CITS3401 Project 2

Report

By: Nicodemus Ong

Student ID: 22607943

Contents

A) Design and implementation of the graph database.

Building upon project 1, the primary objective of this project was to transform my existing data warehouse, which is currently in a relational database format, into a graph database. To accomplish this, two approaches were considered:

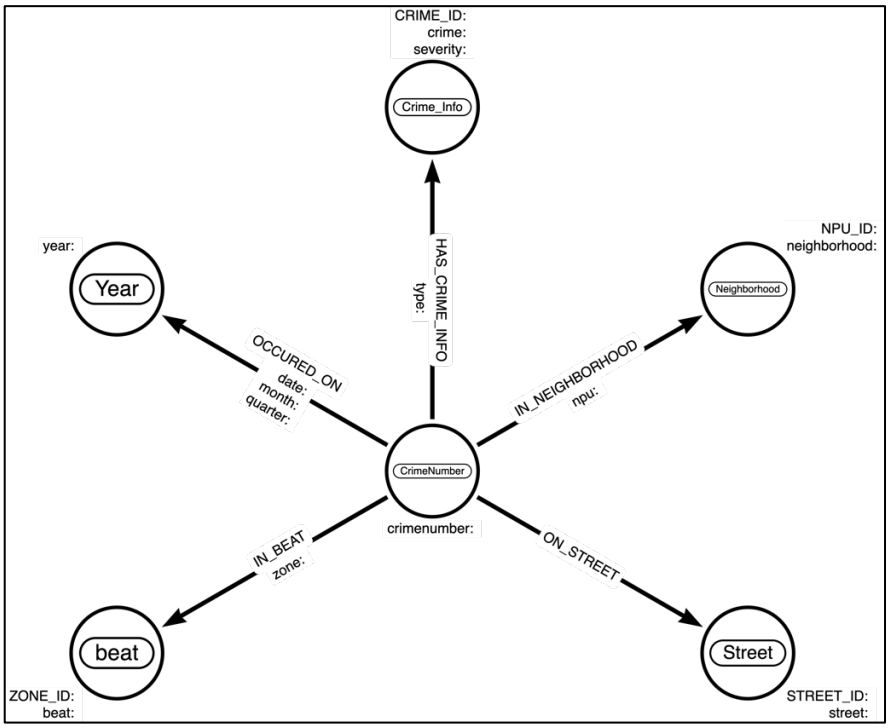
1. The first approach involved directly converting the data warehouse and its existing tables into relevant nodes, labels, and properties:
    1. Each row within the data warehouse would be represented as a node.
    2. The table names would serve as label names.
    3. Joins or foreign keys would be represented as relationships.
2. The second approach entailed designing a graph database structure and adapting the CSV files to meet the requirements of the new graph database design. This approach was chosen for this project, allowing me to first design the structure and schema of the graph database implementation. Subsequently, the data used in project 1 was reprocessed to align with the requirements of the graph database.

In this project, I have opted for the latter approach to create a well-designed structure and schema for my graph database. Additionally, I will modify the previously used data from project 1 to ensure it meets the specific requirements of the graph database.

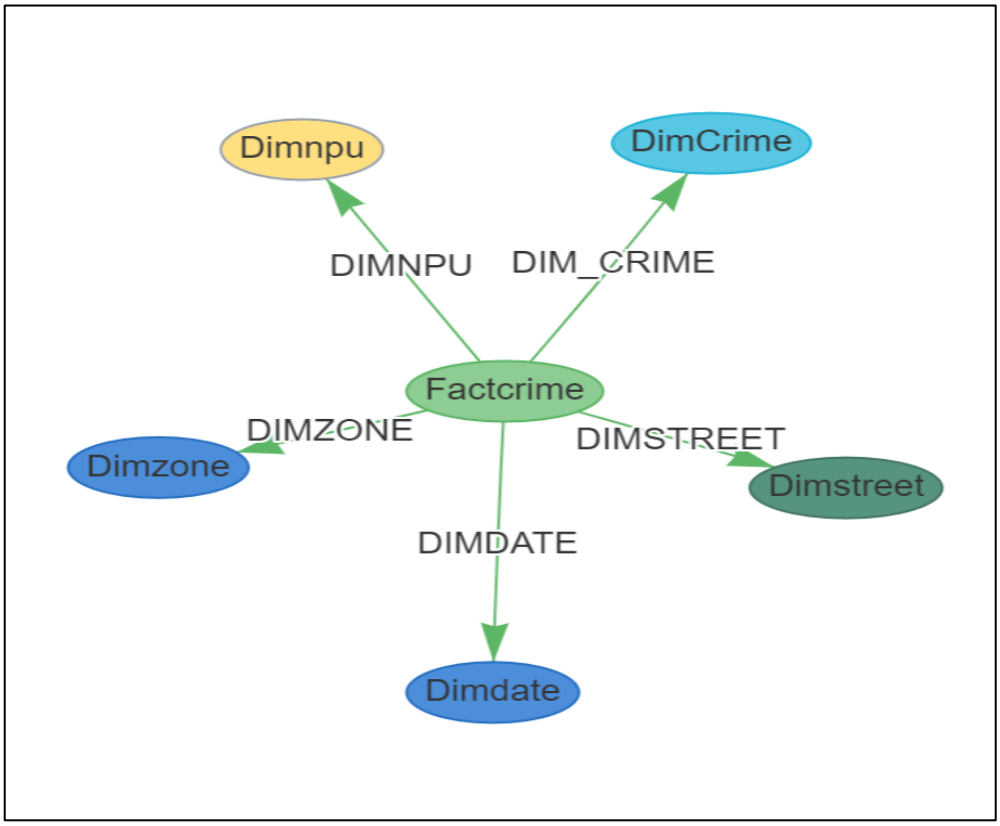
B) Graph database design using the arrow tool in comparison with direct ETL import.

To show the differences of what the 2 approaches in creating the graph database would look like, I will use the *Arrow.app* tool as well as the *Direct ETL Import Tool* in Neo4j.

Arrow.app:



Direct ETL Import Tool:



Referring to the provided screenshots, the *Direct ETL Import Tool* method can be considered a basic approach for converting an existing data warehouse into a functional graph database. It accomplishes this by transforming the rows of dimensional data tables and fact tables into nodes, establishing connections through foreign key relationships. While this implementation results in a functional database, it may not offer the optimal design and can lead to inefficient query performance due to its structure.

Hence, the secondary approach is strongly recommended as it focuses on a design specifically tailored to meet the requirements of the queries, ensuring effective and efficient query execution. One notable improvement in this approach involves the separation of properties and the introduction of additional nodes within the graph database. This enhancement enables dynamic connections and relationships, significantly enhancing the database's ability to query and reference data.

In summary, although the *Direct ETL Import Tool* method functions adequately, it may impose limitations on the graph database's capabilities since it was not designed with the specific queries in mind. Therefore, adopting the secondary approach is highly advantageous for optimizing the graph database's performance and overall functionality.

C) Discussions of the design choices with pros and cons identified.

1. Graph Database Design:
    - **Design Choice**: The chosen design is a graph database model, where nodes represent entities (e.g., *crimes, years, neighborhoods*) and relationships represent connections between these entities.
        - **Pros:**
            - Graph databases are well-suited for handling complex relationships and interconnected data. They can efficiently query relationships and traverse paths between nodes.
            - The graph model allows for flexible and scalable data representation. New nodes and relationships can be easily added as needed.
        - **Cons:**
            - Graph databases may have a steeper learning curve compared to traditional relational databases.
            - They may not be the best choice for simple, tabular data structures with few relationships.
2. Deletion of Existing Nodes:
    - **Design Choice:** The code starts with deleting all existing nodes in the graph using the *"detach delete"* command.
        - **Pros:**
            - Starting with a clean slate ensures that the database is reset and ready to load fresh data.
        - **Cons:**
            - Deleting all nodes can be a time-consuming operation for large databases.
            - If there are any nodes that should not be deleted (e.g., *reference data*), additional steps are required to preserve them.

3. Data Loading:
    - **Design Choice:** The query loads data from CSV files using the "*LOAD CSV*" command and creates nodes for different dimensions (*crime, year, neighborhood, street, beat*).
        - **Pros:**
            - Loading data from CSV files allows for easy integration with external data sources.
            - Creating nodes for different dimensions enables querying and analysing the data based on these dimensions.
        - **Cons:**
            - The query assumes that the CSV files have headers and uses the header names for mapping data. Any inconsistencies or missing headers can cause issues.
            - There is no data validation or transformation performed during the loading process, which can lead to data integrity problems.
4. Node and Relationship Creation:
    - **Design Choice:** The query uses the *"CREATE"* and *"MERGE"* commands to create nodes and relationships between them based on the provided data.
        - **Pros:**
            - Creating nodes and relationships allows for establishing connections between different entities.
            - Using *"MERGE"* ensures that nodes and relationships are created only if they don't already exist, preventing duplicates.
        - **Cons:**
            - The query does not perform any checks or validations to ensure data consistency or integrity during the creation process.
            - There is no handling of potential conflicts or errors that may occur during the creation of nodes and relationships.

Overall, the design choices in this implementation demonstrate the utilization of a graph database to represent and connect various entities related to crimes. However, there are

pitfalls to data validation, error handling, and ensuring data integrity during the loading and creation processes.

D) Graph Database Implementation with Cypher code.

(Note: The CSV and table structure of the dimensional table and fact table are not the same as the ones used in project 1. It has been changed and modified to fit the graph database requirements better. The ETL python script has been provided as *ETL.py.*)

ETL Process of CSV files:

The ETL (Extract, Transform, Load) process for this project involves using a Python script to read and clean data from CSV files. Similar to Project 1, the script performs data preparation tasks on the input data. It then splits the data into separate CSV files for the dimensional tables and the fact table.

However, there is a notable difference in the script compared to the previous project. In this case, certain columns were deliberately left in the fact tables as 'measures' to facilitate the creation of relationships in the Neo4j database.

The reason behind this decision is that there is a limitation in Neo4j's ability to extract specific data from the dimensional tables using foreign key relationships alone. While it might be possible to address this limitation, a solution could not be found in this instance. Therefore, to overcome this challenge, the decision was made to include the relevant columns in the fact table itself.

By including these columns as 'measures' in the fact table, the data can be effectively used to establish relationships in the Neo4j graph database. This approach ensures that the necessary data is available for analysis and querying within the graph database environment, even though it deviates from the typical dimensional modelling principles.

Overall, the ETL process involves reading and cleaning the CSV files, splitting them into dimensional and fact tables, and adapting the fact table structure to accommodate the limitations of Neo4j, thus enabling the creation of relationships within the graph database.

Graph Generation:

1. Before starting the creation, I first ensured that there were no existing nodes and relationships in my graph. To do this I detached and deleted every node in the graph.

```
match (n) detach delete (n);
```

2. To create my graph database, I first started by creating the nodes from my dimensional tables CSVs. The *'LOAD CSV'* command was used to load the CSV as the dataset was only 1000 lines approximately hence it would have suffice for this

requirement. However should your dataset be large, *APOC* would be a better option due to several factors:

a. **Performance:** *APOC* provides optimized procedures specifically designed for handling large datasets. These procedures are implemented in Java and can offer significant performance improvements compared to the built-in *LOAD CSV* functionality.

1. **Parallel Processing:** *APOC* allows for parallel processing of data, which means it can distribute the workload across multiple threads or cores. This parallel execution can significantly speed up the data loading process, especially when dealing with large datasets.
2. **Batching and Chunking:** *APOC* supports batching and chunking techniques that allow for efficient processing of data in smaller portions. This approach helps avoid memory issues that can occur when loading large datasets all at once.
3. **Flexible Data Transformation:** *APOC* provides a wide range of procedures that enable data transformation, validation, and enrichment during the import process. This flexibility is particularly useful when dealing with complex data structures or when additional data pre-processing steps are required.
4. **Error Handling:** *APOC* offers robust error handling capabilities, allowing you to handle and recover from errors encountered during data loading. This feature is crucial when dealing with large datasets where data quality or format issues may arise.

It's important to note that while *APOC* can provide performance benefits for large datasets, the choice between *LOAD CSV* and *APOC* ultimately depends on the specific use case and requirements. *LOAD CSV* is suitable for smaller datasets or simpler data loading tasks, while *APOC* shines when dealing with large, complex datasets that require additional processing capabilities.

```
LOAD CSV WITH HEADERS FROM 'file:///DimCrime.csv' AS row
WITH DISTINCT row.crime AS crime, row.Severity AS severity
CREATE (ci:Crime_Info {
  crime: crime,
  Severity: severity
});

LOAD CSV WITH HEADERS FROM 'file:///DimDate.csv' AS row
WITH DISTINCT row.year AS year
CREATE (y:Year {
  year: year
});

LOAD CSV WITH HEADERS FROM 'file:///DimNPU.csv' AS row
CREATE (n:Neighborhood {
  NEIGHBORHOOD_ID: row.NPU_ID,
  neighborhood: row.neighborhood
});

LOAD CSV WITH HEADERS FROM 'file:///DimStreet.csv' AS row
CREATE (s:Street {
  STREET_ID: row.STREET_ID,
  street: row.street
});

LOAD CSV WITH HEADERS FROM 'file:///DimZone.csv' AS row
CREATE (b:Beat {
  BEAT_ID: row.ZONE_ID,
  beat: row.beat
});
```

3. After the dimensional tables have been loaded in successfully, I then loaded in the Fact table, however unlike the way the dimensional tables were loaded in, the fact table is different. The *MERGE* clause creates a node labelled *CrimeNumber* with the property *crimenumber* based on the number value from the CSV file. The subsequent *MATCH* clauses find existing nodes based on specific conditions. The *MERGE* clauses create relationships between the cn node (*CrimeNumber*) and other nodes based on the matching conditions and properties from the CSV file.

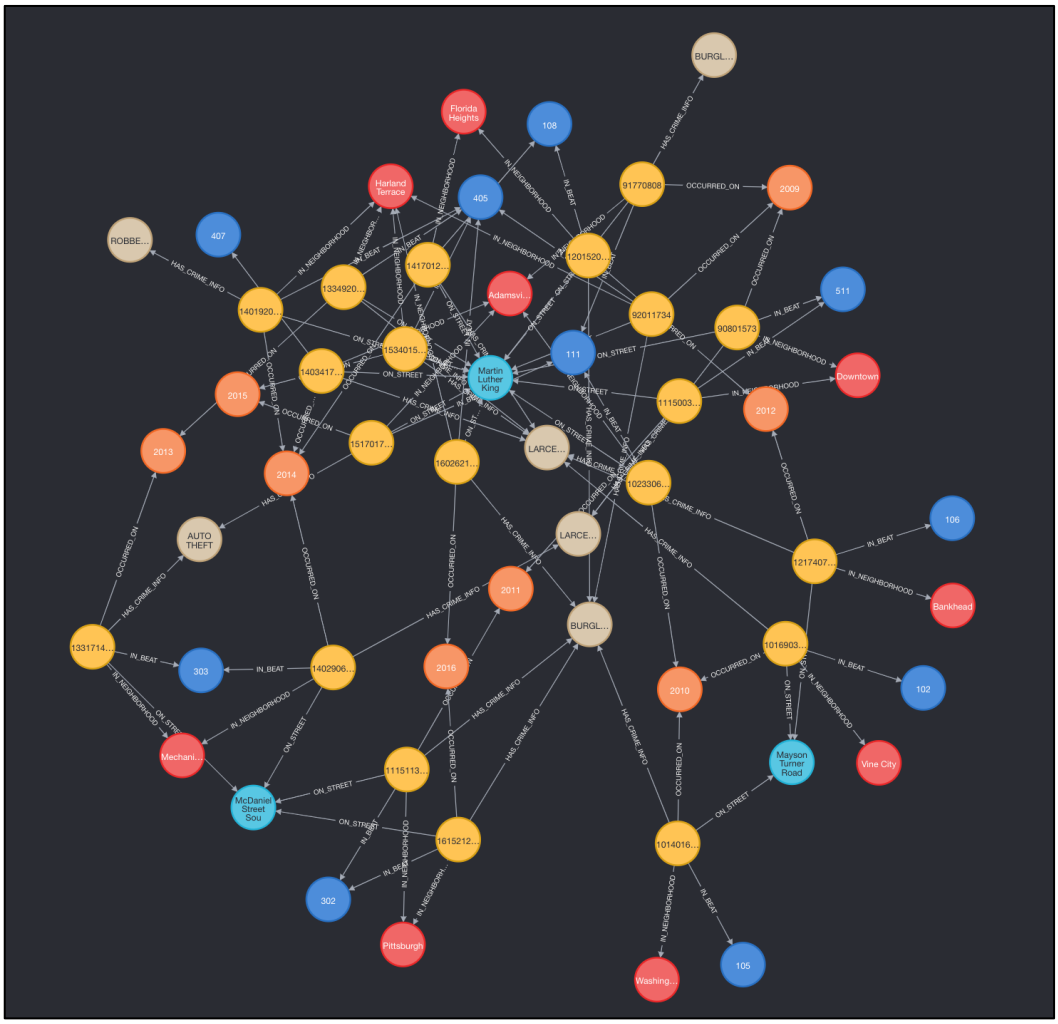
```
LOAD CSV WITH HEADERS FROM 'file:///FactCrime.csv' AS row
MERGE (cn:CrimeNumber {crimenumber: row.number})
WITH cn, row

MATCH (ci:Crime_Info {crime: row.crime})
MATCH (y:Year {year: row.year})
MATCH (n:Neighborhood {NEIGHBORHOOD_ID: row.NPU_ID})
MATCH (s:Street {STREET_ID: row.STREET_ID})
MATCH (b:Beat {BEAT_ID: row.ZONE_ID})

MERGE (cn)-[:HAS_CRIME_INFO {type: row.type}]->(ci)
MERGE (cn)-[:OCCURRED_ON {date: row.date, month:row.month, quarter:row.quarter}]->(y)
MERGE (cn)-[:IN_NEIGHBORHOOD {npu: row.npu}]->(n)
MERGE (cn)-[:ON_STREET]->(s)
MERGE (cn)-[:IN_BEAT {zone: row.zone}]->(b);
```

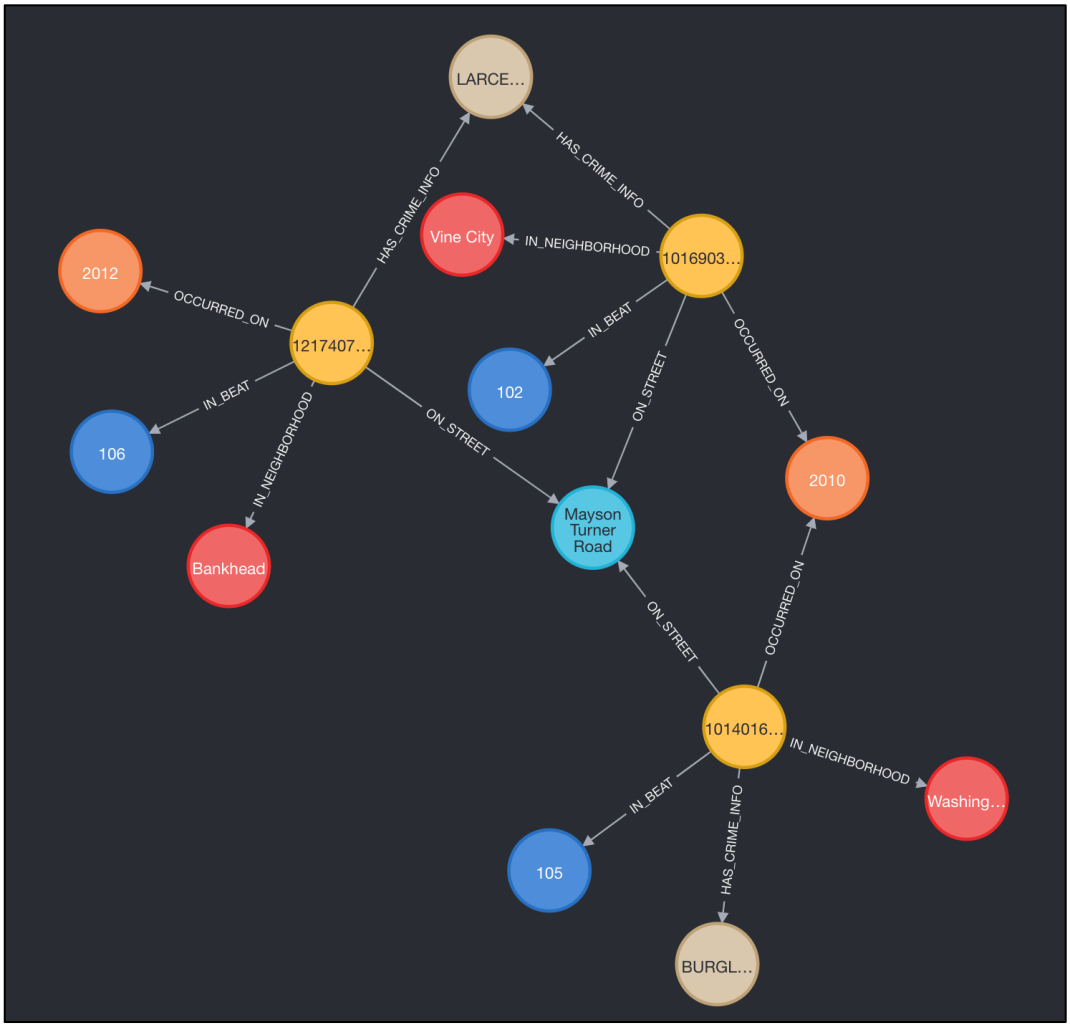
I chose to do the creation of the *CrimeNumber* nodes as well as the relationship to the other nodes together because it was unnecessary to split them up. The resulting

Graph looks something like this. (**Note:** only 100 nodes are being shown for performance reasons.)



A zoomed in version of the graph would show a clearer picture of what the nodes and relationships look like. Each node and relationships have their corresponding

properties as per the configurations that were inputted via the cypher code.



E) Meaningful Graph Database navigation discussed using the Neo4j browser.

Neo4j is a popular graph database management system that allows you to model, store, and query data as a graph. The Neo4j Browser is a web-based tool that provides an interactive interface for querying and visualizing graph data stored in a Neo4j database.

When discussing meaningful graph database navigation, it typically involves performing queries and traversals on the graph data to extract valuable information. Here are a few aspects that make graph database navigation meaningful:

1. **Querying**: The Neo4j Browser allows you to write and execute Cypher queries, a powerful query language specifically designed for working with graph data. By formulating well-crafted queries, you can retrieve specific patterns, relationships, and properties from the graph, enabling you to gain insights and answer complex questions about the data.
2. **Traversal**: Graph databases excel at modelling and representing relationships between entities. With the Neo4j Browser, you can navigate and traverse the graph by following relationships between nodes. Traversals can help you understand the connectedness, paths, and patterns within your data, allowing you to explore various perspectives and uncover meaningful insights.
3. **Visualization**: The Neo4j Browser provides visualization capabilities, allowing you to view and interact with the graph data in a visual form. Visualizations can help you understand the structure, relationships, and clusters within the data. By customizing the visualization settings and applying styling options, you can make the graph representation more meaningful and informative.
4. **Analysing Results**: After executing queries or traversals, the Neo4j Browser displays the results in tabular or graphical formats. Meaningful graph database navigation involves analysing and interpreting these results to extract meaningful insights, patterns, or trends. By understanding the data presented in the results, you can draw conclusions or make informed decisions based on the information discovered.

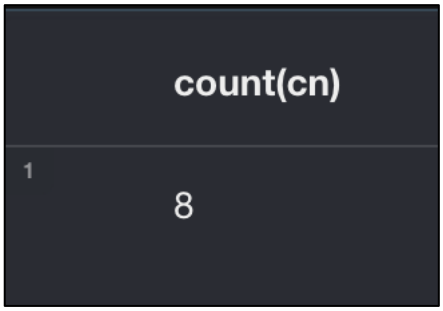
In summary, meaningful graph database navigation using the Neo4j Browser involves leveraging the querying, traversal, visualization, and analytical capabilities of the tool to explore and understand the graph data stored in a Neo4j database. It helps uncover valuable insights, patterns, and relationships within the data, leading to a deeper understanding of the underlying data model and its implications.

F) Graph Database queries with Cypher code

1. How many crimes are recorded for a given crime type in a specified neighbourhood for a particular period?

```
MATCH (cn:CrimeNumber)-[:HAS_CRIME_INFO]->(ci:Crime_Info {crime: 'LARCENY-FROM VEHICLE'})
MATCH (cn)-[:IN_NEIGHBORHOOD]->(n:Neighborhood {neighborhood: 'Midtown'})
MATCH (cn)-[:OCCURRED_ON]->(y:Year {year: '2012'})
return count(cn);

// Result: 8
```



	count(cn)
1	8

   a. The query aims to find the count of crime numbers associated with the crime type *'LARCENY-FROM VEHICLE'* that occurred in the *'Midtown'* neighborhood during the year 2012. It starts by matching a *CrimeNumber* node labelled as cn that has a *HAS_CRIME_INFO* relationship to a *Crime_Info* node labelled as ci. The *Crime_Info* node must have the property crime with the value *'LARCENY-FROM VEHICLE'*. This step ensures that we only consider crime numbers associated with this specific crime type.
   b. Next, the query matches the *cn* node that has an *IN_NEIGHBORHOOD* relationship to a *Neighborhood* node labelled as n. The *Neighborhood* node must have the property *neighborhood* with the value '*Midtown*'. This ensures that we focus on crime numbers that occurred in the '*Midtown*' neighborhood.
   c. Then, the query matches the same *cn* node that has an *OCCURRED_ON* relationship to a *Year* node labelled as *y*. The *Year* node must have the property *year* with the value *'2012'*. This step filters the crime numbers to include only those that occurred in the year *2012*.
   d. Finally, the query returns the count of the matched *cn* nodes using *RETURN count(cn);*. This count represents the number of crime numbers that meet all

the specified conditions, i.e., crime numbers associated with *'LARCENY-FROM VEHICLE'*, occurring in *'Midtown'*, and happening in *2012*.

2. Find the neighborhoods that share the same crime types, organise in descending order of the number of common crime types.

```
MATCH (cn1:CrimeNumber)-[:HAS_CRIME_INFO]->(ci:Crime_Info)
MATCH (cn1:CrimeNumber)-[:IN_NEIGHBORHOOD]->(n1:Neighborhood)
WITH n1, COLLECT(DISTINCT ci.crime) AS crimeTypes
MATCH (cn2:CrimeNumber)-[:HAS_CRIME_INFO]->(ci:Crime_Info)
MATCH (cn2:CrimeNumber)-[:IN_NEIGHBORHOOD]->(n2:Neighborhood)
WHERE n1 <> n2
WITH n1, n2, COLLECT(DISTINCT ci.crime) AS otherCrimeTypes, crimeTypes
WITH n1, n2, [crimeType IN crimeTypes WHERE crimeType IN otherCrimeTypes] AS commonCrimeTypes
WITH n1, n2, SIZE(commonCrimeTypes) AS commonCrimeCount
WHERE commonCrimeCount > 0
RETURN n1.neighborhood AS neighborhood1, n2.neighborhood AS neighborhood2, commonCrimeCount
ORDER BY commonCrimeCount DESC
```

Close (⌘W)

	neighborhood1	neighborhood2	commonCrimeCount
1	"Old Fourth Ward"	"Sylvan Hills"	7
2	"Midtown"	"Sylvan Hills"	7
3	"Downtown"	"Sylvan Hills"	7
4	"Sylvan Hills"	"Old Fourth Ward"	7
5	"Sylvan Hills"	"Midtown"	7
6	"Sylvan Hills"	"Downtown"	7

a. The query aims to find pairs of neighborhoods that share common crime types. It starts by matching *CrimeNumber* nodes labelled as *cn1* that have a *HAS_CRIME_INFO* relationship to *Crime_Info* nodes labelled as *ci*. Additionally, it matches *cn1* nodes that have an *IN_NEIGHBORHOOD* relationship to *Neighborhood* nodes labelled as *n1*. This step collects the distinct crime types associated with the *cn1* nodes, grouping them by the neighborhood *n1*.

b. Next, the query continues by matching another set of *CrimeNumber* nodes labelled as *cn2* that have a *HAS_CRIME_INFO* relationship to *Crime_Info* nodes labelled as *ci*. Similarly, it matches *cn2* nodes that have an *IN_NEIGHBORHOOD* relationship to *Neighborhood* nodes labelled as *n2*. This step is performed separately to ensure we compare different neighborhoods.

c. Then, the query filters the results by excluding pairs of neighborhoods where *n1* is the same as *n2 (n1 <> n2)*. This ensures we only consider distinct pairs of neighborhoods.
d. Afterward, the query groups the common crime types between the neighborhoods. It collects the distinct crime types associated with *cn2* nodes, grouping them by the neighborhood pair *(n1, n2)*. The collected crime types are stored in the variable *otherCrimeTypes*, and the previously collected crime types for *n1* are stored in the variable *crimeTypes*.
e. Next, the query filters the results further by checking if there are any common crime types between the neighborhoods. It creates a list called *commonCrimeTypes*, which includes only the crime types that are present in both *crimeTypes* and *otherCrimeTypes* lists.
f. Then, the query calculates the count of common crime types for each neighborhood pair and stores it in the variable *commonCrimeCount* using the *SIZE()* function.
g. Finally, the query filters the results to only include neighborhood pairs where the common crime count is greater than 0. It returns the neighborhood names for *n1* and *n2*, as well as the common crime count. The results are ordered in descending order based on the common crime count.
h. In summary, the query retrieves pairs of neighborhoods that share common crime types. It identifies the distinct crime types for each neighborhood, compares them between different neighborhoods, calculates the count of common crime types, and returns the neighborhood pair along with the common crime count.

3. Return the top 5 neighborhoods for a specified crime for a specified duration.

```
MATCH (ci:Crime_Info {crime: 'LARCENY-NON VEHICLE'})
MATCH(cn:CrimeNumber)-[:HAS_CRIME_INFO]->(ci)
MATCH(cn:CrimeNumber)-[:IN_NEIGHBORHOOD]->(n:Neighborhood)
return n.neighborhood,COUNT(*) AS crime_count
ORDER BY crime_count DESC
LIMIT 5
```

	n.neighborhood	crime_count
1	"Downtown"	24
2	"Midtown"	13
3	"Lenox"	12
4	"Berkeley Park"	12
5	"Old Fourth Ward"	10

a. The query aims to find the top 5 neighborhoods with the highest count of crimes of the type *'LARCENY-NON VEHICLE'*. It starts by matching a *Crime_Info* node labelled as *ci* that has the property *'crime'* with the value *'LARCENY-NON VEHICLE'*. This step ensures that we focus on crimes specifically of this type.

b. Next, the query matches *CrimeNumber* nodes labelled as *cn* that have a *HAS_CRIME_INFO* relationship to the previously matched *ci* node. This ensures that we consider crime numbers associated with the specific crime type.

c. Additionally, the query matches the same *cn* nodes that have an *IN_NEIGHBORHOOD* relationship to a *Neighborhood* node labelled as *n*. This step associates the crime numbers with their respective neighborhoods.

d. Afterwards, the query returns the neighborhood names (*n.neighborhood*) and calculates the count of crimes in each neighborhood using the *COUNT(*)* function, which counts the number of crime numbers associated with each neighborhood.
e. The results are then ordered in descending order based on the crime count (*crime_count*) to identify the neighborhoods with the highest crime counts at the top.
f. Finally, the query applies a *LIMIT* of 5 to retrieve only the top 5 neighborhoods with the highest crime counts.
g. In summary, the query identifies the neighborhoods with the highest count of crimes of the type *'LARCENY-NON VEHICLE'*. It matches the relevant *Crime_Info* and *CrimeNumber* nodes, associates them with the respective neighborhoods, calculates the crime count for each neighborhood, and returns the top 5 neighborhoods based on the crime count.

4. Find the type of crimes for each property type.

```
MATCH (cn:CrimeNumber)-[hci:HAS_CRIME_INFO]->(ci:Crime_Info)
WITH hci.type AS propertyType, COLLECT(DISTINCT ci.crime) AS crimeTypes
RETURN propertyType, crimeTypes
```

	propertyType	crimeTypes
1	"house_number"	["LARCENY-NON VEHICLE", "AUTO THEFT", "BURGLARY-RESIDENCE", "LARCENY-FROM VEHICLE", "BURGLARY-NONRES
2	"shop"	["LARCENY-NON VEHICLE", "AUTO THEFT", "BURGLARY-RESIDENCE", "LARCENY-FROM VEHICLE", "BURGLARY-NONRES
3	"building"	["LARCENY-NON VEHICLE", "AUTO THEFT", "BURGLARY-RESIDENCE", "LARCENY-FROM VEHICLE", "BURGLARY-NONRES
4	"amenity"	["LARCENY-NON VEHICLE", "AUTO THEFT", "BURGLARY-RESIDENCE", "LARCENY-FROM VEHICLE", "BURGLARY-NONRES
5	"road"	["LARCENY-NON VEHICLE", "AUTO THEFT", "BURGLARY-RESIDENCE", "LARCENY-FROM VEHICLE", "BURGLARY-NONRES
6	"tourism"	["LARCENY-NON VEHICLE", "AUTO THEFT", "LARCENY-FROM VEHICLE", "BURGLARY-NONRES"]

a. The query aims to retrieve the types of crimes associated with each property type. It starts by matching *CrimeNumber* nodes labelled as *cn* that have a *HAS_CRIME_INFO* relationship (identified as *hci*) to *Crime_Info* nodes labelled as *ci*. This step ensures that we consider crime numbers associated with specific crime information.

b. Then, the query proceeds to the *WITH* clause, which allows us to manipulate the intermediate results. It selects the *hci.type* property value (representing the property type) and collects the distinct crime types (*ci.crime*) associated with each property type using the *COLLECT(DISTINCT ci.crime)* function. This step groups the crime types by their respective property types.

c. Finally, the query returns the *propertyType (hci.type)* and *crimeTypes* as the result. This will provide a list of property types along with the corresponding distinct crime types associated with each property type.

d. In summary, the query retrieves the property types and their associated distinct crime types. It matches *CrimeNumber* and *Crime_Info* nodes, groups the crime types by property type, and returns the property type along with the list of distinct crime types for each property type.

5. Which month of a specified year has the highest crime rate? Return one record each for each beat.

```
MATCH (cn:CrimeNumber)-[o:OCCURRED_ON]->(y:Year {year: '2014'})
MATCH (cn)-[:IN_BEAT]->(b:Beat)
WITH cn, b.beat AS beat, y.year AS year, o.month AS month, COUNT(cn) AS crimeCount
ORDER BY beat, crimeCount DESC
WITH beat, COLLECT({cn: cn, year: year, month: month, crimeCount: crimeCount})[0] AS highestMonth
RETURN beat, highestMonth.year AS year, highestMonth.month AS month, highestMonth.crimeCount AS crimeCount, highestMonth.cn AS crimeNumber
```

	beat	year	month	crimeCount	crimeNumber
1	"102"	"2014"	"May"	1	{ "identity": 6839, "labels": ["CrimeNumber"], "properties": { "crimenumber": "141381245" }, "elementId": "6839" }
2	"103"	"2014"	"November"	1	{ "identity": 6968, "labels": ["CrimeNumber"],

a. The query aims to find the month with the highest crime count for each beat in the year 2014. It starts by matching *CrimeNumber* nodes labelled as *cn* that have an *OCCURRED_ON* relationship (identified as *o*) to a *Year* node labelled as y. The *Year* node must have the property *'year'* with the value *'2014'*. This step ensures that we focus on crimes that occurred in the specified year.

b. Next, the query matches the previously matched *cn* nodes that have an *IN_BEAT* relationship to a *Beat* node labelled as *b*. This associates the crime numbers with their respective beats.

c. Afterwards, the query uses the *WITH* clause to manipulate the intermediate results. It selects *cn, b.beat* (representing the beat), *y.year* (representing the year), and *o.month* (representing the month). Additionally, it calculates the count of crime numbers associated with each beat using the *COUNT(cn)* function, identified as *crimeCount*. This step groups the crime numbers by

their respective beats and calculates the crime count for each beat in the year 2014.

d. Then, the results are ordered by beat and the crime count in descending order. This ensures that for each beat, the crime count is sorted in descending order, allowing us to find the month with the highest crime count for each beat.
e. Finally, the query applies another *WITH* clause to further manipulate the intermediate results. It selects beat and uses the *COLLECT({cn: cn, year: year, month: month, crimeCount: crimeCount})[0]* expression to collect the *crime numbers, year, month,* and *crime* count as a map. This collects the details for the highest crime count month for each beat.
f. The query concludes by returning the *beat, highestMonth.year, highestMonth.month, highestMonth.crimeCount,* and *highestMonth.cn* as the result. This provides the *beat, year, month, crime count,* and *crime number* for the month with the highest crime count for each beat in the year 2014.
g. In summary, the query identifies the month with the highest crime count for each beat in the year 2014. It matches *CrimeNumber* nodes, associates them with the respective *beats* and *years*, calculates the crime count for each beat in the specified year, orders the results, and returns the beat, year, month, crime count, and crime number for the highest crime count month for each beat.

6. Identify the months with the highest and lowest crime rates for a specified neighborhood.

```
MATCH (n:Neighborhood {neighborhood: 'Midtown'})<-[:IN_NEIGHBORHOOD]-(cn:CrimeNumber)-[o:OCCURRED_ON]->(y:Year)
WITH n, y.year AS year, o.month AS month, COUNT(cn) AS crimeCount
ORDER BY year, crimeCount DESC
WITH n, year, COLLECT({month: month, crimeCount: crimeCount}) AS monthlyCrimeCounts
WITH n, year, monthlyCrimeCounts[0] AS highest, monthlyCrimeCounts[-1] AS lowest
RETURN n.neighborhood AS neighborhood, year,
       highest.month AS highestMonth, highest.crimeCount AS highestCrimeCount,
       lowest.month AS lowestMonth, lowest.crimeCount AS lowestCrimeCount
```

	neighborhood	year	highestMonth	highestCrimeCount	lowestMonth	lowestCrimeCount
1	"Midtown"	"2009"	"April"	3	"May"	1
2	"Midtown"	"2010"	"October"	3	"January"	1
3	"Midtown"	"2011"	"December"	3	"June"	1
4	"Midtown"	"2012"	"June"	3	"October"	1
5	"Midtown"	"2013"	"October"	3	"March"	1

a. The query begins with a *MATCH* clause that matches the nodes representing the *'Midtown'* neighborhood. It then traverses the relationship labelled *'IN_NEIGHBORHOOD'* in the reverse direction to find all connected *'CrimeNumber'* nodes. Next, it follows the *'OCCURRED_ON'* relationship to connect the crime nodes with the corresponding *'Year'* nodes. This pattern of nodes and relationships that match the specified criteria is the result of the *MATCH* clause.

b. In the next step, the *WITH* clause is used to define the variables that will be retained for further processing. It selects the *'Neighborhood'* node as *'n'*, extracts the *'year'* property from the *'Year'* node as *'year'*, the *'month'* property from the *'OCCURRED_ON'* relationship as *'month'*, and calculates the count of *'CrimeNumber'* nodes as *'crimeCount'*. These variables will be used in subsequent operations.

c. The query then proceeds to sort the results by the *'year'* property in ascending order and the *'crimeCount'* property in descending order. This ordering ensures that the results are grouped by year and that the highest crime count comes first for each year.

d. In the next step, the query continues with the *WITH* clause to group the data by *'n'* (neighborhood) and *'year'*. It collects the *'month'* and *'crimeCount'* values into a list called *'monthlyCrimeCounts'*. This list contains objects with properties *'month'* and *'crimeCount'* for each result, representing the crime counts for each month within a year.
e. Another *WITH* clause is used to define variables *'highest'* and *'lowest'*. It assigns the first element of the *'monthlyCrimeCounts'* list to *'highest'* and the last element to *'lowest'*. These variables represent the highest and lowest crime counts for each year.
f. Finally, the query concludes with a *RETURN* clause. It specifies what to return as the result of the query. It selects the *'neighborhood'* property of the *'n'* node, the *'year'* variable, and the *'month'* and *'crimeCount'* properties of the *'highest'* and *'lowest'* objects. These values represent the neighborhood, year, highest crime count, and corresponding month, as well as the lowest crime count and corresponding month for each year.
g. In summary, this query retrieves crime statistics for the *'Midtown'* neighborhood from the crime graph database. It sorts the results by year and crime count and returns the highest and lowest crime counts with their respective months for each year.

7. Find the top property type associated with each crime type in a specified neighborhood.

```
MATCH (n:Neighborhood {neighborhood: 'Midtown'})
WITH n
MATCH (n)<-[:IN_NEIGHBORHOOD]-(cn:CrimeNumber)-[hci:HAS_CRIME_INFO]->(ci:Crime_Info)
WITH n, ci.crime AS crime, cn, hci
WITH n, crime, hci.type AS propertyType, count(*) AS count
ORDER BY crime, count DESC
WITH n, crime, collect({propertyType: propertyType, count: count})[0] AS topPropertyType
RETURN n.neighborhood AS neighborhood, crime, topPropertyType.propertyType AS topPropertyType, topPropertyType.count AS count
ORDER BY crime
```

	neighborhood	crime	topPropertyType	count
1	"Midtown"	"AUTO THEFT"	"house_number"	4
2	"Midtown"	"BURGLARY-NONRES"	"house_number"	2
3	"Midtown"	"BURGLARY-RESIDENCE"	"house_number"	1
4	"Midtown"	"LARCENY-FROM VEHICLE"	"house_number"	20
5	"Midtown"	"LARCENY-NON VEHICLE"	"house_number"	6

a. The query aims to identify the top property type associated with each crime in the specified neighborhood *'Midtown'*. It starts by matching a *Neighborhood* node labelled as *n* with the property *'neighborhood'* having the value *'Midtown'*.

b. Next, the query uses the *WITH* clause to carry the matched *Neighborhood* node *(n)* forward in the query. This allows us to continue building the query based on the specified neighborhood.

c. Then, the query matches *CrimeNumber* nodes (labelled as *cn*) that have an *IN_NEIGHBORHOOD* relationship to the previously matched *Neighborhood* node *(n)*. Additionally, these *CrimeNumber* nodes are connected via the *HAS_CRIME_INFO* relationship (identified as *hci*) to *Crime_Info* nodes (labelled as *ci*). This step associates the crime numbers with the specified neighborhood and their corresponding crime information.

d. Afterwards, the query continues with the *WITH* clause, which allows us to manipulate the intermediate results. It selects the *neighborhood (n)*, the crime type (*ci.crime*), the property type (*hci.type*), and calculates the count of occurrences (*count(*)*) for each combination of neighborhood, crime, and property type. This step groups the results by neighborhood, crime, and

property type, allowing us to determine the count of occurrences for each combination.

e. Then, the query proceeds to order the results by crime and count in descending order. This ensures that for each crime, the property types are sorted based on the count of occurrences, with the highest count appearing first.

f. The query then uses another *WITH* clause to further manipulate the intermediate results. It selects the *neighborhood (n)*, the crime type (*crime*), and uses the *collect({propertyType: propertyType, count: count})[0]* expression to collect the property types along with their corresponding counts as a map. This collects the top property type with the highest count for each crime.

g. Finally, the query returns the neighborhood *(n.neighborhood)*, the crime type (*crime*), the top property type (*topPropertyType.propertyType*), and the count (*topPropertyType.count*) as the result. This provides the specified neighborhood, along with the associated crime type, the top property type, and the count for each crime. The results are ordered by crime to maintain a consistent order.

h. In summary, the query identifies the top property type associated with each crime in the specified neighborhood *'Midtown'*. It matches the *Neighborhood* node, associates it with the *CrimeNumber* and *Crime_Info* nodes occurring in that neighborhood, calculates the count of occurrences for each crime and property type, and returns the neighborhood, crime type, top property type, and count for each crime in a consistent order.

G) Discussion on capabilities of graph databases in comparison with their relational counterparts, what can you do in graph databases/data warehouses that the other cannot.

Graph databases/data warehouses offer several capabilities that set them apart from relational databases. One key advantage is their ability to model and navigate relationships between entities. Graph databases represent relationships using nodes and edges, providing a natural and intuitive way to express connections. This allows for the direct representation of complex relationships and enables efficient navigation through the graph. Traversing relationships in a graph database is straightforward and does not require multiple joins across tables, making it easier to explore and analyse interconnected data.

Another advantage of graph databases is their flexible schema. Unlike relational databases that require predefined tables and explicit schema modifications, graph databases have a flexible structure. New nodes, edges, or properties can be added to the graph without affecting existing data. This flexibility is particularly beneficial in scenarios where the data model is not fully known in advance or when dealing with data that may have varying attributes or relationships.

Graph databases excel in deep relationship analysis. They provide specialized graph algorithms and query languages designed to uncover insights from interconnected data. These algorithms can identify groups of entities that frequently interact, determine the most influential nodes or edges in the graph, and cluster similar entities based on their relationships. These capabilities enable the discovery of patterns, clusters, and key entities within the graph that may not be easily identifiable using traditional relational databases.

When it comes to performance in connected data queries, graph databases shine. Traversing relationships in a graph is efficient due to the use of indexes and caching mechanisms that store the graph's structure. This allows for faster querying of connected data compared to relational databases, which may require complex join operations across multiple tables. Graph databases are particularly well-suited for tasks such as finding paths, analysing connectivity patterns, and retrieving related entities, making them a valuable tool in scenarios where the relationships between data points are critical for analysis.

Scalability and performance in highly connected data scenarios are also areas where graph databases excel. They are designed to handle large-scale and highly interconnected datasets. Graph databases can distribute data across multiple servers and perform parallel processing, enabling high-performance queries even on massive graphs. This scalability

allows organizations to efficiently analyse and extract insights from interconnected data at scale, supporting applications such as social network analysis, recommendation systems, and fraud detection that involve large and complex networks of relationships.

Graph databases provide a unified and connected view of data, simplifying data integration. By representing relationships as edges, graph databases can model connections between entities from different domains or systems. This simplifies the integration of data from various sources and enables the analysis of complex relationships that span multiple datasets. Graph databases make it easier to seamlessly integrate and query interconnected data sets, setting them apart from relational databases, which typically require complex join operations to combine data from different tables.

In summary, graph databases/data warehouses offer capabilities that go beyond what relational databases can provide. They excel in modelling and navigating relationships, have a flexible schema, enable deep relationship analysis, perform well in querying interconnected data, scale efficiently for highly connected datasets, and simplify data integration. These capabilities make graph databases valuable tools for analysing complex interconnected data and extracting insights from the relationships within the data.

H) Discuss how graph Data Science can be applied in at least one useful application of this graph database.

Graph Data Science is a powerful approach that leverages graph databases and graph algorithms to analyse and extract insights from interconnected data. It allows us to model and explore relationships between entities, uncover patterns, and gain a deeper understanding of the data.

In the given graph database, we have information about crimes, years, neighborhoods, streets, and beats. Let's explore how Graph Data Science can be applied to analyse crime data and derive useful insights:

1. Crime Pattern Analysis:
    a. By leveraging graph algorithms such as community detection, centrality analysis, and graph clustering, we can identify crime patterns within the dataset. For example:
        i. **Community detection:**
            - We can identify groups of crimes that frequently occur together, suggesting underlying connections or shared characteristics among them.
        ii. **Centrality Analysis:**
            - By calculating centrality measures like degree centrality or betweenness centrality, we can identify the most influential crimes, streets, or neighborhoods within the network.
        iii. **Clustering:**
            - Using graph clustering algorithms, we can group similar crimes based on attributes like severity, location, or modus operandi. This can help in understanding crime clusters and their characteristics.
2. Predictive Analysis:
    a. By combining the graph database with machine learning techniques, we can build predictive models to forecast crime occurrences. Graph-based features such as neighborhood connectivity, street attributes, or historical crime patterns can be utilized. For instance:
        i. **Link Prediction:**

- Predicting potential links between crimes and other entities based on similarity metrics can help identify potential future crime hotspots.

ii. **Temporal Analysis:**

- Analysing crime trends over time and identifying seasonal patterns or long-term trends can contribute to predictive modelling.

3. Crime Network Analysis:
    a. Graph Data Science can help in uncovering hidden relationships and networks among criminals, streets, or neighborhoods. By employing techniques such as social network analysis, we can gain insights into:
        i. Criminal Networks:
        - Identifying key individuals involved in criminal activities, their relationships, and roles within the network.

        ii. Crime Propagation:
        - Investigating how crimes spread across neighborhoods or streets, identifying influential nodes or edges in the network.
4. Resource Allocation and Intervention Strategies:
    a. By integrating external data sources like demographics, socioeconomic factors, or law enforcement resources, graph analysis can aid in optimizing resource allocation and devising effective intervention strategies. For example:
        i. Identifying High-Risk Areas:
        - By combining crime data with demographic information, we can identify neighborhoods with higher crime rates and allocate resources accordingly.

        ii. Intervention Planning:
        - Understanding the relationships between crimes=, neighborhoods, and streets can inform targeted intervention strategies, such as increased police presence or community outreach programs.

These are just a few examples of how Graph Data Science can be applied to analyse the given crime graph database. The flexibility and expressiveness of graph databases allow for complex queries, iterative analysis, and the integration of various data sources, making it a valuable tool for understanding and addressing real-world challenges related to crime analysis and prevention.