Concordia University
Department of Computer Science and Software Engineering

Analyzing Car Accidents Using Relational and NoSQL Databases
A Data-Driven Study on the Correlation Between Weather and Accidents

SOEN 363 – Data Systems for Software Engineers
Instructor: Ali Jannatpour
Winter 2025

Team AVM Query

Manel Hellou – 40284245

Ana Rostam – 40187433

Victor Bruson – 40284702

Ryo Sato – 40279749

Team's Github repository

Date of Submission: April 14, 2025

Introduction

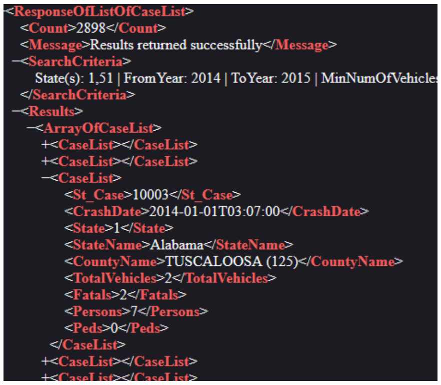
This project explores the correlation between weather conditions and fatal car accidents in the U.S. during 2010 and 2011, aiming to determine whether severe weather leads to more accidents by integrating data through both relational and NoSQL databases.

We collected real-world data from two APIs: one for accident reports and another for historical weather. The data was stored using PostgreSQL for structured relational analysis, and Neo4j for visualizing and querying relationships between entities like weather, location, and accident type.

PostgreSQL was chosen for its robust support for structured data and complex queries, while Neo4j was used for its strength in uncovering patterns across connected data. This project also gave us hands-on experience in data integration, modeling, and analysis using both relational and graph-based approaches.

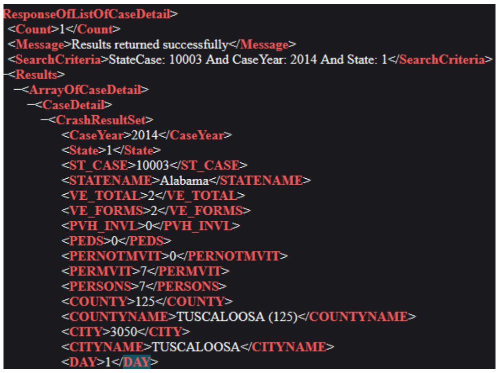
Data Sources

The first data source is the national highway traffic safety administration. We can either get a general list of crash:

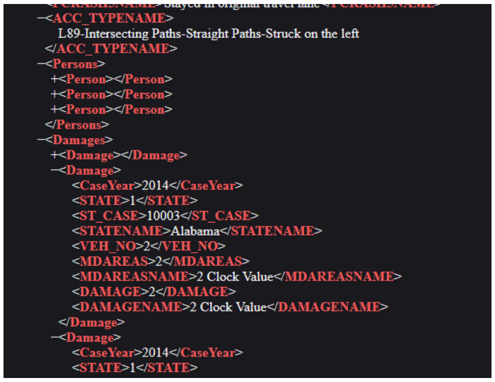


```
<ResponseOfListOfCaseList>
  <Count>2898</Count>
  <Message>Results returned successfully</Message>
 –<SearchCriteria>
    State(s): 1,51 | FromYear: 2014 | ToYear: 2015 | MinNumOfVehicles
  </SearchCriteria>
 –<Results>
   –<ArrayOfCaseList>
     +<CaseList></CaseList>
     +<CaseList></CaseList>
     –<CaseList>
        <St_Case>10003</St_Case>
        <CrashDate>2014-01-01T03:07:00</CrashDate>
        <State>1</State>
        <StateName>Alabama</StateName>
        <CountyName>TUSCALOOSA (125)</CountyName>
        <TotalVehicles>2</TotalVehicles>
        <Fatals>2</Fatals>
        <Persons>7</Persons>
        <Peds>0</Peds>
      </CaseList>
     +<CaseList></CaseList>
     +<CaseList></CaseList>
```

Or complete detail of a crash like the vehicles inside, the people inside and the crash condition:



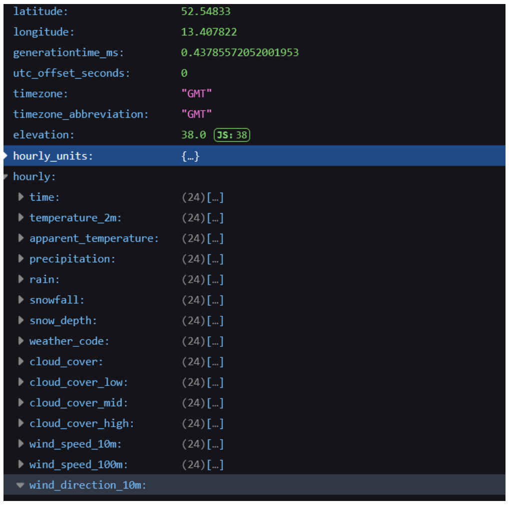
```
ResponseOfListOfCaseDetail>
 <Count>1</Count>
 <Message>Results returned successfully</Message>
 <SearchCriteria>StateCase: 10003 And CaseYear: 2014 And State: 1</SearchCriteria>
–<Results>
  –<ArrayOfCaseDetail>
    –<CaseDetail>
      –<CrashResultSet>
          <CaseYear>2014</CaseYear>
          <State>1</State>
          <ST_CASE>10003</ST_CASE>
          <STATENAME>Alabama</STATENAME>
          <VE_TOTAL>2</VE_TOTAL>
          <VE_FORMS>2</VE_FORMS>
          <PVH_INVL>0</PVH_INVL>
          <PEDS>0</PEDS>
          <PERNOTMVIT>0</PERNOTMVIT>
          <PERMVIT>7</PERMVIT>
          <PERSONS>7</PERSONS>
          <COUNTY>125</COUNTY>
          <COUNTYNAME>TUSCALOOSA (125)</COUNTYNAME>
          <CITY>3050</CITY>
          <CITYNAME>TUSCALOOSA</CITYNAME>
          <DAY>1</DAY>
```



```
    <PCRASH5NAME>Stayed in original travel lane</PCRASH5NAME>
  −<ACC_TYPENAME>
      L89-Intersecting Paths-Straight Paths-Struck on the left
   </ACC_TYPENAME>
  −<Persons>
     +<Person></Person>
     +<Person></Person>
     +<Person></Person>
   </Persons>
  −<Damages>
     +<Damage></Damage>
     −<Damage>
          <CaseYear>2014</CaseYear>
          <STATE>1</STATE>
          <ST_CASE>10003</ST_CASE>
          <STATENAME>Alabama</STATENAME>
          <VEH_NO>2</VEH_NO>
          <MDAREAS>2</MDAREAS>
          <MDAREASNAME>2 Clock Value</MDAREASNAME>
          <DAMAGE>2</DAMAGE>
          <DAMAGENAME>2 Clock Value</DAMAGENAME>
       </Damage>
     −<Damage>
          <CaseYear>2014</CaseYear>
          <STATE>1</STATE>
```

The DML for the API is the getCrashDataV2.mjs file. I gets a list of the crashes and goes accident per accident and saves the right data in the corresponding table. When the data is inserted, the id is returned to create the relationship table.

For the second source, we chose the historical weather api from open-meteo.com. To get the request we send the location and time and the api respond with the the hourly info for the day:



```
latitude:                 52.54833
longitude:                13.407822
generationtime_ms:        0.43785572052001953
utc_offset_seconds:       0
timezone:                 "GMT"
timezone_abbreviation:    "GMT"
elevation:                38.0 JS: 38
▸ hourly_units:           {…}
▾ hourly:
  ▸ time:                   (24)[…]
  ▸ temperature_2m:         (24)[…]
  ▸ apparent_temperature:   (24)[…]
  ▸ precipitation:          (24)[…]
  ▸ rain:                   (24)[…]
  ▸ snowfall:               (24)[…]
  ▸ snow_depth:             (24)[…]
  ▸ weather_code:           (24)[…]
  ▸ cloud_cover:            (24)[…]
  ▸ cloud_cover_low:        (24)[…]
  ▸ cloud_cover_mid:        (24)[…]
  ▸ cloud_cover_high:       (24)[…]
  ▸ wind_speed_10m:         (24)[…]
  ▸ wind_speed_100m:        (24)[…]
  ▾ wind_direction_10m:
```

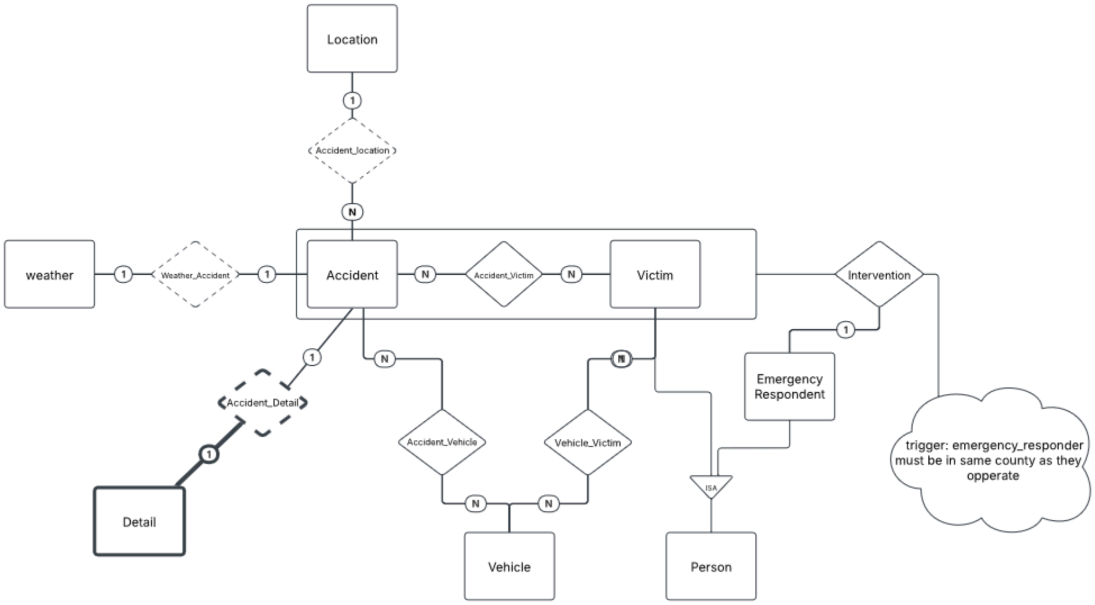
In the weather dml file, the file queries for all the accident without weather and do a call for each and insert the data for each hour.

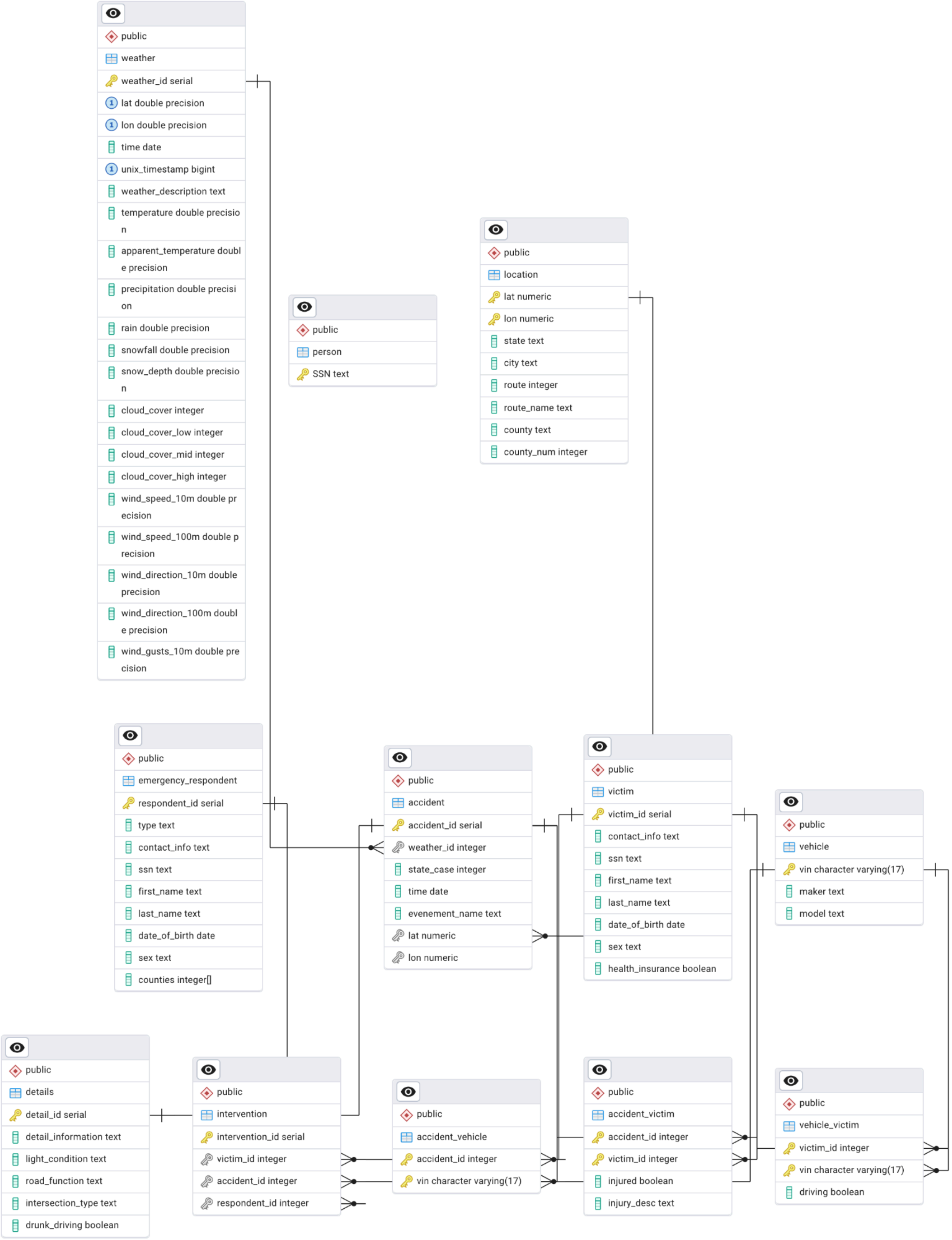
Link between the two

When both tables are filled, a query is ran to find the matching weather (by lat,lon and time) in the weather table and add the weather id to the accident.

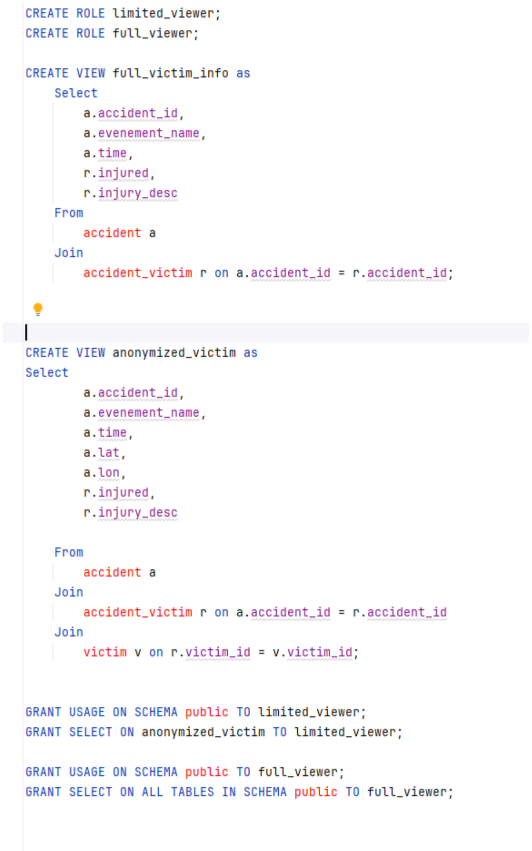
ER Data Model

In this figure, you can see the general architecture of our relational database.

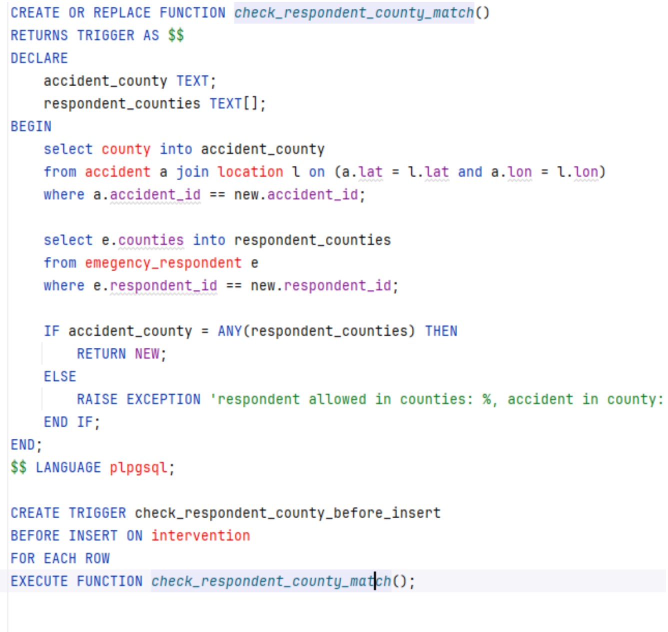

Location
1
Accident_location
N
weather
1
Weather_Accident
1
Accident
N
Accident_Victim
N
Victim
Intervention
1
1
Accident_Detail
1
Detail
N
N
Emergency
Respondent
trigger: emergency_responder
must be in same county as they
opperate
Accident_Vehicle
Vehicle_Victim
N
N
ISA
Vehicle
Person



The dashed diamond are relationship tables that are created using the one table approach. Since they are either one to one or one to many relations, it was easy to integrate with reference keys. For in the three cases Accident has foreign keys that link to the correct location and weather. Furthermore, the detail has the foreign key of the accident for primary key as there is alwas one detail for each accident and it is a weak entity relation. For the relationship between victim, accident and vehicle, we decided to go with the three binary relation instead of ternary because the relation doesnt always hav the three entities, sometime a vehicle hit is empty or a vehicle hit a pedestrian (not in a car). Since a PK cannot be null the ternary relationship is not optimal. We implemented a ISA relationship with person which can be either a emergency_responder or victim. Since people in our database are always either, we set the person table as a template. For the aggregation, the entity emergency_respondent is linked to the relation victim_accident. In this aggregation we added a trigger where we have a function that verifies if the respondent operate in the county of the accident. If not the emergency respondent cannot be in the intervention (hypothetical legislations and law). We also created two view to see the accident and injury, one anonymized that joins only the injury and accident and the full view that joins the victim too. For some reason the full view doesnt have victim info in the select but it is intended to be there.



```sql
CREATE ROLE limited_viewer;
CREATE ROLE full_viewer;

CREATE VIEW full_victim_info as
    Select
        a.accident_id,
        a.evenement_name,
        a.time,
        r.injured,
        r.injury_desc
    From
        accident a
    Join
        accident_victim r on a.accident_id = r.accident_id;


CREATE VIEW anonymized_victim as
Select
        a.accident_id,
        a.evenement_name,
        a.time,
        a.lat,
        a.lon,
        r.injured,
        r.injury_desc

    From
        accident a
    Join
        accident_victim r on a.accident_id = r.accident_id
    Join
        victim v on r.victim_id = v.victim_id;


GRANT USAGE ON SCHEMA public TO limited_viewer;
GRANT SELECT ON anonymized_victim TO limited_viewer;

GRANT USAGE ON SCHEMA public TO full_viewer;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO full_viewer;
```



```sql
CREATE OR REPLACE FUNCTION check_respondent_county_match()
RETURNS TRIGGER AS $$
DECLARE
    accident_county TEXT;
    respondent_counties TEXT[];
BEGIN
    select county into accident_county
    from accident a join location l on (a.lat = l.lat and a.lon = l.lon)
    where a.accident_id == new.accident_id;

    select e.counties into respondent_counties
    from emegency_respondent e
    where e.respondent_id == new.respondent_id;

    IF accident_county = ANY(respondent_counties) THEN
        RETURN NEW;
    ELSE
        RAISE EXCEPTION 'respondent allowed in counties: %, accident in county:
    END IF;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER check_respondent_county_before_insert
BEFORE INSERT ON intervention
FOR EACH ROW
EXECUTE FUNCTION check_respondent_county_match();
```

NoSQL Data Model

During our transition from PostgreSQL to Neo4j, we restructured our relational schema into a graph-based model, in which entities were modeled as node labels, tuples as nodes with properties and relationships between entities became edges.

The overall data structure remained conceptually similar to the relational model, but we simplified and omitted certain constructs, listed below, to better suit the graph-based model:

- Removal of the ISA relationship

In the relational database, the victim and emergency_respondent entities were children of the person entity. In the graph-based database, the person entity is no longer modeled; only victim and emergency_respondent remain and they are treated as distinct node labels, each with their own attributes.

- Storage of weak entity inside main entity

The details entity was modeled as a weak entity in an identifying relationship with the accident entity in the relational database. In Neo4j, we merged the contents of details directly into the accident node. As shown in Figure #, attributes that were formerly part of the details table (e.g., detail_information, light_condition, road_function, etc.) are now stored as direct attributes of the accident node.

In regards to relationships, all former relationships, whether they were modeled using the 1-table approach or the 2-table approach, have been converted to edges in Neo4j.

- 1-table relationship example

The relationship between accident and weather was modeled by storing the primary key of weather (weather_id) as a foreign key inside the accident table. Now that relationship is represented by a direct edge between the accident and weather nodes (see Figure #).

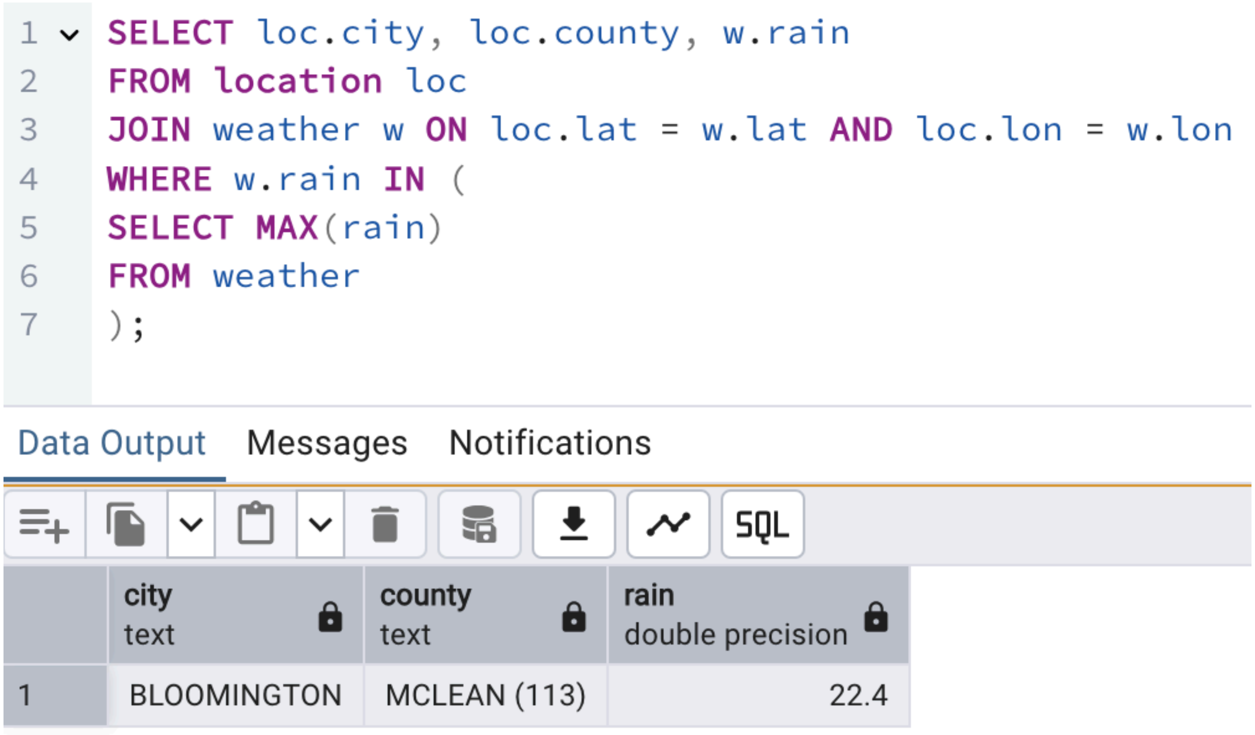
- 2-table relationship example

The relationship between accident and vehicle was made by storing the primary key of each entity in a separate table. In Neo4j, this relationship is modeled by a direct edge between the accident and vehicle nodes (see Figure #).

Demonstrating Major Queries

Queries from the relational database

- Sample join query

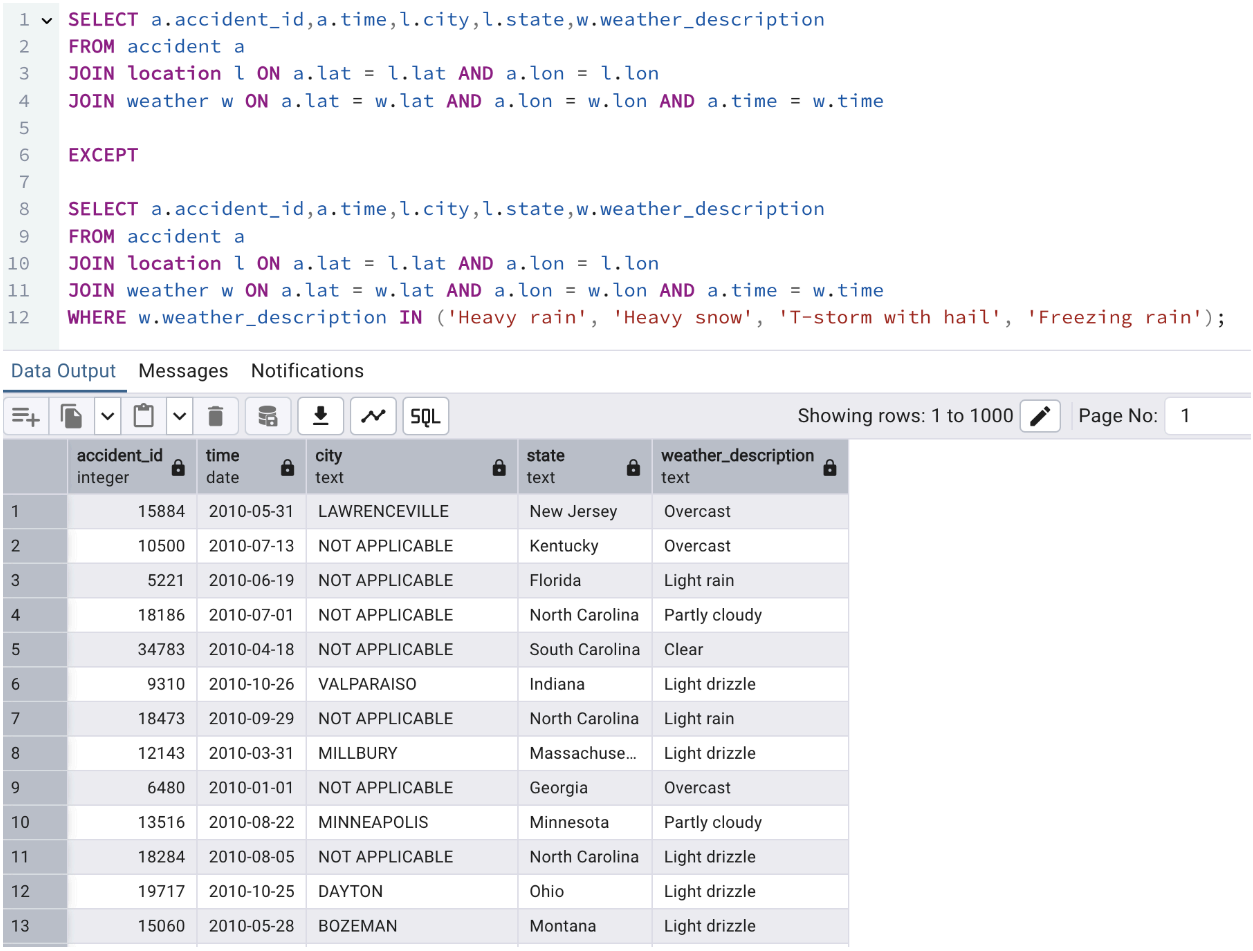


```sql
SELECT loc.city, loc.county, w.rain
FROM location loc
JOIN weather w ON loc.lat = w.lat AND loc.lon = w.lon
WHERE w.rain IN (
SELECT MAX(rain)
FROM weather
);
```

	city text	county text	rain double precision
1	BLOOMINGTON	MCLEAN (113)	22.4

This query retrieves the city and county names along with the rainfall amount from the location and weather table for the location that recorded the highest amount of rainfall.
It does so by performing a JOIN between the two tables on their common coordinates (latitude and longitude).
The WHERE clause filters the results to only get the one where the rainfall value matches the maximum rainfall value obtained from the subquery.
This query is useful to identify extreme weather conditions in specific areas.

- Sample set operation query



```sql
SELECT a.accident_id,a.time,l.city,l.state,w.weather_description
FROM accident a
JOIN location l ON a.lat = l.lat AND a.lon = l.lon
JOIN weather w ON a.lat = w.lat AND a.lon = w.lon AND a.time = w.time

EXCEPT

SELECT a.accident_id,a.time,l.city,l.state,w.weather_description
FROM accident a
JOIN location l ON a.lat = l.lat AND a.lon = l.lon
JOIN weather w ON a.lat = w.lat AND a.lon = w.lon AND a.time = w.time
WHERE w.weather_description IN ('Heavy rain', 'Heavy snow', 'T-storm with hail', 'Freezing rain');
```

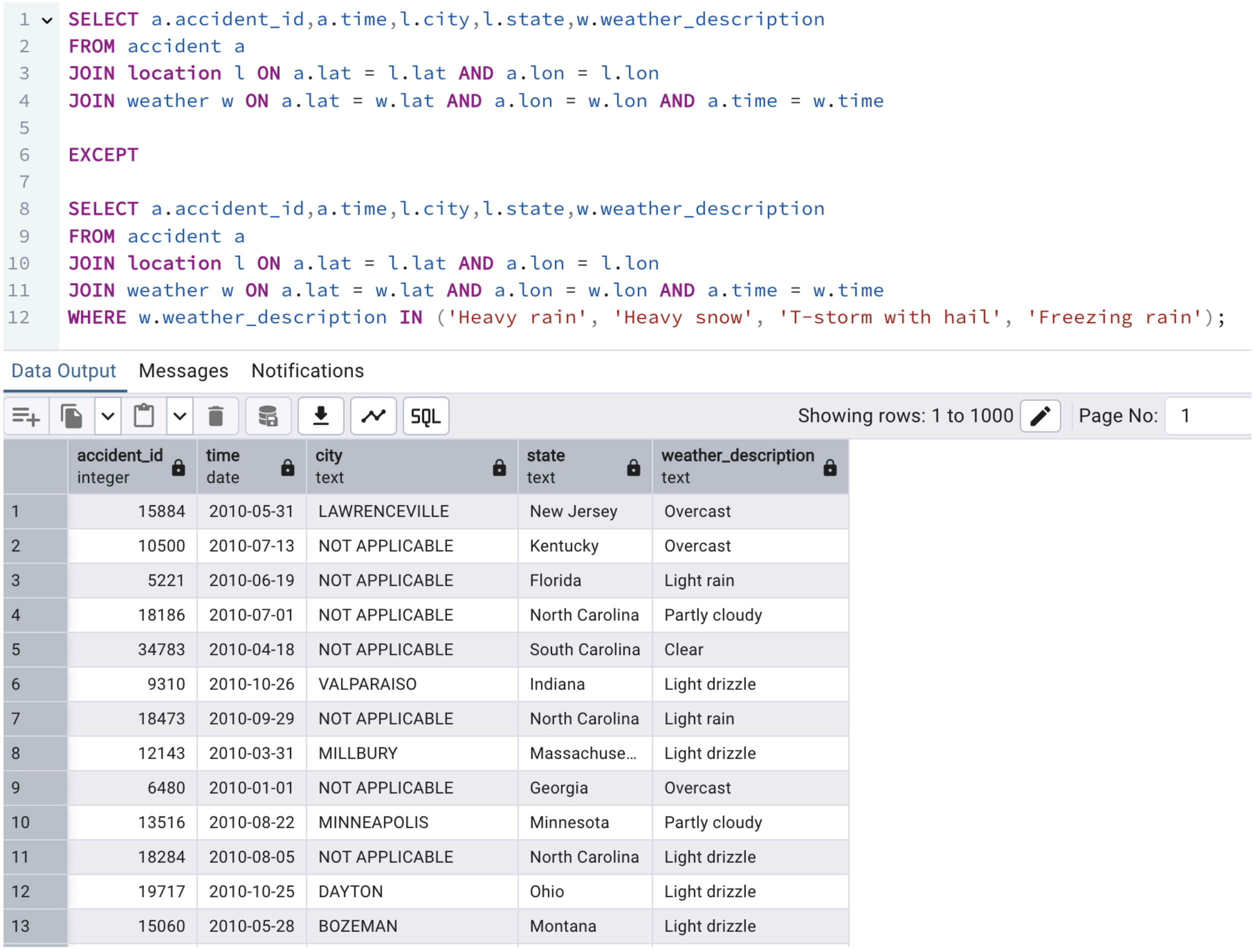
	accident_id integer	time date	city text	state text	weather_description text
1	15884	2010-05-31	LAWRENCEVILLE	New Jersey	Overcast
2	10500	2010-07-13	NOT APPLICABLE	Kentucky	Overcast
3	5221	2010-06-19	NOT APPLICABLE	Florida	Light rain
4	18186	2010-07-01	NOT APPLICABLE	North Carolina	Partly cloudy
5	34783	2010-04-18	NOT APPLICABLE	South Carolina	Clear
6	9310	2010-10-26	VALPARAISO	Indiana	Light drizzle
7	18473	2010-09-29	NOT APPLICABLE	North Carolina	Light rain
8	12143	2010-03-31	MILLBURY	Massachuse...	Light drizzle
9	6480	2010-01-01	NOT APPLICABLE	Georgia	Overcast
10	13516	2010-08-22	MINNEAPOLIS	Minnesota	Partly cloudy
11	18284	2010-08-05	NOT APPLICABLE	North Carolina	Light drizzle
12	19717	2010-10-25	DAYTON	Ohio	Light drizzle
13	15060	2010-05-28	BOZEMAN	Montana	Light drizzle

This query starts by selecting accident information (accident_id, time, city, state and weather_description) by joining the accident, location and weather table using matching latitude, longitude and time values.

The query then uses the EXCEPT clause to exclude any accident that happened during severe weather conditions ('Heavy rain', 'Heavy snow', 'T-storm with hail', 'Freezing rain').

The final result is a list of accidents that occurred during normal weather conditions, helping highlight cases where severe weather was not a contributing factor to accidents.

- Sample view



```sql
SELECT a.accident_id,a.time,l.city,l.state,w.weather_description
FROM accident a
JOIN location l ON a.lat = l.lat AND a.lon = l.lon
JOIN weather w ON a.lat = w.lat AND a.lon = w.lon AND a.time = w.time

EXCEPT

SELECT a.accident_id,a.time,l.city,l.state,w.weather_description
FROM accident a
JOIN location l ON a.lat = l.lat AND a.lon = l.lon
JOIN weather w ON a.lat = w.lat AND a.lon = w.lon AND a.time = w.time
WHERE w.weather_description IN ('Heavy rain', 'Heavy snow', 'T-storm with hail', 'Freezing rain');
```

	accident_id integer	time date	city text	state text	weather_description text
1	15884	2010-05-31	LAWRENCEVILLE	New Jersey	Overcast
2	10500	2010-07-13	NOT APPLICABLE	Kentucky	Overcast
3	5221	2010-06-19	NOT APPLICABLE	Florida	Light rain
4	18186	2010-07-01	NOT APPLICABLE	North Carolina	Partly cloudy
5	34783	2010-04-18	NOT APPLICABLE	South Carolina	Clear
6	9310	2010-10-26	VALPARAISO	Indiana	Light drizzle
7	18473	2010-09-29	NOT APPLICABLE	North Carolina	Light rain
8	12143	2010-03-31	MILLBURY	Massachuse...	Light drizzle
9	6480	2010-01-01	NOT APPLICABLE	Georgia	Overcast
10	13516	2010-08-22	MINNEAPOLIS	Minnesota	Partly cloudy
11	18284	2010-08-05	NOT APPLICABLE	North Carolina	Light drizzle
12	19717	2010-10-25	DAYTON	Ohio	Light drizzle
13	15060	2010-05-28	BOZEMAN	Montana	Light drizzle

In this query, a view called severe_weather_accidents is created to simplify access to accidents that occurred during severe weather conditions.

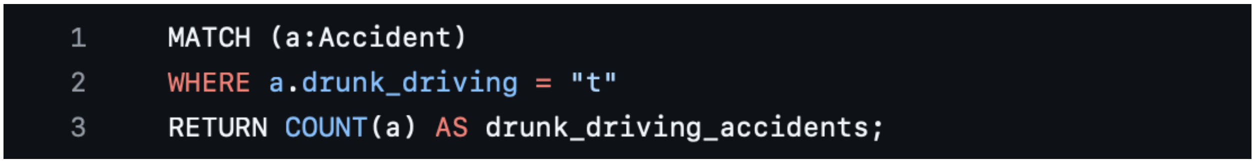
The view joins the accident, location and weather tables using common coordinates (latitude and longitude) and returns information about the accidents (accident_id, time, city, state, county and weather_description).

The WHERE clause filters accidents that happened during severe weather conditions ('Heavy rain', 'Heavy snow', 'T-storm with hail', 'Freezing rain').

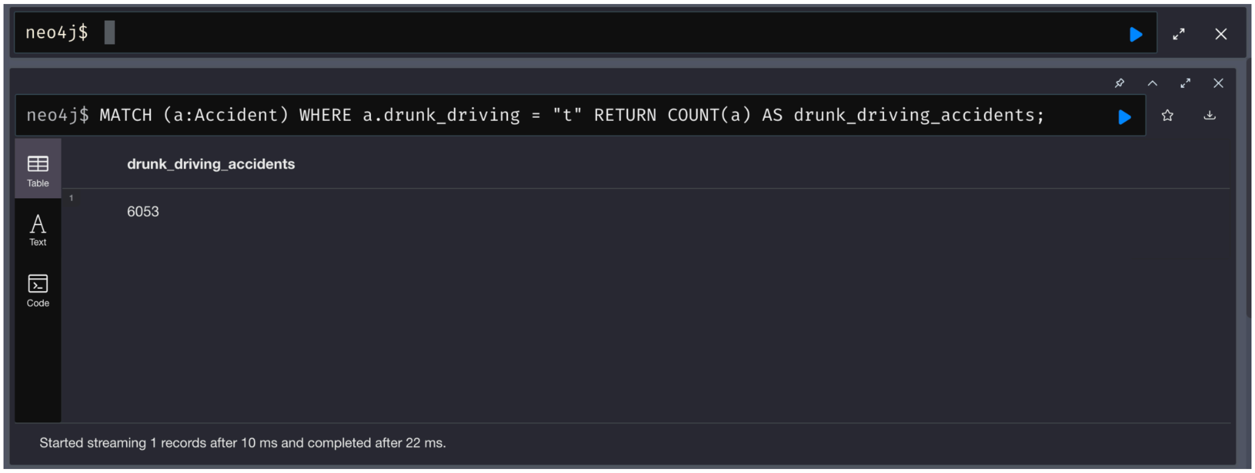
After creating the view, the “SELECT * FROM severe_weather_accidents” statement is used to display its content.

Queries from the graph-based database

- Aggregate data



```
MATCH (a:Accident)
WHERE a.drunk_driving = "t"
RETURN COUNT(a) AS drunk_driving_accidents;
```



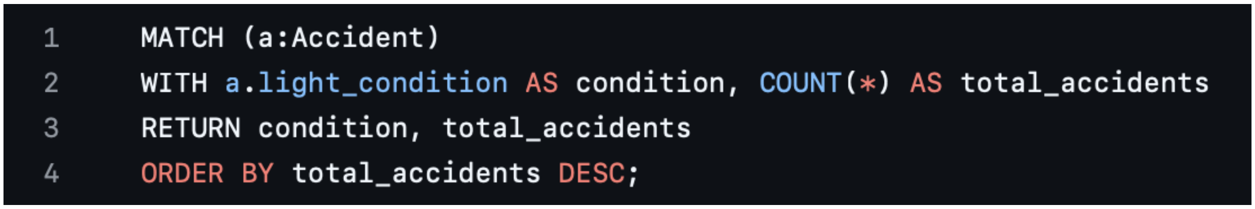
```
neo4j$

neo4j$ MATCH (a:Accident) WHERE a.drunk_driving = "t" RETURN COUNT(a) AS drunk_driving_accidents;
```

	drunk_driving_accidents
1	6053

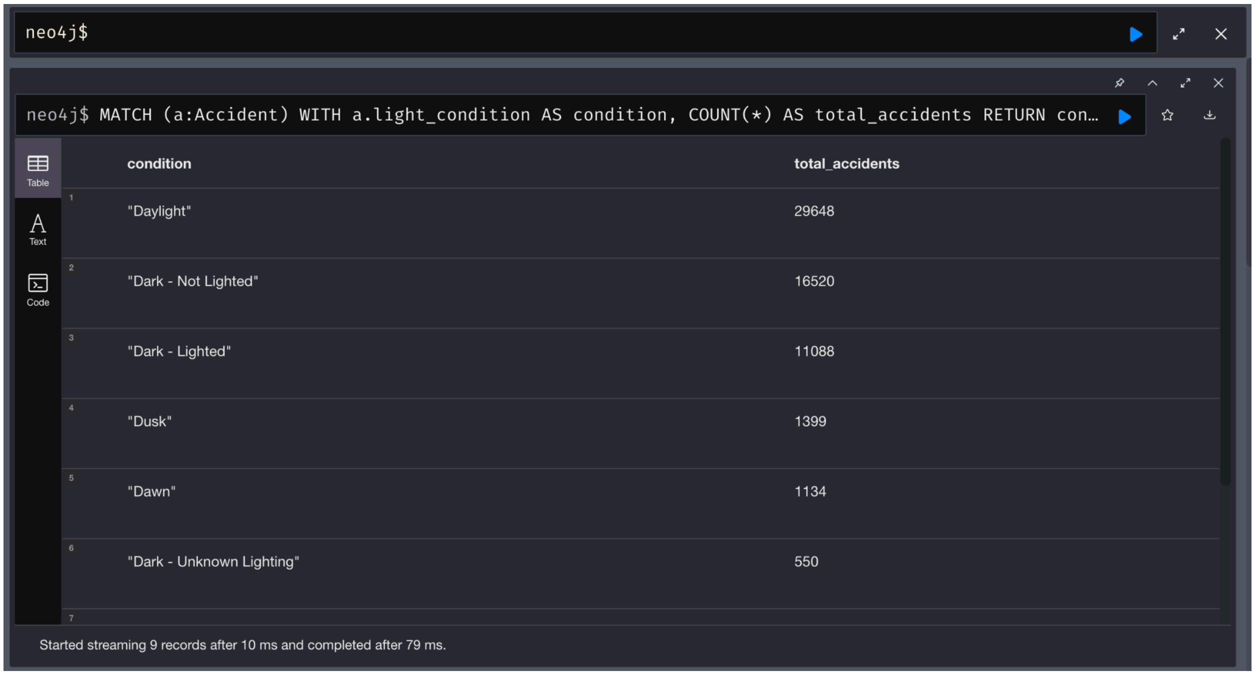
```
Started streaming 1 records after 10 ms and completed after 22 ms.
```

This query matches all the nodes labeled “Accident” and then filters these nodes using a WHERE clause to select only those where the drunk_driving is set to true, indicating that the accident involved alcohol. The query finally returns the count of these filtered accidents using the COUNT function.

- Simulation of relational group by



```
MATCH (a:Accident)
WITH a.light_condition AS condition, COUNT(*) AS total_accidents
RETURN condition, total_accidents
ORDER BY total_accidents DESC;
```



```
neo4j$
```

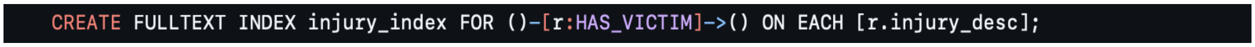
```
neo4j$ MATCH (a:Accident) WITH a.light_condition AS condition, COUNT(*) AS total_accidents RETURN con…
```

	condition	total_accidents
1	"Daylight"	29648
2	"Dark - Not Lighted"	16520
3	"Dark - Lighted"	11088
4	"Dusk"	1399
5	"Dawn"	1134
6	"Dark - Unknown Lighting"	550
7		

Started streaming 9 records after 10 ms and completed after 79 ms.

This query replicates the functionality of a GROUP BY clause in a relational database. It first matches all the nodes labeled “Accident”, then uses the WITH clause to group these records by lighting condition and counts the total number of accidents within each group. The results are displayed in descending order based on the number of accidents to better see which lighting conditions are most frequently associated with car crashes.

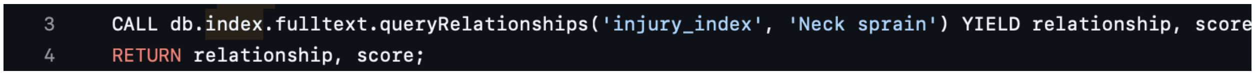
- Indexes



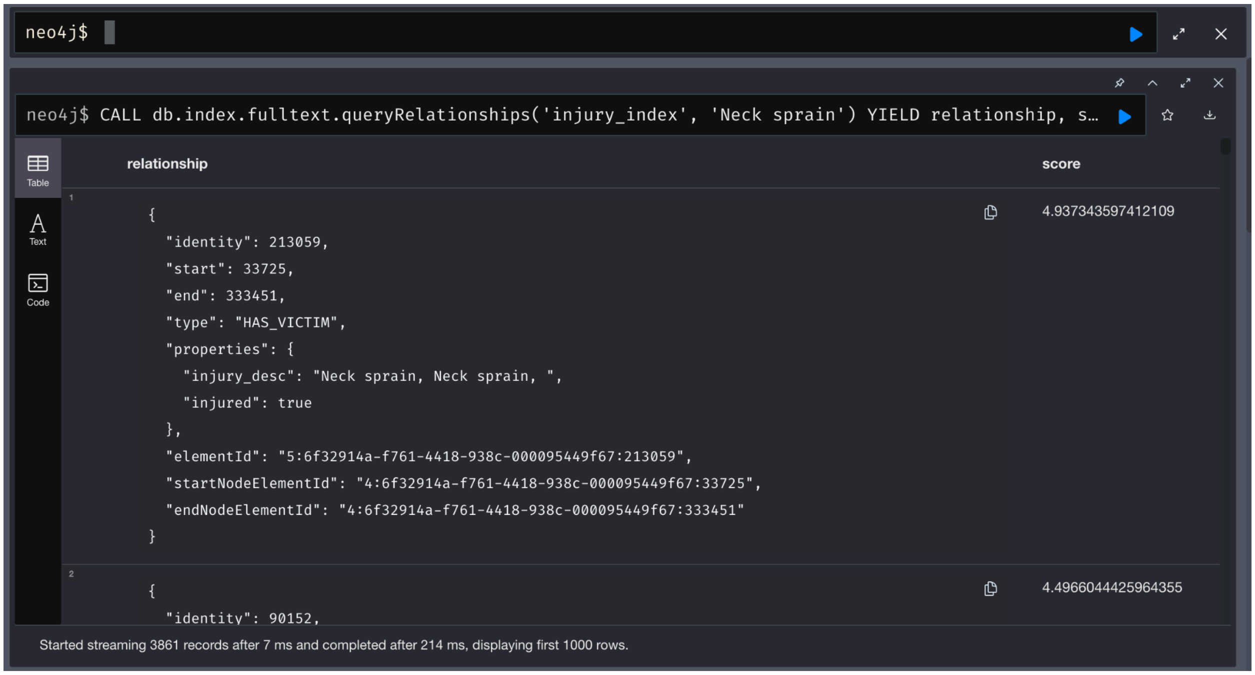
```
CREATE FULLTEXT INDEX injury_index FOR ()-[r:HAS_VICTIM]->() ON EACH [r.injury_desc];
```

A full-text index named “injured_index” was created on the “injury_desc” property of the “HAS_VICTIM” relationship to enable efficient and flexible keyword-based searches. This allows for faster queries of injury descriptions and shows the relevance scoring (more on that below).

- Full-text search



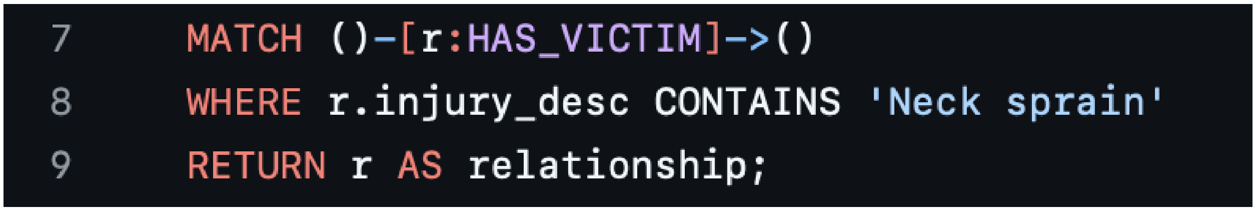
```
CALL db.index.fulltext.queryRelationships('injury_index', 'Neck sprain') YIELD relationship, score
RETURN relationship, score;
```



```
neo4j$

neo4j$ CALL db.index.fulltext.queryRelationships('injury_index', 'Neck sprain') YIELD relationship, s…

        relationship                                                              score
1
          {                                                                      4.937343597412109
            "identity": 213059,
            "start": 33725,
            "end": 333451,
            "type": "HAS_VICTIM",
            "properties": {
              "injury_desc": "Neck sprain, Neck sprain, ",
              "injured": true
            },
            "elementId": "5:6f32914a-f761-4418-938c-000095449f67:213059",
            "startNodeElementId": "4:6f32914a-f761-4418-938c-000095449f67:33725",
            "endNodeElementId": "4:6f32914a-f761-4418-938c-000095449f67:333451"
          }
2
          {                                                                      4.4966044425964355
            "identity": 90152,

Started streaming 3861 records after 7 ms and completed after 214 ms, displaying first 1000 rows.
```

To perform a more flexible search across relationship properties, a full-text search query was implemented. The query uses "CALL db.index.fulltext.queryRelationships" to search for relationships that match the term "Neck sprain" within the previously created full-text index name "injury_index" (see bullet point above). The query returns matching relationships as well a relevance score, which indicates how closely each result matches the search term.

A query doing the same thing but without indexing can be seen below:



```
MATCH ()-[r:HAS_VICTIM]->()
WHERE r.injury_desc CONTAINS 'Neck sprain'
RETURN r AS relationship;
```

While it achieves the same result as using a full-text index, it is slower because it performs a direct scan of all matching relationships instead of simply using an optimized index.
The query with the full-text index executes in 214 ms, whereas the one without executes in 245 ms.

The Migration Process

For the migration of data from our relational Postgresql database to the Neo4j Nosql graph database, we started by exporting the data from each table into a csv format file. Since Neo4j's remote access doesn't allow direct local files import, the files were uploaded on our github repository. This allowed us to use the raw Github URLs in our cypher queries so that Neo4j could import the data from a public url. To import the data we used the LOAD CSV WITH HEADERS command in cypher. The following is an example of one of our queries:

```
LOAD CSV WITH HEADERS FROM
'https://raw.githubusercontent.com/anarostam/AVMQuery_soen363/main/phase2/csv/emergency
_respondent.csv' AS row
        CALL {
        WITH row
        CREATE (:EmergencyRespondent {
                respondent_id: toInteger(row.respondent_id),
                type: row.type,
                contact_info: row.contact_info,
                ssn: row.ssn,
                first_name: row.first_name,
                last_name: row.last_name,
                date_of_birth: date(row.date_of_birth),
                sex: row.sex,
                counties: [x IN split(replace(replace(row.counties, '{', ''), '}', ''), ',') | toInteger(x)]
        })
        } IN TRANSACTIONS OF 50 ROWS;
```

We did the same process for all other tables. for relationship tables, the queries were slightly different since we didn't need to create new nodes so we only created relationships between the existing nodes using the MERGE statements.

Some changes were made to the structure of tables and data storage structure during the migration. The first change was made to the details table which was a weak entity to the accident table. We merged these two tables to eliminate the weak entity in our database. This made it easier to access accident details in a single place and improved query performance. The second change was made to the ISA relationship between the person, victim, and emergency-respondent entities. The person entity, which was the super entity, was deleted. The victim and emergency respondent remained unchanged and kept separately. This was done to keep the data and its storage more consistent.

This migration helped us to shift to a more flexible and have more efficient queries using Neo4j.

Challenges

During phase 1 of the project, one of the main challenges we faced was related to the data population process. The API we initially selected had a strict daily access limit, which made it difficult for us to retrieve the data within our project timeline. To overcome this, we switched to a different API that offered more flexibility toward data access. While the new API had some limitations, it allowed us to work more efficiently in terms of time. Another issue we faced during phase 1 was with the use of the AWS free tier. We exceeded the free tier usage, which meant we could no longer host our database instance. As a solution, we used local databases and shared the backup with each other to keep the data consistent. After the share of backup we were able to continue working on other parts of the project.

During the migration process in phase 2 of the project, we faced additional challenges due to the size of the data files. Large CSV files and the heavy load of data caused Neo4j's server to hang or crash multiple times during import operations. To fix this, we split the larger CSV files into smaller chunks (around 10000 to 50000 rows each based on the CSV size) to reduce the processing load. We also modified the cypher query to process the data in smaller batches using the IN TRANSACTIONS OF clause. This helped to prevent memory overload and ensure more stable data imports.

The challenges helped us to adapt to the new situations and find solutions quickly, giving us a better understanding of working with data at scale.

Conclusion

This project set out to explore whether bad weather leads to more fatal car accidents, but our PostgreSQL analysis showed that most accidents occurred during good weather, with 169,550 in good conditions compared to just 536 in bad. This suggests our initial assumption may have been incorrect, possibly because people tend to avoid driving in severe weather.

From Neo4j, we ran a query that simulates an aggregation grouped by light conditions, and found that accidents were more frequent during daylight than in conditions such as dawn or darkness. While this might suggest accidents are more common in better conditions, it is also true that more people are on the road during those times, which is a factor worth exploring in future work.

Neo4j proved especially useful for mapping and querying real-world relationships in the data, which made it a great fit for our topic. Overall, we learned how to gather real-world data, design and query both relational and graph databases, and better understand how to approach meaningful questions through data.