Data design and modelling, second assignment, MongoDB

Joy Albertini, Alessio Giovagnini, Jacob Salvi

Building the database

We used python to import the CSV file, parse It and insert the result inside a Mongo database.
We created a main collection “Restaurants” from the content of the CSV file, containing sub documents.
The code can be found at: https://github.com/JacobSalvi/DDM-ass2

Restaurants:

The collection “Restaurants” contains following fields:
- restaurant_link, the link of the restaurant, of string type
- restaurant_name, of string type
- claimed, of string type
- awards, a list of strings each representing an award
- keywords, a list of strings
- features, a list of strings
- Position, a sub document
- Popularity, a sub document
- Price, a sum document
- FoodInfo, a sub document
- Schedule, a sub document
- Review, a sub document
- Rating, a sub document

The fields, awards, keywords and features were saved as strings in the csv, during the parsing we spilt them to transform them into lists.

Position:

The sub-document “Position” contains:
- continent, a string
- country, a string
- region, a string
- province, a string
- city, a string
- address, a string
- latitude, a float
- longitude, a float

The CSV contained a field “original_location” which was a string encoded array containing the same fields already present is the position plus the continent. Therefore we extracted it and put in its own field.

Popularity:

The sub-document “Popularity” contains:
- popularity_detailed, a string
- popularity_generic, a string
- top_args, a list of strings

We split the top_args of the CSV into a list.

PriceInfo:

The sub-document “PriceInfo” contains:
- price_level, a string
- min_price, a int
- max_price, a int

We broke the CSV field “price_range” into two fields “min_price” and “max_price” since in our opinion this way it is easier to query and modify the range.

FoodInfo:

The sub-document “FoodInfo” contains:
- meals, a list of strings
- cuisines, a list of strings
- special_diets, a list of strings
- vegetarian_friendly, a strings
- vegan_option, a string
- gluten_free, a string

We split some fields into lists since in the CSV they are string encoded, comma separated list of strings.

Schedule:

The sub-document “schedule” contains:
- original_open_hours, a sub-document
- open_days_per_week, a float
- open_hours_per_week, a float
- working_shifts_per_week, a float

Since “original_ope_hours” was a string encoded json document we parsed it into a sub object.

Review:

The sub-document “Review” contains:
- total_reviews_count, a float
- default_language, a string
- reviews_count_in_default_language, a float

Ratings:

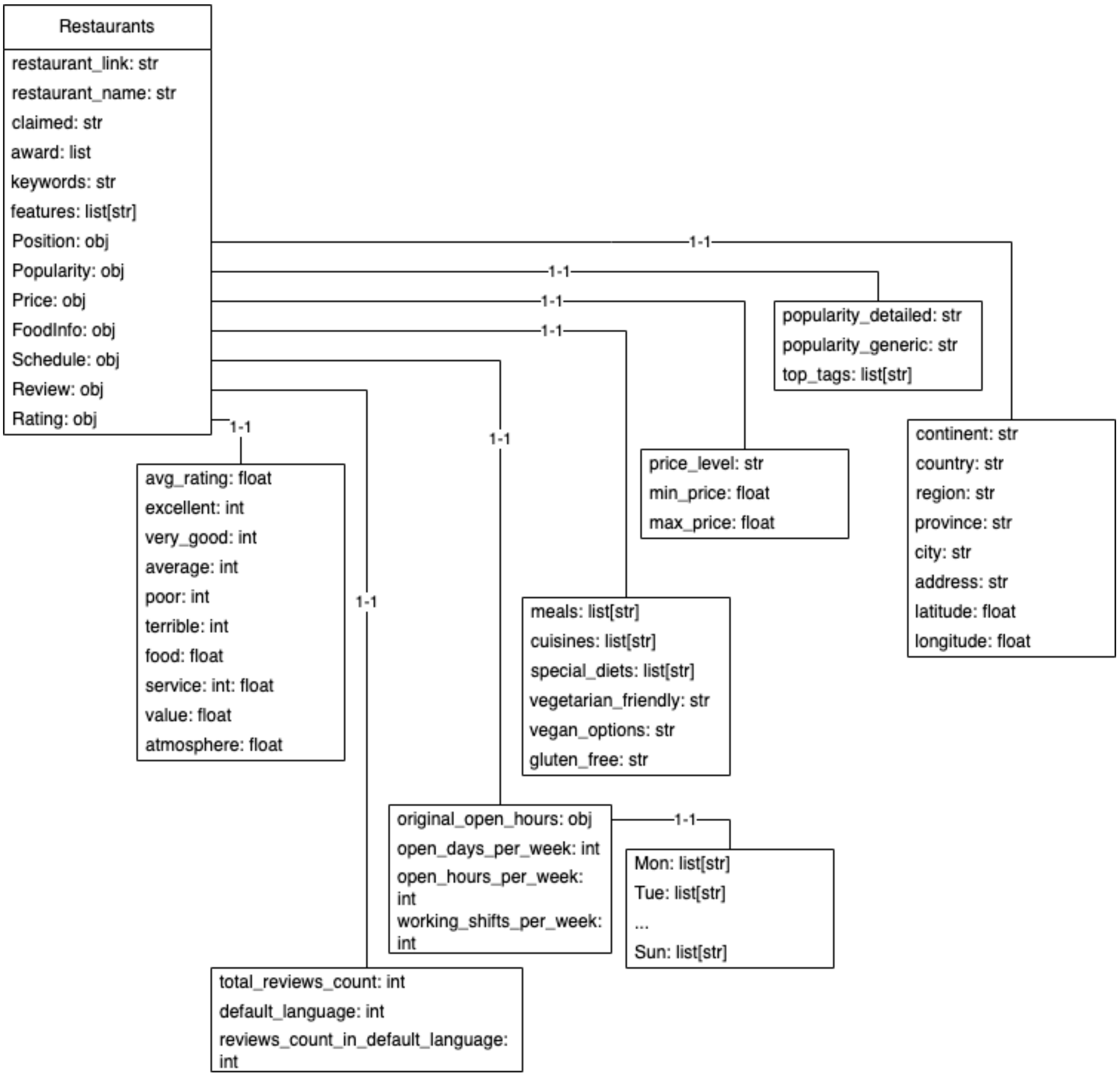
The sub-document “Ratings” contains:
- avg_rating, a float
- excellent, a float
- very_good, a float
- average, a float
- poor, a float
- terrible, a float
- food, a float
- service, a float
- value, a float
- atmosphere, a float

We put here a example document:

```
{
  "_id": {
    "$oid": "String"
  },
  "restaurant_link": "String",
  "restaurant_name": "String",
  "claimed": "String",
  "awards": [“String”],
  "keywords": [“String”],
  "features": [
    "String"
  ],
  "Position": {
    "continent": "String",
    "country": "String",
    "region": "String",
    "province": "String",
    "city": "String",
    "address": "String",
    "latitude": 45.961674,
    "longitude": 1.169131
  },
  "Popularity": {
    "popularity_detailed": "String",
    "popularity_generic": "String",
    "top_tags": [
      "String"
    ]
  },
  "Price": {
    "price_level": "String",
    "min_price": null,
    "max_price": null
  },
  "FoodInfo": {
    "meals": [
      "String"
    ],
    "cuisines": [
      "String"
    ],
    "special_diets": [“String”],
    "vegetarian_friendly": "N",
    "vegan_options": "N",
    "gluten_free": "N"
  },
  "Schedule": {
    "original_open_hours": {},
    "open_days_per_week": null,
    "open_hours_per_week": null,
    "working_shifts_per_week": null
  },
  "Review": {
    "total_reviews_count": 0,
    "default_language": "String",
    "reviews_count_in_default_language": 0
  },
  "Rating": {
    "avg_rating": 0,
    "excellent": 0,
    "very_good": 0,
    "average": 0,
    "poor": 0,
    "terrible": 0,
    "food": 0,
    "service": 0,
    "value": 0,
    "atmosphere": 0
  }
}
```

ER model

This is the ER diagram representing the relations between documents and sub-documents.



Queries

We tested all of the queries and commands on a MacBook Pro 2023, with a M2 CPU, 16 GB of ram, running Sonoma 14.

1 Get vegan restaurants in cities:

The first query returns all the vegetarian friendly restaurants, in the cities specified by the argument, which serve gluten free food.

```
def get_vegan_restaurants_in_cities(self, cities: list[str]):
    result = self.__db["Restaurants"].find({"FoodInfo.vegetarian_friendly": "Y",
                                        "FoodInfo.gluten_free": "Y",
                                        "Position.city": {"$in": cities}
                                        })
    return result
```

It took 0.30 milliseconds.

2 Get custom rating:

This query creates a custom rating based on some fields of Rating and uses it to sort the results. Limits to 30 the result if not the result will not be visible too many prints.

```
def sort_with_weighted_rating(self, country: str, pretty: bool):
    cursor = self.__db["Restaurants"].aggregate([
        {"$match": {"Position.country": country}},
        {"$project": {
            "restaurant_link": 1,
            "weightedRating": {
                "$add": [
                    {"$ifNull": ["$Rating.food", 0]},
                    {"$ifNull": ["$Rating.atmosphere", 0]},
                    {"$ifNull": ["$Rating.value", 0]},
                    {"$ifNull": ["$Rating.service", 0]}
                ]
            }
        }},
        {"$sort": {"weightedRating": -1}},
        {"$limit": 30}
    ])
    if not pretty:
        return [el for el in cursor]
    else:
        return prettify(
            [{"restaurant link": el["restaurant_link"], "weighted rating":
el["weightedRating"]} for el in cursor])
```

It took 864 milliseconds.

3 Get English speaking always open restaurants:

This queries return the English speaking restaurants with the given amount of open days, minimum reviews, minimum price and maximum price.

```
def get_english_speaking_always_open_restaurants(self, open_days: int, reviews: int,
min_price: int, max_price: int):
    cursor = self.__db["Restaurants"].find({"Schedule.open_days_per_week": open_days,
                                             "Review.total_reviews_count": {"$gte":
reviews},
                                             "Review.default_language": "English",
                                             "Price.min_price": {"$gte": min_price},
                                             "Price.max_price": {"$lte": max_price}})
    return [el for el in cursor]
```

It took 0.37 milliseconds.

4 Search restaurant in radius:

This query returns all the restaurant within a given distance from a position specified as a longitude and latitude.

```
def search_restaurants_in_radius(self, my_latitude: float, my_longitude: float, max_distance: float)
-> list:
    """
    find all restaurants in an area, Warning, the database seem to have incorrect values!!!
    :param my_latitude: latitude of center point to search
    :param my_longitude: longitude of center point to search
    :param max_distance: maximum distance from center point of search (is in degree, so very small
value
    should be provided, 1 deg of lat is around 111km, 1 del of long is 71 km)
    :return: list of restaurants links
    """
    expression = {
        "$sqrt": {
            "$add": [
                {
                    "$pow": [
                        {
                            "$subtract": [my_longitude, "$Position.longitude"]
                        },
                        2
                    ]
                },
                {
                    "$pow": [
                        {
                            "$subtract": [my_latitude, "$Position.latitude"]
                        },
                        2
                    ]
                }
            ]
        }
    }
    restaurants = self.__db["Restaurants"].find({"$expr": {"$lte": [expression,
max_distance]}}).limit(10)
    return [restaurant.get("restaurant_link") for restaurant in restaurants]
```

It took 62.22 milliseconds.

5 Search popular in city:

This query returns the top three most popular restaurants in a given city. Using the popularity_generic field which is a string.

```
def search_popular_in_city(self, city_name: str, pretty: bool):
    """
    return the 3 most popular places (generic) in a city
    :param pretty: returning the result in human readable format
    :param city_name: name of the city
    :return: list of restaurants link
    """
    restaurants = self.__db["Restaurants"].find({"Popularity.popularity_generic":
                                                {"$regex": f"^#[0-9]\D.*{city_name}$"}})\
        .sort([("Popularity.popularity_generic", 1)]).limit(3)

    if not pretty:
        return [restaurant["restaurant_link"] for restaurant in restaurants]
    else:
        return prettify([{"resturant-name": restaurant["restaurant_name"], "restaurant-link":
restaurant["restaurant_link"]}
                for restaurant in restaurants])
```

It took 574. 81 milliseconds.

6 Search with feature:

This query returns every Restaurant in a given city with a given feature.

```
def search_with_feature(self, feature: str, city: str) -> list:
    """
    filter restaurants in an area that posses a feature
    :param feature: word to search
    :param city: city to search in
    :return: list of restaurants
    """
    restaurants = self.__db["Restaurants"].find({
        'Position.city': city,
        'features': feature
    })
    return [restaurant for restaurant in restaurants]
```

It took 751. 57 milliseconds.

7 Find most expensive restaurant in each country:

This query returns the most important restaurant for each country. *price_level* is used because there are a lot of error in the field max_price, just to filter out some wrong data.

```
def find_most_expensive_restaurant_in_each_country(self, pretty : bool):
    cursor = self.__db["Restaurants"].aggregate([
        # filter only for the resturanr tagged "€€-€€€"
        {"$match": {"Price.price_level": "€€-€€€"}},
        {"$match": {"Price.max_price": {"$exists": True}}},
        {"$sort": {"Price.max_price": -1}},
        # $$ROOT returns the entire document restaurant most expensive for each group
        {"$group": {
            "_id": "$Position.country",
            "most_expensive_restaurant": {"$first": "$$ROOT"}
        }}
    ])
    if not pretty:
        return list(cursor)
    else:
        return prettify([{"Country": row['_id'],
                          "restaurant": row["most_expensive_restaurant"]["restaurant_name"],
                          "Max Price": row["most_expensive_restaurant"]["Price"]["max_price"],
                          "Symbolic price": row["most_expensive_restaurant"]["Price"]["price_level"]
                          }
                         for row in cursor])
```

It took 11815.973 milliseconds.

8 Find most highest rated restaurants in the most popular cities:

This query finds the top 10 most rated restaurants in the top 5 most popular cities. The popularity of a city is determined by the frequency of city in the DB, assumption that the popularity of city is given by the number of restaurant in it.

```
def find_top10_highest_rating_restaurant_in_the_5most_popular_cities(self, 
pretty : bool):
    # Find the most popular cities in the world,
    # in order to do it we assumed that the most popular cities are the cities 
with most entries in the db
    top_cities_cursor = self.__db["Restaurants"].aggregate([
        # $ne filters out string equal to ""
        {"$match": {"Position.city": {"$exists": True, "$ne": ""}}},
        {"$group": {"_id": "$Position.city", "count": {"$sum": 1}}},
        {"$sort": {"count": -1}},
        {"$limit": 5}
    ])

    # Retrive the highest revived restaurant in the most popular cities. The 
reviewed score is determined useing
    # both the average rating of a restaurant and the number of excellent 
reviews
    top_cities = [city['_id'] for city in top_cities_cursor]
    restaurant_cursor = self.__db["Restaurants"].find({
        "Position.city": {"$in": top_cities},
    }).sort([("Rating.avg_rating", -1), ("Rating.excellent", -1)]).limit(10)

    if not pretty:
        return [restaurant for restaurant in restaurant_cursor]
    else:
        return prettify([{
            "City": restaurant["Position"]["city"],
            "Restaurant": restaurant["restaurant_name"],
            "Average rating": restaurant["Rating"]["avg_rating"],
            "Number of excellent ratings": restaurant["Rating"]["excellent"]
        }
            for restaurant in restaurant_cursor])
```

It took 3050.03 milliseconds.

9 Find the top 5 countries with the average highest excellent reviews:

This query finds the top five country by average highest excellent review. It computes the average excellent review by summing each the number of excellent review in a country and divide it by the number restaurant.

```
def get_top5_countries_with_the_highest_average_excellent_reviews(self, pretty : 
bool):
    cursor = self.__db["Restaurants"].aggregate([
        # takes only restaurant with excenllent rating
        {"$match": {"Rating.excellent": {"$exists": True}}},
        # Groups for country, sums up excellent ratings and counting 
restaurants.
        {"$group": {
            "_id": "$Position.country",
            "total_excellents": {"$sum": "$Rating.excellent"},
            "num_restaurants": {"$sum": 1}
        }},
        # Calculate the average of excellent reviews, $project adds field to the
group
        {"$project": {
            "avg_excellent": {"$divide": ["$total_excellents",
"$num_restaurants"]}
        }},
        {"$sort": {"avg_excellent": -1}},
        {"$limit": 5}
    ])
    if not pretty:
        return [result for result in cursor]
    else:
        return prettify([{"Country": row['_id'],
                          "Average of excellent reviews":
math.floor(row['avg_excellent'])}
                                 for row in cursor])
```

It took 1874.02 milliseconds.

10 Find three closest restaurants:

This query finds the three closest restaurants in a city chosen at random, which has between 10 and 100 restaurants. This query would be impossible in mongoDB so I used goopy to retrieve the 3 closest restaurant to each other connect effectively mongoDB with a new service.

```
def find_the_closest_three_restaurant_in_randon_city(self):

    # find a random city with at least 10 restaurant and at most 100
    cities = [
        {"$group": {"_id": "$Position.city", "counts": {"$sum": 1}}},
        {"$match": {"counts": {"$gte": 1000, "$lte": 2000}}},
        {"$sample": {"size": 1}}
    ]
    city = list(self.__db["Restaurants"].aggregate(cities))
    city_name = city[0]["_id"] if city else None
    number_of_resturant = city[0]["counts"]

    if city_name is None:
        raise ValueError("city name is none")

    # Create a map with all restaurants in city,
    # containing the key restaurant_name, values: position latitude, position.longitude
    query = {
        "Position.city": city_name,
        "Position.latitude": {"$exists": True},
        "Position.longitude": {"$exists": True}
    }
    restaurants_positions = list(self.__db["Restaurants"].find(query,
                                                             {"restaurant_name": 1,
"Position.latitude": 1,
                                                              "Position.longitude":
1}))

    if restaurants_positions is None:
        raise ValueError("no positions for restaurant")

    min_distance = float("inf")
    triple = None
    # tool to find 3 resturants with smallest location fast, mongoDB could not have done
it
    for r1, r2, r3 in itertools.combinations(restaurants_positions, 3):
        location1 = (r1["Position"]["latitude"], r1["Position"]["longitude"])
        location2 = (r2["Position"]["latitude"], r2["Position"]["longitude"])
        location3 = (r3["Position"]["latitude"], r3["Position"]["longitude"])
        distance = great_circle(location1, location2, location3).meters

        if distance < min_distance:
            min_distance = distance
            triple = (r1["restaurant_name"], r2["restaurant_name"],
r3["restaurant_name"])

    print(
        f"in City {city_name} with number of restaurants {number_of_resturant}. The
closest restaurants between "
        f"each other are: {triple[0]},{triple[1]} and {triple[2]}, "
        f"distance between the them is {math.floor(min_distance)} m")
```

It took 2572.02 milliseconds.

Commands

1 Update ratings:

It update the rating of a given restaurant.

```
def update_ratings(self, restaurant_link: str, rating: Rating):
    """
    update the rating of a restaurant
    :param restaurant_link: the link to the restaurant
    :param rating: new rating to add
    :return:
    """
    old_rating = self.__db["Restaurants"].find_one({"restaurant_link":
restaurant_link}).get("Rating")
    if not rating or not old_rating:
        print(f"Restaurant link: {restaurant_link} not found in DB")
        return
    old_rating_table: dict = {
        "excellent": old_rating.get("excellent"),
        "very_good": old_rating.get("very_good"),
        "average": old_rating.get("average"),
        "poor": old_rating.get("poor"),
        "terrible": old_rating.get("terrible"),
    }
    old_rating_table[rating.name] = old_rating_table[rating.name] + 1

    total = 0
    review_count = 0
    for key, val in old_rating_table.items():
        review_count = review_count + val
        total = total + (val * Rating[key].value)
    new_average = total / review_count
    self.__db["Restaurants"].update_one({"restaurant_link": restaurant_link},
                                        {"$set": {
                                            "Rating.avg_rating": new_average,
                                            f"Rating.{rating.name}": old_rating_table[rating.name]
                                        },
                                            "$inc": {"Review.total_reviews_count": 1}
                                        })
```

It took 4.01 milliseconds.

2 Update Restaurant feature:

It adds a a feature to a restaurant feature set.

```
def update_restaurant_feature(self, restaurant_link: str, new_feature: str):
    """
    add a feature to a restaurant if it do not exist
    :param restaurant_link: link to restaurant
    :param new_feature: feature to add
    """
    self.__db["Restaurants"].update_one({"restaurant_link": restaurant_link}, {
        "$addToSet": {"features": new_feature}
    })
```

It took 3.44 milliseconds.

3 Add weekend availability:

It adds the feature “openDuringTheWeekEnd” to every restaurant which is open during Saturday and Sunday.

```
def add_weekend_availability(self):
    self.__db["Restaurants"].update_many(filter={
        "Schedule.original_open_hours.Sat": {"$exists": True},
        "Schedule.original_open_hours.Sun": {"$exists": True}
    },
        update={"$push": {"features": "openDuringTheWeekEnd"}})
    return
```

It took 5671.51 milliseconds.

4 Increase price for restaurants with seating:

For every restaurant in a given city, which has the features “Seating” and “ServesAlcohol” open at least five days a week which serve French cuisine, it will increase the minimum price by the given amount. If the minimum price was null it will set it to the given minimum_price.

```
def increase_price_for_restaurants_with_seating(self, city: str, minimum_price: int, increase: int):
    self.__db["Restaurants"].update_many(filter={"Position.city": city,
                                                  "features": {"$all": ["Seating", "ServesAlcohol"]},
                                                  "FoodInfo.cuisines": {"$in": ["French"]},
                                                  "Schedule.open_days_per_week": {"$gte": 5}
                                                  },
                                          update=[{
                                              "$set": {
                                                  "Price.min_price": {
                                                      "$switch": {
                                                          "branches": [
                                                              {"case":
                                                                   {"$eq": ["Price.min_price", None]},
                                                               "then": minimum_price
                                                               }
                                                          ],
                                                          "default": {"$sum": ["Price.min_price", increase]}
```

```
                                                            }
                                                        }

                                                    },
                                                }])
        return
```

It took 576.23 milliseconds.

5 Give to each restaurant link to other similar priced Restaurant:

Save in each restaurant an array of maximum 4 restaurant in the same city which have the same price level as the first one. This would allow an application to show easily recommended restaurant when someone click on a specific restaurant. Unfortunately this command is very slow, I tried to improve the performance by caching and using bulk_write in the DB but is not enough to allow this query to update the entire DB. So instead I decided to specify a city to work with.

```python
def update_restaurant_by_assigning_a_similarly_priced_resturant_to_each_other_in_Osnabruck(self):
    price_levels = ["€", "€€-€€€", "€€€€"]
    to_be_updated = []

    for level in price_levels:
        all_restaurants = list(self.__db["Restaurants"].find(
            # IMPORTANT: unfortunately I could only do it in small cities,
            # for large cities or the entirety of the db takes too much time
            {"Position.city": "Osnabruck", "Price.price_level": level},
            {"restaurant_link": 1}
        ))
        all_resturant_links = [r["restaurant_link"] for r in all_restaurants]

        # Assign similar restaurants
        for restaurant_link in all_resturant_links:
            # [:4] limit to 4 the operation --> imprve performance
            similar_priced_restaurant = [r for r in all_resturant_links if r != restaurant_link][:4]
            to_be_updated.append(UpdateOne(
                {"restaurant_link": restaurant_link},
                {"$set": {"similar_priced_restaurants": similar_priced_restaurant}}
            ))

    # write all in bulk should improve performance
    if to_be_updated:
        self.__db[“Restaurants”].bulk_write(to_be_updated)
```

It took 3003 milliseconds.

This a simple query to show that the DB was correctly updated.

```python
def print_restaurants_connection_in_Osnabruck(self):
    restaurants = self.__db["Restaurants"].find({"Position.city": "Osnabruck"})
    return prettify([{
        "Restaurant Name": el.get("restaurant_name"),
        "Restaurant Link": el.get("restaurant_link"),
        "Similar Priced Restaurants Links": el.get("similar_priced_restaurants", [])
    } for el in restaurants if "similar_priced_restaurants" in el and
el["similar_priced_restaurants"]])
```