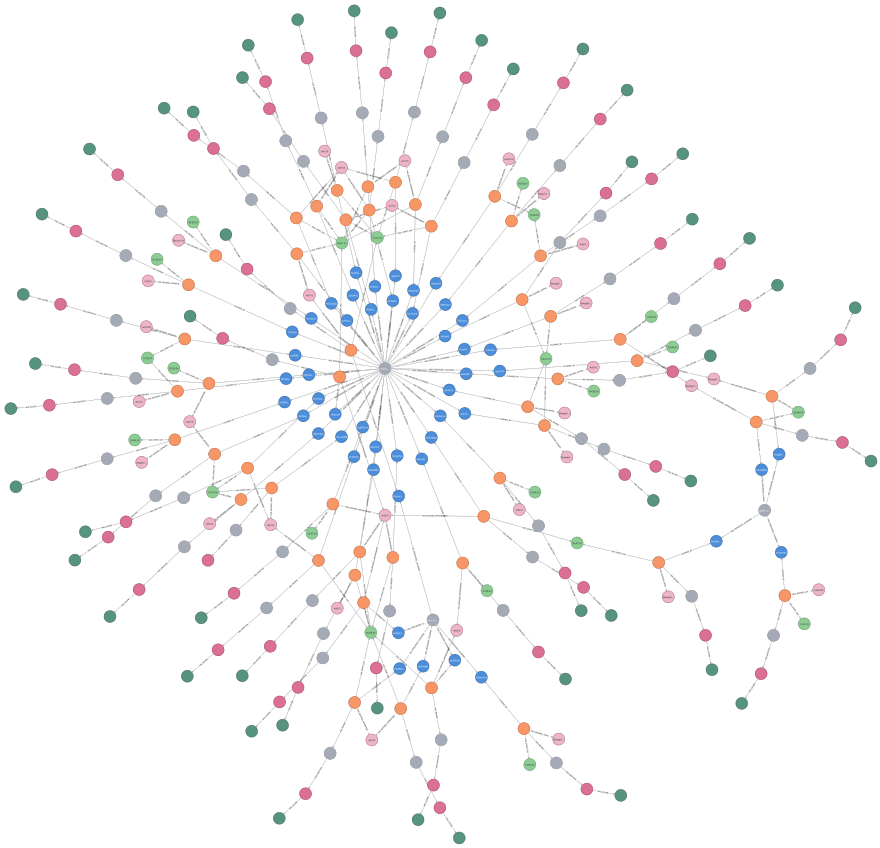
Thot Reviews

Dominik Pichler
SS24, 192.116 Knowledge Graphs @ TU Wien
Student ID: 01530718



1. Introduction

1.1 Scenario

Short-Term Renting business (STR) is hard, but without the right monitoring tools for customer satisfaction, it is even harder (then it has to be). This Project utilizes modern Knowledge-Graph-based Approaches to assist hotels and STR-businesses in **identifying key issues** regarding their cleaning services and customer satisfactions. In particular, it aims to identify whether certain apartments or cleaning staff form clusters or sources of exceptionally good or bad customer experiences.

1.1.1 (LO11)Proposed Analytics and Solutions

For this reason, this project aims to provide a presentation layer (via a `streamlit` application) that displays the following information to the user:

1. A list of cleaning personal that is linked to the best/worst customer experiences.
2. A list of apartements that are linked to the best/worst customer experiences.
3. An analysis to identify if certain cleaning people or appartements became a central node in a node of bad customer experiences or form a cluster utilizing utilizing *Deep Modularity Networks*.
4. A list of cleaners that are assumed to have a high record of bad cleaning quality, which is determined through the use of a Graph Embedding-Technique (*TransE*) that learns whether a review indicates a bad cleaning quality.

Eventually, this insight could then be used to infer insights for improvements in cleaning protocols, appartements and eventually customer satisfaction.

1.2 Background

In the hotel/STR business, a common SaaS Stack is the combination of *Kross Booking* that provides PMS + Channel Manager + Booking Engine in one solution, in combination with *TimeTac* that allows for smart time tracking of all internal (cleaning) processes. While the above is great for managing daily operations, the amount of data insight that can be extracted out of the box is pretty limited and Hence, business owners of certain scales that use the SaaS stack described above are left with high amounts of manual analytical effort with still limited insights.
Therefore, this project tries to reduce the amount of manual effort needed, as well as to increase the quality of insight possible.

2. Data Source

In order to achieve the previously defined objectives, the following data has been used to construct a Knowledgegraph (based on an ontology, depicted later on in the architecture section) from two APIs. Fortunately, I was able to get access to real data, generated by a large austrian business.

2.1 Raw Data

2.1.1 Kross Booking

As mentioned before, *Kross Booking* is a plattform that works as booking engine for the management of hotels/appartements. In this context, it is used to oversee (and store) all bookings and related activities across all properties. The stored data about the bookings can be fetched via a REST-API that follows the OpenAPI Standard.

2.1.2 TimeTac

Timetac is a plattform that allows to track process times of (cleaning) personnel. In this instance, it is utilized to track and access data regarding who cleaned each apartment, along with the timing and duration of the cleaning. The stored data about cleaning durations can be fetched via a REST-API that follows the OpenAPI Standard.

2.2 Additionally derived Data

In addition, the collected reviews (via *Kross Booking*) are automatically translated and pre-evaluated with a sentiment model.

2.2.1 Translation

As the guests of the appartements can (and have been) writing reviews in more than 150 different languages, I had to start out by automatically translating them to a single language. For this purpose, I used the `src/review_process_utils/review_translor.py` script that utilizes the `googleTrans` package to translate all reviews (if possible) to English.

2.2.2 Sentiment Analysis

In addition, for effective review analysis, a sentiment analysis utilizing `DistilRoBERTa` has been implemented to categorize the reviews along *Paul Ekman's 6 basic dimensions* + one neutral dimension in case no particular emotion has been detected. The corresponding script can be found in `src/Review_Handler.py`

2.2.3 Results

The two additional operations, the translation and sentiment analysis then yielded the following additional review data per Booking_ID for the knowledge graph:

Column Name	Data Type
Booking_ID (PK)	INT
Translated_Review_Text	TEXT
Primary_Emotion	TEXT
Sentiment_Score	FLOAT
Cleaning_Quality	INT

To simplify the data management, this data is also stored in an `AWS RDS`. Of course arguments for storing this data in a NoSQL Table like MongoDB or AWS Dynamo DB can be made, but due to the limited scope of this project I have decided to keep the overhead low and not setup another DB.

2.3 Data for the Knowledge-Graph-Generation

Finally, the resulting data has been stored in the following ABT `ABT_BASE_TABLE_KG_GENERATION` that will be used for building the Knowledge Graph:

Column Name	Data Type	Source
Booking_ID (PK)	INT	KROSS
Start_date_of_stay	TIME STAMP	KROSS
Appartement	STRING	KROSS
Cleaner	STRING	TIMETAC
Translated_Review_Text	TEXT	KROSS

Column Name	Data Type	Source
Primary_Emotion	TEXT	ML Model
Sentiment_Score	FLOAT	ML Model
Quality_Indication	INT	Manual/TransE

Side Notes:
For this demonstration purpose, the production data has been used and been anonymized using `src/data_anonimizer.py` and stored in `data/demo_data.csv` - Given my limited local computational resources, I have opted to work with a small sample of the original data. However, the project is designed to be scalable, allowing for the entire dataset to be efficiently processed by deploying it to AWS with additional computational resources.

3. (LO5,LO7,LO8, LO9) KGMS and KG Construction

As mentioned before, the application utilizes data that has been fetched from *KROSS Booking* and *TimeTac* via their internal APIs is currently stored in a `AWS RDS` in multiple tables using the architecture displayed below:

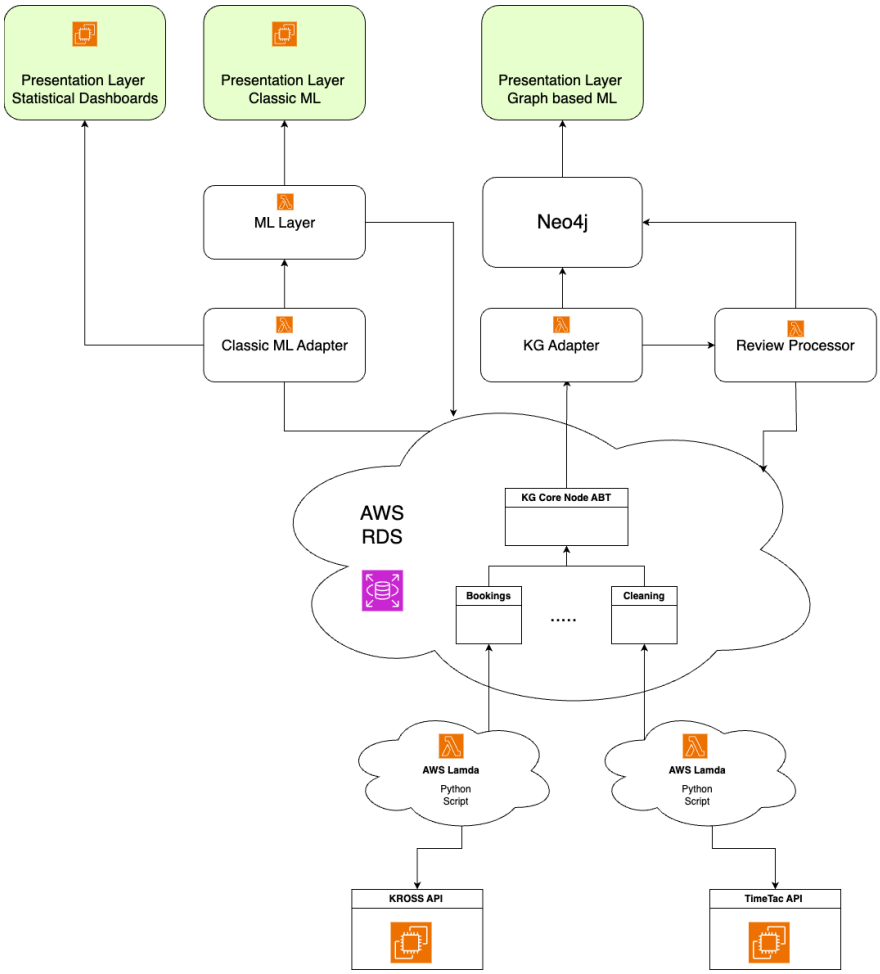


Figure 1: Description of the application architecture

In the beginning the data is being fetched from the two API's utilizing a Python script that runs in a *AWS Lamda Container* that is being executed once per day (at midnight). Once the data has been fetched successfully, is then stored in extraction tables in a *PostgreSQL* DB stored in an *AWS RDS Instance* (serving as central source of truth) and then automatically (via *AWS Lamda* again) processed into the aforementioned ABT.

In the meantime, an adapter (also running in a different *AWS Lamda Container*), is daily fetching new review data from the ABT, sends the reviews to a sentiment model (`emotion_detector.py`) that returns the top sentiment scores for each review.

After that, the data gets transformed into a graph-structure and then added to an on-premise *Neo4J* Database (dockerized) via the following python script `src/KG_Building_Handler.py`. This script iterates over every row in the ABT, transforms it based on the Ontology below and adds it to Neo4j. In case a certain Node already exists (for example, an appartement has already been booked before), the script uses Cypher to determine this and instead of creating a new node of this kind, the already existing ndoe will be used.

Through the process outlined above, the Knowledge Graph (KG) is updated daily with the latest available

data, allowing it to continuously evolve. The resulting KG comprises a collection of nodes and edges. Each row in the ABT corresponds to one such instance displayed below.

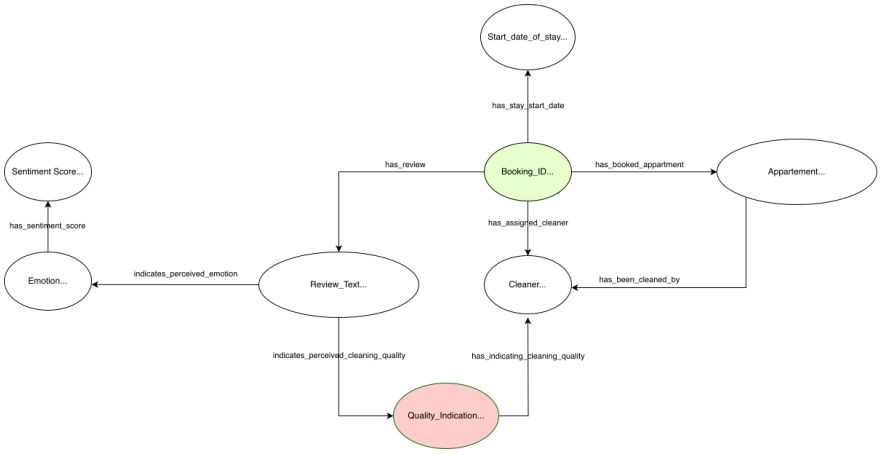


Figure 2: Description of the Knowledgegraph architecture

Side Notes: - In the initial creation phase, edges leading to the *Quality Indication* are available only for a specific subgroup (the training set). These edges are intended to be learned using *TransE*.

3.1 Technologies used

Starting out, EC2 (running *Python* and *PostgreSQL*) and *AWS Lamda* have been chosen for the data engineering side, including job scheduling and classic RDBS (using PostgreSQL as Single Source of Truth). The decision to adopt this technology suite was mainly influenced by it's general purpose, time proven quality, high scalability and wide array of utilities. In addition, it provides a strong architectural backbone for all kind of ML-Application, being it classic, or graph based machine learning, allowing them to flourish in harmony and synergy.

Furthermore *Neo4j* was then chose as a graph database for storing the built Knowledge Graph(s), while other database have been investigated, some being:

- Amazon's own solution - Neptune
- Microsoft's Azure Cosmos DB
- Dgraph
- ArangoDB
- OrientDB
- ...

While each DB provided individual advantages and disadvantages, Neo4j was convincing for this project, mainly due to it's great support for graph data structure, Cypher's amazing syntax, the efficient querying and the docker support, leading to great flexibility, solid performance, and eas of use that was really appealing. The opportunity to add Neo4j in a docker container to the existing technical infrastructure in AWS (leveraging EC2) underlines the flexibility and scalability of this technology.

In addition and out of curiosity (in Vadalog/Datalog), a **cozoDB** has also been setup (can be found in `DataLogMe`) and tested. Due to time constraints for further experiments, I nonetheless stuck to Neo4j and Cypher.

4 Analytics / Methods

4.1 (LO1) Knowledge Graph Embeddings

4.1.1 Introduction

One very important factor for customer satisfaction in this industry is the quality of the appartement cleanings. With increasing numbers of properties under management, assessing this quality can become a very time-consuming and inefficient process. So the idea here is to offer the business owners a application that helps to assess the quality of the appartement cleanings. As this is a very specific use case, a general (not fine-tuned and industry specific) model like *BERT* is assumed to be only of limited help. Therefore, a very small train-dataset (due to the time constraints) consisting of manually labels that indicate whether a review is concerned with cleaning issues, has been created and used to learn the *indicates_perceived_cleaning_quality* relationship from the original ontology with the help of TransE were the logic for the relationship connection should be the following:

$$\text{Quality Indication(Review)} = \begin{cases} \text{great cleaning quality}, & \text{If good cleaning explicitly mentioned} \\ \text{neutral cleaning quality}, & \text{If no problems mentioned in the review} \\ \text{bad cleaning quality}, & \text{Else} \end{cases}$$

Hence, the goal is to solve the following:

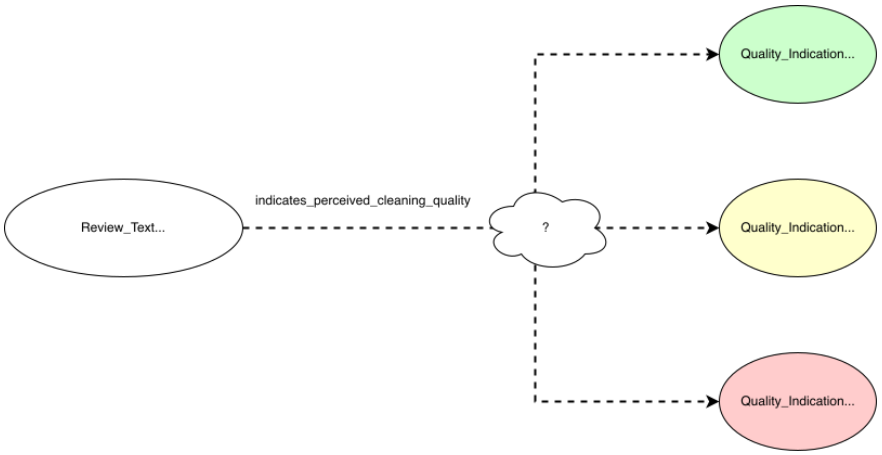


Figure 3: Goal of TransE

4.1.2 Used Model(s)

To address this problem, I chose to use Knowledge Graph (KG) Embeddings. Although I've tested only one algorithm, TransE, the code is designed with the GoF's Strategy Pattern in mind. This approach allows for the quick implementation of additional embedding algorithms available in the *PyKEEN* library, such as *TransF*, *PairRE*, *QuatE*, and many others.

For now *TransE* has been selected as suitable model and trained/learned the following way:

The implementation can be found in `src/Embeddings_Handler.py`.

4.1.3 Embeddings Results

On the very small test-set the Model yielded the following proposed connections. Although the small training set does not yield highly sophisticated results, the framework developed here provides a scalable solution that can be easily adapted for larger training and test sets, as well as for making predictions.

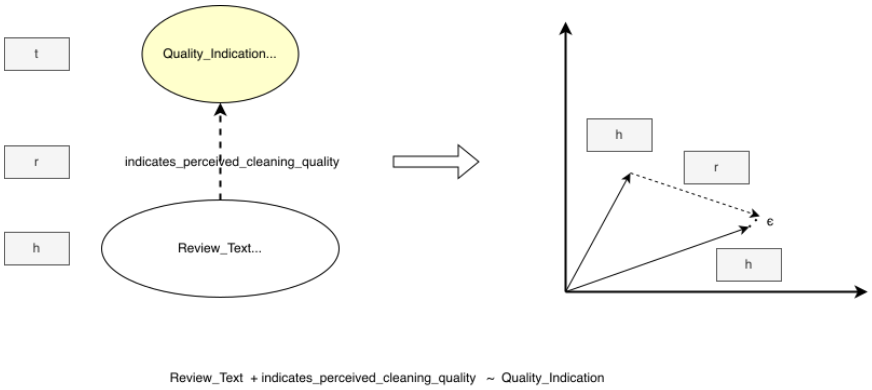


Figure 4: Training of TransE

Review_Text	Edge	Quality_Indication	score
Everything was perfect!	indicates ...	bad_cleaning_quality	-0.608245
Good for the price	indicates ...	bad_cleaning_quality	-0.475592
Great stay	indicates ...	bad_cleaning_quality	-0.642992
Great stay, thank you.	indicates ...	bad_cleaning_quality	-0.475699
Great value. We slept 4 adults in two double beds...	indicates ...	bad_cleaning_quality	-0.701354
It was a great base for my travels, thank you!	indicates ...	bad_cleaning_quality	-0.514512
Nice and comfortable place.	indicates ...	bad_cleaning_quality	-0.594642
Nice place to stay. Good value for money.	indicates ...	bad_cleaning_quality	-0.520994
Nice quiet hotel.	indicates ...	great_cleaning_quality	-0.988858
Our stay was just amazing. clean room.	indicates ...	great_cleaning_quality	-0.762633

4.2 (LO3) GNNs and the KG

4.2.1 Introduction

The initial analysis focuses on high-density regions and grouping. Specifically, I aim to determine if the entire graph can be clustered into meaningful clusters. For this task, I based my approach on the paper by Tsitsulin et.al. (2023) In this paper, the authors have compared the following different methods, including their basic properties and introduced their own Methode *Deep Modularity Networks* (**DMoN**).

Method	End-to-end	Unsup.	Node pooling	Sparse	Soft assign.	Stable	Complexity
Graclus	N	Y	Y	Y	N	Y	$O(dn + m)$
DiffPool	Y	Y	Y	N	Y	N	$O(dn^2)$
AGC	N	Y	Y	N	N	N	$O(dn^2k)$
DAEGC	N	Y	Y	Y	N	N	$O(dnk)$
SDCN	N	Y	Y	Y	N	N	$O(d^2n + m)$
NOCD	Y	Y	Y	Y	Y	Y	$O(dn + m)$
Top-k	Y	N	N	Y	N	Y	$O(dn + m)$
SAG	N	N	Y	N	N	N	$O(dn + m)$
MinCut	Y	Y	Y	Y	Y	N	$O(d^2n + m)$
DMoN	Y	Y	Y	Y	Y	Y	$O(d^2n + m)$

4.2.2 Used Model(s)

Intrigued by their claims, I wanted to test **DMoN** on my own knowledge graph. Therefore, with the help of **PyTorch Geometric** I wrote a script to run this method on my own KG. This script can be found in `src/GNN_Handler.py`.
PyTorch Geometric was chosen over other Frameworks like DGl and Graphnets due its high compatability (seamless integration into the PyTorch ecosystem), its dedicated CUDA kernels for sparse data and mini-batch, its strong community support and its research-orientation.

4.2.3 GNN Results

Unfortunate, due to time constraints, I was not able to finish this part (for now).

4.3 (LO2) Logic Based Reasoning on the KG

After testing the effectiveness of the *TransE Embeddings*, logical queries have been developed and executed to answer the analytics questions proposed in the introduction:

4.3.1 List of cleaning personal that is linked to the best/worst customer experiences:

For this purpose, I designed the following logical query:

```
// Match cleaning personnel and their associated reviews and emotions
MATCH (r:Reinigungsmitarbeiter)<-[:CLEANED_BY]-(b:Booking)-[:HAS_REVIEW]->(rev:Review)
-[:HAS_EMOTION]->(em:Emotion)
WHERE em.text IN ['joy', 'disgust'] // Filter for relevant emotions

// Aggregate emotion counts and total cleanings
WITH r, em.text AS emotion, count(em) AS emotionCount, count(DISTINCT b) AS totalCleanings
ORDER BY r.name

// Calculate joy-to-disgust ratio
WITH r,
     sum(CASE WHEN emotion = 'joy' THEN emotionCount ELSE 0 END) AS joyCount,
     sum(CASE WHEN emotion = 'disgust' THEN emotionCount ELSE 0 END) AS disgustCount,
     totalCleanings

WITH r,
     joyCount,
     disgustCount,
     totalCleanings,
     CASE WHEN disgustCount = 0 THEN joyCount ELSE joyCount * 1.0 / disgustCount END
     AS joyDisgustRatio

// Order by joy-to-disgust ratio to rank performers
ORDER BY joyDisgustRatio DESC

// Return ranked list of performers with total cleanings
RETURN r.name AS cleaner,
       joyDisgustRatio AS ratio,
       totalCleanings
```

4.3.2 List of apartments that are linked to the best/worst customer experiences.

For this purpose, I designed the following logical query:

```
// Match apartments and their associated reviews and emotions
MATCH (a:Appartment)<-[:HAS_BOOKED_APPARTEMENT]-(b:Booking)-[:HAS_REVIEW]->(rev:Review)
-[:HAS_EMOTION]->(em:Emotion)
WHERE em.text IN ['joy', 'disgust'] // Filter for relevant emotions

// Aggregate emotion counts and total bookings
WITH a,
     em.text AS emotion,
     count(em) AS emotionCount,
     count(DISTINCT b) AS totalBookings
ORDER BY a.name

// Calculate joy-to-disgust ratio
WITH a,
     sum(CASE WHEN emotion = 'joy' THEN emotionCount ELSE 0 END) AS joyCount,
     sum(CASE WHEN emotion = 'disgust' THEN emotionCount ELSE 0 END) AS disgustCount,
     totalBookings

WITH a,
     joyCount,
     disgustCount,
     totalBookings,
     CASE WHEN disgustCount = 0 THEN joyCount ELSE joyCount * 1.0 / disgustCount END
     AS joyDisgustRatio

// Order by joy-to-disgust ratio to rank apartments
ORDER BY joyDisgustRatio DESC

// Return ranked list of apartments with total bookings
RETURN a.name AS apartment,
       joyDisgustRatio AS ratio,
       totalBookings
```

4.3.2 List of apartments that are linked to the best/worst customer experiences.

For this purpose, I designed the following logical query:

```
// Match apartments and their associated reviews and emotions
MATCH (a:Appartment)<-[:HAS_BOOKED_APPARTEMENT]-(b:Booking)-[:HAS_REVIEW]->(rev:Review)
-[:HAS_EMOTION]->(em:Emotion)
WHERE em.text IN ['joy', 'disgust'] // Filter for relevant emotions

// Aggregate emotion counts and total bookings
WITH a,
     em.text AS emotion,
     count(em) AS emotionCount,
     count(DISTINCT b) AS totalBookings
```

```
ORDER BY a.name

// Calculate joy-to-disgust ratio
WITH a,
     sum(CASE WHEN emotion = 'joy' THEN emotionCount ELSE 0 END) AS joyCount,
     sum(CASE WHEN emotion = 'disgust' THEN emotionCount ELSE 0 END) AS disgustCount,
     totalBookings

WITH a,
     joyCount,
     disgustCount,
     totalBookings,
     CASE WHEN disgustCount = 0 THEN joyCount ELSE joyCount * 1.0 / disgustCount END
     AS joyDisgustRatio

// Order by joy-to-disgust ratio to rank apartments
 ORDER BY joyDisgustRatio DESC

// Return ranked list of apartments with total bookings
RETURN a.name AS apartment,
       joyDisgustRatio AS ratio,
       totalBookings
```

4.3.3 A analysis to identify if certain cleaning people or appartements became a central node in a node of dissatisfaction or form a cluster.

I identified this via:

```
// Find clusters of dissatisfaction based on negative emotions
MATCH (a:Appartment)<-[:HAS_BOOKED_APPARTEMENT]-(b:Booking)-[:HAS_REVIEW]->
(rev:Review)-[:HAS_EMOTION]->(em:Emotion)
WHERE em.text = 'disgust'
RETURN a.name AS apartment, count(DISTINCT b) AS bookingsWithDisgust
ORDER BY bookingsWithDisgust DESC;
```

```
// Similarly for cleaning personnel
MATCH (r:Reinigungsmitarbeiter)<-[:CLEANED_BY]-(b:Booking)-[:HAS_REVIEW]->
(rev:Review)-[:HAS_EMOTION]->(em:Emotion)
WHERE em.text = 'disgust'
RETURN r.name AS cleaner, count(DISTINCT b) AS bookingsWithDisgust
ORDER BY bookingsWithDisgust DESC;
```

4.3.4 Results/Summary of Logic-based Representation

The results of 4.2.1 - 4.2.3 can be viewed in a streamlit application that can be started via navigating into `src` and and running

```
python streamlit run Central_Dashboard.py
```

Additional Thoughts: While the Knowledge Graph is currently not being updated based on the results of the logic based reasoning, in the future, this might be a great extension. For example Nodes for high performing appartements or cleaning personal could be introduced to identify (or at least reason about) factors that contribute to this high performance.

4.3.5 (LO6) Thoughts on Scalable Reasoning

While these queries are very fest at small scales like those present in this project, this does generally not hold true in large scale information retrieval systems (build around KGs). As the amount of data, and thereby the size of the Knowledge Graphs increases, this could quickly lead to highly expensive computations and significantly longer execution times. In some cases, queries may even fail if the computational system can no longer provide the necessary resources. Hence, many researchers have worked on building solutions that that scale very well with increasing KG/data size.

Thereby, generally speaking, researchers focused on two big areas:

1. **The System** itself, including hardware utilization, distributed computation, efficient data storages, and computation on the low level. In this project, I have used fully dockerized solutions that can be easily deployed to highly optimized, (distributed) systems like AWS. This allows for high scaling. In addition, I used Neo4J which utilizes in-memory graph projections and parallelized graph algorithms and thereby lets me run queries very fast.
2. **The logical** queries, through the introduction of highly scalable solutions like *Vadalog*, *BOOM (Berkeley Orders Of Magnitude)* or *LogicBlox*.

For now, the system runs in reasonable time, but in case of significant Graph-growth, combining the solutions mentioned above will provide a suitable solution.

5 Results

5.1 Presentation Layer

In order to present some of the determined results, I decided to use *Streamlit* to create a small dashboard, that can then be used in a real life application as **customer satisfaction and cleaning quality monitor**. I chose *Streamlit* mainly due to its ease of use, it's excellence when it comes to rapid prototyping that still comes with very good user experience that can be designed in a typical pythonic way. The thereby built dashboard can be found under `src/dashboards/monitoring_dashboard.py`

5.2 (LO12)Reflections

The application designed above, displays the versatility of Knowledge Graphs and the vastness of possibility of interaction or even joint application with "classic" ML. For example the in **4.1.3** derived attributes can be used for further (classic) ML modelling or logic based reasining. Hence, this application displays a highly cooperative setting for all kinds of different ML-/Logic-based Reasoning & Learning, where the outputs of each model can be further processed with other models.

In the bigger picture, classic ML based Knowledge and Logic-based knowledge can interact and thereby form solutions that perform way better then each individual approach on it's own. One prominent example is are *graph-based Retrieval-Augmented Generation (RAG) systems* that use Knowledge Graphs in order to improve the performance of (large) language models and information retrieval systems. Therefore, less trustworthy language systems gain trustworthiness by a reduced danger of hallucination due to the Graph-RAG systems.

5.2.1 (LO10) Other Applications of (Financial) Knowledge Graphs

Due to my work as Lead ML Engineer at the *Austrian Federal Computing Centre / Austrian Federal Ministry of Finance* I came across many interesting and helpful *Financial KG Applications*, mainly in the area of fraud detection and prevention. Here, a typical case of tax fraud/theft is the so called *Value Added Tax Carousel* Due to the nature of this kind of fraud, it is very important to identify potential fraudulent activities before they reach their full scale. As, in order to "successfully" steal the VAT, those fraudulent companies have to be organised in large network-structures. Out of curiosity, I started a different project that is concerned with exactly those special geometric network structures for fraud detection. The up and coming project can be found here: *Don't steal my taxes*

How to run

1. Start the Neo4j database via: `docker-compose up -d`
2. Install all needed packages from the `requirements.txt`
3. Fill the Neo4j database with the (demo) data In case you want to work with the demo data, just run the `populate_KG_with_demo_data` function in `src/KG_Building_Handler.py`.

et voilà, visit http://localhost:7474/browser/ for the Neo4j Database. If you want to see the visual output of the Logic-based-reasoning, run `streamlit run Central_dashboard.py`for the *Streamlit Dashboard*. Enjoy the show.

Further functionality can be found in the `src` folder and can be easily extended/modified to each and everyones needs. Every script necessary to replicate the productive ETL System in AWS can be found in the `ETL` directory. Due to time constraints, I had to refrain from writing detailed description for this package. Especially as it can be assumed that the setup is not too complicated.