



**UNIVERSITY OF WESTERN AUSTRALIA**

**CITS3401 PROJECT 2**

Graph Database Design: Fatalities

**Kennaldy Lukman**

**23 May 2025**

## Table of Contents

<b>I. Introduction</b>	<b>2</b>
<b>II. Design and Implementation of Graph Database</b>	<b>3</b>
2.1 Design of Graph Database	3
2.1.1 Nodes	3
2.1.2 Relationships	4
2.1.3 Design Strengths and Flaws	4
<b>III. ETL Process</b>	<b>5</b>
3.1 Extract and Transform Process in Python	5
3.2 Loading to Neo4j	7
<b>IV. Cypher Query Results</b>	<b>9</b>
4.1 Specified Cypher Queries	9
4.2 Other Cypher Queries	13
<b>V. Application of Graph Data Science (GDS)</b>	<b>16</b>
5.1 What is GDS?	16
5.2 Practicality of Database to GDS	16
5.3 Practical Application of GDS	16

## **I. Introduction**

Road safety is still a major problem in various countries all around the world, with millions dying each year due to fatal accidents. This is particularly true for Australia, which is currently facing a problem of rising road fatalities, where over 1300 people were involved in fatal crashes in 2024.<sup>[1]</sup> In light of this situation, it is crucial for the Australian government to find the reasons for these fatal accidents, in order to reduce the unnecessary loss of Australian lives.

A data-driven approach can provide some insights on the factors that influence road safety. This report outlines the design of a graph database based on the historical data of road fatalities in Australia, using various tools such as the Arrow App and Neo4j, a graph database. Through the insights found using queries in the database, and the application of graph data science, recommendations can be proposed to the Australian government, aimed at tackling the problem of rising road fatalities.

## II. Design and Implementation of Graph Database

### 2.1 Design of Graph Database

#### 2.1.1 Nodes

The graph database contains several nodes centered around the *Crash* node, and is interconnected by relationships. These nodes are entities that can be found from the original dataset.

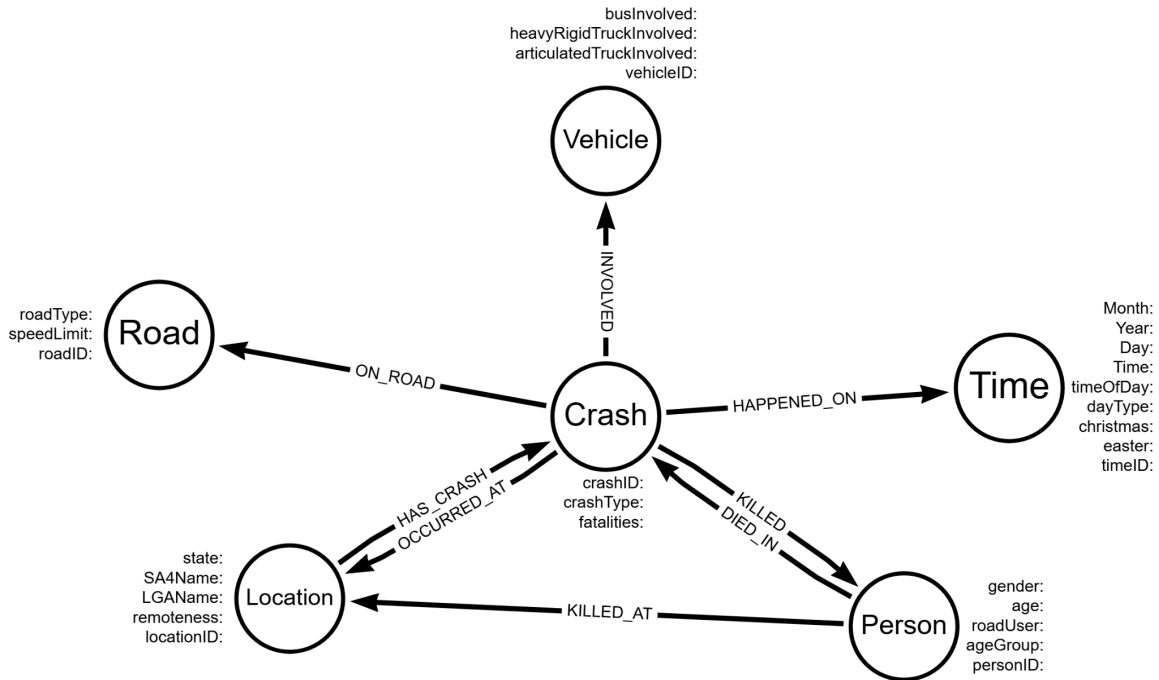


Fig 1.1 Graph Database Schema

The following are the entities (nodes) that can be found in our graph database:

- *Crash*: each crash event, using *crashID* as an identifier. Includes the number of fatalities involved in each crash.
- *Person*: people involved in the fatalities.
- *Location*: geographical attributes
- *Road*: road classification (e.g. Arterial Road, Highway, etc), including speed limits
- *Vehicle*: involvement of large vehicles, such as buses and trucks
- *Time*: time period, including holiday periods (e.g. Christmas, Easter)

Each entity, except for *Crash*, is provided with a surrogate key, and these surrogate keys will be used to form relationships between entities. For the node *Crash*, we will be using the *crashID* as the foreign key. Other than the entity *Crash*, each node is atomic and reusable. For example, multiple crashes may use the same locations and time periods.

### 2.1.2 Relationships

The graph schema above indicates the following relationships between nodes:

- *OCCURRED\_AT* (*Crash* → *Location*): where the crash occurred
  - *HAS\_CRASH* (*Location* → *Crash*) can be derived from the above relationship.
- *KILLED* (*Crash* → *Person*): people killed in the crash
  - Similarly, the relationship *DIED\_IN* (*Person* → *Crash*) can be derived.
- *ON\_ROAD* (*Crash* → *Road*): the road type where the crash occurred
  - This is **not the same** as the *OCCURRED\_AT* relationship.
- *INVOLVED* (*Crash* → *Vehicle*): vehicles involved in the crash
- *HAPPENED\_ON* (*Crash* → *Time*): time period of the crash

The relationships *HAS\_CRASH* and *DIED\_IN* are useful for graph database traversals in between nodes. Each of the relationships will contain foreign keys that link entities together.

### 2.1.3 Design Strengths and Flaws

The graph schema above clearly displays several strengths. Firstly, the central *Crash* node is connected to all the relevant contextual information, making any database queries relatively simple. The relationships are meaningful as it reflects real-life concepts, which enhances readability and clarity. The schema is relatively flexible, meaning that addition of several properties, even entities, may have minimal impact on the whole structure of the database.

However, the graph schema also displays several weaknesses. Notably, there are redundancies in the relationship, such as *HAS\_CRASH* and *DIED\_IN*, which can possibly affect query performance. Furthermore, there is a lack of properties in all the relationships, which decreases specificity of the database.

### III. ETL Process

#### 3.1 Extract and Transform Process in Python

The dataset is taken from a modified version of the Fatalities – December 2024 dataset, already converted into CSV format. This dataset is then read and transformed to the relevant tables in Python.

```
# reading base dataset
df = pd.read_csv('Project2_Dataset.csv')
df.head()
```

✓ 0.0s

	ID	Crash ID	State	Month	Year	Dayweek	Time	Crash Type	Number Fatalities	Bus Involvement	...	Age	National Remoteness Areas	SA4 Name 2021	National LGA Name 2024	National Road Type
0	1	20241115	NSW	12	2024	Friday	4:00	Single	1	No	...	74	Inner Regional Australia	Riverina	Wagga Wagga	Arterial Road
1	2	20241125	NSW	12	2024	Friday	6:15	Single	1	No	...	19	Inner Regional Australia	Sydney - Baulkham Hills and Hawkesbury	Hawkesbury	Local Road
2	3	20246013	TAS	12	2024	Friday	9:43	Single	1	No	...	33	Inner Regional Australia	Launceston and North East	Northern Midlands	Local Road
3	4	20241002	NSW	12	2024	Friday	10:35	Single	1	No	...	32	Outer Regional Australia	New England and North West	Armidale	National or State Highway
4	5	20243185	QLD	12	2024	Friday	13:00	Single	1	No	...	61	Inner Regional Australia	Toowoomba	Lockyer Valley	National or State Highway

5 rows × 25 columns

Fig 3.1 Top: Extracting data from the modified Fatalities 2024 dataset. Bottom: Clearing duplicates and checking for null values

```
df = df.drop_duplicates()
df.isna().sum()
```

✓ 0.0s

ID	0
Crash ID	0
State	0
Month	0
Year	0
Dayweek	0
Time	0
Crash Type	0
Number Fatalities	0
Bus Involvement	0
Heavy Rigid Truck Involvement	0
Articulated Truck Involvement	0
Speed Limit	0
Road User	0
Gender	0
Age	0
National Remoteness Areas	0
SA4 Name 2021	0
National LGA Name 2024	0
National Road Type	0
Christmas Period	0
Easter Period	0
Age Group	0
Day of week	0
Time of day	0
dtype: int64	

After ensuring that there are no issues in the dataset, such as null and duplicate values, the tables for each node and relationship can be created. Each node table undergoes the same process; attributes are extracted from the main dataset, then any duplicates in the tables are removed, and they are given a surrogate key, which acts as a foreign key. The *Crash* node

table, as discussed in the previous section, is not given any surrogate key, and the *Crash ID* will act as foreign key instead. The tables are then converted into CSV files.

```
# crash node
df_crash = df[['Crash ID', 'Crash Type', 'Number Fatalities']]
df_crash = df_crash.drop_duplicates()
df_crash.to_csv('nodes_relationships/node_crash.csv', index=False)
df_crash
```

✓ 0.0s Python

Fig 3.2 Top: creating the node table Crash. Bottom: creating the node table Person, with the surrogate key Person ID

```
# person node
df_person = df[['Gender', 'Age', 'Road User', 'Age Group']]
df_person = df_person.drop_duplicates().reset_index(drop=True)
df_person['Person ID'] = df_person.index + 1
df_person.to_csv('nodes_relationships/node_person.csv', index=False)
df_person
```

✓ 0.0s Python

	Crash ID	Crash Type	Number Fatalities		Gender	Age	Road User	Age Group	Person ID
0	20241115	Single	1	0	Male	74	Driver	65_to_74	1
1	20241125	Single	1	1	Female	19	Driver	17_to_25	2
2	20246013	Single	1	2	Female	33	Driver	26_to_39	3
3	20241002	Single	1	3	Female	32	Driver	26_to_39	4
4	20243185	Single	1	4	Female	61	Passenger	40_to_64	5
...	...	...	...	...	...	...	...	...	...
10485	20144079	Single	1	816	Female	64	Motorcycle rider	40_to_64	817
10486	20145055	Single	1	817	Female	43	Pedal cyclist	40_to_64	818
10487	20144007	Single	1	818	Male	86	Motorcycle rider	75_or_older	819
10488	20145072	Single	1	819	Female	49	Pedal cyclist	40_to_64	820
10489	20145108	Single	1	820	Male	33	Pedal cyclist	26_to_39	821
9683 rows × 3 columns				821 rows × 5 columns					

Fig 3.3 Left: the Crash node table. Right: the Person node table.

The relationship table undergoes a different process. Each relationship table contains the foreign keys that are associated with the entities. Therefore, after each node table is created, they are merged together with the main dataset, in a process that is similar to creating a fact table. This table is then filtered to only output the keys that are relevant to the relationship. Similarly, these tables are then converted into CSV files. There are special cases in some relationships, such as the relationships between *Crash* and *Person*, where there are two relationships that involve the same primary keys: *KILLED* and *DIED\_IN*. In this case, since both relationships use the same foreign keys, we do not need any special treatments in the extraction and transformation process.

```
# killed relationship
df_killed = df.merge(df_person, on=['Gender', 'Age', 'Road User', 'Age Group'])
df_killed = df_killed[['Crash ID', 'Person ID']]
df_killed.to_csv('nodes_relationships/rel_killed.csv', index=False)
df_killed
```

✓ 0.0s

	Crash ID	Person ID
0	20241115	1
1	20241125	2
2	20246013	3
3	20241002	4
4	20243185	5
...	...	...
10485	20144079	62
10486	20145055	187
10487	20144007	451
10488	20145072	278
10489	20145108	33

10490 rows × 2 columns

Fig 3.4 The creation of the relationship table for KILLED and DIED\_IN.

### 3.2 Loading to Neo4j

The CSV files will then be loaded to a Neo4j database. The node tables are loaded first before the relationships. A sample Cypher query for loading a node is shown below. In this case, this query is used to load the *Person* node.

```
// Person node
LOAD CSV WITH HEADERS FROM 'file:///node_person.csv' AS row
CREATE (d:Person {
    gender: row.`Gender`,
    age: toInteger(row.`Age`),
    roadUser: row.`Road User`,
    ageGroup: row.`Age Group`,
    personID: row.`Person ID`
})
```

This process is repeated until all the nodes are loaded into the database. Afterwards, the relationship tables are then loaded into the database. A sample Cypher query for loading the *INVOLVED* relationship (*Crash* → *Vehicle*) relationship is shown below.



```
// INVOLVED (Crash -> Vehicle)
LOAD CSV WITH HEADERS FROM 'file:///rel_involved.csv' AS row
MATCH (c:Crash {crashID: row.`Crash ID`}),
(n:Vehicle {vehicleID: row.`Vehicle ID`})
CREATE (c)-[:INVOLVED]->(n)
```

There are some special case relationships between entities, as stated previously. For example, there are two relationships between *Crash* and *Person*: *KILLED* (*Crash* → *Person*) and *DIED\_IN* (*Person* → *Crash*). While these relationships don't require any special treatment in the previous process, because both of them use the same foreign key references, some modification is required in the Cypher query so that the relationship is portrayed properly in the graph database. The modified Cypher query can be seen below:

```
// KILLED and DIED_IN (Crash <-> Person)
LOAD CSV WITH HEADERS FROM 'file:///rel_killed.csv' AS row
MATCH (c:Crash {crashID: row.`Crash ID`}),
(n:Person {personID: row.`Person ID`})
CREATE (c)-[:KILLED]->(n)
CREATE (n)-[:DIED_IN]->(c)
```

Once all the tables have been loaded, the Neo4j database is ready for querying.

## IV. Cypher Query Results

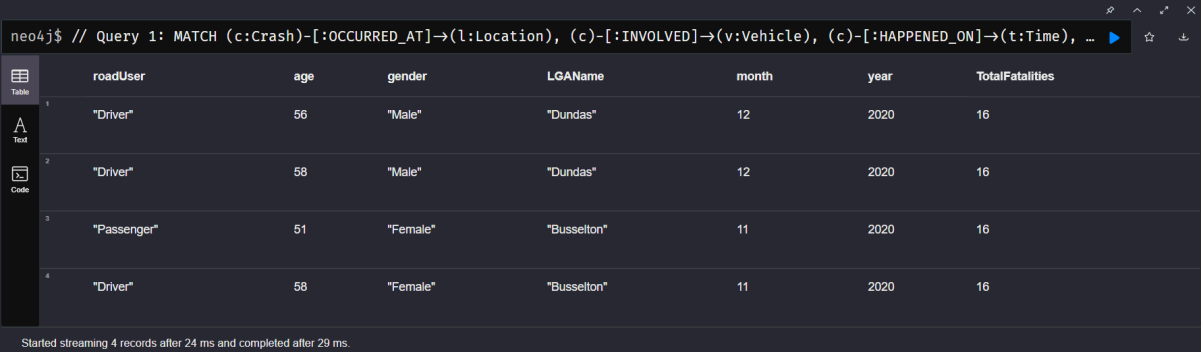
Based on the Neo4j database created above, these are the following Cypher queries that may provide some insights on the factors behind road fatalities in Australia.

### 4.1 Specified Cypher Queries

1. 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**.

*// Query 1:*

```
MATCH (c:Crash)-[:OCCURRED_AT]->(l:Location),
      (c)-[:INVOLVED]->(v:Vehicle),
      (c)-[:HAPPENED_ON]->(t:Time),
      (c)-[:KILLED]->(p:Person)
WHERE c.fatalities > 1
      AND l.state = 'WA'
      AND v.articulatedTruckInvolved = 'Yes'
      AND t.year >= 2020 AND t.year <= 2024
RETURN
      p.roadUser AS roadUser,
      p.age AS age,
      p.gender AS gender,
      l.LGName AS LGName,
      t.month AS month,
      t.year AS year,
      SUM(c.fatalities) AS TotalFatalities
```



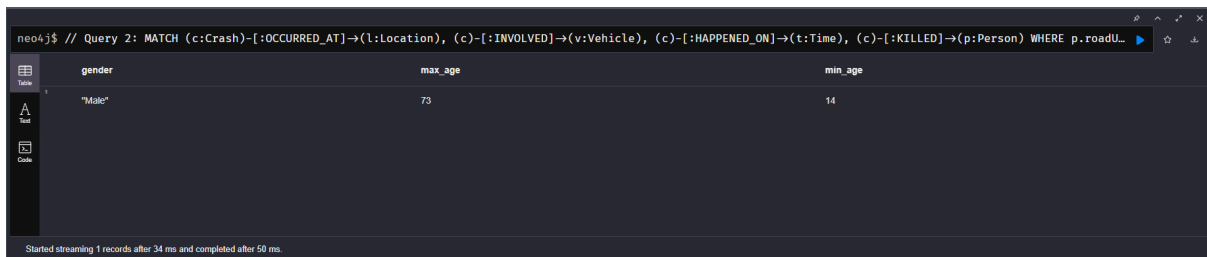
	roadUser	age	gender	LGName	month	year	TotalFatalities
1	"Driver"	56	"Male"	"Dundas"	12	2020	16
2	"Driver"	58	"Male"	"Dundas"	12	2020	16
3	"Passenger"	51	"Female"	"Busselton"	11	2020	16
4	"Driver"	58	"Female"	"Busselton"	11	2020	16

Started streaming 4 records after 24 ms and completed after 29 ms.

Fig 4.1 Output of query 1

2. 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**.

```
// Query 2:
MATCH (c:Crash)-[:OCCURRED_AT]->(l:Location),
      (c)-[:INVOLVED]->(v:Vehicle),
      (c)-[:HAPPENED_ON]->(t:Time),
      (c)-[:KILLED]->(p:Person)
WHERE p.roadUser = 'Motorcycle rider'
      AND (t.christmas = 'Yes' OR t.easter = 'Yes')
      AND l.remoteness = 'Inner Regional Australia'
WITH p.age AS age, p.gender AS gender
RETURN
  gender,
  MAX(age) AS max_age,
  MIN(age) AS min_age
```



gender	max_age	min_age
"Male"	73	14

Fig 4.2 Output of query 2

- 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**.

```
// Query 3:
MATCH (c:Crash)-[:OCCURRED_AT]->(l:Location),
      (c)-[:HAPPENED_ON]->(t:Time),
      (c)-[:KILLED]->(p:Person)
WHERE p.ageGroup = '17_to_25'
      AND t.year = 2024
WITH l.state AS state, t.dayType AS dayType, COUNT(*) AS Count,
      AVG(p.age) AS age
RETURN
  state,
  COALESCE(COLLECT(CASE WHEN dayType = 'Weekday' THEN Count
END)[0], 0) AS Weekday,
  COALESCE(COLLECT(CASE WHEN dayType = 'Weekend' THEN Count
END)[0], 0) AS Weekend,
  round(AVG(age), 2) AS avg_age
```

	state	Weekday	Weekend	avg_age
1	"NSW"	62	128	20.7
2	"QLD"	40	33	20.93
3	"SA"	23	3	21.25
4	"TAS"	6	6	22.17
5	"VIC"	42	20	20.52
6	"ACT"	1	0	19.0
7	"NT"	1	0	20.0

Fig 4.3 Output of query 3

- 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**.

**// Query 4:**

```

MATCH (c:Crash)-[:OCCURRED_AT]->(l:Location),
      (c)-[:HAPPENED_ON]->(t:Time),
      (c)-[:KILLED]->(p:Person),
      (c)-[:ON_ROAD]->(r:Road)
WHERE l.state = 'WA'
      AND (t.day = 'Friday' AND t.dayType = 'Weekend')
      AND c.crashType = 'Multiple'
      AND (p.gender = 'Male' OR p.gender = 'Female')
RETURN
  DISTINCT c.crashID AS crashID,
  l.SA4Name AS SA4,
  l.remoteness AS remoteness,
  r.roadType AS roadType

```

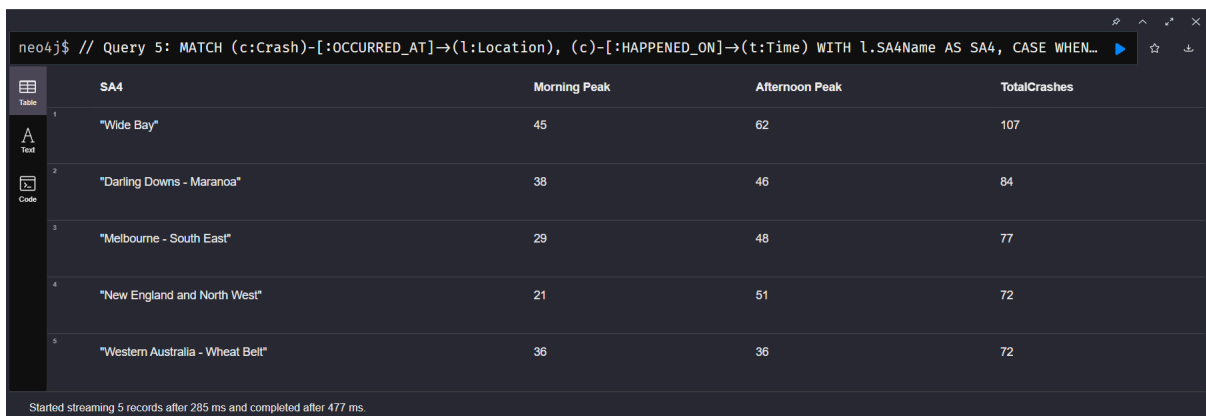
	crashID	SA4	remoteness	roadType
1	"20205094"	"Perth - South West"	"Major Cities of Australia"	"National or State Highway"
2	"20175132"	"Perth - South West"	"Major Cities of Australia"	"Local Road"
3	"20195098"	"Perth - South East"	"Major Cities of Australia"	"Local Road"
4	"20155048"	"Western Australia - Outback (North)"	"Very Remote Australia"	"National or State Highway"
5	"20165169"	"Western Australia - Wheat Belt"	"Very Remote Australia"	"Local Road"
6	"20185119"	"Western Australia - Outback (North)"	"Very Remote Australia"	"Arterial Road"
7	"20205063"	"Perth - North East"	"Major Cities of Australia"	"Local Road"

Fig 4.4 Output of query 4

- 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).

```
// Query 5:
MATCH (c:Crash)-[:OCCURRED_AT]->(l:Location),
      (c)-[:HAPPENED_ON]->(t:Time)
WITH l.SA4Name AS SA4,
      CASE
        WHEN t.time >= time("07:00") AND t.time <= time("09:00")
      THEN "Morning"
        WHEN t.time >= time("16:00") AND t.time <= time("18:00")
      THEN "Afternoon"
        ELSE null
      END AS Peak
WHERE Peak IS NOT NULL
WITH
      SA4,
      SUM(CASE WHEN Peak = "Morning" THEN 1 ELSE 0 END) AS
MorningPeak,
      SUM(CASE WHEN Peak = "Afternoon" THEN 1 ELSE 0 END) AS
AfternoonPeak
RETURN
      SA4,
      MorningPeak AS `Morning Peak`,
      AfternoonPeak AS `Afternoon Peak`,
      (MorningPeak + AfternoonPeak) AS TotalCrashes
ORDER BY TotalCrashes DESC
LIMIT 5
```



The screenshot shows a Neo4j query result interface. The query is: `neo4j$ // Query 5: MATCH (c:Crash)-[:OCCURRED_AT]->(l:Location), (c)-[:HAPPENED_ON]->(t:Time) WITH l.SA4Name AS SA4, CASE WHEN...`. The result is a table with 5 rows and 4 columns: SA4, Morning Peak, Afternoon Peak, and TotalCrashes. The rows are ordered by TotalCrashes in descending order.

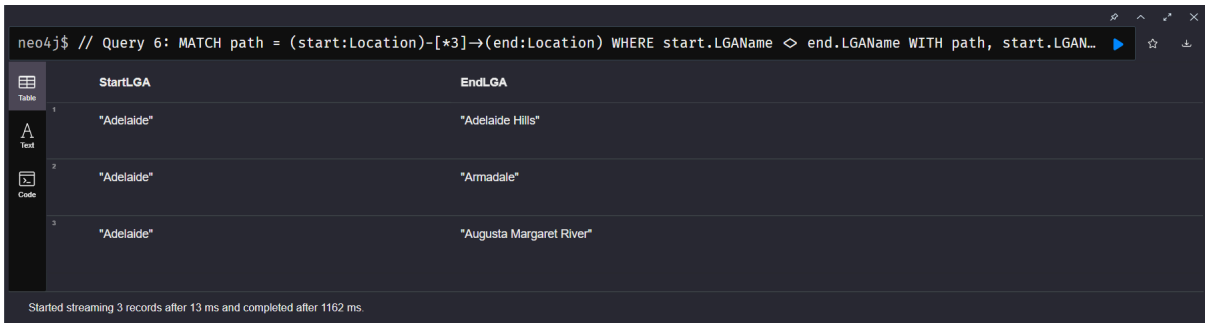
	SA4	Morning Peak	Afternoon Peak	TotalCrashes
1	"Wide Bay"	45	62	107
2	"Darling Downs - Maranoa"	38	46	84
3	"Melbourne - South East"	29	48	77
4	"New England and North West"	21	51	72
5	"Western Australia - Wheat Belt"	36	36	72

Started streaming 5 records after 285 ms and completed after 477 ms.

Fig 4.5 Output of query 5

- 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.

```
// Query 6:
MATCH path = (start:Location)-[*3]->(end:Location)
WHERE start.LGaname <> end.LGaname
WITH path, start.LGaname AS StartLGA, end.LGaname AS EndLGA
ORDER BY StartLGA, EndLGA
RETURN path
LIMIT 3
```



	StartLGA	EndLGA
1	"Adelaide"	"Adelaide Hills"
2	"Adelaide"	"Armadale"
3	"Adelaide"	"Augusta Margaret River"

Started streaming 3 records after 13 ms and completed after 1162 ms

Fig 4.6 Output of query 6

## 4.2 Other Cypher Queries

- Are there more fatalities during **holiday periods**? Compare **each state**.

```
// Query 7:
MATCH (c:Crash)-[:HAPPENED_ON]->(t:Time),
      (c)-[:OCCURRED_AT]->(l:Location)
WITH c.fatalities AS fatalities,
     l.state AS state,
     CASE
       WHEN t.christmas = "Yes" THEN "Christmas"
       WHEN t.easter = "Yes" THEN "Easter"
       ELSE "Non-Holiday"
     END AS period
RETURN state,
       period,
       SUM(fatalities) AS TotalFatalities
ORDER BY state, period
```

neo4j\$ // Query 7: MATCH (c:Crash)-[:HAPPENED\_ON]->(t:Time), (c)-[:OCCURRED\_AT]->(l:Location) WITH c.fatalities AS fatalities, l.state ...

	state	period	TotalFatalities
1	"ACT"	"Holiday"	4
2	"ACT"	"Non-Holiday"	116
3	"NSW"	"Holiday"	417
4	"NSW"	"Non-Holiday"	6040
5	"NT"	"Holiday"	41
6	"NT"	"Non-Holiday"	717
7	"QLD"	"Holiday"	172
8	"QLD"	"Non-Holiday"	3487
9	"SA"	"Holiday"	124

Fig 4.7 Outputs of query 7.

neo4j\$ // Query 7: MATCH (c:Crash)-[:HAPPENED\_ON]->(t:Time), (c)-[:OCCURRED\_AT]->(l:Location) WITH c.fatalities AS fatalities, l.state ...

	state	period	TotalFatalities
8	"QLD"	"Non-Holiday"	3487
9	"SA"	"Holiday"	124
10	"SA"	"Non-Holiday"	1612
11	"TAS"	"Holiday"	40
12	"TAS"	"Non-Holiday"	530
13	"VIC"	"Holiday"	154
14	"VIC"	"Non-Holiday"	3052
15	"WA"	"Holiday"	52
16	"WA"	"Non-Holiday"	1906

Started streaming 16 records after 38 ms and completed after 64 ms.

- Find the total number of crashes by each road type in 2020-2024, in speed zones greater than 40 km/h, that involves any large vehicle.

```
// Query 8
MATCH (c:Crash)-[:ON_ROAD]->(r:Road),
      (c)-[:INVOLVED]->(v:Vehicle),
      (c)-[:HAPPENED_ON]->(t:Time)
WHERE r.speedLimit > 40
      AND (t.year >= 2020 AND t.year <= 2024)
WITH r.roadType AS roadType,
      COUNT(*) AS TotalCrashes,
      CASE
        WHEN v.busInvolved = "Yes" THEN "Bus"
        WHEN v.heavyRigidTruckInvolved = "Yes" THEN "Heavy Rigid"
```

```

Truck"
    WHEN v.articulatedTruckInvolved = "Yes" THEN "Articulated
Truck"
    ELSE null
END AS vehicleInvolved
WHERE vehicleInvolved IS NOT NULL
RETURN roadType,
    SUM(TotalCrashes) AS TotalCrashes

```

neo4j\$ // Query 8: MATCH (c:Crash)-[:ON\_ROAD]→(r:Road), (c)-[:INVOLVED]→(v:Vehicle), (c)-[:HAPPENED\_ON]→(t:Time) WHERE r.speedLimit ...

	roadType	TotalCrashes
1	"Arterial Road"	159
2	"Local Road"	103
3	"National or State Highway"	2090
4	"Sub-arterial Road"	130
5	"Collector Road"	28
6	"Access road"	28
7	"Pedestrian Thoroughfare"	1
8	"Busway"	3

Fig 4.8 Output of query 8



## V. Application of Graph Data Science (GDS)

### 5.1 What is GDS?

According to the Neo4j website, **graph data science (GDS)** is an analytics and machine learning solution that analyses relationships in the data to discover more in-depth insights and improve predictions that cannot be discovered through traditional relational data models.<sup>[2]</sup> This can be achieved as GDS primarily focuses on the relationships and connections between data points, and not the data points themselves, to detect patterns, rank importance, and communities, among other aspects that can be found in a graph database.

### 5.2 Practicality of Database to GDS

GDS can be applied in the above graph database. The database is focused on road fatalities that occurred in Australia in the period 2020-2024, and contains many relationships that somewhat reflect real-world scenarios. The database is also relatively centralized around the node *Crash*, increasing connectivity and as such making the graph database more ideal for graph algorithms that are dependent on connectedness.

### 5.3 Practical Application of GDS

One practical application of GDS to the above graph database is graph similarity analysis. In this case, GDS will be applied to find crash nodes that are relatively similar to each other. The algorithm being used for GDS will be the **K-Nearest Neighbors (KNN) algorithm**. The KNN algorithm works by computing a distance value for all node pairs in the graph and creates new relationships between each node and its  $k$  nearest neighbors. The distance is calculated based on the node properties.<sup>[3]</sup>

To begin GDS on the database, a projected graph needs to be created, and it can be created by running the query below:

```
// Creating Projected Graph
CALL gds.graph.project(
  'crashGraph',
  ['Crash', 'Location', 'Vehicle', 'Time', 'Road', 'Person'],
  {
    OCCURRED_AT: {},
    INVOLVED: {},
    HAPPENED_ON: {},
    ON_ROAD: {},
    KILLED: {},
```

```

        KILLED_AT: {}
    }
)

```

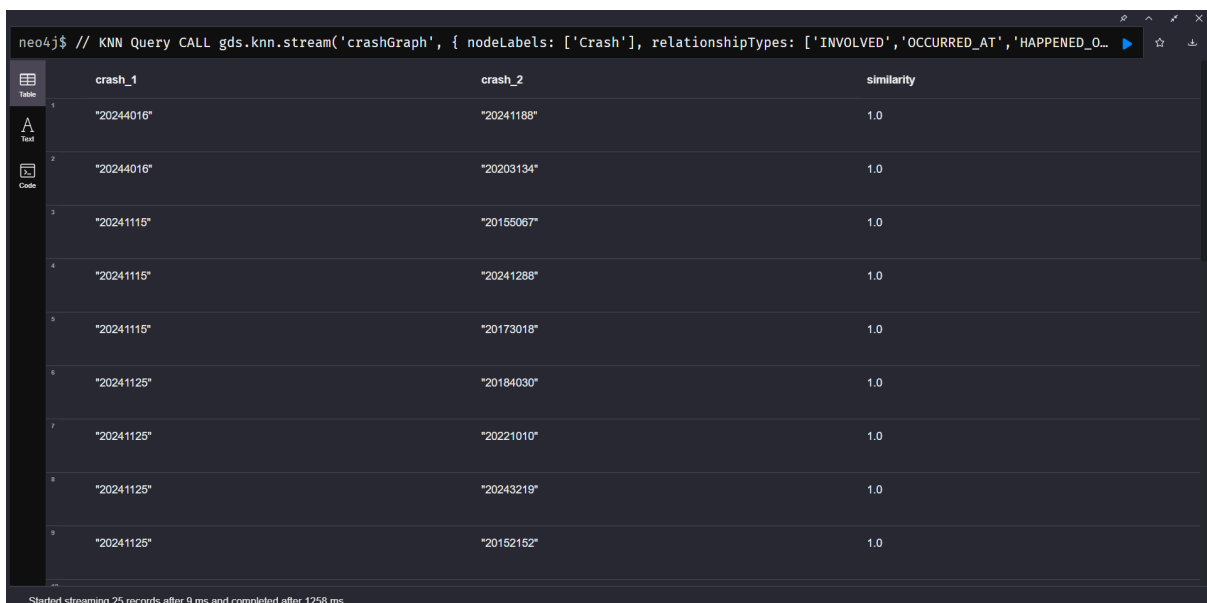
After running the above, we can immediately use KNN to find the most similar nodes.

```

// KNN Query
CALL gds.knn.stream('crashGraph', {
  nodeLabels: ['Crash'],
  relationshipTypes: ['INVOLVED', 'OCCURRED_AT', 'HAPPENED_ON'],
  topK: 5,
  nodeProperties: ['embedding'],
  similarityCutoff: 0.5
})
YIELD node1, node2, similarity
RETURN
  gds.util.asNode(node1).crashID AS crash_1,
  gds.util.asNode(node2).crashID AS crash_2,
  similarity
ORDER BY similarity DESC
LIMIT 25

```

The query above calls for *Crash* nodes that occur under similar circumstances. In this case, the algorithm is looking for crashes that involve **similar vehicles** (*INVOLVED*), **location** (*OCCURRED\_AT*), and **time of occurrence** (*HAPPENED\_ON*). By setting  $k = 5$ , the algorithm will be returning the 5 most similar crashes for each crash. A small output of the above query has been shown below:



The screenshot shows the Neo4j query interface with the following query executed: `CALL gds.knn.stream('crashGraph', { nodeLabels: ['Crash'], relationshipTypes: ['INVOLVED', 'OCCURRED_AT', 'HAPPENED_ON'] })`. The output is a table with three columns: `crash_1`, `crash_2`, and `similarity`. The table displays 9 rows of results, all with a similarity of 1.0. The first column contains crash IDs, and the second column contains the IDs of the most similar crashes. The status bar at the bottom indicates that 25 records were streamed after 9 ms and completed after 1258 ms.

	crash_1	crash_2	similarity
1	"20244016"	"20241188"	1.0
2	"20244016"	"20203134"	1.0
3	"20241115"	"20155067"	1.0
4	"20241115"	"20241288"	1.0
5	"20241115"	"20173018"	1.0
6	"20241125"	"20184030"	1.0
7	"20241125"	"20221010"	1.0
8	"20241125"	"20243219"	1.0
9	"20241125"	"20152152"	1.0

Fig 5.1 Query output for KNN analysis

From the output, the crash ID **“20241115”** is similar to the crashes with the following crash IDs: **“20155067”**, **“20241288”**, and **“20173018”**. All similarity scores are **1.0**, which means a perfect similarity. This can be possible due to three reasons: firstly, the dataset contains over 10,000 data, and a higher dataset means it is more likely for nodes to share attributes with other nodes, increasing their similarity to each other. Secondly, only three relationships are being used above, instead of all the relationships. The narrowed scope of the relationships being captured increases the likelihood of two nodes to be very similar to each other. Lastly, Neo4j also automatically rounds numbers with a large decimal place (e.g. 0.999999989...) to 1.0, which effectively treats near-identical nodes as perfectly similar nodes.

In practicality, this is very useful, as determining crashes that have similar patterns means that the Australian government can come up with a prevention strategy that can effectively tackle multiple different factors that relate to road accidents.

## References

- [1] Australian Government. (2024, November 11). *Monthly road deaths*. National Road Safety Data Hub.  
<https://datahub.roadsafety.gov.au/progress-reporting/monthly-road-deaths>
- [2] *Graph Data Science*. (n.d.). Neo4j Graph Data Platform.  
<https://neo4j.com/product/graph-data-science/>
- [3] *K-Nearest Neighbors - Neo4j Graph Data Science*. (n.d.). Neo4j Graph Data Platform.  
<https://neo4j.com/docs/graph-data-science/current/algorithms/knn/>