CITS 5504 - Data Warehousing
Project 2 Report

Graph Database Design and Cypher Query

Boya Zhang (24324257)

23 May 2025

Content

1. Introduction

Objective

This project is designed to solve business problems using graph database modelling techniques on a real-world dataset about traffic fatalities in Australia. The goal is to explore the road crash data, find patterns, and support decision-making for road safety.

This project continues from Project 1. Project 1 used the raw dataset and built a data warehouse. In this project, I apply graph database modelling, which performs well on queries involving highly connected data and is highly flexible.

Dataset

The dataset used in this project is provided in the official project specification [1]. It is obtained from the ARDD: Fatalities-December 2024 Excel file. The cleaned version of the dataset has already removed all missing or unknown values. The dataset is stored in CSV format.

It has 10,490 rows and 25 columns. The data covers all jurisdictions in Australia. Each row represents a person who died in a crash between 2014 and 2024, and each crash links to other information such as vehicles, time, people, road conditions, and locations, which makes a graph database suitable for this dataset.

Methodology

This project follows a step-by-step approach:
1. Graph Modelling: Design a property graph schema for the provided dataset
2. ETL Process: Use Python to transform the dataset into separate node and relationship CSV files
3. Graph Implementation: Import the CSV files into Neo4j and build the graph structure using Cypher
4. Cypher Queries: Write and run Cypher to answer the required and self-designed queries
5. Graph Data Science: Research and explore how graph algorithms and analytics can be applied to this traffic safety data

Tools used in this project include:
- Arrows App: Draw the graph schema
- Python: Perform ETL, data transformation, and generate CSV files
- Neo4j Desktop: Implement the property graph
- Cypher: Query language used to interact with Neo4j

Disclaimer: Some parts of the project involved assistance from generative AI (ChatGPT) [2]. I used it mainly to help with Python function explanations and graph database algorithm explanations. In writing, it was also used to adjust wording. All decisions on data modelling, query logic, and analysis were made by myself based on my understanding of the project requirements.

2. Graph Database Modelling and Design

This dataset is highly connected. Each fatal crash involves people, vehicles, time, location, and road characteristics. These elements naturally form relationships, which makes the data suitable for a graph-based approach. I use Neo4j to build a property graph model for this project, following a design that balances clarity, performance, and extensibility.

2.1 Modelling Strategy and Structure Mapping

Step1: Understand the Dataset

The dataset contains 25 columns. Each row represents one person who died in a crash, along with information about the crash itself, the vehicle involved, the road and location, and the time of the fatal crash. I group the fields into six categories:

- Crash-related: Crash ID, Crash Type, Number of Fatalities
- Person: Age, Gender, Road User Type, Age Group
- Vehicle involvement: Bus, Rigid Truck, Articulated Truck
- Time: Year, Month, Day of Week, Time of Day, Holiday Flags
- Location: State, SA4, LGA, Remoteness
- Road: Speed Limit, National Road Type

I use this grouping to identify potential nodes and properties for the graph. Fields that represent independent entities can be modelled as nodes. Fields that simply describe an entity can be stored as properties.

Step 2: Research and Modelling Guidance

To guide my design decisions, I refer to several sources.

Neo4j's official modelling guidelines recommend designing around how users ask questions, not just how data is stored in traditional relational databases. Their approach suggests using nodes for real-world entities and properties for descriptive attributes, unless those attributes are frequently reused or queried across relationships [3].

Robinson et al. in Graph Databases further emphasise avoiding over-normalisation, which can lead to unnecessarily complex queries and longer traversal paths [4]. They recommend keeping related information in the same node where possible, to support more efficient filtering and aggregation.

AWS also advises modelling graphs based on access patterns, which questions users want to answer, rather than rigid entity structures [5]. This idea reinforces my decision to group frequently filtered fields (like time and region) into compact nodes that simplify queries.

I also review two traffic knowledge graph studies:

- Zhang et al. [6] built a graph model where time is represented as a unified node. This approach improves filtering for seasonal or time-based crash analysis. I adopt the same strategy in creating a TimeInfo node that combines time attributes.

- Yuan et al. [7] proposed modelling location as a single spatial node with multi-level geographic data stored as attributes. This help me design the Location node to include LGA, SA4, state, and remoteness in one place, avoiding hierarchical joins that don't always apply due to inconsistent region coding.

In addition, a recent study by Gu et al. [8] used graph-based association rule mining to analyse crash patterns. Their work shows that graph structures can capture hidden multi-factor relationships more effectively than flat data tables. This supports my decision to develop a flexible and extensible graph model that can later incorporate new factors like weather or road control measures.

Step 3: Node vs Property Decisions

A central modelling choice is to decide what should be a node and what should remain a property. Some fields, like Crash, Person, and Vehicle, are clearly distinct entities and are modelled as nodes. Descriptive fields like Gender, Age, and Speed Limit are used as properties.

Table 2.1 summarises this classification across the dataset.

Table 2.1. Field-to-Graph Mapping Summary

Dataset Column	Mapped To Node	Comment
Crash ID	Crash	Unique crash event
Crash Type, Number of Fatalities	Crash	Crash-level attributes
Gender, Age, Age Group, Road User Type	Killed_Person	Victim characteristics
Bus, Rigid Truck, Articulated Truck	Vehicle	Current info is minimal; modelled as node for future extensibility
Speed Limit, Road Type	Road	Could later include more detailed road data
Year, Month, Time of Day, Holidays	TimeInfo	Aggregated from date/time attributes
SA4, LGA, State, Remoteness	Location	Combined to reflect spatial context

Others require more consideration. For example, the **Vehicle** and **Road** fields contain limited information in the current dataset. However, they represent physical entities that could later be expanded. For instance, if future data includes vehicle make, model, or crash speed, or if road conditions or traffic controls are added, having these as nodes will make the graph easier to scale without major restructuring. Based on this, both are modelled as nodes.

However, time and location fields require more careful design, as there are multiple ways to model them depending on how much normalisation is needed. The next section explains and compares two design versions.

2.2 Time and Location Design Justification

2.2.1 Version 1: Normalised Structure

In the first version, I separate time and location into multiple nodes:

- Date: year, month, day of week
- TimeDetail: time of day, holiday flags
- Location: broken down into LGA, SA4, State, and Remoteness, linked through relationships like BELONGS_TO

This design aims to normalise repeated values and enable reuse. However, it makes queries more complex and required deeper traversal. Also, in the dataset, the regional fields don't always follow consistent hierarchies. For example, LGA and SA4 mappings are especially inconsistent. Therefore, I have to remove the relationship between LGA and SA4, and add another relationship between Crash and SA4.

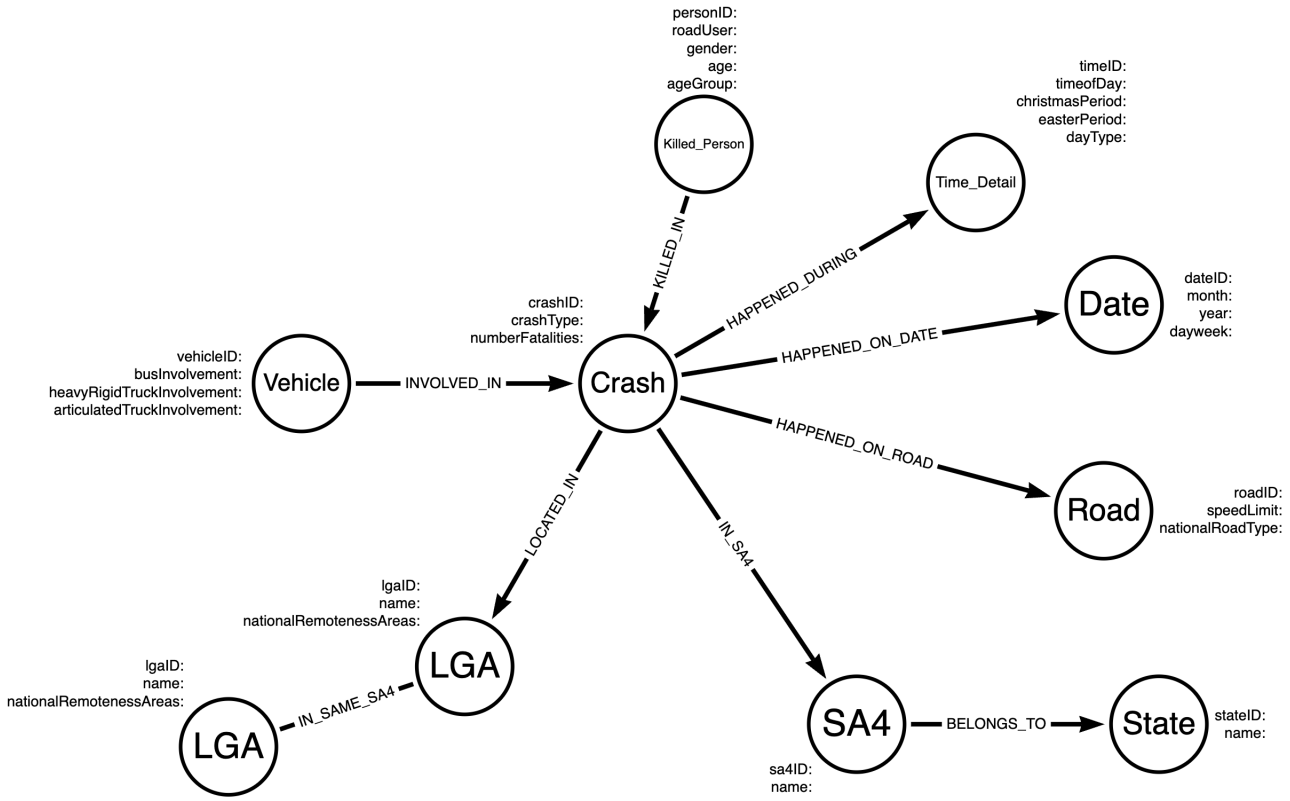


Figure 2.1. Early Graph Schema – Version 1

Version 2: Flattened Structure

In the second version, I simplify the model by combining related fields:

- TimeInfo: all time-related attributes (year, month, time of day, holiday flags)
- Location: all spatial fields (LGA, SA4, state, remoteness)

This design avoids complex joins and supports more efficient queries. It aligns with Zhang et al.'s [6] and Yuan et al.'s [7] strategies, but is adapted to my dataset's limitations by avoiding rigid hierarchies. Each crash links directly to a Location and TimeInfo node, which store all relevant attributes as properties.

This results in a star-shaped graph, with Crash at the centre and connected to:

- KILLED_IN → Killed_Person
- INVOLVED_IN → Vehicle
- HAPPENED_ON → Road
- HAPPENED_DURING → TimeInfo
- LOCATED_IN → Location

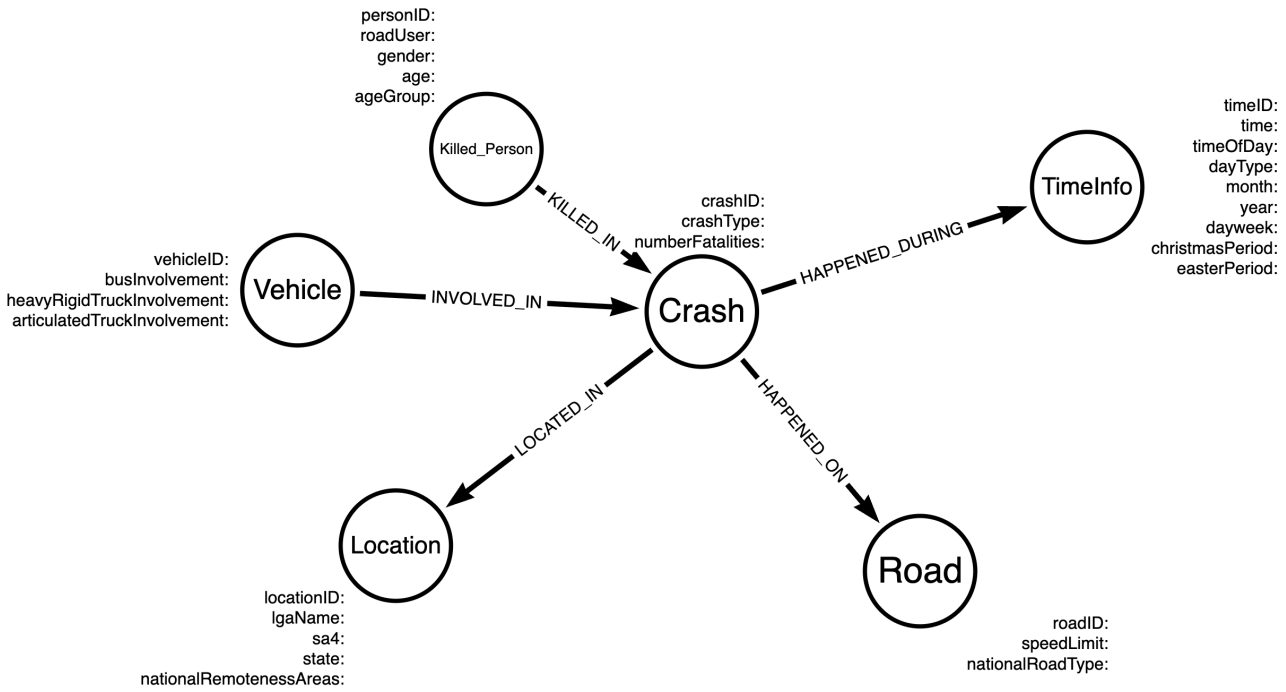


Figure 2.2. Final Graph Schema – Used in this Project

2.2.3 Why I Chose Version 2

I test both versions using the same set of queries. The results are shown in Table 2.3.

Table2.2 Execution Time

Query	Version 1	Version 2
Query a	28ms	27ms
Query b	24ms	23ms
Query c	27ms	25ms
Query d	23	17ms
Query e	60ms	45ms
Query g	20ms	17ms

Version 2 consistently perform better, especially for queries involving both time and region filtering. It also produces shorter and clearer Cypher queries. While the performance difference in milliseconds is small, the improvement in query complexity and maintainability is meaningful.

2.3 Evaluation

The final graph model provides a practical balance of simplicity, performance, and flexibility. Its strengths include:

- A clean star-shaped structure centred on Crash.
- Simplified nodes (TimeInfo, Location) that group related fields.

- Flexibility: The model supports future extensions (e.g., weather, vehicle speed, traffic controls)
- Design is supported by academic and industry modelling recommendations

The main limitations are:

- Flattened nodes lose some structural hierarchy
- The dataset does not allow modelling person-vehicle relationships.
- Adding new entities still requires changes to the schema.

Overall, the model supports all current project queries and provides a strong base for further development or integration with graph analytics.

3. ETL process

This section explains how I convert the original dataset into node and relationship CSV files for Neo4j. The process is done in a Jupyter Notebook using Python and pandas. The goal is to prepare clean, connected data that matches the graph design in Section 2.

3.1 Dataset and Preparation

The source file is Project2_Dataset_Corrected.csv. It has 10,490 rows and 25 columns. Each row represents one fatal crash case between 2014 and 2024. The dataset includes crash details, vehicle types, person info, road data, location, and time.

I start by checking the dataset:

- All columns have full data, so I do not need to drop any rows.
- I check the data types. The Time column is changed to HH:mm format using string padding. This makes time-based grouping easier later.

	ID	Crash ID	State	Month	Year	Dayweek	Time	Crash Type	Number Fatalities	Bus Involvement	...	Age	National Remoteness Areas	SA4 Name 2021	Na LGA
0	1	20241115	NSW	12	2024	Friday	4:00	Single	1	No	...	74	Inner Regional Australia	Riverina	
1	2	20241125	NSW	12	2024	Friday	6:15	Single	1	No	...	19	Inner Regional Australia	Sydney - Baulkham Hills and Hawkesbury	Hawk
2	3	20246013	TAS	12	2024	Friday	9:43	Single	1	No	...	33	Inner Regional Australia	Launceston and North East	Mi
3	4	20241002	NSW	12	2024	Friday	10:35	Single	1	No	...	32	Outer Regional Australia	New England and North West	Ar

Figure 3.1 – First five rows of the original dataset loaded in Jupyter Notebook

I use Python's apply() function with a lambda expression to format the time values. This part of the logic is clarified with the help of ChatGPT [2], particularly in understanding how to handle string padding and null checks in pandas.

```
# Standardise the 'Time' column to HH:mm format (e.g., 4:00 → 04:00)
crash_df['Time'] = crash_df['Time'].apply(
    lambda t: f"{t.split(':')[0].zfill(2)}:{t.split(':')[1].zfill(2)}" if pd.notnull(t) and ':' in t else t
)
```

Figure 3.2 – Code snippet for formatting and standardising the Time column

3.2 Node File Creation

I create one node file for each key entity: Crash, Killed_Person, Vehicle, TimeInfo, Road, and Location. For each node, I:

1. Select the related columns.
2. Remove duplicate rows to ensure each node is unique.
3. Add an ID column if needed.

Table3.1 Code and output showing how Crash and TimeInfo nodes are created

Node Type	Columns Used	Output File
Crash	Crash ID, Crash Type, Number Fatalities	crash_node.csv
Killed_Person	Gender, Age, Road User, Age Group	killed_person_node.csv
Vehicle	Bus Involvement, Heavy Rigid Truck, Articulated Truck	vehicle_node.csv
TimeInfo	Time, Day of Week, Year, Month, Holiday Flags	timeinfo_node.csv
Road	Speed Limit, National Road Type	road_node.csv
Location	State, SA4, LGA, Remoteness	location_node.csv

Crash Node

Each crash is unique by Crash ID. I include Crash Type and Number Fatalities as properties. I drop duplicates because there may be multiple people killed in the same crash.

```
# Create Crash node file
# Extract relevant columns and remove duplicate crash records
crash_node_table = crash_df[['Crash ID', 'Crash Type', 'Number Fatalities']].drop_duplicates()

# Export the result to a CSV file
crash_node_table.to_csv('csv_full_datasets/crash_node.csv', index=False)

# Print the number of unique crash nodes
print(f"Crash nodes: {crash_node_table.shape}")
```

Figure 3.3 – Sample codes for creating node

Crash ID	Crash Type	Number Fatalities
20241115	Single	1
20241125	Single	1
20246013	Single	1
20241002	Single	1
20243185	Single	1
20244016	Single	1
20243168	Single	1
20246003	Single	1

Figure 3.4 – Sample records in the crash_node.csv file

Vehicle Node

This node uses three columns about vehicle involvement: Bus, Heavy Rigid Truck, and Articulated Truck. I remove duplicate combinations to keep unique combinations and add a vehicleID.

Killed_Person Node

This node uses person info like Gender, Age, Age Group, and Road User. I rename the column ID to personID. No duplicates are dropped, since each row already represents one person.

TimeInfo Node

This node includes fields like Year, Month, Dayweek, Time of day, and holiday period. I combine them into one node and remove duplicate time patterns. I add a timeID.

Road Node

This node uses Speed Limit and National Road Type. Each unique pair is one road type. I drop duplicates and add a roadID.

Location Node

I merge State, SA4, LGA, and Remoteness into one node with four properties. I drop duplicates so each unique location appears once. I add a locationID.

3.3 Relationship File Creation

I then create the relationship files based on how nodes are connected in the graph model. Each file contains two columns: one for the start node ID and one for the end node ID. I name these files using the format rel_[relationship_name].csv.

Table 3.2 how relationship files are built using IDs

Relationship	From -> To	Output File
KILLED_IN	Killed_Person -> Crash	rel_killed_in.csv
INVOLVED_IN	Vehicle -> Crash	rel_involved_in.csv
HAPPENED_DURING	Crash -> TimeInfo	rel_happened_during.csv

Relationship	From -> To	Output File
HAPPENED_ON	Crash -> Road	rel_happened_on.csv
LOCATED_IN	Crash -> Location	rel_located_in.csv

File Name	Type	Description
rel_killed_in.csv	Relationship	Person-to-Crash
rel_involved_in.csv	Relationship	Vehicle-to-Crash
rel_happened_during.csv	Relationship	Crash-to-Time
rel_happened_on.csv	Relationship	Crash-to-Road
rel_located_in.csv	Relationship	Crash-to-Location

KILLED_IN

Links a person to the crash they were killed in. Each row has a personID and crashID.

INVOLVED_IN

Links vehicle info to a crash. I match vehicle involvement fields to vehicleID.

HAPPENED_DURING

Links crash to time info. I join on all time fields to get the right timeID.

```
# Create HAPPENED_DURING relationship between Crash and TimeInfo
# Join based on all time-related fields to get corresponding timeID
rel_happened_during_table = crash_df[['Crash ID', 'Time', 'Time of day', 'Day of week', 'Month', 'Year',
                              'Dayweek', 'Christmas Period', 'Easter Period']].drop_duplicates()

rel_happened_during_table = rel_happened_during_table.merge(
    timeinfo_node_table,
    on=['Time', 'Time of day', 'Day of week', 'Month', 'Year', 'Dayweek',
        'Christmas Period', 'Easter Period'],
    how='left'
)

rel_happened_during_table = rel_happened_during_table[['Crash ID', 'timeID']].rename(columns={'Crash ID': 'crashID'})
rel_happened_during_table.to_csv('csv_full_datasets/rel_happened_during.csv', index=False)
print(f"HAPPENED_DURING relationships: {len(rel_happened_during_table)}")
```

Figure 3.5 – Code snippet showing .merge() used to create HAPPENED_ON relationship

crashID	timeID
20241115	1
20241125	2
20246013	3
20241002	4
20243185	5
20244016	6
20243168	7
20246003	8
20246001	9
20244081	10

Figure 3.6 – Sample records in rel_happened_on.csv relationship file

HAPPENED_ON

Links crash to road info using speed and road type to match roadID.

LOCATED_IN

Links crash to location info using State, SA4, LGA, and Remoteness.

3.4 Output Summary

At the end of the process, I have:

- 6 node files: Crash, Killed_Person, Vehicle, TimeInfo, Road, Location
- 5 relationship files: KILLED_IN, INVOLVED_IN, HAPPENED_DURING, HAPPENED_ON, LOCATED_IN

All files are saved as CSV and follow the property graph structure.

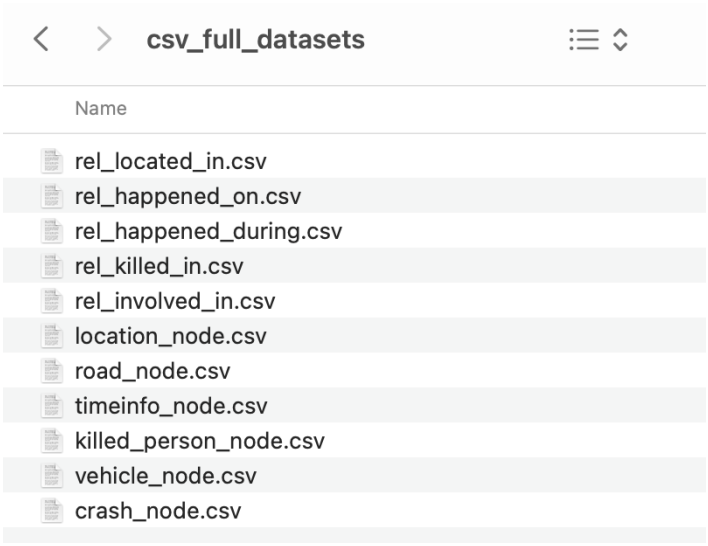


Figure 3.7 – Output folder with all 6 node and 5 relationship CSV files

3.5 Data Processing Decisions

1. I drop duplicates in most nodes to keep only one row per unique entity (e.g., crash, location).
2. I format time to make queries easier (e.g., filter by time of day).
3. I generate IDs like vehicleID, timeID, and locationID to link data clearly.
4. I combine fields like State, SA4, and LGA into one node to simplify the graph and make queries more efficient.

After testing, I find that all queries run fast enough without filtering. So I keep the full dataset and do not apply any filters in this version of the ETL.

4. Graph Database Implementation

This section describes how I implement the graph database using Neo4j Desktop. I use Cypher scripts to load the cleaned CSV files into the graph, following the model designed in Section 2.

4.1 Create Graph Database and Import CSV Files

I creat a new Neo4j DBMS named Crash-Graph-DB and activate it in Neo4j Desktop. All node and relationship CSV files generated in the ETL process are copied into the /import folder.

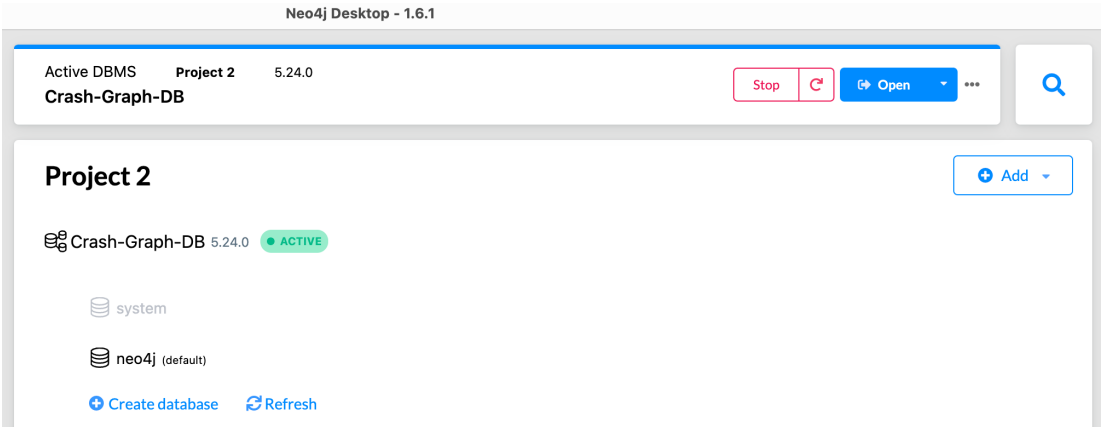


Figure 4.1.1 – Creating and activating the DBMS

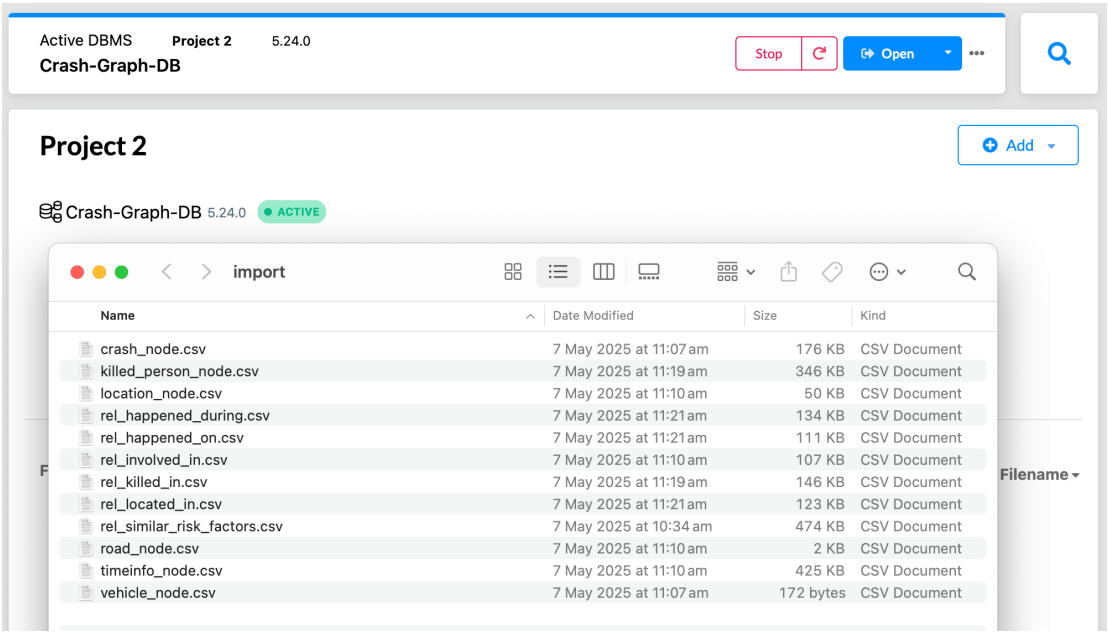


Figure 4.1.2 – All CSV files placed in the import folde

Once all the files are ready, I open the database in Neo4j Browser, where I run Cypher scripts to load the data.

4.2 Load Node Tables

I use LOAD CSV WITH HEADERS to load each node type into the graph. Each file contains one type of entity, and I used Cypher to define node labels and properties.

Example - **Crash Node**

```
// 1. Create Crash nodes
// Each crash has a crash ID, type, and total number of fatalities
LOAD CSV WITH HEADERS FROM 'file:///crash_node.csv' AS row
CREATE (:Crash {
  crashID: row.`Crash ID`,
  crashType: row.`Crash Type`,
  numberFatalities: toInteger(row.`Number Fatalities`)
});
```

Table

Added 9683 labels, created 9683 nodes, set 29049 properties, completed after 188 ms.

Figure 4.2.1 – Cypher script to load Crash nodes

Other node types (Vehicle, Killed_Person, TimeInfo, Road, and Location) are loaded in the same way, using their respective CSV files and appropriate property mappings.

4.3 Load Relationship Tables

After all nodes are created, I use Cypher MATCH and CREATE to establish the relationships between them.

Example: Vehicle -> [INVOLVED_IN] -> Crash

```
// 1. Vehicle → Crash
LOAD CSV WITH HEADERS FROM 'file:///rel_involved_in.csv' AS row
MATCH (v:Vehicle {vehicleID: row.vehicleID})
MATCH (c:Crash {crashID: row.crashID})
CREATE (v)-[:INVOLVED_IN]→(c);
```

Table

Created 9683 relationships, completed after 11899 ms.

Figure 4.3.1 – Cypher query to create INVOLVED_IN relationship

The same process was used for INVOLVED_IN, HAPPENED_DURING, HAPPENED_ON, and LOCATED_IN relationships.

4.4 Graph Database Summary

After importing the data, I verified that all node labels and relationships were created correctly using Neo4j Browser.

Node Labels and Relationship Types

The screenshot below (Figure 4.4.1) shows all node labels and relationship types in the graph. There are 30,470 nodes and 49,222 relationships in total.

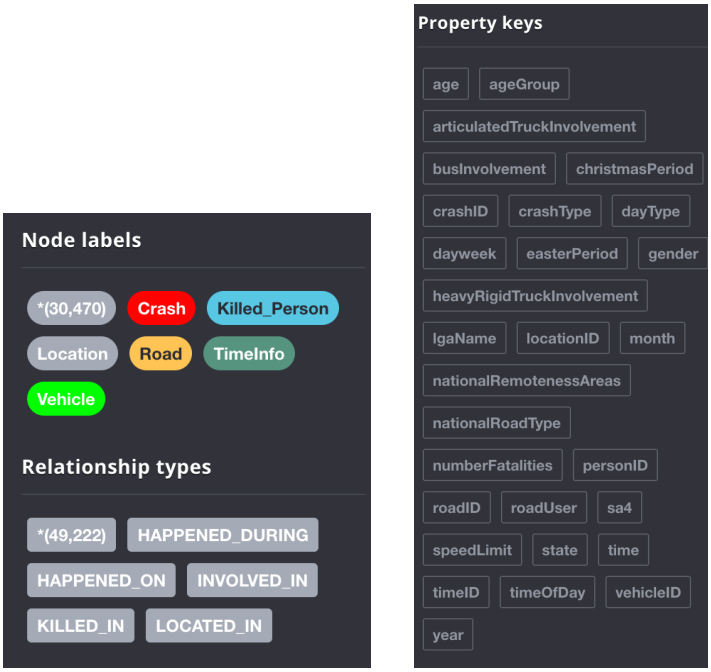


Figure 4.4.1 – Node labels, relationship types and property keys in Neo4j Browser

Graph Schema

I also run the CALL db.schema.visualization() command to show the structure of the graph. This helps confirm that all six node types are connected with the correct relationships.

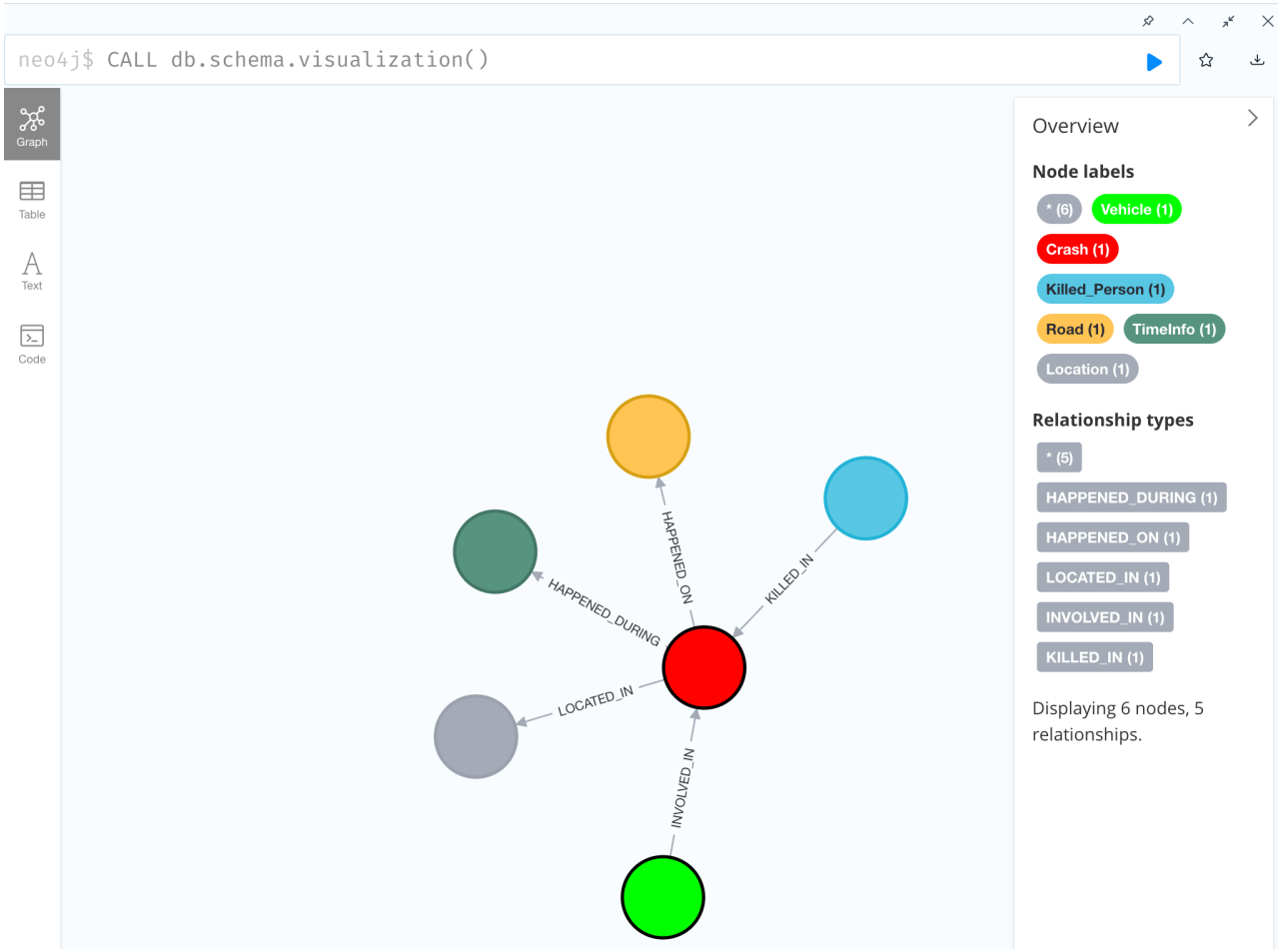


Figure 4.4.2 – Schema visualisation of the graph database

4.4.3 Summary Table of Nodes and Relationships

Table 4.4.1 – Summary of nodes and relationships

Node	# of Nodes	One Label	Description
Crash	9,683	Crash	Each crash event
Vehicle	7	Vehicle	Type of vehicle involved
Killed_Person	10,490	Killed_Person	People who died in crashes
TimeInfo	9,424	TimeInfo	Time, date, and holiday info
Road	80	Road	Road type and speed limit
Location	786	Location	State, region, remoteness area

Relationship	# of Relationships	Description
INVOLVED_IN	9,683	Vehicle -> Crash
KILLED_IN	10,490	Killed_Person -> Crash
HAPPENED_DURING	9,683	Crash -> TimeInfo

Relationship	# of Relationships	Description
HAPPENED_ON	9,683	Crash -> Road
LOCATED_IN	9,683	Crash -> Location

5. Cypher Queries

This section includes the Cypher queries used to answer the required questions in project brief. Each query includes the Cypher code and result screenshots.

5.1 Query a

Query

Find all crashes in WA from 2020-2024 where articulated trucks were involved and multiple fatalities (Number Fatalities>1) occurred. For each crash, provide the road user, age of each road user, gender of each road user, LGA Name, month and year of the crash, and the total number of fatalities.

Cypher code

```
MATCH (v:Vehicle)-[:INVOLVED_IN]→(c:Crash),
      (p:Killed_Person)-[:KILLED_IN]→(c),
      (c)-[:HAPPENED_DURING]→(t:TimeInfo),
      (c)-[:LOCATED_IN]→(l:Location)
WHERE l.state = 'WA'
  AND t.year ⩾ 2020 AND t.year ⩽ 2024
  AND v.articulatedTruckInvolvement = 'Yes'
  AND c.numberFatalities > 1
RETURN DISTINCT
    p.roadUser AS `Road User`,
    p.age AS `Age`,
    p.gender AS `Gender`,
    l.lgaName AS `LGA Name`,
    t.month AS `Month`,
    t.year AS `Year`,
    c.numberFatalities AS `Total Number of Fatalities`
ORDER BY `Year`, `Month`;
```

Output

```
neo4j$ MATCH (v:Vehicle)-[:INVOLVED_IN]→(c:Crash), (p:Killed_Person)-[:KILLED_IN…
```

Table | Text | Code

	Road User	Age	Gender	LGA Name	Month	Year	Total Number of Fatalities
1	"Driver"	58	"Female"	"Busselton"	11	2020	2
2	"Passenger"	51	"Female"	"Busselton"	11	2020	2
3	"Driver"	56	"Male"	"Dundas"	12	2020	2
4	"Driver"	58	"Male"	"Dundas"	12	2020	2

Figure 5.1: Output for Query a

5.2 Query b

Query

Find the maximum and minimum age for female and male motorcycle riders who were involved in fatal crashes during the Christmas Period or Easter Period in inner regional Australia. Output the following information: gender, maximum age and minimum age. (Hint: Zero results is a meaningful result in itself.)

Cypher code

```
MATCH (p:Killed_Person)-[:KILLED_IN]→(c:Crash),
      (c)-[:HAPPENED_DURING]→(t:TimeInfo),
      (c)-[:LOCATED_IN]→(l:Location)
WHERE p.roadUser = 'Motorcycle rider'
  AND (t.christmasPeriod = 'Yes' OR t.easterPeriod = 'Yes')
  AND l.nationalRemotenessAreas = 'Inner Regional Australia'
RETURN
  p.gender AS `Gender`,
  MAX(p.age) AS `Maximum Age`,
  MIN(p.age) AS `Minimum Age`;
```

Output

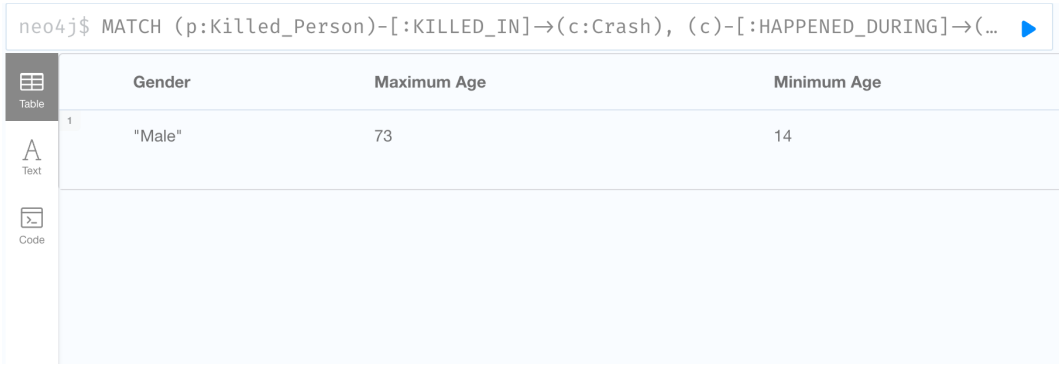


Figure 5.2: Output for Query b

5.3 Query c

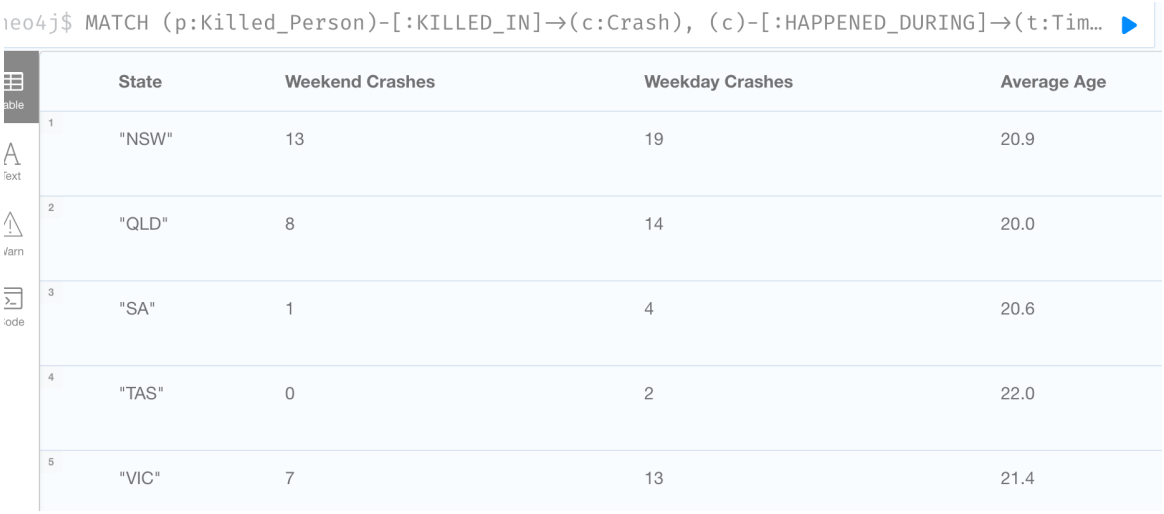
Query

How many young drivers (Age Group = '17_to_25') were involved in fatal crashes on weekends vs. weekdays in each state during 2024? Output 4 columns: State name, weekends, weekdays, and the average age for all young drivers (Age Group = '17_to_25') who were involved in fatal crashes in each State.

Cypher code

```cypher
MATCH (p:Killed_Person)-[:KILLED_IN]→(c:Crash),
      (c)-[:HAPPENED_DURING]→(t:TimeInfo),
      (c)-[:LOCATED_IN]→(l:Location)
WHERE p.ageGroup = '17_to_25'
  AND p.roadUser = 'Driver'
  AND t.year = 2024
WITH DISTINCT p.personID AS pid, p.age AS age, t.dayType AS dayType, l.state AS stateName
RETURN
  stateName AS `State`,
  COUNT(CASE WHEN dayType = 'Weekend' THEN 1 END) AS `Weekend Crashes`,
  COUNT(CASE WHEN dayType = 'Weekday' THEN 1 END) AS `Weekday Crashes`,
  ROUND(AVG(age), 1) AS `Average Age`
ORDER BY `State`;
```

Output



	State	Weekend Crashes	Weekday Crashes	Average Age
1	"NSW"	13	19	20.9
2	"QLD"	8	14	20.0
3	"SA"	1	4	20.6
4	"TAS"	0	2	22.0
5	"VIC"	7	13	21.4

Figure 5.3: Output for Query c

5.4 Query d

Query

Identify all crashes in WA that occurred Friday (but categorised as a weekend) and resulted in multiple deaths, with victims being both male and female. For each crash, output the SA4 name, national remoteness areas, and national road type.

Cypher code

```
MATCH (p:Killed_Person)-[:KILLED_IN]→(c:Crash),
      (c)-[:HAPPENED_DURING]→(t:TimeInfo),
      (c)-[:LOCATED_IN]→(l:Location),
      (c)-[:HAPPENED_ON]→(r:Road)
WHERE l.state = 'WA'
  AND t.dayweek = 'Friday'
  AND t.dayType = 'Weekend'
  AND c.numberFatalities > 1
WITH c,
     l.sa4 AS sa4Name,
     l.nationalRemotenessAreas AS remoteness,
     r.nationalRoadType AS roadType,
     COLLECT(p.gender) AS genders
WHERE 'Male' IN genders AND 'Female' IN genders
RETURN DISTINCT
  sa4Name AS `SA4 Name`,
  remoteness AS `National Remoteness Areas`,
  roadType AS `National Road Type`
ORDER BY `SA4 Name`;
```

Output

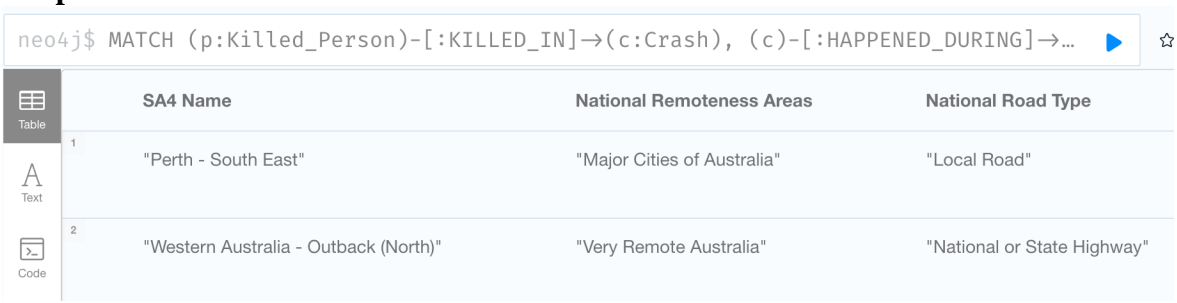


Figure 5.4: Output for Query d

5.5 Query e

Query

Find the top 5 SA4 regions where the highest number of fatal crashes occur during peak hours (Time between 07:00-09:00 and 16:00-18:00). For each SA4 region, output the name of the region and the separate number of crashes that occurred during morning peak hours and afternoon peak hours (Renamed Morning Peak and Afternoon Peak).

Cypher code

```cypher
MATCH (c:Crash)-[:HAPPENED_DURING]→(t:TimeInfo),
      (c)-[:LOCATED_IN]→(l:Location)
WHERE (t.time ⩾ '07:00' AND t.time ⩽ '09:00')
   OR (t.time ⩾ '16:00' AND t.time ⩽ '18:00')
WITH l.sa4 AS sa4Name,
     CASE
       WHEN t.time ⩾ '07:00' AND t.time ⩽ '09:00' THEN 'Morning Peak'
       ELSE 'Afternoon Peak'
     END AS peakPeriod
RETURN
  sa4Name AS `SA4 Name`,
  COUNT(CASE WHEN peakPeriod = 'Morning Peak' THEN 1 END) AS `Morning Peak`,
  COUNT(CASE WHEN peakPeriod = 'Afternoon Peak' THEN 1 END) AS `Afternoon Peak`,
  COUNT(*) AS `Total Crashes`
ORDER BY `Total Crashes` DESC
LIMIT 5;
```

Output

	SA4 Name	Morning Peak	Afternoon Peak	Total Crashes
1	"Wide Bay"	31	47	78
2	"South Australia - South East"	26	32	58
3	"Melbourne - South East"	23	34	57
4	"Capital Region"	23	30	53
5	"New England and North West"	18	34	52

Figure 5.5: Output for Query e

5.6 Query f

Query

Find paths with a length of 3 between any two LGAs. Return the top 3 paths, including the starting LGA and ending LGA for each path. Order results alphabetically by starting LGA and then ending LGA.

Cypher code

```cypher
MATCH path = (a:Location)-[*3]-(b:Location)
WHERE a.lgaName < b.lgaName
RETURN
  a.lgaName AS StartLGA,
  b.lgaName AS EndLGA,
  length(path) AS PathLength
ORDER BY StartLGA, EndLGA
LIMIT 3;
```

Output

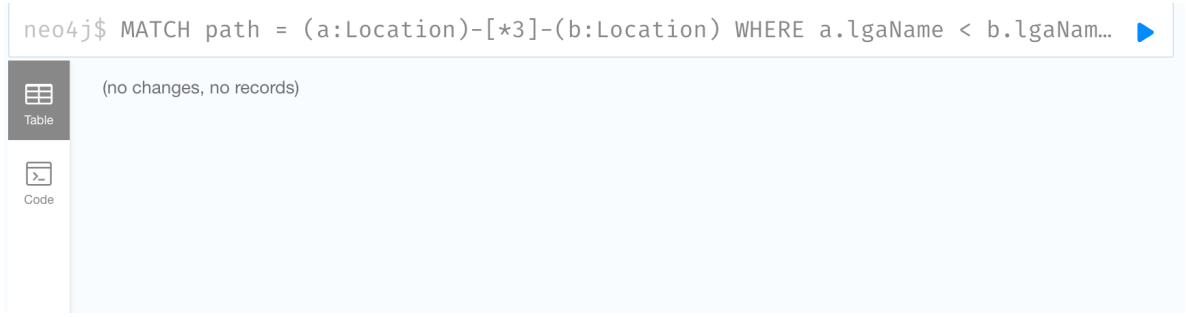


Figure 5.6: Output for Query f

This query returns no result because in the current graph design, there are no direct or indirect connections between different Location (LGA) nodes. Each Crash node is connected to only one Location, and Location nodes are not linked to each other.

As a result, there is no valid path of length 3 between any two LGA nodes. To support such queries in future, an additional layer of abstraction (e.g., shared crashes) would need to be modeled explicitly.

5.7 Query g

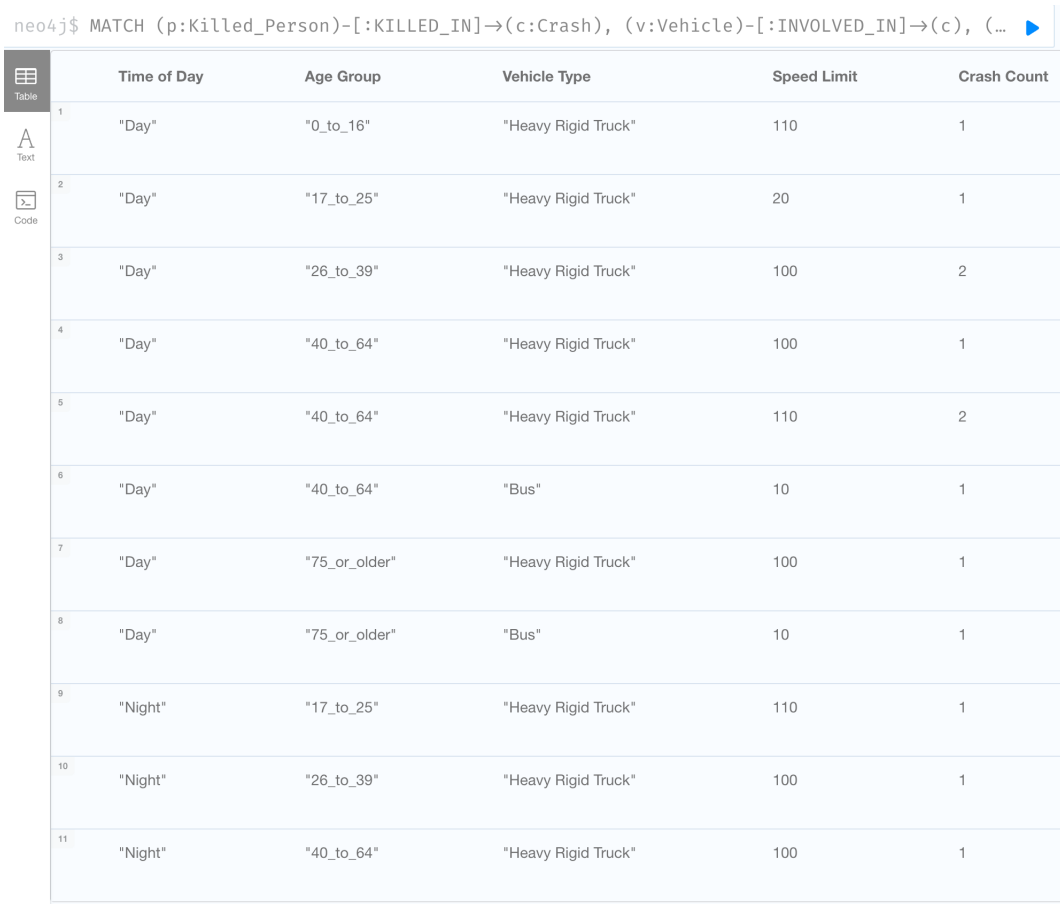
Query

Find all weekday fatal crashes involving pedestrians where either buses or heavy rigid trucks were present in speed zones less than 40 or greater than/equal to 100. Group these crashes by unique combinations of time of day, age group, vehicle type (bus or heavy rigid truck), and speed limitation. For each group, count the number of crashes that occurred. Output a table showing time of day, age group, vehicle type, crash count, and speed limitation, sorted first by time of day (ascending) and then by age group (ascending).

Cypher code

```cypher
MATCH (p:Killed_Person)-[:KILLED_IN]→(c:Crash),
      (v:Vehicle)-[:INVOLVED_IN]→(c),
      (c)-[:HAPPENED_DURING]→(t:TimeInfo),
      (c)-[:HAPPENED_ON]→(r:Road)
WHERE p.roadUser CONTAINS 'Pedestrian'
  AND t.dayType = 'Weekday'
  AND (v.busInvolvement = 'Yes' OR v.heavyRigidTruckInvolvement = 'Yes')
  AND (r.speedLimit < 40 OR r.speedLimit ⩾ 100)
WITH DISTINCT
      c.crashID AS crashID,
      t.timeOfDay AS `Time of Day`,
      p.ageGroup AS `Age Group`,
      CASE
        WHEN v.busInvolvement = 'Yes' THEN 'Bus'
        ELSE 'Heavy Rigid Truck'
      END AS `Vehicle Type`,
      r.speedLimit AS `Speed Limit`
RETURN
  `Time of Day`,
  `Age Group`,
  `Vehicle Type`,
  `Speed Limit`,
  COUNT(*) AS `Crash Count`
ORDER BY `Time of Day`, `Age Group`;
```

Output



	Time of Day	Age Group	Vehicle Type	Speed Limit	Crash Count
1	"Day"	"0_to_16"	"Heavy Rigid Truck"	110	1
2	"Day"	"17_to_25"	"Heavy Rigid Truck"	20	1
3	"Day"	"26_to_39"	"Heavy Rigid Truck"	100	2
4	"Day"	"40_to_64"	"Heavy Rigid Truck"	100	1
5	"Day"	"40_to_64"	"Heavy Rigid Truck"	110	2
6	"Day"	"40_to_64"	"Bus"	10	1
7	"Day"	"75_or_older"	"Heavy Rigid Truck"	100	1
8	"Day"	"75_or_older"	"Bus"	10	1
9	"Night"	"17_to_25"	"Heavy Rigid Truck"	110	1
10	"Night"	"26_to_39"	"Heavy Rigid Truck"	100	1
11	"Night"	"40_to_64"	"Heavy Rigid Truck"	100	1

Figure 5.7: Output for Query g

6. Self-designed Queries

In this section, two self-designed queries are designed to further explore t he relationships in the graph structure.

6.1 Query 1

Query: Fatal Pedestrian Crashes Involving Young or Female Victims During Day vs Night

Find all fatal crashes from 2020 involving either young pedestrians (Age Group = '17_to_25') or female pedestrians, that happened on roads with a speed limit of 100 km/h or more and involved a bus, heavy rigid truck, or articulated truck. Group the results by state and time of day (split into day vs night), and count the number of crashes in each group.

Why this query is designed

This query is designed to explore whether fatal crashes involving vulnerable pedestrians (young people or females) are more likely to happen at night or during the day, and in which states. It focuses on high-speed roads and large vehicles, which are often involved in more severe crashes. By simplifying the time into day vs night, it gives a clear comparison of crash risk across different states and times of day.

Cypher code

```
MATCH (p:Killed_Person)-[:KILLED_IN]→(c:Crash),
      (c)←[:INVOLVED_IN]-(v:Vehicle),
      (c)-[:HAPPENED_DURING]→(t:TimeInfo),
      (c)-[:HAPPENED_ON]→(r:Road),
      (c)-[:LOCATED_IN]→(loc:Location)
WHERE p.roadUser = 'Pedestrian'
  AND (
    p.ageGroup = '17_to_25' OR
    p.gender = 'Female'
  )
  AND t.year ⩾ 2020
  AND (
    v.busInvolvement = 'Yes' OR
    v.heavyRigidTruckInvolvement = 'Yes' OR
    v.articulatedTruckInvolvement = 'Yes'
  )
  AND r.speedLimit ⩾ 100
WITH
  loc.state AS `State`,
  CASE
    WHEN t.timeOfDay IN ['Evening', 'Night', 'Early Morning'] THEN 'Night'
    ELSE 'Day'
  END AS `Time of Day`,
  COUNT(DISTINCT c) AS `Crash Count`
RETURN
  `State`, `Time of Day`, `Crash Count`
ORDER BY `State` ASC, `Time of Day` DESC;
```

Output

```
neo4j$ MATCH (p:Killed_Person)-[:KILLED_IN]→(c:Crash), (c)←[:INVOLVED_IN]
```

Table | Text | Code

	State	Time of Day	Crash Count
1	"NSW"	"Night"	4
2	"NSW"	"Day"	2
3	"NT"	"Night"	1
4	"QLD"	"Night"	2
5	"QLD"	"Day"	2
6	"VIC"	"Day"	2

Figure 6.1: Output for Query 1

The query returned 6 rows. Each row shows a state, whether the crashes happened during the day or at night, and how many crashes happened under those conditions.

From the result:

- NSW had the most night-time crashes (4), and 2 during the day.
- QLD had 2 crashes at night and 2 during the day.
- VIC had 2 crashes during the day.
- NT had 1 crash at night.

This shows that fatal pedestrian crashes involving vulnerable people and large vehicles do not only happen at night. Some states (like NSW) may need to focus more on night-time road safety.

6.2 Query 2:

Query: Holiday Crashes Involving Large Vehicles in Regional and Remote Areas

Find all fatal crashes from 2020 that occurred during the Christmas or Easter periods, involved a bus, heavy rigid truck, or articulated truck, and happened in regional or remote areas (excluding major cities). Group the results by holiday period, time of day, road type, and remoteness level, and count the number of crashes and total fatalities in each group.

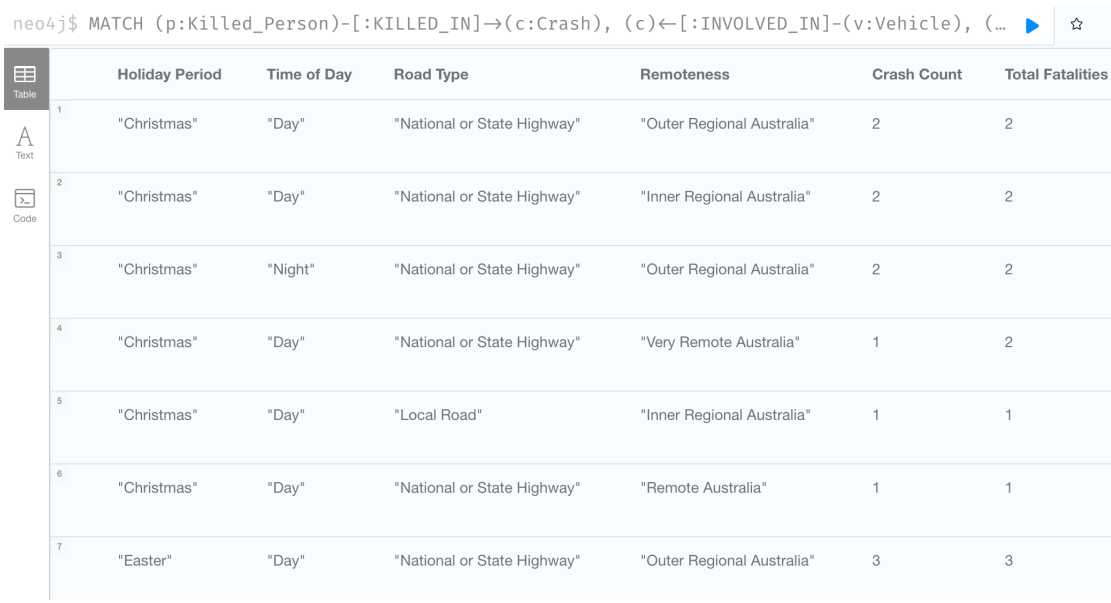
Why this query is designed

This query is designed to examine fatal crashes that happen during national holiday periods, which are times when road traffic often increases and people travel longer distances. It focuses specifically on crashes involving large vehicles and filters out those in major cities, to highlight risks in regional and remote areas. By grouping by holiday period, time of day, road type, and remoteness, the query helps identify patterns in when and where these crashes are most likely to happen. This information can help target road safety campaigns during public holidays.

Cypher code

```
MATCH (p:Killed_Person)-[:KILLED_IN]→(c:Crash),
      (c)←[:INVOLVED_IN]-(v:Vehicle),
      (c)-[:HAPPENED_ON]→(r:Road),
      (c)-[:LOCATED_IN]→(loc:Location),
      (c)-[:HAPPENED_DURING]→(t:TimeInfo)
WHERE t.year ⩾ 2020
  AND (
    t.christmasPeriod = 'Yes' OR
    t.easterPeriod = 'Yes'
  )
  AND (
    v.busInvolvement = 'Yes' OR
    v.heavyRigidTruckInvolvement = 'Yes' OR
    v.articulatedTruckInvolvement = 'Yes'
  )
  AND loc.nationalRemotenessAreas <> 'Major Cities of Australia'
WITH
  CASE
    WHEN t.christmasPeriod = 'Yes' THEN 'Christmas'
    ELSE 'Easter'
  END AS `Holiday Period`,
  t.timeOfDay AS `Time of Day`,
  r.nationalRoadType AS `Road Type`,
  loc.nationalRemotenessAreas AS `Remoteness`,
  c AS crash,
  p AS victim
WITH
  `Holiday Period`, `Time of Day`, `Road Type`, `Remoteness`,
  COUNT(DISTINCT crash) AS `Crash Count`,
  COUNT(victim) AS `Total Fatalities`
RETURN
  `Holiday Period`, `Time of Day`, `Road Type`, `Remoteness`, `Crash Count`, `Total
Fatalities`
ORDER BY `Holiday Period`, `Total Fatalities` DESC, `Crash Count` DESC;
```

Output



	Holiday Period	Time of Day	Road Type	Remoteness	Crash Count	Total Fatalities
1	"Christmas"	"Day"	"National or State Highway"	"Outer Regional Australia"	2	2
2	"Christmas"	"Day"	"National or State Highway"	"Inner Regional Australia"	2	2
3	"Christmas"	"Night"	"National or State Highway"	"Outer Regional Australia"	2	2
4	"Christmas"	"Day"	"National or State Highway"	"Very Remote Australia"	1	2
5	"Christmas"	"Day"	"Local Road"	"Inner Regional Australia"	1	1
6	"Christmas"	"Day"	"National or State Highway"	"Remote Australia"	1	1
7	"Easter"	"Day"	"National or State Highway"	"Outer Regional Australia"	3	3

Figure 6.2: Output for Query 2

The query returned 7 rows. Each row shows a combination of holiday period, time of day, road type, and remoteness level. For each group, it shows how many crashes happened and how many people were killed.

From the result:

- Most crashes happened during the Christmas period, especially on national or state highways.
- These crashes mostly occurred in outer regional, inner regional, and very remote areas.
- Crashes occurred both during the day and at night, with no clear pattern.
- Only one Easter-related group appeared, with 3 crashes and 3 fatalities in an outer regional area during the day.

This shows that fatal crashes involving large vehicles during holiday periods are more common in regional and remote areas, particularly on highways. Christmas appears to be more dangerous than Easter based on the number of crashes. These results suggest that regional road safety efforts should be increased during the Christmas period, especially on long-distance highways.

7. Graph Data Science Application

This project uses Neo4j to model the graph database in a star structure. Cypher queries can answer more targeted questions and mainly rely on predefined filters, but this approach is not ideal for discovering unknown patterns. At the same time, traditional crash analyses often study one crash's factors (such as driver age or road type) in isolation. However, fatal crashes often happen when several risks interact. For example, a high-speed road may not be dangerous by itself, but when combined with night-time and a young driver, it could be highly risky.

Graph Data Science (GDS) provides a way to detect deeper structures in connected data. Instead of isolated rows in a table, it helps us see these connections. Neo4j's GDS library provides several useful algorithms for this purpose [9]. In this section, I propose a GDS-based pipeline with three steps:

1. Build a crash-to-crash similarity graph
2. Find clusters of similar crashes (Louvain community detection)
3. Identify the most significant risk factor within those clusters (PageRank centrality)

7.1 Step1: Crash-to-Crash Similarity Graph

In my graph schema, each Crash node connects to several dimensions of information:

- Vehicle (e.g., heavyRigidTruckInvolvement, articulatedTruckInvolvement)
- TimeInfo (e.g., timeOfDay, dayType, easterPeriod)
- Road (e.g., speedLimit, nationalRoadType)
- Location (e.g., nationalRemotenessAreas, LGAName)
- Killed_Person (e.g., age, gender)

To apply clustering algorithms, I first need to build a new crash-to-crash similarity graph based on this star-shaped schema. In this similarity graph, each node is still a crash, but now crashes are connected to each other based on shared high-risk characteristics.

I can set up a weighted similarity score and a threshold. For example, shared attributes have different weights: night-time (0.3), speed ≥ 100 km/h (0.4), and truck involvement (0.25). If the total score is above the threshold (e.g., 0.65), these two crashes will be connected with a weighted edge. This process produces a crash similarity network showing how similar two crashes are.

7.2 Step2: Finding Similar Crash Clusters

The next step is to apply a clustering algorithm to this similarity graph. Neo4j provides several community detection methods, including K-means, Louvain, Label Propagation, Strongly Connected Components, and HDBSCAN [9]. I also consulted ChatGPT [2] to better understand the differences between these algorithms and how they apply to graph data. After comparing their features, I chose the Louvain algorithm.

Louvain does not require specifying the number of clusters in advance and supports weighted edges. It uses modularity to measure how well nodes are grouped. For example, a modularity score above 0.3 usually indicates meaningful structure. I also reviewed research by Lu et al. [10], which shows that Louvain performs well on large-scale networks and produces strong modularity values, even with millions of nodes.

After applying the Louvain algorithm, it might detect:

- community 0: night high-speed heavy truck
- community 1: city holiday crashes
- community 2: young driver weekends crashes.

Once the crash clusters are detected, they can be used to help local governments understand system-level risk patterns.

7.3 Step3: Identifying Influential Risk Factors

After grouping crashes, the next step is to identify which risk factors are most influential within each crash group. Neo4j provides centrality algorithms to evaluate the importance of individual nodes in a network [9].

PageRank assigns each node a score that increases if it is connected to other important nodes. This creates a ranking of risk factors based on network influence, not just frequency. A factor may not appear frequently, but if it connects many important or severe crashes, its PageRank score will be high. PageRank uses the idea of a random walker. At each step, the walker can follows a link or jumps to a random node. The chance of continuing the walk is controlled by a damping factor, usually set to 0.85 [11].

We can run PageRank within each community detected in Step 2. For example, if Louvain finds three communities, I can apply PageRank separately inside each one to find the most central risk factors for each crash type. For example, a rural road may not appear in many crashes. But if those crashes also involve speeding and night-time driving, the road could still have a high PageRank score.

Police departments can use PageRank scores to prioritise road-time combinations that have the most influence on serious crashes by placing more patrols or warning signs in those areas.

7.4 Application

This system-level framework can support a wide range of stakeholders. Governments can target risk zones, not just isolated crash sites. Police can focus resources on high-impact combinations of roads and time conditions identified through centrality analysis.

However, this method also depends on the quality and information within the dataset. In the future, this analysis could incorporate live data, such as traffic volume, weather, or event schedules. It

could also support real-time alerts. The long-term goal is to shift from only studying crashes after they happen to preventing them before they occur.

8. Conclusion

This project designs and builds a graph database for fatal crash data in Australia. The final graph schema includes six node types (Crash, Killed_Person, Vehicles, Road, TimeInfo, and Location), and they are connected through clear relationships. This star-shaped structure supports all required and self-designed queries. It is flexible enough to allow new nodes and relationships to be added in the future.

Compared to a relational database, this graph model makes it easier to explore complex connected relationships. Section 7 discusses how graph data science algorithms, such as community detection and centrality, can be applied to this graph model.

There are still some limitations that can be improved in future work. The dataset only includes fatal crashes and lacks relationships between people and vehicles. In the future, the dataset can be extended to include weather data, road information, vehicle names, etc. In Section 7, GDS is only discussed theoretically without implementation. This can be developed in the future to provide more practical support for real-world safety analysis.

References

[1] UWA CSSE. “Data Warehouse Project 2 - S1 2025.” The University of Western Australia. https://csse-uwa.gitbook.io/data-warehouse-project-2-s1-2025 (accessed Apr. 14, 2025).

[2] ChatGPT (2025), OpenAI. Accessed May 22, 2025. [Online]. Available: https://chat.openai.com/

[3] Neo4j. “Graph Data Modelling Tips.” Neo4j.com. https://neo4j.com/docs/getting-started/data-modeling/modeling-tips/ (accessed May 2, 2025).

[4] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2015.

[5] Amazon Web Services. “Graph Databases on AWS.” AWS.com. https://aws.amazon.com/nosql/graph/#topic-0 (accessed May 10, 2025).

[6] L. Zhang, X. Li, and M. Wang, “Analysis of traffic accident based on knowledge graph,” *J. Adv. Transp.*, vol. 2022, Article ID 3915467, 2022.

[7] D. Yuan, Y. Liu, X. Zhou, and J. Chen, “Architecture and application of traffic safety management knowledge graph based on Neo4j,” *Sustainability*, vol. 15, no. 9786, 2023.

[8] C. Gu, L. Zhao, and J. Li, “Multivariate analysis of roadway multi-fatality crashes using association rules and graph structures,” *PLOS ONE*, vol. 17, no. 10, 2022.

[9] Neo4j. “Graph Data Science Library Documentation.” Neo4j.com. https://neo4j.com/docs/graph-data-science/current/ (accessed May 20, 2025).

[10] H. Lu, M. Halappanavar, and A. Kalyanaraman, “Parallel heuristics for scalable community detection,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 6, pp. 1629–1642, 2017.

[11] L. Page, S. Brin, R. Motwani, and T. Winograd, “The PageRank citation ranking: Bringing order to the Web,” Tech. Rep., Stanford InfoLab, 1999.