Warehouse	USA	10407	2005-04-22	S24_2887	1952 Citroen-15CV	Exoto Designs	

Neo4j graph:

In [57]:
```
# 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 [58]:
```
# 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 }
        )

"""

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

Neo4j: Testing

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

In [51]:
```
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[51]:

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

In [47]:
```
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[47]:

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

In [48]:

```
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[48]:

	country	customerNumber	customerName	orderNumber	quantityOrdered	orderDate	priceEach	productCode	productNam
0	Spain	141	Euro+ Shopping Channel	10412	60	2005-05-03	157.49	S18_3232	1992 Ferra 360 Spide re
1	USA	362	Gifts4AllAges.com	10414	60	2005-05-06	72.58	S24_3151	1912 For Model Deliver Wago
2	Australia	282	Souveniers And Things Co.	10420	60	2005-05-29	60.26	S24_1046	1970 Chev Chevelle S 45

In [59]:

```
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[59]:

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

In [ ]: