ADM Project

-the design and the development of the large-scale data management/data processing layer for a specific application

Patryk Jan Sozański

University of Genoa

February 2021

Part I
- design and development requirements

Domain

Short-term housing rental data.

Related application

The application for analysing short-term housing rental market and people's travel preferences in specific areas.

Application's details

The application is mainly used for the purpose of business intelligence, data mining and reporting.

The application inter alia analyses the decisions of tourists by searching and comparing different types of patterns (content-based filtering, collaborative filtering) in order to comprehensively meet the needs of today's travel sector, which will allow to develop businesses by evaluating the attractiveness of offers and aiding the offer creation process.

The application is read intensive and should allow for complex (due to the complex relationships between many entities) point and multipoint ad hoc queries with a reasonable execution time.

System requirements

In order to make overall processing more efficient, support availability and balance load, partitioning is needed. The partitioning can be performed taking the area as the partitioning key and thus creating partitions of size from several to several dozen MBs.

Replication is needed to improve and optimize system performance by reducing load, improving efficiency, providing fault tolerance and ensuring high availability.

Given the intensity of read operations, the system should ensure high availability.

Considering the purpose of the application, it is important to ensure data consistency since the data inconsistency can lead for instance to misinformed business decisions. However, providing it only for each subsystem, not the whole network after partitioning, shall be sufficient as each partition will accumulate an autonomous dataset and the queries will be mainly performed within it.

Details about the dataset

Datasets available on the Inside Airbnb website that is sourced from publicly available information from the Airbnb site.

Inside Airbnb has collected data on listings, hosts and reviews. The data is divided into separate datasets based on area.

All datasets have relational data model and each of them is composed of multiple data tables with many columns (up to 73 columns).

The datamodel of the presented data is not the most suitable for the application because does not support in an effective way complex relationships-based queries and it has to be changed what will be presented in the second part of the project.

The data has been analyzed, cleansed and aggregated.

The size of each dataset varies form several to several dozen MBs.

Examples of entities, relationships and typical workload queries

The datastore system should contain entities like the following examples: Country, City, Neighbourhood, Host, Listing, Property, Amenity, Review, User etc.

The above-mentioned entities shall be connected one to other by a complex net of relationships. Giving just short example of the relationships between entities, they should be like:

- between Users: isFriendTo, travelsWith,
- between a User and a Review: wrote,
- between a Review and a Listing: reviews,
- between a User and a Listing: likes,
- between a Host and a Listing: hosts,
- between a Listing and a Property: isAbout,
- between a Property and an Amenity: has,
- between a Listing and a Neighbourhood: isInNeighbourhood,

and others that will be described in the second part of the project.

Such entities and relationships shall allow to execute in an effective way complex queries like for instance generating personalized recommendations:

- content-based filtering: show all the listings in a city that are available for a number of people with whom Alice travels and have been reviewed by any of Alice's friends and additionally have an amenity that Alice's needs,

- collaborative filtering: show all the listings that have been reviewed by users who also reviewed a listing that has been reviewed by Alice,

and others that will be described in the second part of the project.

Chosen NoSQL system

The most suitable for the application NoSQL system shall be Neo4j, since it executes in the most effective way the queries based on data with many complex relationships between numerous entities such as content-based filtering or collaborative filtering, examples of which are presented above. Neo4j, in contrast to other no-graph-based NoSQL systems, allows the engine to navigate connections between nodes in constant time.

Neo4j is a CA database that guarantees high data consistency and availability by balancing load among read replicas.

Neo4j delivers the fast read performance, while still protecting the data integrity. It combines native graph storage, scalable architecture optimized for speed, and transnational compliance to ensure predictability of relationship-based queries.

In view of the above, partitioning, that is not well-supported by Neo4j, can be rejected since it is less important for the application than other characteristics of the system and it can be implemented on the application level.

Part II
- designing and developing a NoSQL data store

Conceptual schema

The figure below presents the ERD for the aforementioned required system.

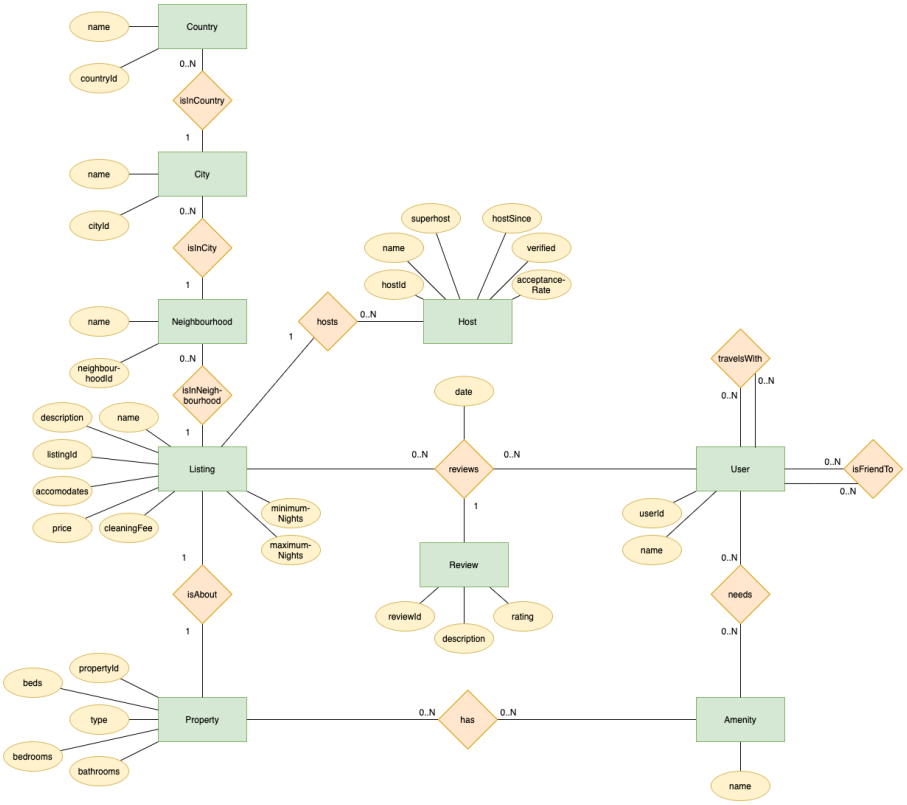

Figure 1: Conceptual schema for the domain

Such a schema has to be modified in order to be implemented as a graph data store, since the ERD includes constructs that cannot be mapped directly to a GDBS, which consists of only nodes and binary edges.

Therefore, before mapping the ERD schema to a logical schema, some of the original ERD constructs have to be adjust to semantically equivalent ones. In general, adjustments are needed for the following constructs: ternary relationships, aggregation (whole-parts) relationships, and inheritance (is-a) relationships.

In this example there is only one element that has to be adjusted - the ternary relationship between "Listing", "User" and "Review". The ternary relationship is mapped to a weak entity, with binary relations to the entities involved in the ternary relation.

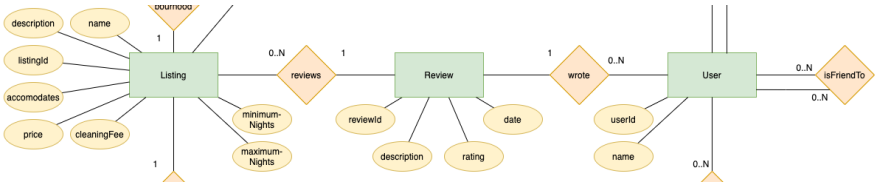

Figure 2: Fragment of the adjusted conceptual schema

Workload

1. Show the most popular neighbourhood in a city.
2. Show all the listings in a city that are hosted by superhosts.
3. Show all the listings positively reviewed (rating at least 3) by Alice.
4. Show all the amenities of the places that were positively reviewed (rating at least 3) by Alice.
5. Show all the listings in a city that were reviewed positively (rating at least 3) by the Alice's friends.
6. Show all the listings in a neighbourhood that have all the amenities that the people with whom Alice travels need.
7. Show all the listings in a city that were reviewed positively (rating at least 3) by the Alice's friends and are hosted by a verified superhost that has at least 3 years of experience.
8. Show all the listings that have been reviewed positively (rating at least 3) by users who also reviewed positively a listing that has been reviewed positively by Alice.
9. Show all the listings in a city that are available for a number of people with whom Alice travels and have been reviewed positively (rating at least 3) by any of Alice's friends and additionally have an amenity that Alice needs.
10. Show all the listings in a city that are available for a number of people with whom Alice travels and were reviewed positively (rating at least 3) by the Alice's friends and are hosted by a verified superhost that has at least 3 years of experience and have all the amenities that Alice needs.
11. Show all the listings in a city that are available for a number of people with whom Alice travels and are hosted by a verified superhost that has at least 3 years of experience and have all the amenities that Alice and the people with whom Alice travels need.

Logical schema

Based on the workload, a logical schema for representing the conceptual schema was designed. As shown below, it consists of 10 different node labels and 12 different relationship types. In the rectangles next to the nodes (or relationships) all the common properties for each label (or relationship type) are presented. Neo4j is "schema optional" - some data integrity constraints can be defined. In this case to avoid creating new redundant nodes, uniqueness constraints are defined and are represented on the logical schema by underlining.

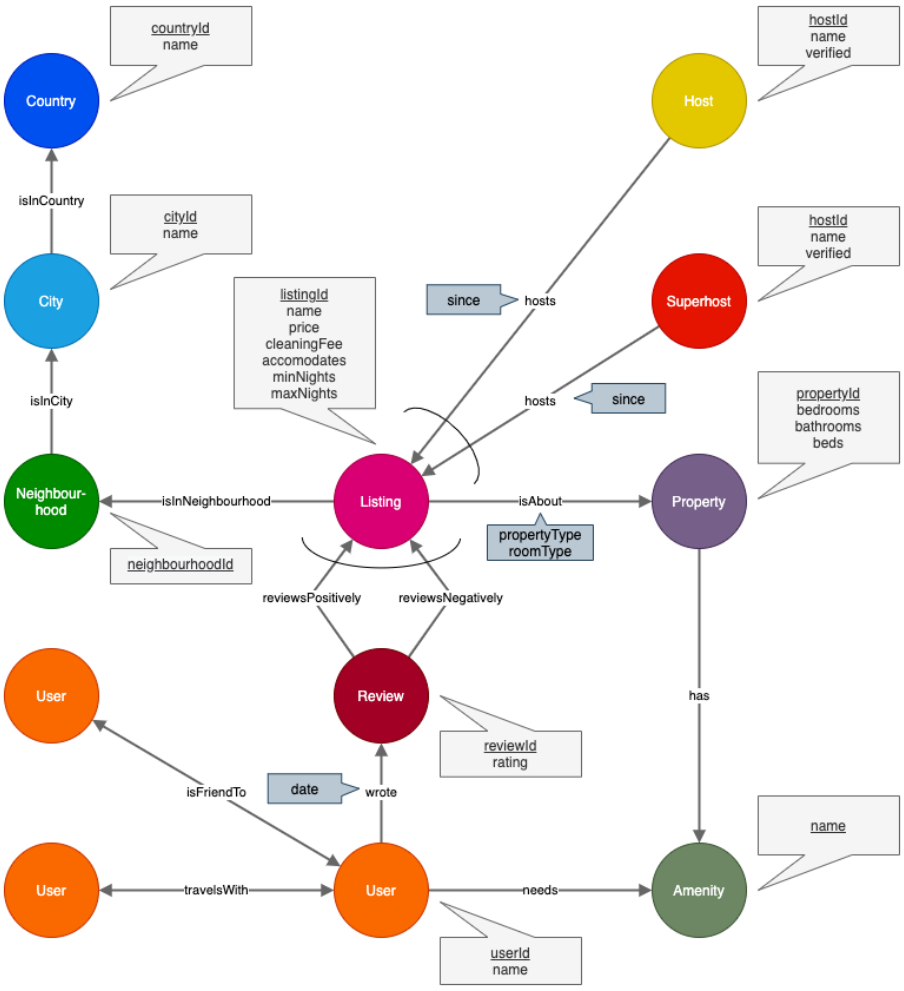

Figure 3: Logical schema for representing the conceptual schema

In order to make some queries faster, "Review" nodes can be connected to "Listing" nodes by two different types of relationships depending on the value

of the "Review.rating" property. For the same purposed "Listing" node can be connected to either a "Host" node or a "Superhost" node.

All the relationships besides "travelsWith" and "isFriendTo" are directed. Unfortunately, Neo4j doesn't support undirected relationships. However, the notion of undirected edges can be used at query time.

System configurations

Some information in the following section is just a theoretical discussion about the system configuration that could be set if the system was implemented on a cluster. Due to hardware limitations, in the next part will be presented the implementation of only one server.

Partitioning

Partitioning in Neo4j is difficult and still not well-supported. Anyway, storing related nodes on the same server is better for graph traversal, because traversing a graph when the nodes are on different machines is not good for performance.

The best solution for the system is application-level partitioning that partitions the data from the application side using domain-specific knowledge. Application-level partitioning needs to understand that nodes are stored on physically different databases.

Since the workload applies only to the nodes from one specific region, in the example of the implemented system all the nodes that relate to one city (area) shall be stored on one server.

Scalability, availability and consistency

Even though data partitioning is made on the application level, running the system on a cluster made of multiple servers that store entire graph is useful for other reasons.

One of them is load balancing that is important to maintain big throughput in read intensive, analytical applications by sending queries in parallel to different replicas.

Another reason is providing some data redundancy in order to minimize the risk of loosing the data in case of possible breakdown.

Replication also increases the system's availability and makes it possible to retrieve query results, even when a group of nodes is unavailable, by sending the request to the next available node in the cluster.

Neo4j uses master-slave replication model by configuring several slave databases to be exact replicas of a single master database. This approach makes the system horizontally scalable what results in a speed-up of read operations by enabling to handle more read load than a single node.

Neo4j is a CA datastore system and when it is run on a single node it provides transactions that by default are ACID-compliant. However, when it is run on a cluster, transactions are still atomic, consistent and durable, but writes are eventually propagated to the slaves nodes that are always available for read.

Since the application is read intensive, in the implemented datastore system writes to slave nodes are not allowed to minimize the risk of data inconsistency.

Instance of the dataset in the selected system

LOAD CSV command allows to import data from CSV files, using Cypher to define the model we want to create.

It is possible to examine CSV file without creating any instance using the command shown below.

```
LOAD CSV WITH HEADERS FROM "file:///Austin/listings.csv" AS row
RETURN row LIMIT 10
```

Figure 4: Examining the CSV file

By specifying WITH HEADERS, the row object becomes a directory. LOAD CSV allows to iterate over each row in the file, creating data for each row.

In the following section are presented all the Cypher commands used to create the instance of the dataset in Neo4j.

Listings

```
LOAD CSV WITH HEADERS FROM "file:///Austin/listings.csv" AS row
WITH row WHERE row.id IS NOT NULL
CREATE (l:Listing {listingId: row.id})
SET l.name = row.name,
    l.price = toFloat(substring(row.price, 1)),
    l.cleaningFee = toFloat(substring(row.cleaning_fee, 1)),
    l.accommodates = toInteger(row.accommodates),
    l.minNights = toInteger(row.minimum_nights),
    l.maxNights = toInteger(row.maximum_nights)
```

Figure 5: Creating "listing" nodes

```
neo4j$ CREATE INDEX ON :Listing(listingId);
```

Figure 6: Creating index on listingId

```
neo4j$ CREATE INDEX ON :Listing(price);
```

Figure 7: Creating index on price

Host

```
neo4j$ CREATE CONSTRAINT ON (h:Host) ASSERT h.hostId IS UNIQUE;
```

Figure 8: Creating unique constraint on hostId

```
LOAD CSV WITH HEADERS FROM "file:///Austin/listings.csv" AS row
WITH row WHERE row.host_id IS NOT NULL and row.host_is_superhost = "f"
MERGE (h:Host {hostId: row.host_id})
ON CREATE SET h.name      = row.host_name,
              h.about     = row.host_about,
              h.verified = CASE WHEN row.host_identity_verified = "t" THEN True ELSE False END
WITH row, h
MATCH (l:Listing {listingId: row.id})
MERGE (h)-[:hosts {since: date(row.host_since)}]→(l);
```

Figure 9: Creating "host" nodes

Superhost

```
neo4j$ CREATE CONSTRAINT ON (s:Superhost) ASSERT s.hostId IS UNIQUE;
```

Figure 10: Creating unique constraint on hostId

```
LOAD CSV WITH HEADERS FROM "file:///Austin/listings.csv" AS row
WITH row WHERE row.host_id IS NOT NULL and row.host_is_superhost = "t"
MERGE (s:Superhost {hostId: row.host_id})
ON CREATE SET s.name      = row.host_name,
              s.about     = row.host_about,
              s.verified = CASE WHEN row.host_identity_verified = "t" THEN True ELSE False END
WITH row, s
MATCH (l:Listing {listingId: row.id})
MERGE (s)-[:hosts {since: date(row.host_since)}]→(l);
```

Figure 11: Creating "superhost" nodes

Neighborhood

```
neo4j$ CREATE CONSTRAINT ON (n:Neighborhood) ASSERT n.neighborhoodId IS UNIQUE;
```

Figure 12: Creating unique constraint on neighborhoodId

```
LOAD CSV WITH HEADERS FROM "file:///Austin/listings.csv" AS row
WITH row WHERE row.id IS NOT NULL
MATCH (l:Listing {listingId: row.id})
MERGE (n:Neighborhood {neighborhoodId: coalesce(row.neighbourhood_cleansed, "NA")})
ON CREATE SET n.name = row.neighbourhood
MERGE (l)-[:isInNeighborhood]→(n);
```

Figure 13: Creating "neighborhood" nodes

City

```
neo4j$ CREATE CONSTRAINT ON (c:City) ASSERT c.cityId IS UNIQUE;
```

Figure 14: Creating unique constraint on cityId

```
LOAD CSV WITH HEADERS FROM "file:///Austin/listings.csv" AS row
WITH row WHERE row.id IS NOT NULL
MATCH (n:Neighborhood {neighborhoodId: coalesce(row.neighbourhood_cleansed, "NA")})
MERGE (c:City {cityId: coalesce(left(row.zipcode, 2), left(row.neighbourhood_cleansed, 2))})
ON CREATE SET c.name = row.city
MERGE (n)-[:isInCity]→(c);
```

Figure 15: Creating "city" nodes

Country

```
neo4j$ CREATE CONSTRAINT ON (c:City) ASSERT c.cityId IS UNIQUE;
```

Figure 16: Creating unique constraint on countryId

```
LOAD CSV WITH HEADERS FROM "file:///Austin/listings.csv" AS row
WITH row WHERE row.id IS NOT NULL
MATCH (c:City {cityId: left(row.zipcode, 2)})
MERGE (s:Country {countryId: row.country_code})
ON CREATE SET s.name = row.country
MERGE (c)-[:isInCountry]→(s);
```

Figure 17: Creating "country" nodes

Property

```
LOAD CSV WITH HEADERS FROM "file:///Austin/listings.csv" AS row
WITH row WHERE row.id IS NOT NULL
MERGE (p:Property {propertyId: row.id})
ON CREATE SET   p.bedrooms = toInteger(row.bedrooms),
                p.bathrooms = toInteger(row.bathrooms),
                p.beds =  toInteger(row.beds)
WITH row, p
MATCH (l:Listing {listingId: row.id})
MERGE (l)-[:isAbout {propertyType: row.property_type, roomType: row.room_type}]→(p);
```

Figure 18: Creating "property" nodes

```
neo4j$ CREATE INDEX ON :Property(propertyId);
```

Figure 19: Creating index on propertyId

Amenity

```
neo4j$ CREATE CONSTRAINT ON (a:Amenity) ASSERT a.name IS UNIQUE;
```

Figure 20: Creating unique constraint on name

```
LOAD CSV WITH HEADERS FROM "file:///Austin/listings.csv" AS row
WITH row WHERE row.id IS NOT NULL
MATCH (p:Property {propertyId: row.id})
WITH p, split(replace(replace(replace(row.amenities, "{", ""), "}", ""), "\"", ""), ",") AS
amenities
UNWIND amenities AS amenity
MERGE (a:Amenity {name: amenity})
MERGE (p)-[:has]→(a);
```

Figure 21: Creating "amenity" nodes

Review

```
neo4j$ CREATE CONSTRAINT ON (r:Review) ASSERT r.reviewId IS UNIQUE;
```

Figure 22: Creating unique constraint on reviewId

```
LOAD CSV WITH HEADERS FROM "file:///Austin/reviews.csv" AS row
CREATE (r:Review {reviewId: row.id})
SET r.rating = round(((rand() * 10) / 2))
```

Figure 23: Creating "review" nodes

```
LOAD CSV WITH HEADERS FROM "file:///Austin/reviews.csv" AS row
MATCH (r:Review {reviewId: row.id})
WHERE r.rating ⩾ 3
WITH row, r
MATCH (l:Listing {listingId: row.listing_id})
MERGE (r)-[:reviewsPositively]→(l)
```

Figure 24: Creating "positivelyReviews" relationships

```
LOAD CSV WITH HEADERS FROM "file:///Austin/reviews.csv" AS row
MATCH (r:Review {reviewId: row.id})
WHERE r.rating < 3
WITH row, r
MATCH (l:Listing {listingId: row.listing_id})
MERGE (r)-[:reviewsNegatively]→(l)
```

Figure 25: Creating "negativelyReviews" relationships

User

```
neo4j$ CREATE CONSTRAINT ON (u:User) ASSERT u.userId IS UNIQUE;
```

Figure 26: Creating unique constraint on userId

```
LOAD CSV WITH HEADERS FROM "file:///Austin/reviews.csv" AS row
MERGE (u:User {userId: row.reviewer_id})
SET u.name = row.reviewer_name
WITH row, u
MATCH (r:Review {reviewId: row.id})
MERGE (u)-[:wrote {date: date(row.date)}]→(r);
```

Figure 27: Creating "user" nodes

```
MATCH (u:User), (a:Amenity)
WITH COLLECT(u) AS us, COLLECT(a) AS aa
WITH apoc.coll.randomItems(us, 1300) AS us, apoc.coll.randomItem(aa) AS a
FOREACH(u in us | CREATE (u)-[:needs]→(a));
```

Figure 28: Creating "needs" relationships

```
neo4j$ MATCH (a:Amenity)←[x:needs]-(u:User)-[:needs]→(a) DELETE x
```

Figure 29: Deleting doubled "needs" relationships

```
MATCH (u1:User), (u2:User)
WITH u1, u2
WHERE rand() < 0.01
MERGE (u1)←[:isFriendTo]→(u2);
```

Figure 30: Creating "isFriendTo" relationships

```
neo4j$ MATCH (u1)-[x:isFriendTo]-(u2)-[:isFriendTo]-(u1) DELETE x
```

Figure 31: Deleting doubled "isFriendTo" relationships

```
neo4j$ MATCH (u1)-[x:isFriendTo]-(u1) DELETE x
```

Figure 32: Deleting "isFriendTo" relationships between the same node

At the end "travelsWith" relationships were created following the same steps as in the case of "isFriendTo" relationships.

After all the steps the datamodel was verified using CALL db.schema.visualization() command.

Figure 33: Implemented datamodel

Specifying and implementing the workload's operations in the language supported by the system

1

Show the most popular neighbourhood in a city.

```
MATCH ()-[:isInNeighborhood]→(n)-[:isInCity]→({name: "Austin"})
WITH n, COUNT(*) AS num ORDER BY num DESC
RETURN n.neighborhoodId, n.name, num LIMIT 1
```

Figure 34: Query no. 1

	n.neighborhoodId	n.name	num
1	"78704"	*null*	1601

Figure 35: Result no. 1

2

Show all the listings in a city that are hosted by superhosts.

```
MATCH (:Superhost)-[:hosts]→(l)-[:isInNeighborhood]→()-[:isInCity]→({name: "Austin"})
RETURN l
```

Figure 36: Query no. 2

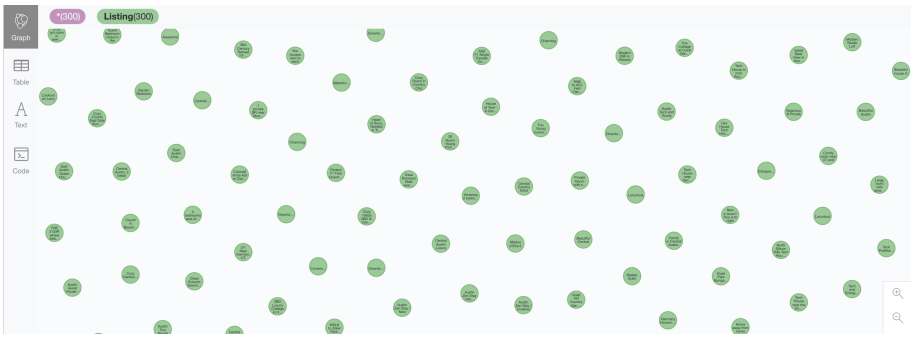

Figure 37: Result no. 2

3

Show all the listings positively reviewed (rating at least 3) by Alice.

```
MATCH (u:User {userId: "2796131"})-[:wrote]-()-[:reviewsPositively]→(l)
RETURN l
```

Figure 38: Query no. 3

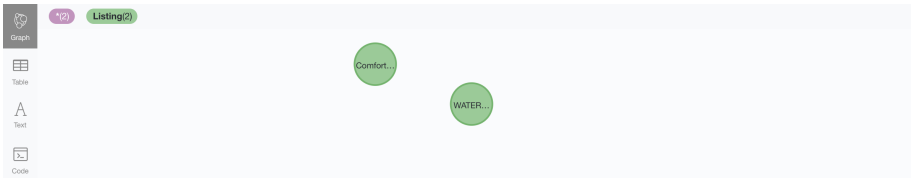

Figure 39: Result no. 3

4

Show all the amenities of the places that were positively reviewed (rating at least 3) by Alice.

```
MATCH (u:User {userId: "2796131"})-[:wrote]-()-[:reviewsPositively]→()-[:isAbout]→()-[:has]→(a)
RETURN a.name
```

Figure 40: Query no. 4

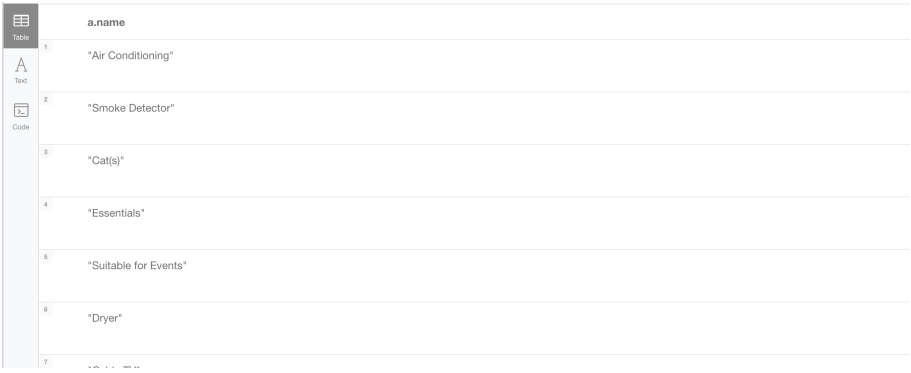

Figure 41: Result no. 4

5

Show all the listings in a city that were reviewed positively (rating at least 3) by the Alice's friends.

```
MATCH (u:User {userId: "2796131"})-[:isFriendTo]-()-[:wrote]→()-[:reviewsPositively]→(l)
RETURN l
```

Figure 42: Query no. 5

Figure 43: Result no. 5

6

Show all the listings in a neighbourhood that have all the amenities that the people with whom Alice travels need.

```
MATCH (:User {userId: "2796131"})-[:travelsWith]-()-[:needs]-(n)
WITH COLLECT(n) as needs
MATCH (:Neighborhood {name: "Circle C"})←[:isInNeighborhood]-(l)-[:isAbout]→(p)
WHERE ALL(n IN needs WHERE (p)-[:has]→(n))
RETURN l
```

Figure 44: Query no. 6

Figure 45: Result no. 6

7

Show all the listings in a city that were reviewed positively (rating at least 3) by the Alice's friends and are hosted by a verified superhost that has at least 3 years of experience.

```
MATCH (:User {userId: "2796131"})-[:isFriendTo]-()-[:wrote]→()-[:reviewsPositively]→(l)←
[x:hosts]-(:Superhost {verified: True})
WHERE date().year - date(x.since).year > 3
RETURN l
```

Figure 46: Query no. 7

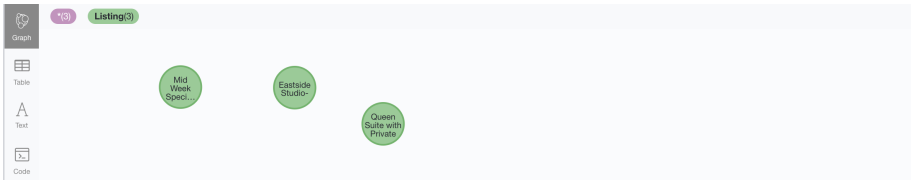

Figure 47: Result no. 7

8

Show all the listings that have been reviewed positively (rating at least 3) by users who also reviewed positively a listing that has been reviewed positively by Alice.

```
MATCH (:User {userId: "2796131"})-[:wrote]→()-[:reviewsPositively]→()←[:reviewsPositively]-()←
[:wrote]-()-[:wrote]→()-[:reviewsPositively]→(l)
RETURN l
```

Figure 48: Query no. 8

Figure 49: Result no. 8

9

Show all the listings in a city that are available for a number of people with whom Alice travels and have been reviewed positively (rating at least 3) by any of Alice's friends and additionally have an amenity that Alice needs.

```
MATCH (u:User {userId: "2796131"})-[:travelsWith]-(p)
WITH COUNT(p) AS num
MATCH (a)←[:needs]-(u:User {userId: "2796131"})
WITH num, a
MATCH (a)←[:has]-()←[:isAbout]-(l)-[:isInNeighborhood]→()-[:isInCity]→({name: "Austin"})
WHERE l.accommodates > num
WITH l
MATCH (l)←[:reviewsPositively]-()←[:wrote]-()-[:isFriendTo]-(u:User {userId: "2796131"})
RETURN l
```

Figure 50: Query no. 9

Figure 51: Result no. 9

10

Show all the listings in a city that are available for a number of people with whom Alice travels and were reviewed positively (rating at least 3) by the Alice's friends and are hosted by a verified superhost that has at least 3 years of experience and have all the amenities that Alice needs.

```
MATCH (u:User {userId: "2796131"})-[:travelsWith]-(p)
WITH COUNT(p) AS num
MATCH (a)<-[:needs]-(u:User {userId: "2796131"})
WITH num, COLLECT(a) AS aa
MATCH (l)-[:isInNeighborhood]->()-[:isInCity]->({name: "Austin"})
WHERE l.accommodates > num AND ALL(a IN aa WHERE (a)<-[:has]-()<-[:isAbout]-(l))
WITH l
MATCH (l)<-[:reviewsPositively]-()<-[:wrote]-()-[:isFriendTo]-(u:User {userId: "2796131"})
WITH l
MATCH (l)<-[x:hosts]-(:Superhost {verified: True})
WHERE date().year - date(x.since).year > 3
RETURN l
```

Figure 52: Query no. 10

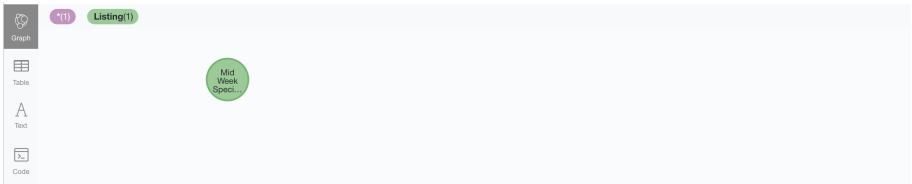

Figure 53: Result no. 10

11

Show all the listings in a city that are available for a number of people with whom Alice travels and are hosted by a verified superhost that has at least 3 years of experience and have all the amenities that Alice and the people with whom Alice travels need.

```
MATCH (:User {userId: "2796131"})-[:travelsWith]-(p)
WITH COUNT(p) AS num
MATCH (a)←[:needs]-(:User {userId: "2796131"})-[:travelsWith]-()-[:needs]→(x)
WITH num, COLLECT(x) + COLLECT(a) AS ax
MATCH (:Superhost {verified: True})-[x:hosts]→(l)-[:isInNeighborhood]→()-[:isInCity]→({name:
"Austin"})
WHERE l.accommodates > num AND ALL(a IN ax WHERE (a)←[:has]-()←[:isAbout]-(l)) AND date().year -
date(x.since).year > 3
RETURN l
```

Figure 54: Query no. 11

Figure 55: Result no. 11