DATA PROCESSING AND ANALYTICS

(Project report on Data Processing and Analytics)

Project on:

“Data Processing and Analytics Topics”

Done By:

Bidhan Pant

Completed On

12th May 2023

Table of Contents

Table of Figures

Table of Tables

Part A

The first part of the assignment focuses on your understanding and implementation of various database technologies. You are given three use case scenarios as below:

Use-case 1

A database system for a bus network of the town needs to be designed. There are several bus companies, with their names, addresses, phone numbers and emails, which operate different routes. You need to record the length and the name of the route. A route can be operated by a single company only. Each route can have multiple bus stops, for which you need to record their address, type and coordinates. A single bus stop can serve multiple routes. The database also needs to record the passengers who have registered for the unified ticketing app used to buy the bus tickets. Their names, email addresses, phone numbers, as well as the discount status are recorded. Passengers' trips are recorded as well. This includes the date, time, tariff and start/end bus stops. Finally, you need to record the buses, including their models, mileage, and years. Buses can serve multiple routes, and the start and end dates of the service for each route are recorded too.

TASK (Part A):

Choose one of the above use case scenarios and:

1. Develop an Entity-Relationship model of the information requirements for the selected scenario.

2. Translate your model into an equivalent relational schema. Specify all relation headings, indicating primary and foreign keys.

3. Implement at least three important entities from your relational schema in SQL, MongoDB and Neo4J (could be different three entities for each technology) and generate sample data for all of the implementations. Show samples of generated data.

4. Explain why you chose the particular entities to implement with each technology. 5. Come up with

5 test cases for each database technology and implement those using queries. Show the queries code and the output for each database technology.

Part A:

1. Entity Relationship Model

1.1. Entities and Attributes

- BusCompany: It contains all the details of bus companies and have five attributes company_id (Primary Key), name, address, phone_number and email.

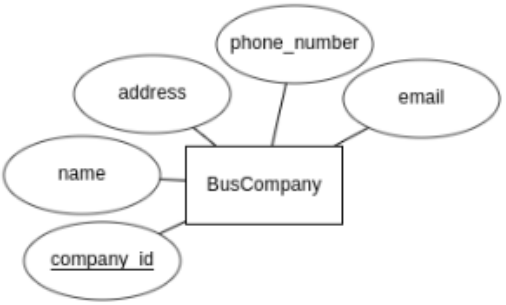


Figure 1: Bus company attribute

- Bus: It contains all the details of buses and have four attributes bus_id (Primary Key), model, mileage, and year

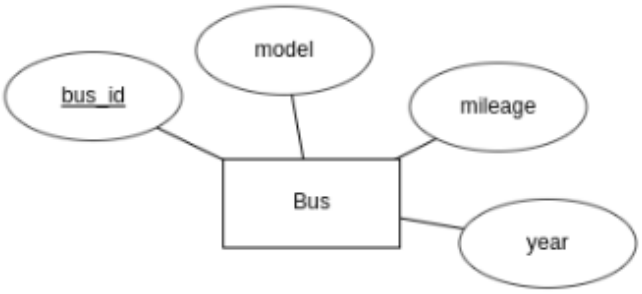


Figure 2: Bus attribute

- Route: It contains all the details of routes and have three attributes route_id (Primary Key), name, and length of the route.

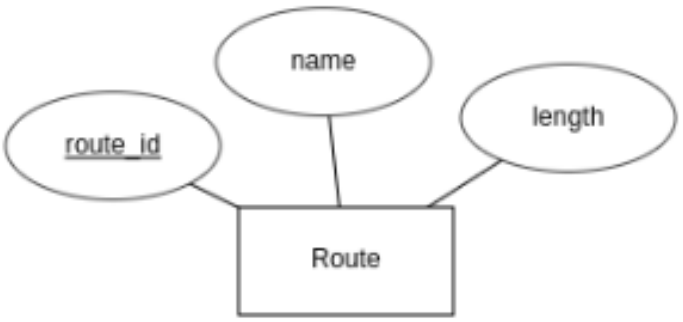


Figure 3: Route attribute

- BusStop: It contains all the details of bus stops and have four attributes StartStop_id (Primary Key), address, type and composite coordinates. Furthermore, coordinates are divided into two more sides i.e. latitude and longitude

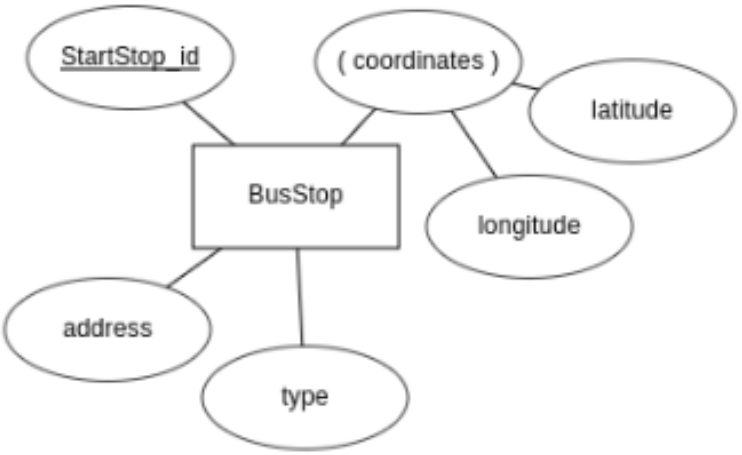


Figure 4: BusStop attribute

- Trip: It contains all the details about trip, and it is a weak entity because it is derived from the combination of two entities and it has five main attributes trip_id (Primary Key), name, tariff, date, and time.

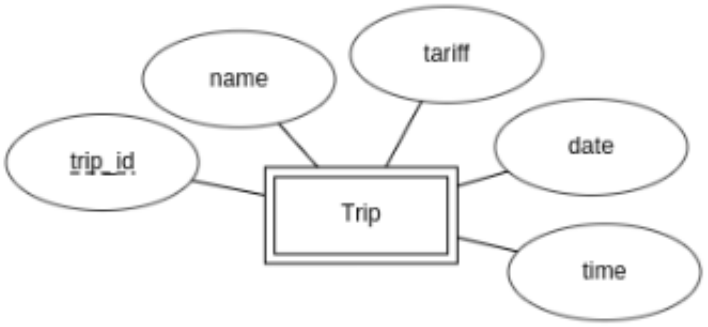


Figure 5: Trip attribute

- Passenger: It contains all the details of passengers and have four attributes passenger_id (Primary Key), name (composite), email, and phone_number (multi-valued).

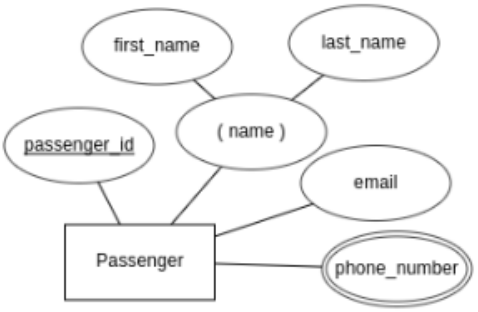


Figure 6: Passenger attribute

- Address: It contains all the details of passengers address and have four attributes address_id (Primary Key), building, street, and zipcode.

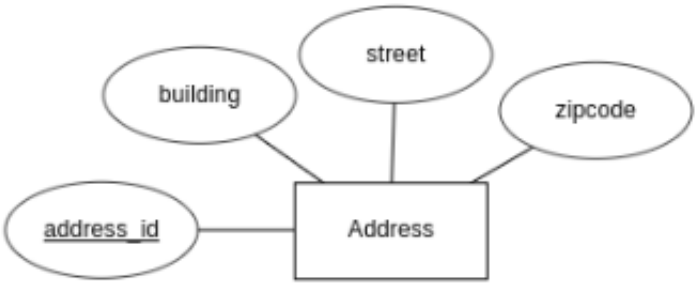


Figure 7: Address attribute

- Discount: It contains all the details of passenger's discount status and have three attributes discount_id (Primary Key), discount_status, and discount_type.

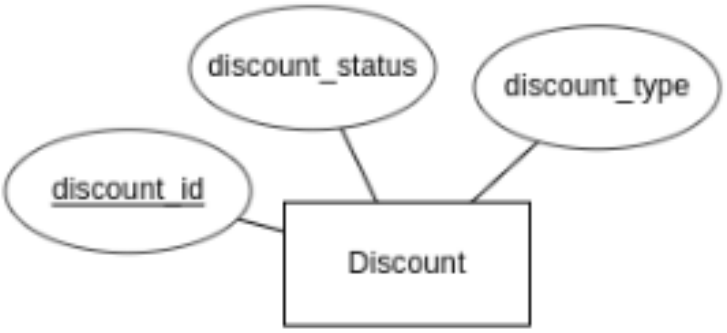


Figure 8: Discount attribute

1.2. Special Attributes

There are three special attributes in our database. One multivalued and two composite attributes:

Multivalued attribute is an attribute that have multiple value for a single entity. In our database we have created passenger's phone number as a multivalued because passenger may have multiple phone numbers.

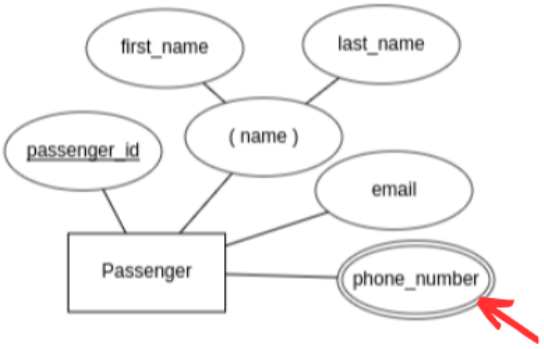


Figure 9: Multivalued attribute

A *composite attribute* is an attribute that is further sub-divided into smaller parts. In our database we have two composite attributes i.e., name of a passenger from passenger table, which is further sub-divided into first name and last name and coordinates from the bus stop table which is further sub-divided into latitude and longitude.

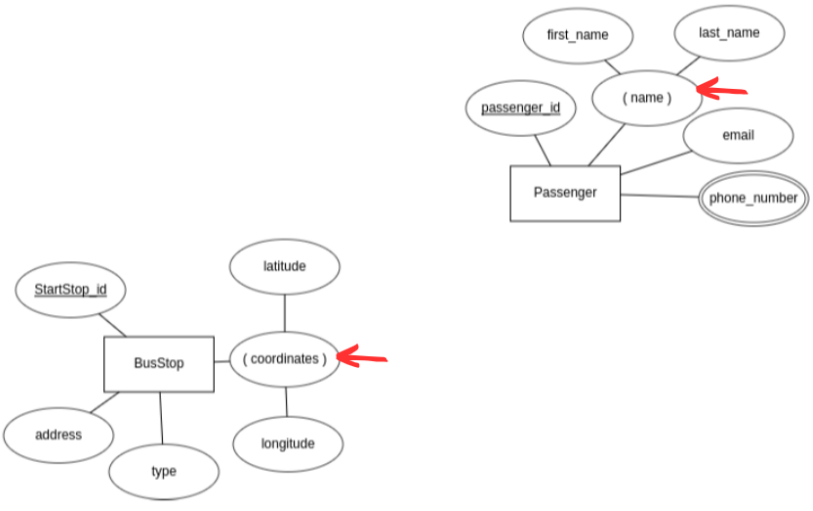


Figure 10: Composite attribute

1.3. Relations and Cardinalities

Following are the relations in our database:

1.3.1. One-to-one relationships

We have two one-to-one relationships in our database i.e., passenger entity with address entity and passenger entity with discount entity.

In a relationship between passenger and address entities, each passenger in a database is associated with only one address entity, and each address entity can be associated with only one passenger entity. The relationship is mandatory on address side as passenger must have address.

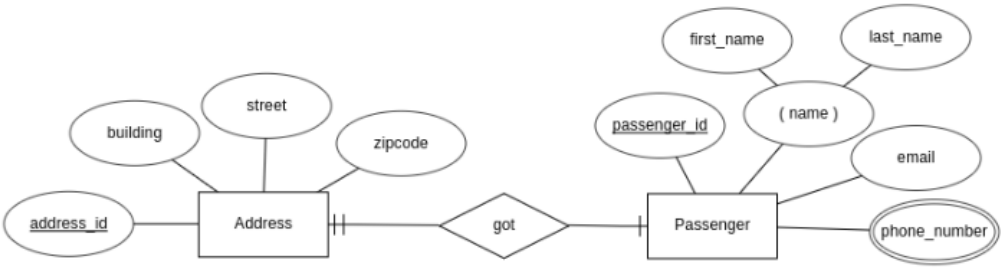


Figure 11: One-to-one relationship between Address and Passenger attributes

In above case, the address entity has three main attributes i.e., building, street and zip code. These attributes represent the different parts of the passenger's address.

By having a one-to-one relationship between the Passenger and Address entities, we can ensure that each passenger has a unique address, and vice versa. This relationship can be useful in situations where we need to quickly retrieve a passenger's address or update a passenger's address without affecting other entities in the system.

In a relationship between passenger and discount entities, each passenger in a database is associated with only one discount entity, and each discount entity can be associated with only one passenger entity. The relationship is optional because not all passengers may have a discount associated with their account.

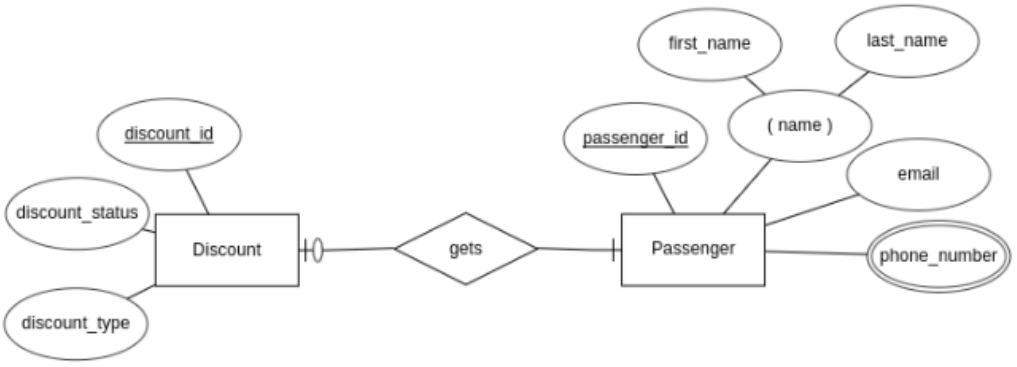


Figure 12: One-to-one relationship between Discount and Passenger attributes

In the above case, discount status and discount type are two attributes of the Discount entity. These attributes would represent the various types of discounts available to a passenger, such as student or child discounts.

We can assure that each passenger has at least one discount connected with their account by creating a one-to-one optional link between the Passenger and Discount entities. This can be handy when we need to rapidly retrieve a passenger's discount status or when we need to adjust a passenger's discount without affecting other entities in the system. The relationship's optional aspect allows for data flexibility, as not all passengers may have a discount connected with their account.

1.3.2. One-to-many relationship

We have two one-to-many relationship in our database i.e., bus company entity with address entity and passenger entity with discount entity.

In a relationship between bus company and bus entities, each bus company in a database are associated with many buses entity, and each bus entity is associated with only one bus company entity. The cardinality is optional because some Bus Company entities may operate only one Bus entity.

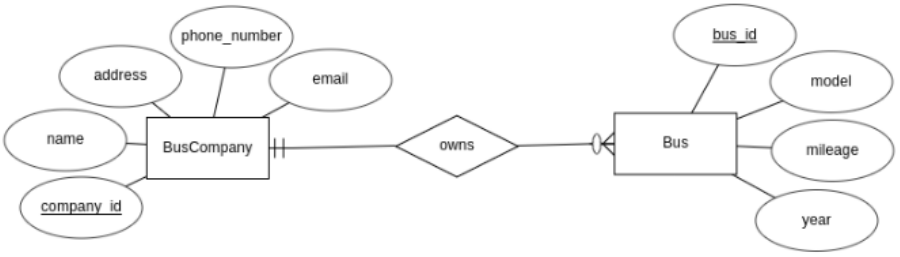


Figure 13: One-to-many relationship between Bus company and Bus

In a relationship between route and trip entities, each route in a database is associated with many trips entity, and each trip entity is associated with only one route entity. The cardinality is optional on the trip side of the relationship means that not all Routes will necessarily have multiple trips, it means that some routes are associated with only one trip. However, every Trip must be associated with a Route, so the cardinality on the Route side is mandatory.

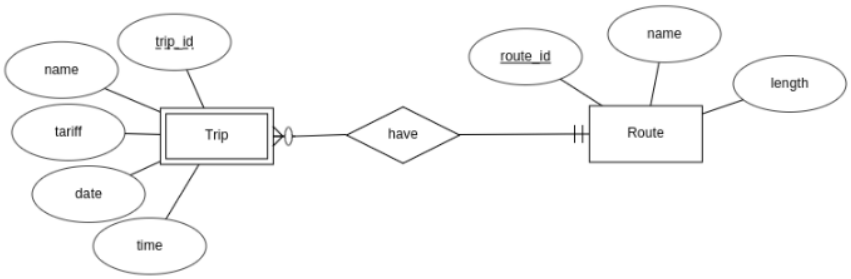


Figure 14: : One-to-many relationship between Trip and Route

1.3.3. Many-to-many relationship

We have three many-to-many relationship in our database i.e., bus entity with route entity, route entity with bus stop entity and bus stop entity with trip entity.

The relationship between bus and route entities is many-to-many. A route can serve more than one bus, and vice versa. We create a junction table to represent this relation in this scenario we have created ‘serves’ as a junction table which contains foreign keys from both bus and route tables. The cardinality is optional on both sides because certain routes can service one or zero buses and some buses can serve one or zero routes.

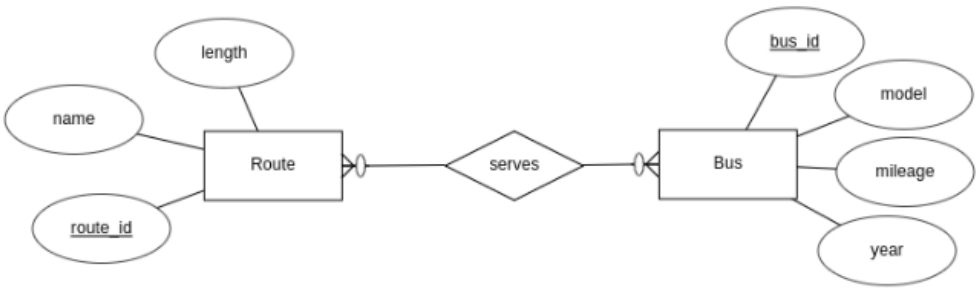


Figure 15: Many-to-many relationship between Route and Bus

The relationship between route and bus stop entities is many-to-many. This is because a route can have more than one bus stops and bus stops can served by multiple routes. We create a junction table to represent this relation in this scenario we have created ‘can_have’ as a junction table which contains foreign keys from both route and bus stop tables. The cardinality is, we can have mandatory on the route side (i.e., a route must have at least one stop), but optional on the stop side (i.e., a stop can be served by zero or many routes). This represents the reality that some stops may be served by just one route, but others may be served by several.

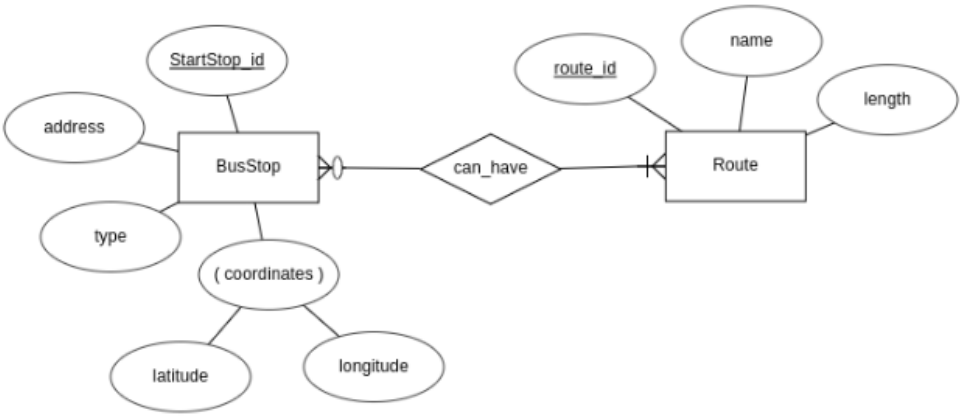


Figure 16: Many-to-many relationship between BusStop and Route

The relationship between bus stop and trip entities is many-to-many. This is because a bus stop can have more than one trip and trip can have multiple bus stops. We create a junction table to represent this relation in this scenario we have created ‘must_have’ as a junction table which contains foreign keys from both bus stop and trip tables. The cardinality is, every trip must have at least one bus stop, and we can have mandatory participation from the trip entity side. However, because a bus stop is not always included in a trip, we can have optional participation from the bus stop entity side. As a result, the relationship's cardinality would be one trip to many stops and one stop to many journeys.

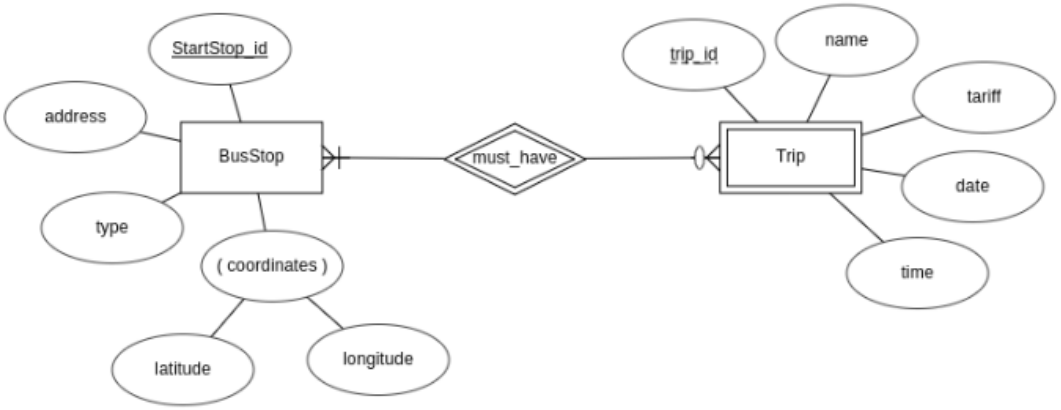


Figure 17: Many-to-many relationship between BusStop and Trip

1.4. Assumptions

When designing a database system, assumptions must be considered. They contribute to the system's scope and requirements, as well as ensuring that all participants understand its purpose and functionality. There are several assumptions that must be made in the context of a bus network database system in order to accurately represent the data and relationships between different entities. For this scenario, the following assumptions were made:

1. There is no need to record the timing of the buses; only the start and end dates of service for each route are required: The system doesn't need to keep record of the daily or weekly timing of the buses. Instead, it only needs to keep track of the start and end dates for each bus's service on a specific route.
2. Only one discount status for each passenger: A passenger can have only one discount status at a time, such as child, student, disability, etc. one can't have student and disability discount status at once.
3. The tariff for the trip is fixed for regular passenger: The cost of the tariff depends on if the passenger falls under the discount criteria. For those who falls under discount category they will get discount according to discounted rate but to the regular passengers the cost of the tariff is fixed.

1.5.

2. Conversion into Relational Schema

2.1. Entity Relation Diagram (ERD)

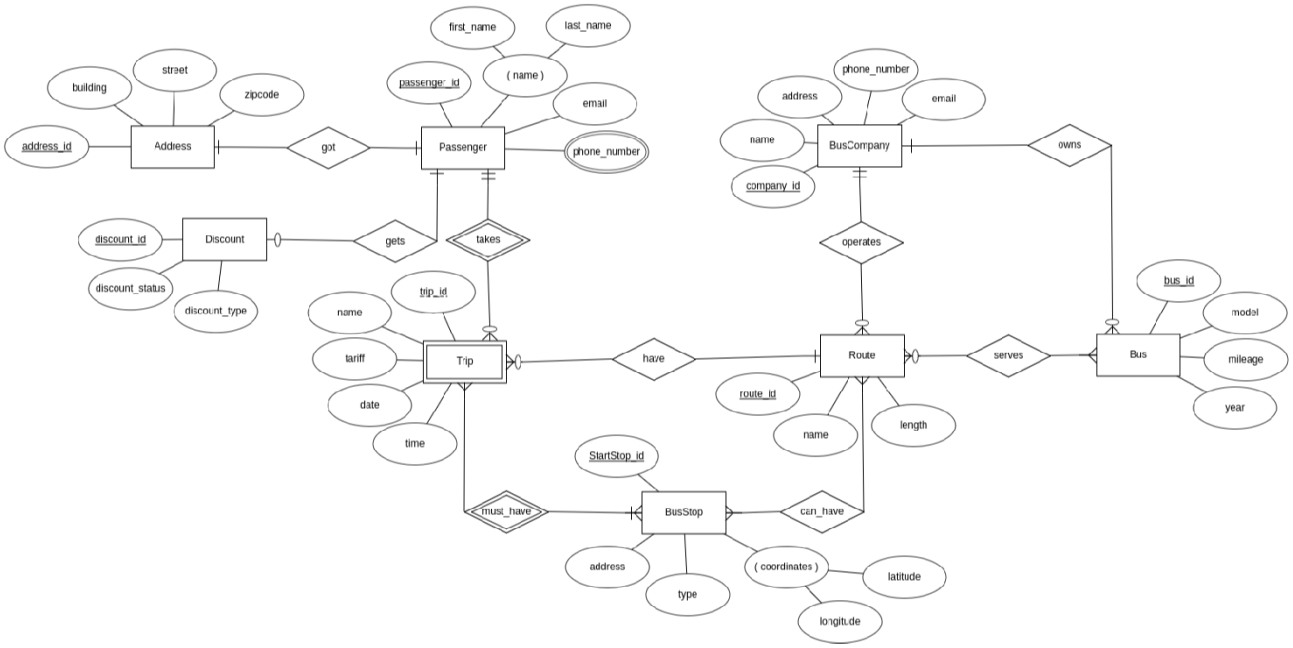


Figure 18: Entity Relation Diagram (ERD)

2.2. Mapping ER to Relational Schema

2.2.1. Regular Entities

- BusCompany
- Bus
- Route
- BusStop
- Passenger
- Address
- Discount

Outcomes:

- BusCompany (**company_id**, name, address, phone_number, email)
- Bus (**bus_id**, model, mileage, year)
- Route (**route_id**, name, length)
- BusStop (**StartStop_id**, address, type, coordinates)
- Passenger (**passenger_id**, name, email)
- Address(**address_id**, building, street, zipcode)
- Discount(**discount_id**, discount_status, discount_type)

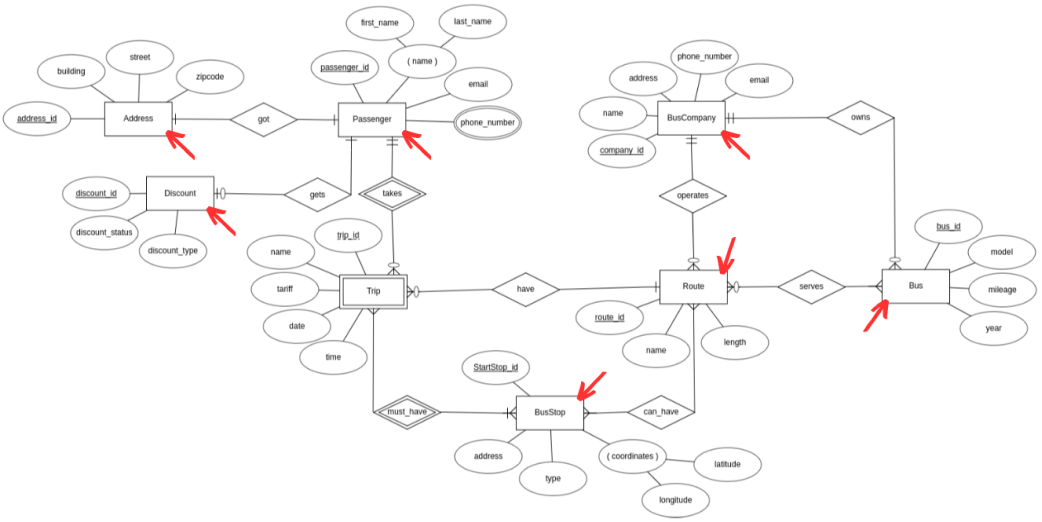


Figure 19: Regular entities

2.2.2. Weak Entities

- Trip

Outcome:

Trip (**trip_id, passenger_id***, name, tariff, time, date, route_id*)

- trip_id, passenger_id: primary key
- passenger_id: foreign key referencing Passenger
- route_id: foreign key referencing Route

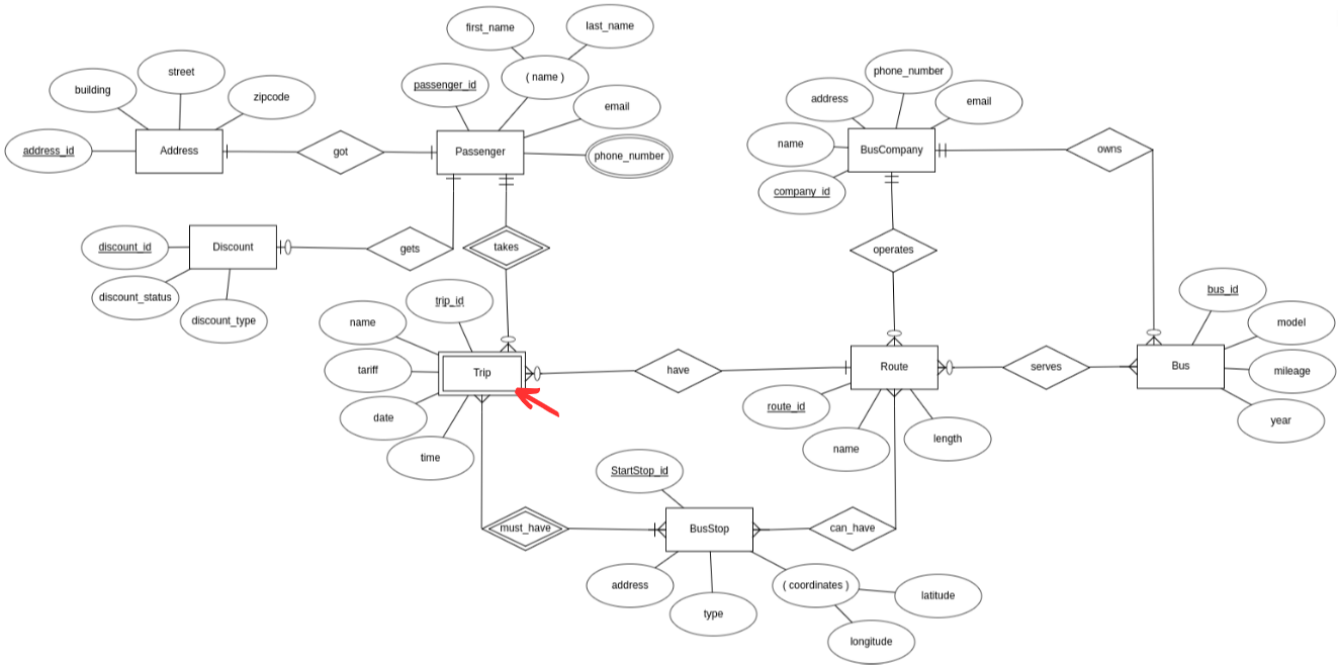


Figure 20: Weak entities

2.2.3. One-to-one relationships

- got
- gets

Outcome:

Passenger(**<u>passenger_id</u>**, first_name, last_name, email, **address_id***, **discount_id***)

- passenger_id: primary key
- address_id: foreign key referencing Address
- discount_id: foreign key referencing Discount

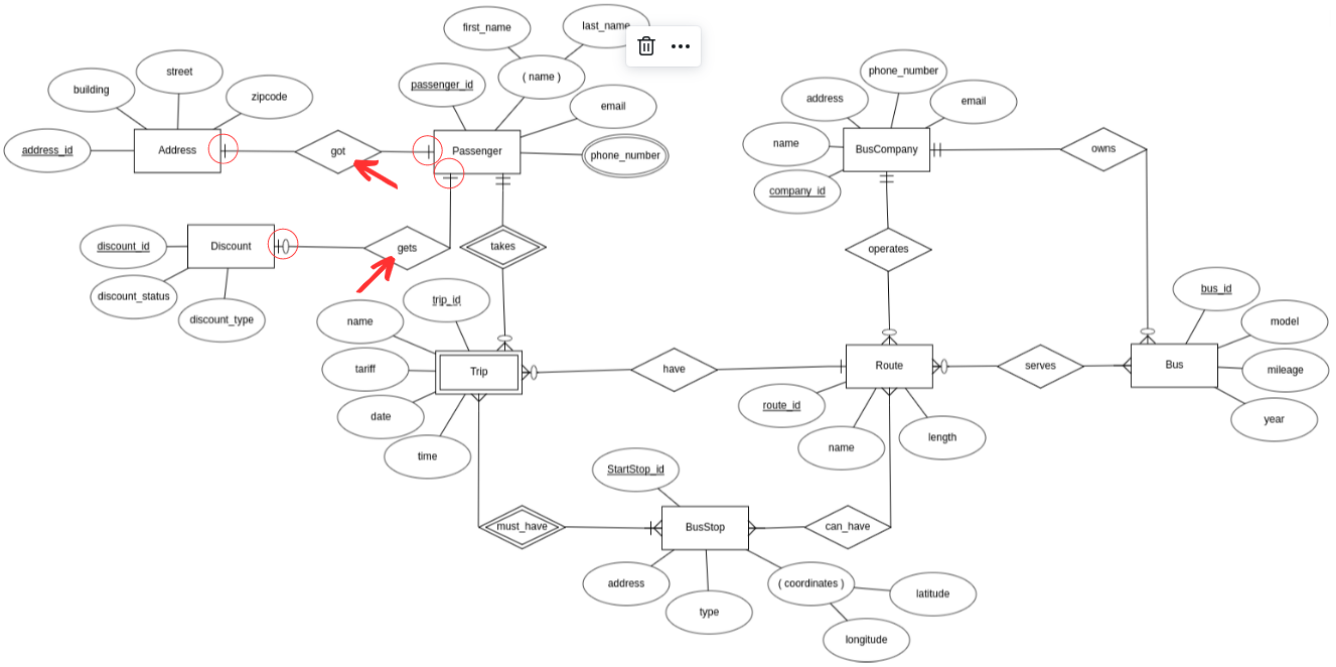


Figure 21: One-to-one relationship

2.2.4. One-to-many relationships

- owns
- operates

Outcomes:

Bus (**bus_id**, model, mileage, year, **company_id***)

- bus_id: primary key
- company_id: foreign key referencing BusCompany

Route (**route_id**, name, length, **company_id***)

- route_id: primary key
- company_id: foreign key referencing BusCompany

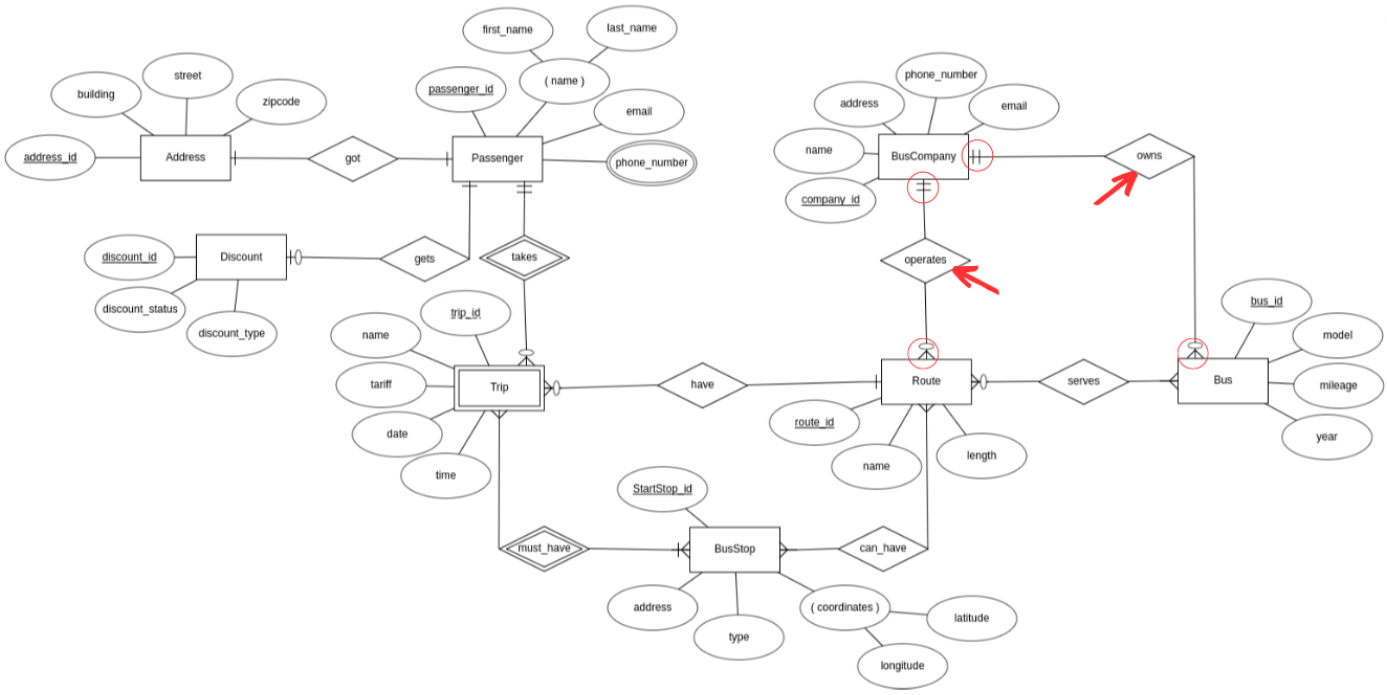


Figure 22: One-to-many relationship

2.2.5. Many-to-many relationships

- serves
- can_have
- must have

Outcomes:

serves (**route_id*, bus_id***)

can_have (**route_id*, StartStop_id***)

must_have(**(trip_id*, passenger_id*), StartStop_id***)

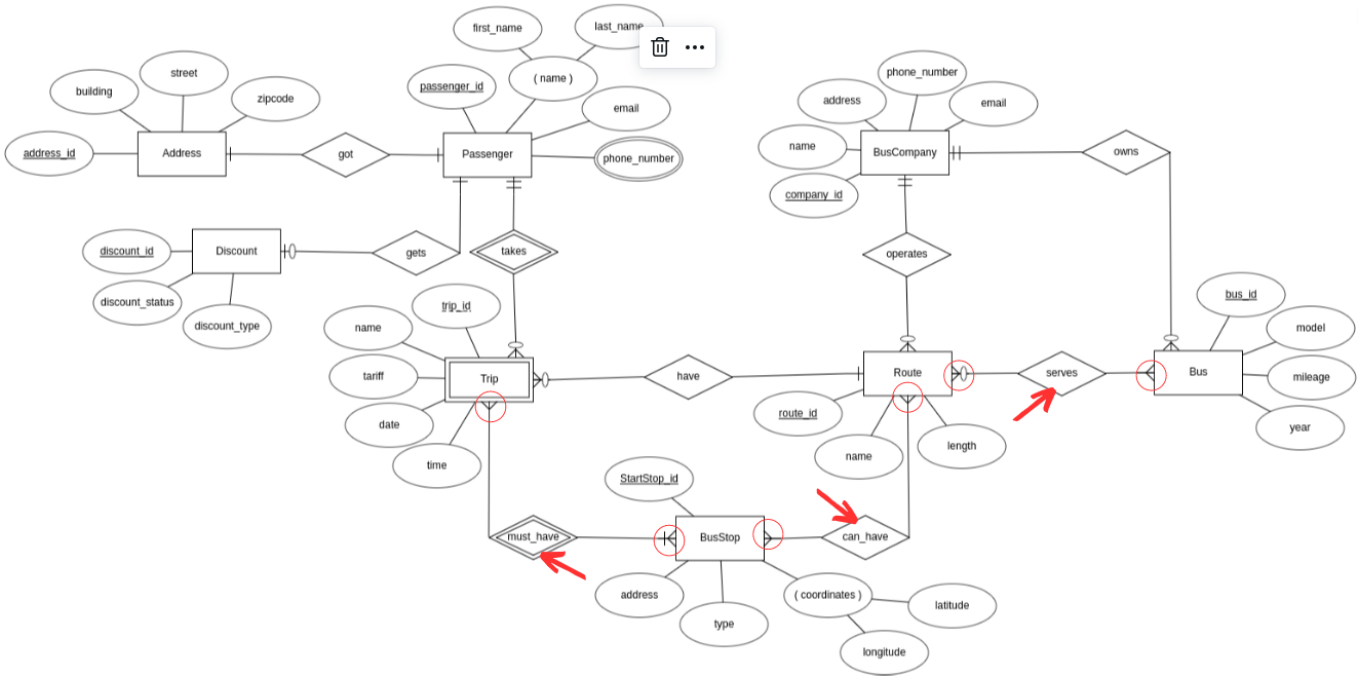


Figure 23: Many-to-many relationship

2.2.6. Multivalued attributes

- Passenger_phone_number

Outcomes:

Passenger_phone_number(**phone_number, passenger_id***)

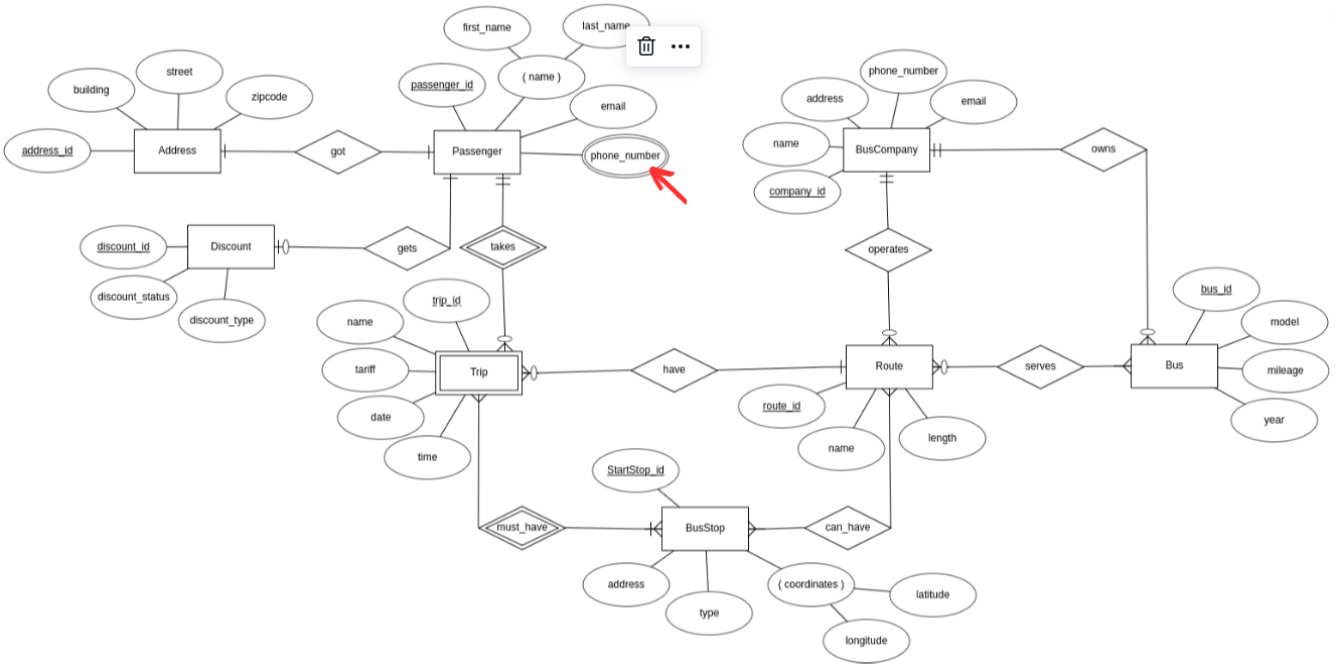


Figure 24: Multivalued attribute

2.2.7. Final outcome

- BusCompany(**company_id**, name, address, phone_number, email)
- Bus(**bus_id**, model, mileage, year, **company_id***)
- serves(**route_id*, bus_id***)
- Route(**route_id**, name, length, **company_id***)
- can_have(**route_id*, StartStop_id***)
- BusStop(**StartStop_id**, address, type, latitude, longitude)
- Discount(**discount_id**, discount_type, discount_status)
- Address(**address_id**, building, street, zipcode)
- Passenger(**passenger_id**, first_name, last_name, email, **address_id***, **discount_id***)
- Passenger_phone_number(**phone_number**, **passenger_id***)
- Trip(**trip_id, passenger_id***, name, tariff, time, date, **route_id***)
- must_have(**(trip_id, passenger_id)***, **StartStop_id***)

3. Conversion into Databases and sample data

According to the task assigned we have taken three entities for SQL (BusCompany, Bus and Route) and four entities for Neo4J (BusCompany, Bus, Route and Trip) and MangoDB (Address, Discount, Passenger and Trip).

3.1. SQL

3.1.1. Creating Bus Company table:

```
-- Creating BusCompany table
CREATE TABLE BusCompany
(
company_id VARCHAR(10) NOT NULL,
name VARCHAR(50) NOT NULL,
address VARCHAR(100) NOT NULL,
phone_number INT NOT NULL,
email VARCHAR(50) NOT NULL,
CONSTRAINT pk_BusCompany PRIMARY KEY (company_id)
);
```

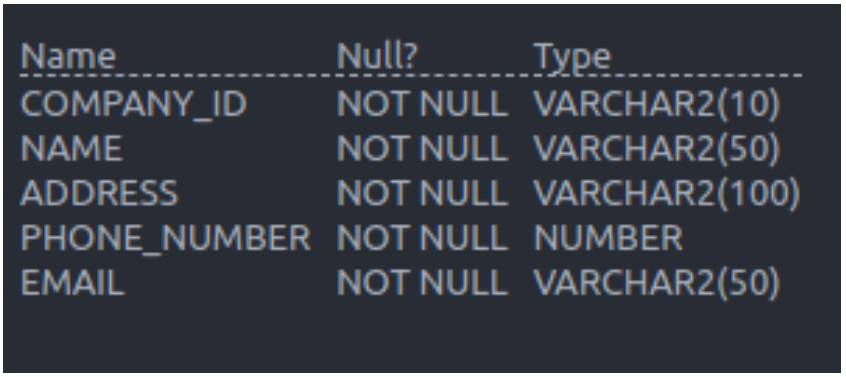
Figure 25: Creating bus company table

Above SQL command creates a table name “BusCompany” with five columns: company_id, name, address, phone_number and email.

- The “company_id” is a string data type with a maximum length of 10 characters and is marked as “NOT NULL” which means that it must contain some value for every row in the table. This column is also marked as a primary key of the table using the “CONSTRAINT pk_BusCompany PRIMARY KEY”, which assure that each row in the table has unique “company_id” value.
- The “name” column is marked as “NOT NULL” and is a string data type with maximum length of 50 characters.
- The “address” column is marked as “NOT NULL” and is a string data type with maximum length of 100 characters.
- The “phone_number” column is marked as “NOT NULL” and is an integer data type.

- The “email” column is marked as “NOT NULL” and is a string data type with maximum length of 50 characters.

Output:



Name	Null?	Type
COMPANY_ID	NOT NULL	VARCHAR2(10)
NAME	NOT NULL	VARCHAR2(50)
ADDRESS	NOT NULL	VARCHAR2(100)
PHONE_NUMBER	NOT NULL	NUMBER
EMAIL	NOT NULL	VARCHAR2(50)

Figure 26: Bus company table structure

Now we have created a bus company table, let’s add some sample data to the table.

```
-- BusCompany table
INSERT INTO BusCompany (company_id, name, address, phone_number, email)
VALUES ('bc1', 'Bournemouth Transport Ltd', '21 Depot Road, Bournemouth', 01202451210, 'info@bournemouthtransport.co.uk');
INSERT INTO BusCompany (company_id, name, address, phone_number, email)
VALUES ('bc2', 'Yellow Buses', 'Yeomans Way, Bournemouth', 01202636000, 'info@yellowbuses.co.uk');
INSERT INTO BusCompany (company_id, name, address, phone_number, email)
VALUES ('bc3', 'Morebus', 'Wilts & Dorset House, 25 Poole Road, Bournemouth', 03450728090, 'info@morebus.co.uk');
INSERT INTO BusCompany (company_id, name, address, phone_number, email)
VALUES ('bc4', 'Wilts & Dorset', 'Yeomans Way, Bournemouth', 01202673555, 'customerservices@wilts.co.uk');
INSERT INTO BusCompany (company_id, name, address, phone_number, email)
VALUES ('bc5', 'Go South Coast', 'Enterprise House, Walworth Industrial Estate, Andover', 01264710555, 'info@gosouthcoast.co.uk');
```

Figure 27: Bus company table dummy data

Outcome:

COMPANY_ID	NAME	ADDRESS	PHONE_NUMBER	EMAIL
bc1	Bournemouth Transport Ltd	21 Depot Road, Bournemouth	1202451210	info@bournemouthtransport.co.uk
bc2	Yellow Buses	Yeomans Way, Bournemouth	1202636000	info@yellowbuses.co.uk
bc3	Morebus	Wilts & Dorset House, 25 Poole Road, Bournemouth	3450728090	info@morebus.co.uk
bc4	Wilts & Dorset	Yeomans Way, Bournemouth	1202673555	customerservices@wilts.co.uk
bc5	Go South Coast	Enterprise House, Walworth Industrial Estate, Andover	1264710555	info@gosouthcoast.co.uk

Figure 28: Bus company table details

3.1.2. Creating Bus table:

```
-- Creating Bus table
CREATE TABLE Bus
(
bus_id VARCHAR(10) NOT NULL,
model VARCHAR(50) NOT NULL,
mileage INT NOT NULL,
year INT NOT NULL,
company_id VARCHAR(10) NOT NULL,
CONSTRAINT pk_bus PRIMARY KEY (bus_id),
CONSTRAINT fk_company_id FOREIGN KEY (company_id) REFERENCES BusCompany(company_id)
);
```

Figure 29: Creating Bus table

Above SQL command creates a table name “Bus” with five columns: bus_id, model, mileage, year, and company_id.

- The “bus_id” is a string data type with a maximum length of 10 characters and is marked as “NOT NULL” which means that it must contain some value for every row in the table. This column is also marked as a primary key of the table using the “CONSTRAINT pk_bus PRIMARY KEY”, which assure that each row in the table has unique “company_id” value.
- The “model” column is marked as “NOT NULL” and is a string data type with maximum length of 50 characters. The “mileage” and year column is marked as “NOT NULL” and is an integer data type.
- The FOREIGN KEY constraint "fk_company_id" is applied to the "company_id" column. This means that the "company_id" column must have values from the "BusCompany" table's "company_id" column. This constraint enforces the relationship between the tables "Bus" and "BusCompany," ensuring that every bus in the "Bus" table is owned by a corporation in the "BusCompany" table.

Output:

Name	Null?	Type
BUS_ID	NOT NULL	VARCHAR2(10)
MODEL	NOT NULL	VARCHAR2(50)
MILEAGE	NOT NULL	NUMBER
YEAR	NOT NULL	NUMBER
COMPANY_ID	NOT NULL	VARCHAR2(10)

Figure 30: Bus table structure

Now we have created a bus table, let's add some sample data to the table.

```
-- Bus table
INSERT INTO Bus (bus_id, model, mileage, year, company_id)
VALUES ('b1', 'Mercedes-Benz Citaro', 25, 2019, 'bc1');
INSERT INTO Bus (bus_id, model, mileage, year, company_id)
VALUES ('b2', 'Volvo B7L', 30, 2010, 'bc2');
INSERT INTO Bus (bus_id, model, mileage, year, company_id)
VALUES('b3', 'Scania N230UD', 20, 2013, 'bc3');
INSERT INTO Bus (bus_id, model, mileage, year, company_id)
VALUES('b4', 'Volvo B7RLE', 15, 2011, 'bc4');
INSERT INTO Bus (bus_id, model, mileage, year, company_id)
VALUES('b5', 'Alexander Dennis Enviro200', 35, 2017, 'bc5');

INSERT INTO Bus (bus_id, model, mileage, year, company_id)
VALUES('b6', '25Tr Irisbus', 17, 1990, 'bc1');
INSERT INTO Bus (bus_id, model, mileage, year, company_id)
VALUES('b7', '26Tr Solaris', 22, 1995, 'bc2');
INSERT INTO Bus (bus_id, model, mileage, year, company_id)
VALUES('b8', '33Tr SOR', 31, 2011, 'bc3');
INSERT INTO Bus (bus_id, model, mileage, year, company_id)
VALUES('b9', '489 Polaris', 24, 2017, 'bc4');
INSERT INTO Bus (bus_id, model, mileage, year, company_id)
VALUES('b10', 'Tata Motors', 29, 2011, 'bc5');

INSERT INTO Bus (bus_id, model, mileage, year, company_id)
VALUES('b11', 'Belkommunmash', 16, 2017, 'bc1');
INSERT INTO Bus (bus_id, model, mileage, year, company_id)
VALUES('b12', 'Aero Bus', 23, 2011, 'bc2');
INSERT INTO Bus (bus_id, model, mileage, year, company_id)
VALUES('b13', 'AGG300', 14, 2017, 'bc3');
INSERT INTO Bus (bus_id, model, mileage, year, company_id)
VALUES('b14', 'Ailsa B55', 34, 2011, 'bc4');
INSERT INTO Bus (bus_id, model, mileage, year, company_id)
VALUES('b15', 'AN440', 40, 2017, 'bc5');
```

Figure 31: Bus table dummy data

Outcome:

BUS_ID	MODEL	MILEAGE	YEAR	COMPANY_ID
b6	25Tr Irisbus	17	1990	bc1
b7	26Tr Solaris	22	1995	bc2
b8	33Tr SOR	31	2011	bc3
b9	489 Polaris	24	2017	bc4
b10	Tata Motors	29	2011	bc5
b11	Belkommunmash	16	2017	bc1
b12	Aero Bus	23	2011	bc2
b13	AGG300	14	2017	bc3
b14	Ailsa B55	34	2011	bc4
b15	AN440	40	2017	bc5
b1	Mercedes-Benz Citaro	25	2019	bc1
b2	Volvo B7L	30	2010	bc2
b3	Scania N230UD	20	2013	bc3
b4	Volvo B7RLE	15	2011	bc4
b5	Alexander Dennis Enviro200	35	2017	bc5

Figure 32: Bus table details

3.1.3. Creating Route table

```
-- Creating Route table
CREATE TABLE Route
(
route_id VARCHAR(10) NOT NULL,
name VARCHAR(50) NOT NULL,
length INT NOT NULL,
company_id VARCHAR(10) NOT NULL,
CONSTRAINT pk_route_id PRIMARY KEY (route_id),
CONSTRAINT fk_company_id1 FOREIGN KEY (company_id) REFERENCES BusCompany(company_id)
);
```

Figure 33: Creating Route table

Above SQL command creates a table name “Route” with five columns: route_id, name, length, and company_id.

- The “route_id” is a string data type with a maximum length of 10 characters and is marked as “NOT NULL” which means that it must contain some value for every row in the table. This column is also marked as a primary key of the table using the “CONSTRAINT pk_route_id PRIMARY KEY”, which assure that each row in the table has unique “route _id” value.
- The “name” column is marked as “NOT NULL” and is a string data type with maximum length of 50 characters. The “length” column is marked as “NOT NULL” and is an integer data type.
- The FOREIGN KEY constraint "fk_company_id1" is applied to the "company_id" column. This means that the "company_id" column must have values from the "BusCompany" table's "company_id" column. This constraint enforces the relationship between the tables "Route" and "BusCompany," ensuring that every route in the "Route" table is served by a corporation in the "BusCompany" table.

Outcome:

Name	Null?	Type
ROUTE_ID	NOT NULL	VARCHAR2(10)
NAME	NOT NULL	VARCHAR2(50)
LENGTH	NOT NULL	NUMBER
COMPANY_ID	NOT NULL	VARCHAR2(10)

Figure 34: Route table structure

Now we have created a route table, let's add some sample data to the table.

```
-- Route table
INSERT INTO Route (route_id, name, length, company_id)
VALUES ('r1', '1A', 5, 'bc1');
INSERT INTO Route (route_id, name, length, company_id)
VALUES ('r2', '5', 7, 'bc2');
INSERT INTO Route (route_id, name, length, company_id)
VALUES ('r3', 'M1', 10, 'bc3');
INSERT INTO Route (route_id, name, length, company_id)
VALUES ('r4', '17', 14, 'bc4');
INSERT INTO Route (route_id, name, length, company_id)
VALUES ('r5', 'X6', 52, 'bc5');

INSERT INTO Route (route_id, name, length, company_id)
VALUES ('r6', '5a', 66, 'bc1');
INSERT INTO Route (route_id, name, length, company_id)
VALUES ('r7', 'M2', 32, 'bc2');
INSERT INTO Route (route_id, name, length, company_id)
VALUES ('r8', '1b', 96, 'bc3');
INSERT INTO Route (route_id, name, length, company_id)
VALUES ('r9', '3x', 48, 'bc4');
INSERT INTO Route (route_id, name, length, company_id)
VALUES ('r10', '6a', 42, 'bc5');

INSERT INTO Route (route_id, name, length, company_id)
VALUES ('r11', '7A', 95, 'bc1');
INSERT INTO Route (route_id, name, length, company_id)
VALUES ('r12', '7B', 67, 'bc2');
INSERT INTO Route (route_id, name, length, company_id)
VALUES ('r13', 'Breezer 40', 78, 'bc3');
INSERT INTO Route (route_id, name, length, company_id)
VALUES ('r44', 'Breezer 60', 80, 'bc4');
INSERT INTO Route (route_id, name, length, company_id)
VALUES ('r15', 'Cango C32, Cango C33', 100, 'bc5');
```

Figure 35: Route table dummy data

Outcome:

ROUTE_ID	NAME	LENGTH	COMPANY_ID
r7	M2	32	bc2
r8	1b	96	bc3
r9	3x	48	bc4
r10	6a	42	bc5
r11	7A	95	bc1
r12	7B	67	bc2
r13	Breezer 40	78	bc3
r44	Breezer 60	80	bc4
r15	Cango C32, Cango C33	100	bc5
r6	5a	66	bc1
r1	1A	5	bc1
r2	5	7	bc2
r3	M1	10	bc3
r4	17	14	bc4
r5	X6	52	bc5

Figure 36: Route table details

3.1.4. Creating Serves Junction table:

```
-- Creating serves table
CREATE TABLE serves
(
route_id VARCHAR(10) NOT NULL,
bus_id VARCHAR(10) NOT NULL,
CONSTRAINT pk_route_bus_id PRIMARY KEY (route_id, bus_id),
CONSTRAINT fk_routeid FOREIGN KEY (route_id) REFERENCES Route(route_id),
CONSTRAINT fk_busid FOREIGN KEY (bus_id) REFERENCES Bus(bus_id)
);
```

Figure 37: Creating Serves table

Serve table is used to represent the many-to-many relationship between the “Route” table and "Bus” table. The table has two columns:

- The “route_id” and “bus_id” is a string data type with a maximum length of 10 characters and is marked as “NOT NULL” which means that it must contain some value for every row in the table. This column is also marked as a primary key of the table using the “CONSTRAINT pk_route_bus_id PRIMARY KEY”, which assure that each row in the table has unique “route _id” and “bus_id” value.
- The FOREIGN KEY constraint "fk_routeid" and “fk_busid” is applied to the "route_id" and “bus_id” column. This means that the "route_id" and “bus_id” columns must have values from the "Route" table's "route_id" and “Bus” table’s “bus_id” columns respectively. These constraints enforce the link between the "serves" table and the "Route" and "Bus" tables, guaranteeing that every "route_id" and "bus_id" combination in the "serves" table represents a legitimate route-bus pairing.

Outcome:

Name	Null?	Type
ROUTE_ID	NOT NULL	VARCHAR2(10)
BUS_ID	NOT NULL	VARCHAR2(10)

Figure 38: Serves table structure

Now we have created a serves junction table, let's add some sample data to the table.

```sql
-- Serves table
INSERT INTO serves (route_id, bus_id)
VALUES ('r1', 'b1');
INSERT INTO serves (route_id, bus_id)
VALUES ('r2', 'b2');
INSERT INTO serves (route_id, bus_id)
VALUES ('r3', 'b3');
INSERT INTO serves (route_id, bus_id)
VALUES ('r4', 'b4');
INSERT INTO serves (route_id, bus_id)
VALUES ('r5', 'b5');

INSERT INTO serves (route_id, bus_id)
VALUES ('r6', 'b6');
INSERT INTO serves (route_id, bus_id)
VALUES ('r7', 'b7');
INSERT INTO serves (route_id, bus_id)
VALUES ('r8', 'b8');
INSERT INTO serves (route_id, bus_id)
VALUES ('r9', 'b9');
INSERT INTO serves (route_id, bus_id)
VALUES ('r10', 'b10');

INSERT INTO serves (route_id, bus_id)
VALUES ('r11', 'b11');
INSERT INTO serves (route_id, bus_id)
VALUES ('r12', 'b12');
INSERT INTO serves (route_id, bus_id)
VALUES ('r13', 'b13');
INSERT INTO serves (route_id, bus_id)
VALUES ('r44', 'b14');
INSERT INTO serves (route_id, bus_id)
VALUES ('r15', 'b15');
```

```sql
INSERT INTO serves (route_id, bus_id)
VALUES ('r15', 'b1');
INSERT INTO serves (route_id, bus_id)
VALUES ('r44', 'b2');
INSERT INTO serves (route_id, bus_id)
VALUES ('r13', 'b3');
INSERT INTO serves (route_id, bus_id)
VALUES ('r12', 'b4');
INSERT INTO serves (route_id, bus_id)
VALUES ('r11', 'b5');

INSERT INTO serves (route_id, bus_id)
VALUES ('r10', 'b6');
INSERT INTO serves (route_id, bus_id)
VALUES ('r9', 'b7');
INSERT INTO serves (route_id, bus_id)
VALUES ('r8', 'b15');
INSERT INTO serves (route_id, bus_id)
VALUES ('r7', 'b9');
INSERT INTO serves (route_id, bus_id)
VALUES ('r6', 'b10');

INSERT INTO serves (route_id, bus_id)
VALUES ('r1', 'b11');
INSERT INTO serves (route_id, bus_id)
VALUES ('r2', 'b12');
INSERT INTO serves (route_id, bus_id)
VALUES ('r3', 'b13');
INSERT INTO serves (route_id, bus_id)
VALUES ('r4', 'b14');
INSERT INTO serves (route_id, bus_id)
VALUES ('r5', 'b15');
```

Figure 39: Serves table dummy data

Outcome:

ROUTE_ID	BUS_ID
r1	b1
r1	b11
r10	b10
r10	b6
r11	b11
r11	b5
r12	b12
r12	b4
r13	b13
r13	b3
r15	b1
r15	b15
r2	b12
r2	b2
r3	b13
r3	b3
r4	b14
r4	b4
r44	b14
r44	b2
r5	b15
r5	b5
r6	b10
r6	b6
r7	b7
r7	b9
r8	b15
r8	b8
r9	b7
r9	b9

Figure 40: Serves table details

3.2. Neo4J

3.2.1. Bus Company

```
//creating Bus_Company node

CREATE (bcompany1: Bus_company {company_id:"bc1", name:"Bournemouth Transport Ltd", address:"21 Depot Road, Bournemouth", number:"01202451210", email:"info@bournemouthtransport.co.uk"})

CREATE (bcompany2: Bus_company {company_id:"bc2", name:"Yellow Buses", address:"Yeomans Way, Bournemouth", number:"01202636000", email:"info@yellowbuses.co.uk"})

CREATE (bcompany3: Bus_company {company_id:"bc3", name:"Morebus", address:"Wilts & Dorset House, 25 Poole Road, Bournemouth", number:"03450728090", email:"info@morebus.co.uk"})

CREATE (bcompany4: Bus_company {company_id:"bc4", name:"Go South Coast", address:"Enterprise House, Walworth Industrial Estate, Andover", number:"012647105550", email:"info@gosouthcoast.co.uk"})

CREATE (bcompany5: Bus_company {company_id:"bc5", name:"National Express", address:"Bournemouth Square", number:"01202453000", email:"info@nationalexpress.co.uk"})
```

Figure 41: Creating bus company node

The code above creates 5 nodes for the "Bus_Company" with the following properties:

Company_id: a unique identifier for a company (string), name: a company's name (string), address: a bus company's physical address (string), number: a company's contact number (string), and email: a company's email address (string).

Outcome:

MATCH (n:Bus_company) RETURN n LIMIT 25;

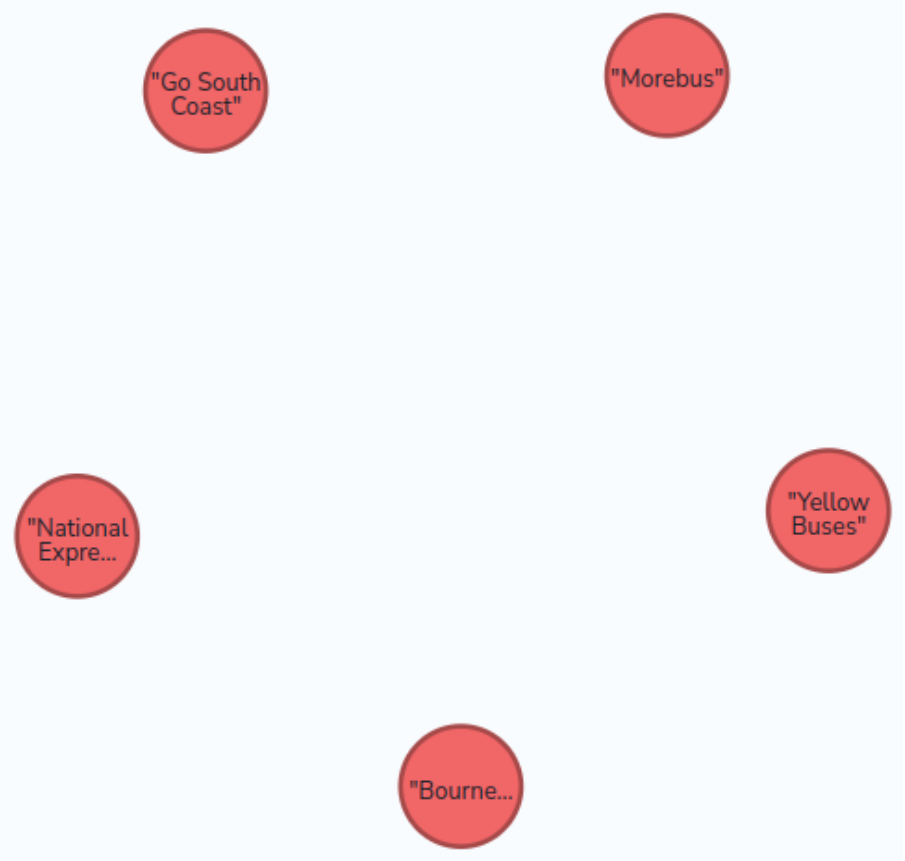


Figure 42: Five bus company nodes

3.2.2. Bus

```
//creating Bus node

CREATE (bus1: Bus {bus_id:"b1", model:"Mercedes-Benz Citaro", mileage:"25", yea
r:"2019"})

CREATE (bus2: Bus {bus_id:"b2", model:"Volvo B7L", mileage:"35", year:"2011"})

CREATE (bus3: Bus {bus_id:"b3", model:"Scania N230UD", mileage:"15", year:"200
0"})

CREATE (bus4: Bus {bus_id:"b4", model:"Volvo B7RLE", mileage:"25.5", year:"199
0"})

CREATE (bus5: Bus {bus_id:"b5", model:"Alexander Dennis Enviro200", mileage:"1
8", year:"1992"})

CREATE (bus6: Bus {bus_id:"b6", model:"25Tr Irisbus", mileage:"17", year:"199
3"})

CREATE (bus7: Bus {bus_id:"b7", model:"26Tr Solaris", mileage:"24", year:"199
5"})

CREATE (bus8: Bus {bus_id:"b8", model:"33Tr SOR", mileage:"27", year:"1998"})

CREATE (bus9: Bus {bus_id:"b9", model:"489 Polaris", mileage:"28", year:"2015"})

CREATE (bus10: Bus {bus_id:"b10", model:"Tata Motors", mileage:"30.5", year:"202
0"})

CREATE (bus11: Bus {bus_id:"b11", model:"Belkommunmash", mileage:"31", year:"202
1"})

CREATE (bus12: Bus {bus_id:"b12", model:"Aero Bus", mileage:"14", year:"1988"})

CREATE (bus13: Bus {bus_id:"b13", model:"AGG300", mileage:"22", year:"2001"})

CREATE (bus14: Bus {bus_id:"b14", model:"Ailsa B55", mileage:"23", year:"2019"})

CREATE (bus15: Bus {bus_id:"b15", model:"AN440", mileage:"36", year:"2022"})
```

Figure 43: Creating bus node

The code above creates 15 nodes for the "Bus" with the following properties:

Bus_id: a unique identifier for a bus (string), model: a description of a bus's model (string), mileage: the number of miles a bus can travel on one liter of diesel/petrol, and year: the year a bus was built (string).

Outcome:

MATCH (n:Bus) RETURN n LIMIT 25;

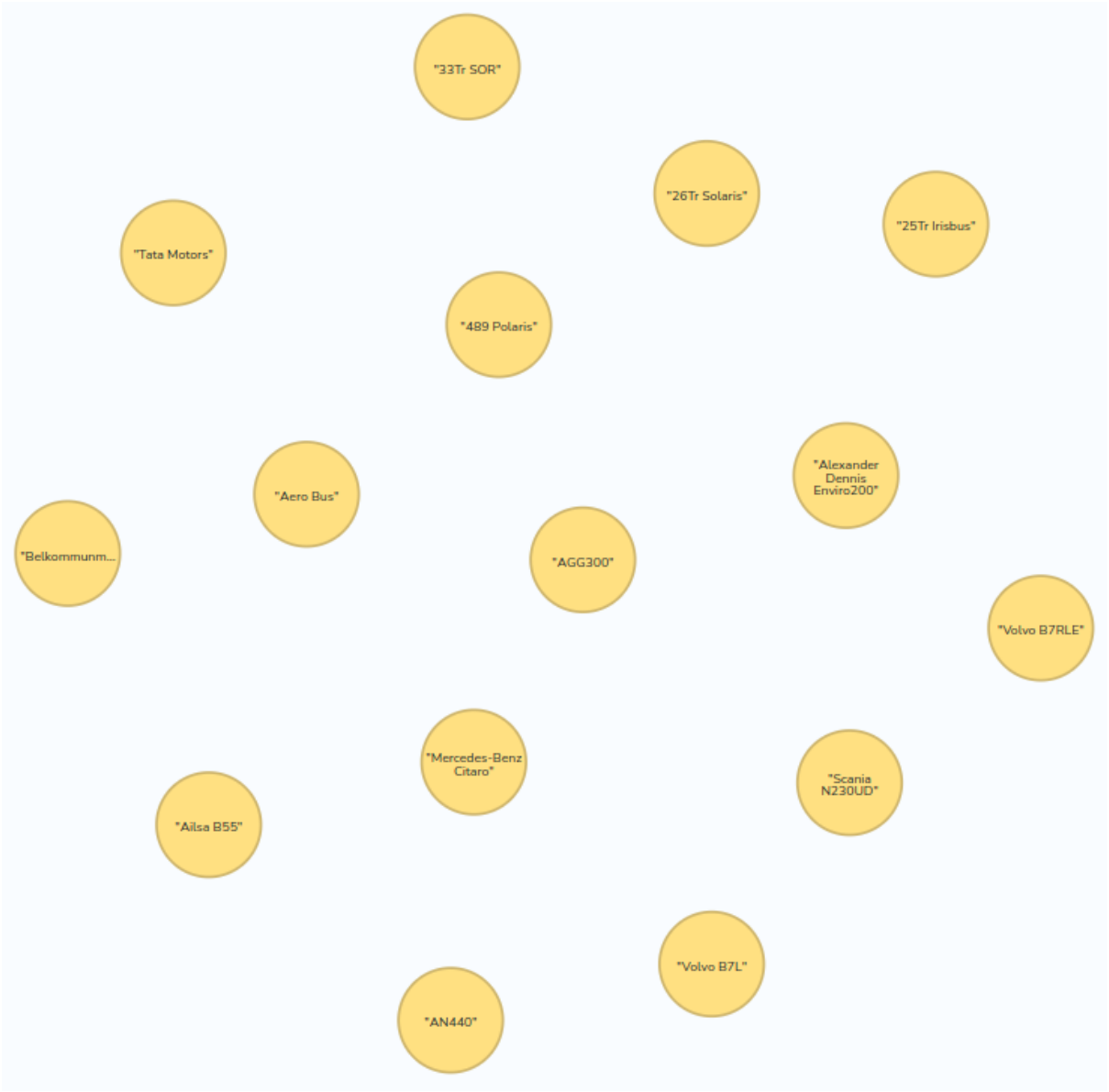


Figure 44: Fifteen bus nodes

3.2.3. Relationship between Bus Company and Bus

```
//creating relationship of Bus Company 1 with Bus

MATCH (bcompany1:Bus_company {company_id: "bc1"})
MATCH (bus1:Bus {bus_id: "b1"})
MATCH (bus6:Bus {bus_id: "b6"})
MATCH (bus11:Bus {bus_id: "b11"})
CREATE (bcompany1)-[:OWNS]->(bus1)
CREATE (bcompany1)-[:OWNS]->(bus6)
CREATE (bcompany1)-[:OWNS]->(bus11)
RETURN bcompany1, bus1, bus6, bus11


//creating relationship of Bus Company 2 with Bus

MATCH (bcompany2:Bus_company {company_id: "bc2"})
MATCH (bus2:Bus {bus_id: "b2"})
MATCH (bus7:Bus {bus_id: "b7"})
MATCH (bus12:Bus {bus_id: "b12"})
CREATE (bcompany2)-[:OWNS]->(bus2)
CREATE (bcompany2)-[:OWNS]->(bus7)
CREATE (bcompany2)-[:OWNS]->(bus12)
RETURN bcompany2, bus2, bus7, bus12


//creating relationship of Bus Company 3 with Bus

MATCH (bcompany3:Bus_company {company_id: "bc3"})
MATCH (bus3:Bus {bus_id: "b3"})
MATCH (bus8:Bus {bus_id: "b8"})
MATCH (bus13:Bus {bus_id: "b13"})
CREATE (bcompany3)-[:OWNS]->(bus3)
CREATE (bcompany3)-[:OWNS]->(bus8)
CREATE (bcompany3)-[:OWNS]->(bus13)
RETURN bcompany3, bus3, bus8, bus13
```

```
//creating relationship of Bus Company 4 with Bus

MATCH (bcompany4:Bus_company {company_id: "bc4"})
MATCH (bus4:Bus {bus_id: "b4"})
MATCH (bus9:Bus {bus_id: "b9"})
MATCH (bus14:Bus {bus_id: "b14"})
CREATE (bcompany4)-[:OWNS]->(bus4)
CREATE (bcompany4)-[:OWNS]->(bus9)
CREATE (bcompany4)-[:OWNS]->(bus14)
RETURN bcompany4, bus4, bus9, bus14


//creating relationship of Bus Company 5 with Bus

MATCH (bcompany5:Bus_company {company_id: "bc5"})
MATCH (bus5:Bus {bus_id: "b5"})
MATCH (bus10:Bus {bus_id: "b10"})
MATCH (bus15:Bus {bus_id: "b15"})
CREATE (bcompany5)-[:OWNS]->(bus5)
CREATE (bcompany5)-[:OWNS]->(bus10)
CREATE (bcompany5)-[:OWNS]->(bus15)
RETURN bcompany5, bus5, bus10, bus15
```

Figure 45: Creating relationship between bus company and bus

The query above creates a connection between Bus Company and their individual buses. Each of the queries shown above has a similar structure:

- First, it matches the bus company node by having a distinctive identification (company_id).
- Second, it matches the node representing that company's buses by their unique identification (bus_id).
- Finally, we utilized the OWNS relationship type to connect the bus company node and each bus node.

The RETURN statement will show the created relationship and the nodes involved in that relationship.

Outcome:

MATCH p=()-[:OWNS]->() RETURN p LIMIT 25;

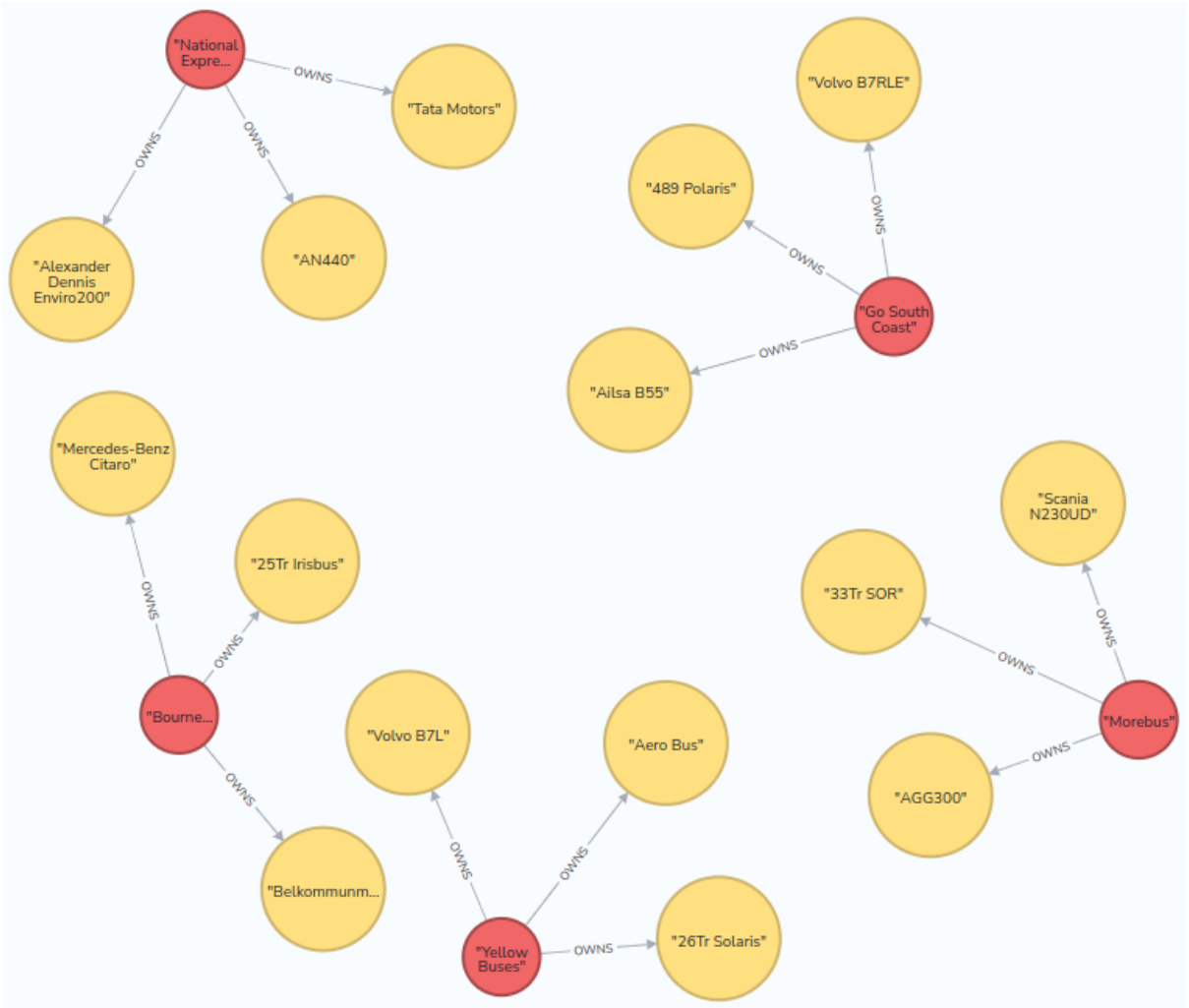


Figure 46: Graphical representation of relation between bus company and bus nodes

3.2.4. Route

```
//creating Route node

CREATE (route1: Route {route_id:"r1", name: "1A", length:"5"})
CREATE (route2: Route {route_id:"r2", name: "5", length:"7"})
CREATE (route3: Route {route_id:"r3", name: "M1", length:"10"})
CREATE (route4: Route {route_id:"r4", name: "17", length:"14"})
CREATE (route5: Route {route_id:"r5", name: "X6", length:"52"})

CREATE (route6: Route {route_id:"r6", name: "5A", length:"66"})
CREATE (route7: Route {route_id:"r7", name: "M2", length:"32"})
CREATE (route8: Route {route_id:"r8", name: "1b", length:"96"})
CREATE (route9: Route {route_id:"r9", name: "3x", length:"42"})
CREATE (route10: Route {route_id:"r10", name: "6a", length:"48"})

CREATE (route11: Route {route_id:"r11", name: "7A", length:"95"})
CREATE (route12: Route {route_id:"r12", name: "7B", length:"67"})
CREATE (route13: Route {route_id:"r13", name: "Breezer 40", length:"40"})
CREATE (route14: Route {route_id:"r14", name: "Breezer 60", length:"80"})
CREATE (route15: Route {route_id:"r15", name: "Cango C32, Cango C33", length:"10
0"})
```

Figure 47: Creating route node

The code above creates 15 nodes for the "Route" with the following properties:

Route_id: a unique identifier for a bus (string), name: name of the route where buses will run (string), and length: total length of a route in miles (string).

Outcome:

MATCH (n:Route) RETURN n LIMIT 25;

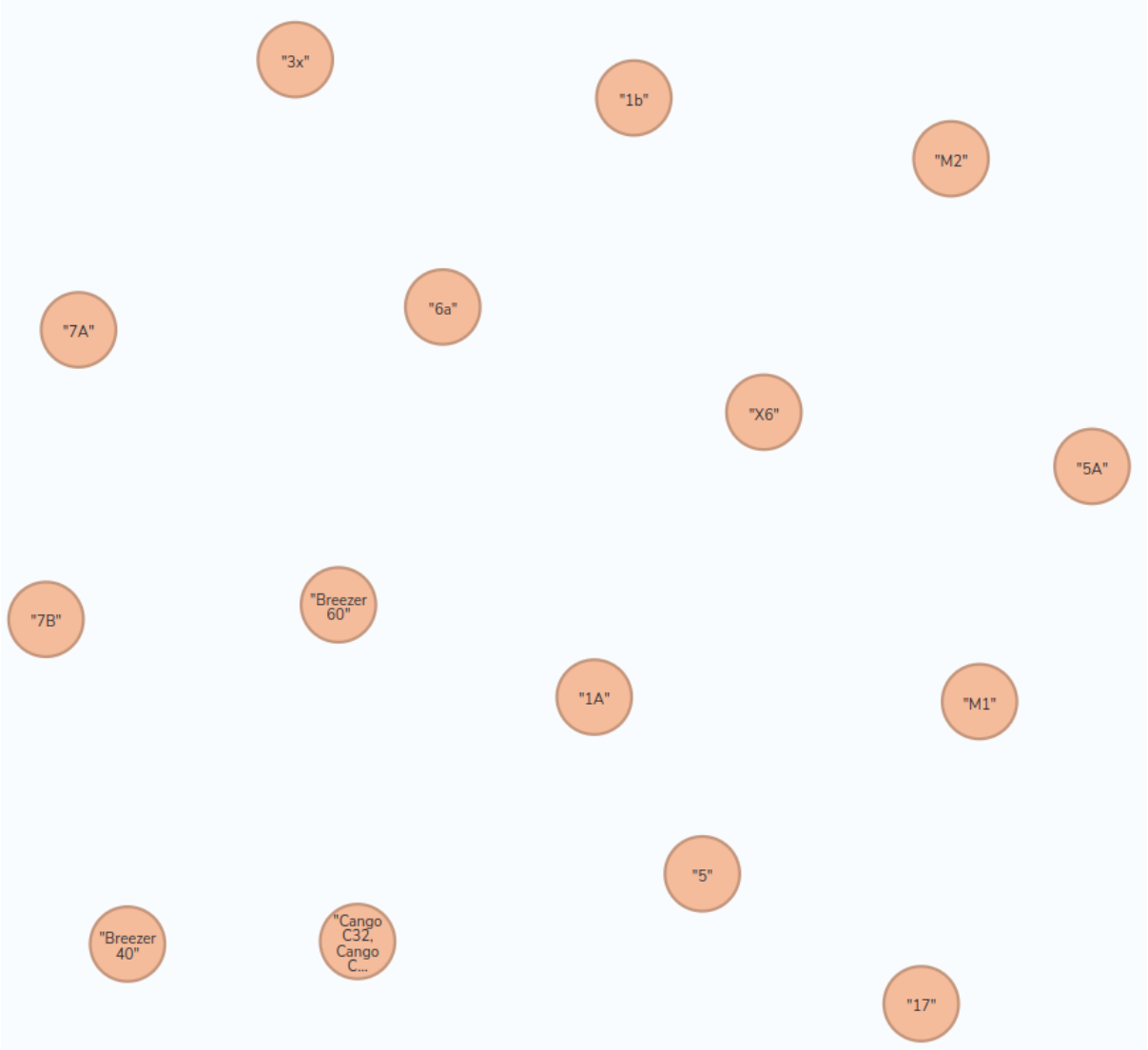


Figure 48: Fifteen route nodes

3.2.5. Relationship between Bus and Route

```
//creating relationship of Bus1 with Routes

MATCH (bus1:Bus {bus_id: "b1"})
MATCH (route1:Route {route_id: "r1"})
MATCH (route2:Route {route_id: "r2"})
MATCH (route3:Route {route_id: "r3"})
MATCH (route4:Route {route_id: "r4"})
MATCH (route5:Route {route_id: "r5"})
CREATE (bus1)-[:SERVES]->(route1)
CREATE (bus1)-[:SERVES]->(route2)
CREATE (bus1)-[:SERVES]->(route3)
CREATE (bus1)-[:SERVES]->(route4)
CREATE (bus1)-[:SERVES]->(route5)
RETURN bus1, route1, route2, route3, route4, route5


//creating relationship of Bus2 with Routes

MATCH (bus2:Bus {bus_id: "b2"})
MATCH (route2:Route {route_id: "r2"})
MATCH (route10:Route {route_id: "r10"})
MATCH (route7:Route {route_id: "r7"})
CREATE (bus2)-[:SERVES]->(route2)
CREATE (bus2)-[:SERVES]->(route10)
CREATE (bus2)-[:SERVES]->(route7)
RETURN bus2, route2, route10, route7


//creating relationship of Bus3 with Routes

MATCH (bus3:Bus {bus_id: "b3"})
MATCH (route14:Route {route_id: "r14"})
MATCH (route12:Route {route_id: "r12"})
MATCH (route10:Route {route_id: "r10"})
CREATE (bus3)-[:SERVES]->(route14)
CREATE (bus3)-[:SERVES]->(route12)
CREATE (bus3)-[:SERVES]->(route10)
RETURN bus3, route14, route12, route10
```

```
//creating relationship of Bus4 with Routes

MATCH (bus4:Bus {bus_id: "b4"})
MATCH (route6:Route {route_id: "r6"})
MATCH (route8:Route {route_id: "r8"})
MATCH (route10:Route {route_id: "r10"})
CREATE (bus4)-[:SERVES]->(route6)
CREATE (bus4)-[:SERVES]->(route8)
CREATE (bus4)-[:SERVES]->(route10)
RETURN bus4, route6, route8, route10


//creating relationship of Bus5 with Routes

MATCH (bus5:Bus {bus_id: "b5"})
MATCH (route12:Route {route_id: "r12"})
MATCH (route14:Route {route_id: "r14"})
MATCH (route1:Route {route_id: "r1"})
MATCH (route3:Route {route_id: "r3"})
CREATE (bus5)-[:SERVES]->(route12)
CREATE (bus5)-[:SERVES]->(route14)
CREATE (bus5)-[:SERVES]->(route1)
CREATE (bus5)-[:SERVES]->(route3)
RETURN bus5, route12, route14, route1, route3


//creating relationship of Bus6 with Routes

MATCH (bus6:Bus {bus_id: "b6"})
MATCH (route5:Route {route_id: "r5"})
MATCH (route7:Route {route_id: "r7"})
MATCH (route9:Route {route_id: "r9"})
MATCH (route11:Route {route_id: "r11"})
MATCH (route13:Route {route_id: "r13"})
MATCH (route15:Route {route_id: "r15"})
MATCH (route2:Route {route_id: "r2"})
MATCH (route4:Route {route_id: "r4"})
CREATE (bus6)-[:SERVES]->(route5)
CREATE (bus6)-[:SERVES]->(route7)
CREATE (bus6)-[:SERVES]->(route9)
CREATE (bus6)-[:SERVES]->(route11)
CREATE (bus6)-[:SERVES]->(route13)
CREATE (bus6)-[:SERVES]->(route15)
CREATE (bus6)-[:SERVES]->(route2)
CREATE (bus6)-[:SERVES]->(route4)
RETURN bus6, route5, route7, route9, route11, route13, route15, route2, route4
```

```
//creating relationship of Bus7 with Routes

MATCH (bus7:Bus {bus_id: "b7"})
MATCH (route6:Route {route_id: "r6"})
MATCH (route8:Route {route_id: "r8"})
MATCH (route10:Route {route_id: "r10"})
MATCH (route12:Route {route_id: "r12"})
MATCH (route14:Route {route_id: "r14"})
MATCH (route1:Route {route_id: "r1"})
CREATE (bus7)-[:SERVES]->(route6)
CREATE (bus7)-[:SERVES]->(route8)
CREATE (bus7)-[:SERVES]->(route10)
CREATE (bus7)-[:SERVES]->(route12)
CREATE (bus7)-[:SERVES]->(route14)
CREATE (bus7)-[:SERVES]->(route1)
RETURN bus7, route6, route8, route10, route12, route14, route1


//creating relationship of Bus8 with Routes

MATCH (bus8:Bus {bus_id: "b8"})
MATCH (route3:Route {route_id: "r3"})
MATCH (route5:Route {route_id: "r5"})
MATCH (route7:Route {route_id: "r7"})
MATCH (route9:Route {route_id: "r9"})
CREATE (bus8)-[:SERVES]->(route3)
CREATE (bus8)-[:SERVES]->(route5)
CREATE (bus8)-[:SERVES]->(route7)
CREATE (bus8)-[:SERVES]->(route9)
RETURN bus8, route3, route5, route7, route9
```

```
//creating relationship of Bus9 with Routes

MATCH (bus9:Bus {bus_id: "b9"})
MATCH (route11:Route {route_id: "r11"})
MATCH (route13:Route {route_id: "r13"})
MATCH (route15:Route {route_id: "r15"})
MATCH (route2:Route {route_id: "r2"})
MATCH (route4:Route {route_id: "r4"})
MATCH (route6:Route {route_id: "r6"})
MATCH (route8:Route {route_id: "r8"})
MATCH (route10:Route {route_id: "r10"})
MATCH (route12:Route {route_id: "r12"})
MATCH (route14:Route {route_id: "r14"})
CREATE (bus9)-[:SERVES]->(route11)
CREATE (bus9)-[:SERVES]->(route13)
CREATE (bus9)-[:SERVES]->(route15)
CREATE (bus9)-[:SERVES]->(route2)
CREATE (bus9)-[:SERVES]->(route4)
CREATE (bus9)-[:SERVES]->(route6)
CREATE (bus9)-[:SERVES]->(route8)
CREATE (bus9)-[:SERVES]->(route10)
CREATE (bus9)-[:SERVES]->(route12)
CREATE (bus9)-[:SERVES]->(route14)
RETURN bus9, route11, route13, route15, route2, route4, route6, route8, route10,
route12, route14
```

```
//creating relationship of Bus10 with Routes

MATCH (bus10:Bus {bus_id: "b10"})
MATCH (route1:Route {route_id: "r1"})
MATCH (route3:Route {route_id: "r3"})
MATCH (route5:Route {route_id: "r5"})
MATCH (route7:Route {route_id: "r7"})
MATCH (route9:Route {route_id: "r9"})
MATCH (route11:Route {route_id: "r11"})
MATCH (route13:Route {route_id: "r13"})
MATCH (route15:Route {route_id: "r15"})
MATCH (route2:Route {route_id: "r2"})
MATCH (route4:Route {route_id: "r4"})
MATCH (route6:Route {route_id: "r6"})
MATCH (route8:Route {route_id: "r8"})
MATCH (route10:Route {route_id: "r10"})
MATCH (route12:Route {route_id: "r12"})
MATCH (route14:Route {route_id: "r14"})
CREATE (bus10)-[:SERVES]->(route1)
CREATE (bus10)-[:SERVES]->(route3)
CREATE (bus10)-[:SERVES]->(route5)
CREATE (bus10)-[:SERVES]->(route7)
CREATE (bus10)-[:SERVES]->(route9)
CREATE (bus10)-[:SERVES]->(route11)
CREATE (bus10)-[:SERVES]->(route13)
CREATE (bus10)-[:SERVES]->(route15)
CREATE (bus10)-[:SERVES]->(route2)
CREATE (bus10)-[:SERVES]->(route4)
CREATE (bus10)-[:SERVES]->(route6)
CREATE (bus10)-[:SERVES]->(route8)
CREATE (bus10)-[:SERVES]->(route10)
CREATE (bus10)-[:SERVES]->(route12)
CREATE (bus10)-[:SERVES]->(route14)
RETURN bus10, route1, route3, route5, route7, route9, route11, route13, route15,
route2, route4, route6, route8, route10, route12, route14
```

```
//creating relationship of Bus11 with Routes

MATCH (bus11:Bus {bus_id: "b11"})
MATCH (route1:Route {route_id: "r1"})
MATCH (route15:Route {route_id: "r15"})
CREATE (bus11)-[:SERVES]->(route1)
CREATE (bus11)-[:SERVES]->(route15)
RETURN bus11, route1, route15


//creating relationship of Bus12 with Routes

MATCH (bus12:Bus {bus_id: "b12"})
MATCH (route5:Route {route_id: "r5"})
MATCH (route10:Route {route_id: "r10"})
MATCH (route15:Route {route_id: "r15"})
CREATE (bus12)-[:SERVES]->(route5)
CREATE (bus12)-[:SERVES]->(route10)
CREATE (bus12)-[:SERVES]->(route15)
RETURN bus12, route5, route10, route15


//creating relationship of Bus13 with Routes

MATCH (bus13:Bus {bus_id: "b13"})
MATCH (route2:Route {route_id: "r2"})
MATCH (route4:Route {route_id: "r4"})
MATCH (route6:Route {route_id: "r6"})
MATCH (route8:Route {route_id: "r8"})
MATCH (route10:Route {route_id: "r10"})
MATCH (route12:Route {route_id: "r12"})
MATCH (route14:Route {route_id: "r14"})
CREATE (bus13)-[:SERVES]->(route2)
CREATE (bus13)-[:SERVES]->(route4)
CREATE (bus13)-[:SERVES]->(route6)
CREATE (bus13)-[:SERVES]->(route8)
CREATE (bus13)-[:SERVES]->(route10)
CREATE (bus13)-[:SERVES]->(route12)
CREATE (bus13)-[:SERVES]->(route14)
RETURN bus13, route2, route4, route6, route8, route10, route12, route14
```

```
//creating relationship of Bus14 with Routes

MATCH (bus14:Bus {bus_id: "b14"})
MATCH (route1:Route {route_id: "r1"})
MATCH (route3:Route {route_id: "r3"})
MATCH (route5:Route {route_id: "r5"})
MATCH (route7:Route {route_id: "r7"})
MATCH (route9:Route {route_id: "r9"})
MATCH (route11:Route {route_id: "r11"})
MATCH (route13:Route {route_id: "r13"})
MATCH (route15:Route {route_id: "r15"})
CREATE (bus14)-[:SERVES]->(route1)
CREATE (bus14)-[:SERVES]->(route3)
CREATE (bus14)-[:SERVES]->(route5)
CREATE (bus14)-[:SERVES]->(route7)
CREATE (bus14)-[:SERVES]->(route9)
CREATE (bus14)-[:SERVES]->(route11)
CREATE (bus14)-[:SERVES]->(route13)
CREATE (bus14)-[:SERVES]->(route15)
RETURN bus14, route1, route3, route5, route7, route9, route11, route13, route15


//creating relationship of Bus15 with Routes

MATCH (bus15:Bus {bus_id: "b15"})
MATCH (route1:Route {route_id: "r1"})
CREATE (bus15)-[:SERVES]->(route1)
RETURN bus15, route1
```

Figure 49: Creating relationship between bus and route nodes

The code above creates relationship between buses and routes. Each of the queries shown above has a similar structure:

In the above queries we used MATCH clause to find the specific nodes that represent the bus and routes based on their unique identifier. Then, we used CREATE clause to create a relationship type i.e., SERVES to create relationship between bus node and route node.

Finally, the RETURN statement will show the created relationship and the nodes involved in that relationship.

Outcome:

MATCH p=()-[:SERVES]->() RETURN p;

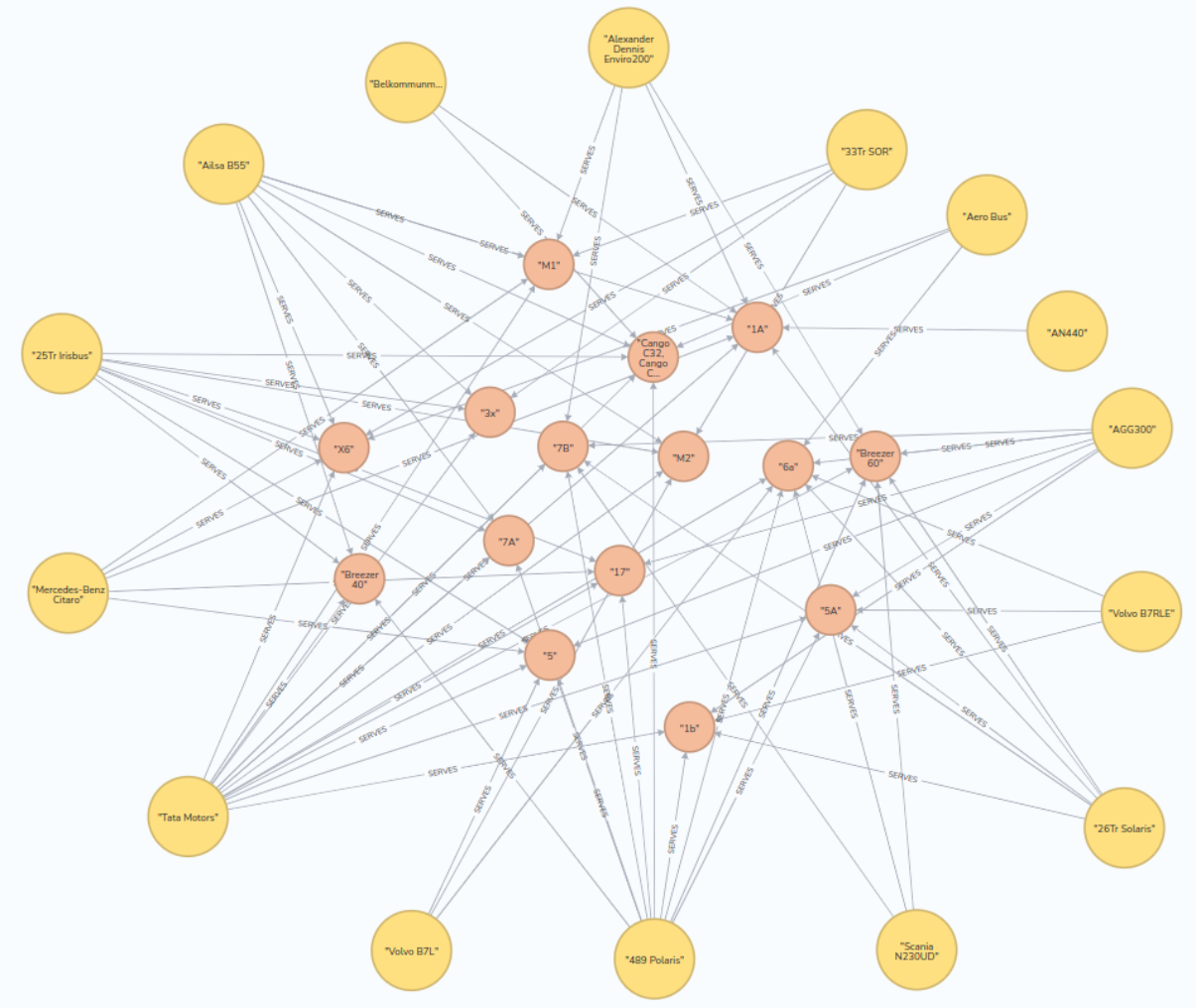


Figure 50: Graphical representation of relation between bus and route nodes

3.2.6. Trip

```
//creating Trip node

CREATE (trip1: Trip {trip_id:"t1", name: "Bournemouth Beach", tariff: "£3.10", t
ime:"25 mins"})
CREATE (trip2: Trip {trip_id:"t2", name: "Bournemouth Pier", tariff: "£2.50", ti
me:"20 mins"})
CREATE (trip3: Trip {trip_id:"t3", name: "Bournemouth Gardens", tariff: "£3.20",
time:"22 mins"})
CREATE (trip4: Trip {trip_id:"t4", name: "Boscombe Beach", tariff: "£1.70", tim
e:"15 mins"})
CREATE (trip5: Trip {trip_id:"t5", name: "Hengistbury Head", tariff: "£2.70", ti
me:"17 mins"})

CREATE (trip6: Trip {trip_id:"t6", name: "Poole Harbour", tariff: "£4.10", tim
e:"30 mins"})
CREATE (trip7: Trip {trip_id:"t7", name: "Christchurch Priory", tariff: "£3.50",
time:"27 mins"})
CREATE (trip8: Trip {trip_id:"t8", name: "Sandbanks Beach", tariff: "£4.50", tim
e:"35 mins"})
CREATE (trip9: Trip {trip_id:"t9", name: "Russell-Cotes Art Gallery and Museum",
tariff: "£2.70", time:"17 mins"})
CREATE (trip10: Trip {trip_id:"t10", name: "Bournemouth Aviation Museum", tarif
f: "£3.50", time:"27 mins"})
```

Figure 51: Creating trip node

The code above creates 10 nodes for the "Trip" with the following properties:

Trip_id: a unique identifier for a trip (string), name: the name of the trip (string), tariff: the amount of money required to reach that destination (string), and time: the total amount of time required to reach that destination (string).

Outcome:

MATCH (n:Trip) RETURN n LIMIT 25;

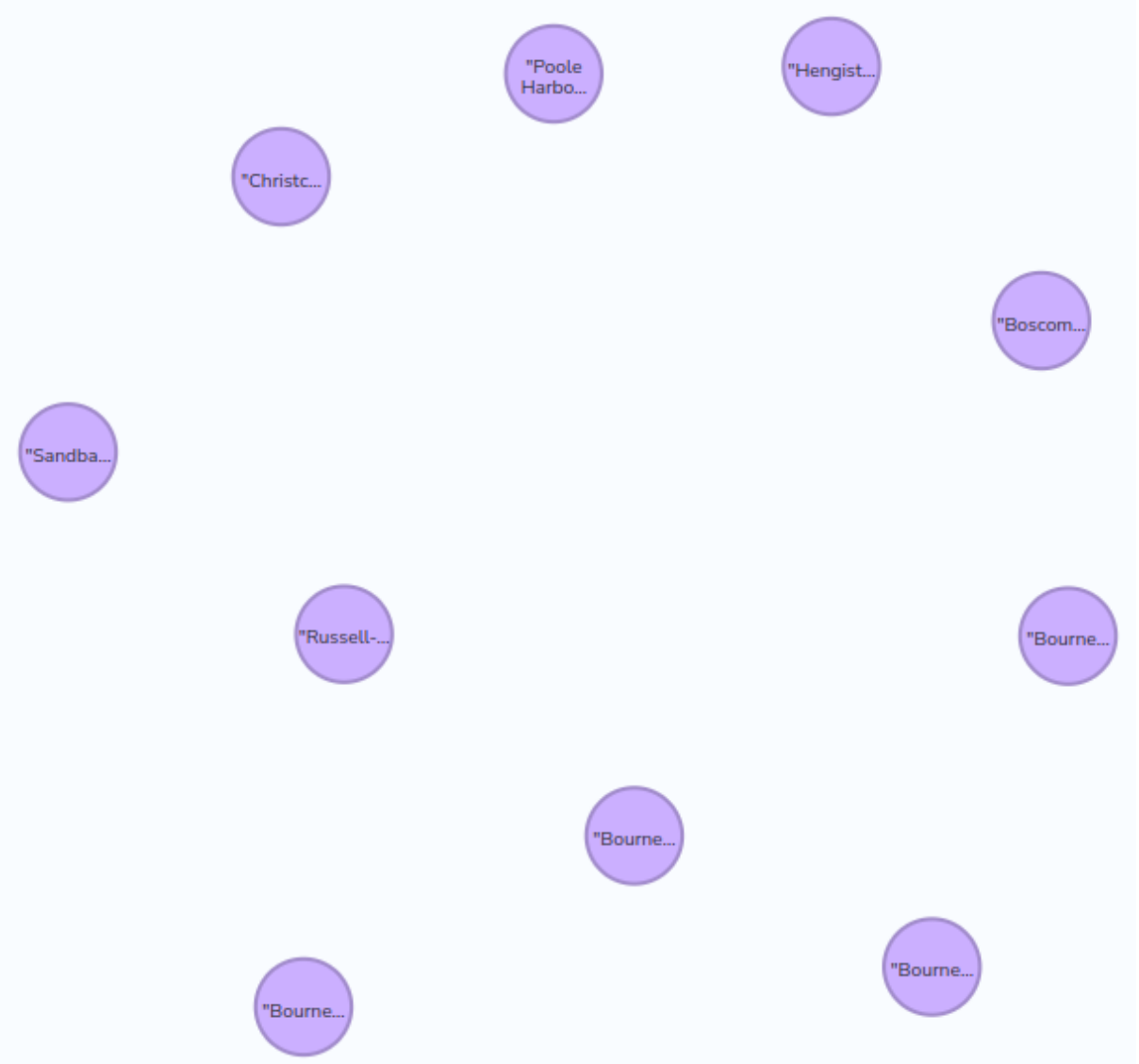


Figure 52: Ten route nodes

3.2.7. Relationship between Route and Trip

```
//creating relationship of Route1 with Trips

MATCH (route1:Route {route_id: "r1"})
MATCH (trip1:Trip {trip_id: "t1"})
MATCH (trip3:Trip {trip_id: "t3"})
MATCH (trip5:Trip {trip_id: "t5"})
CREATE (route1)-[:LEADS_TO]->(trip1)
CREATE (route1)-[:LEADS_TO]->(trip3)
CREATE (route1)-[:LEADS_TO]->(trip5)
RETURN route1, trip1, trip3, trip5


//creating relationship of Route2 with Trips

MATCH (route2:Route {route_id: "r2"})
MATCH (trip7:Trip {trip_id: "t7"})
MATCH (trip9:Trip {trip_id: "t9"})
CREATE (route2)-[:LEADS_TO]->(trip7)
CREATE (route2)-[:LEADS_TO]->(trip9)
RETURN route2, trip7, trip9


//creating relationship of Route3 with Trips

MATCH (route3:Route {route_id: "r3"})
MATCH (trip2:Trip {trip_id: "t2"})
MATCH (trip4:Trip {trip_id: "t4"})
MATCH (trip6:Trip {trip_id: "t6"})
MATCH (trip8:Trip {trip_id: "t8"})
MATCH (trip10:Trip {trip_id: "t10"})
MATCH (trip1:Trip {trip_id: "t1"})
CREATE (route3)-[:LEADS_TO]->(trip2)
CREATE (route3)-[:LEADS_TO]->(trip4)
CREATE (route3)-[:LEADS_TO]->(trip6)
CREATE (route3)-[:LEADS_TO]->(trip8)
CREATE (route3)-[:LEADS_TO]->(trip10)
CREATE (route3)-[:LEADS_TO]->(trip1)
RETURN route3, trip2, trip4, trip6, trip8, trip10, trip1
```

```
//creating relationship of Route4 with Trips

MATCH (route4:Route {route_id: "r4"})
MATCH (trip3:Trip {trip_id: "t3"})
MATCH (trip5:Trip {trip_id: "t5"})
MATCH (trip7:Trip {trip_id: "t7"})
CREATE (route4)-[:LEADS_TO]->(trip3)
CREATE (route4)-[:LEADS_TO]->(trip5)
CREATE (route4)-[:LEADS_TO]->(trip7)
RETURN route4, trip3, trip5, trip7


//creating relationship of Route5 with Trips

MATCH (route5:Route {route_id: "r5"})
MATCH (trip9:Trip {trip_id: "t9"})
MATCH (trip2:Trip {trip_id: "t2"})
CREATE (route5)-[:LEADS_TO]->(trip9)
CREATE (route5)-[:LEADS_TO]->(trip2)
RETURN route5, trip9, trip2


//creating relationship of Route6 with Trips

MATCH (route6:Route {route_id: "r6"})
MATCH (trip4:Trip {trip_id: "t4"})
MATCH (trip6:Trip {trip_id: "t6"})
MATCH (trip8:Trip {trip_id: "t8"})
CREATE (route6)-[:LEADS_TO]->(trip4)
CREATE (route6)-[:LEADS_TO]->(trip6)
CREATE (route6)-[:LEADS_TO]->(trip8)
RETURN route6, trip4, trip6, trip8
```

```
//creating relationship of Route7 with Trips

MATCH (route7:Route {route_id: "r7"})
MATCH (trip1:Trip {trip_id: "t1"})
MATCH (trip2:Trip {trip_id: "t2"})
MATCH (trip3:Trip {trip_id: "t3"})
MATCH (trip4:Trip {trip_id: "t4"})
MATCH (trip5:Trip {trip_id: "t5"})
MATCH (trip6:Trip {trip_id: "t6"})
MATCH (trip7:Trip {trip_id: "t7"})
MATCH (trip8:Trip {trip_id: "t8"})
MATCH (trip9:Trip {trip_id: "t9"})
MATCH (trip10:Trip {trip_id: "t10"})
CREATE (route7)-[:LEADS_TO]->(trip1)
CREATE (route7)-[:LEADS_TO]->(trip2)
CREATE (route7)-[:LEADS_TO]->(trip3)
CREATE (route7)-[:LEADS_TO]->(trip4)
CREATE (route7)-[:LEADS_TO]->(trip5)
CREATE (route7)-[:LEADS_TO]->(trip6)
CREATE (route7)-[:LEADS_TO]->(trip7)
CREATE (route7)-[:LEADS_TO]->(trip8)
CREATE (route7)-[:LEADS_TO]->(trip9)
CREATE (route7)-[:LEADS_TO]->(trip10)
RETURN route7, trip1, trip2, trip3, trip4, trip5,
trip6, trip7, trip8, trip9, trip10


//creating relationship of Route8 with Trips

MATCH (route8:Route {route_id: "r8"})
MATCH (trip1:Trip {trip_id: "t1"})
MATCH (trip5:Trip {trip_id: "t5"})
MATCH (trip10:Trip {trip_id: "t10"})
MATCH (trip2:Trip {trip_id: "t2"})
MATCH (trip6:Trip {trip_id: "t6"})
CREATE (route8)-[:LEADS_TO]->(trip1)
CREATE (route8)-[:LEADS_TO]->(trip5)
CREATE (route8)-[:LEADS_TO]->(trip10)
CREATE (route8)-[:LEADS_TO]->(trip2)
CREATE (route8)-[:LEADS_TO]->(trip6)
RETURN route8, trip1, trip5, trip10, trip2, trip6
```

```
//creating relationship of Route9 with Trips

MATCH (route9:Route {route_id: "r9"})
MATCH (trip6:Trip {trip_id: "t6"})
CREATE (route9)-[:LEADS_TO]->(trip6)
RETURN route9, trip6


//creating relationship of Route10 with Trips

MATCH (route10:Route {route_id: "r10"})
MATCH (trip1:Trip {trip_id: "t1"})
MATCH (trip10:Trip {trip_id: "t10"})
CREATE (route10)-[:LEADS_TO]->(trip1)
CREATE (route10)-[:LEADS_TO]->(trip10)
RETURN route10, trip1, trip10


//creating relationship of Route11 with Trips

MATCH (route11:Route {route_id: "r11"})
MATCH (trip2:Trip {trip_id: "t2"})
MATCH (trip9:Trip {trip_id: "t9"})
MATCH (trip3:Trip {trip_id: "t3"})
MATCH (trip8:Trip {trip_id: "t8"})
CREATE (route11)-[:LEADS_TO]->(trip2)
CREATE (route11)-[:LEADS_TO]->(trip9)
CREATE (route11)-[:LEADS_TO]->(trip3)
CREATE (route11)-[:LEADS_TO]->(trip8)
RETURN route11, trip2, trip9, trip3, trip8


//creating relationship of Route12 with Trips

MATCH (route12:Route {route_id: "r12"})
MATCH (trip4:Trip {trip_id: "t4"})
MATCH (trip7:Trip {trip_id: "t7"})
CREATE (route12)-[:LEADS_TO]->(trip4)
CREATE (route12)-[:LEADS_TO]->(trip7)
RETURN route12, trip4, trip7
```

```
//creating relationship of Route13 with Trips

MATCH (route13:Route {route_id: "r13"})
MATCH (trip5:Trip {trip_id: "t5"})
MATCH (trip6:Trip {trip_id: "t6"})
MATCH (trip10:Trip {trip_id: "t10"})
CREATE (route13)-[:LEADS_TO]->(trip5)
CREATE (route13)-[:LEADS_TO]->(trip6)
CREATE (route13)-[:LEADS_TO]->(trip10)
RETURN route13, trip5, trip6, trip10


//creating relationship of Route14 with Trips

MATCH (route14:Route {route_id: "r14"})
MATCH (trip1:Trip {trip_id: "t1"})
MATCH (trip5:Trip {trip_id: "t5"})
MATCH (trip10:Trip {trip_id: "t10"})
CREATE (route14)-[:LEADS_TO]->(trip1)
CREATE (route14)-[:LEADS_TO]->(trip5)
CREATE (route14)-[:LEADS_TO]->(trip10)
RETURN route14, trip1, trip5, trip10


//creating relationship of Route15 with Trips

MATCH (route15:Route {route_id: "r15"})
MATCH (trip1:Trip {trip_id: "t1"})
CREATE (route15)-[:LEADS_TO]->(trip1)
RETURN route15, trip1
```

Figure 53: Creating Relationship between route and trip nodes

The code above creates relationship between route and trip. Each of the queries shown above has a similar structure:

In the above queries we used MATCH clause to find the specific nodes that represent the route and trip based on their unique identifier. Then, we used CREATE clause to create a relationship type i.e., LEADS_TO to create relationship between route node and trip node.

Finally, the RETURN statement will show the created relationship and the nodes involved in that relationship.

Outcome:

MATCH p=()-[:LEADS_TO]->() RETURN p;

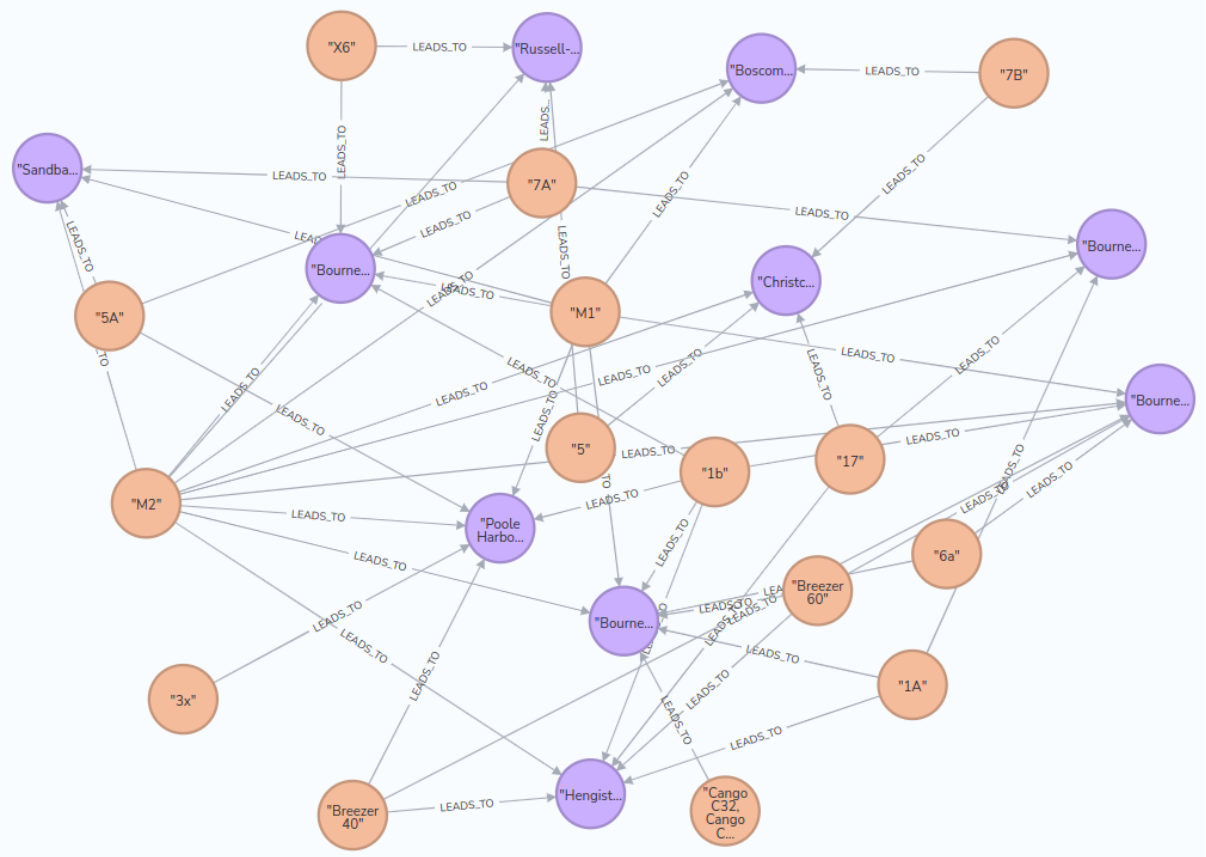


Figure 54: Graphical representation of relation between route and trip

Here is the whole graph database of bus network:

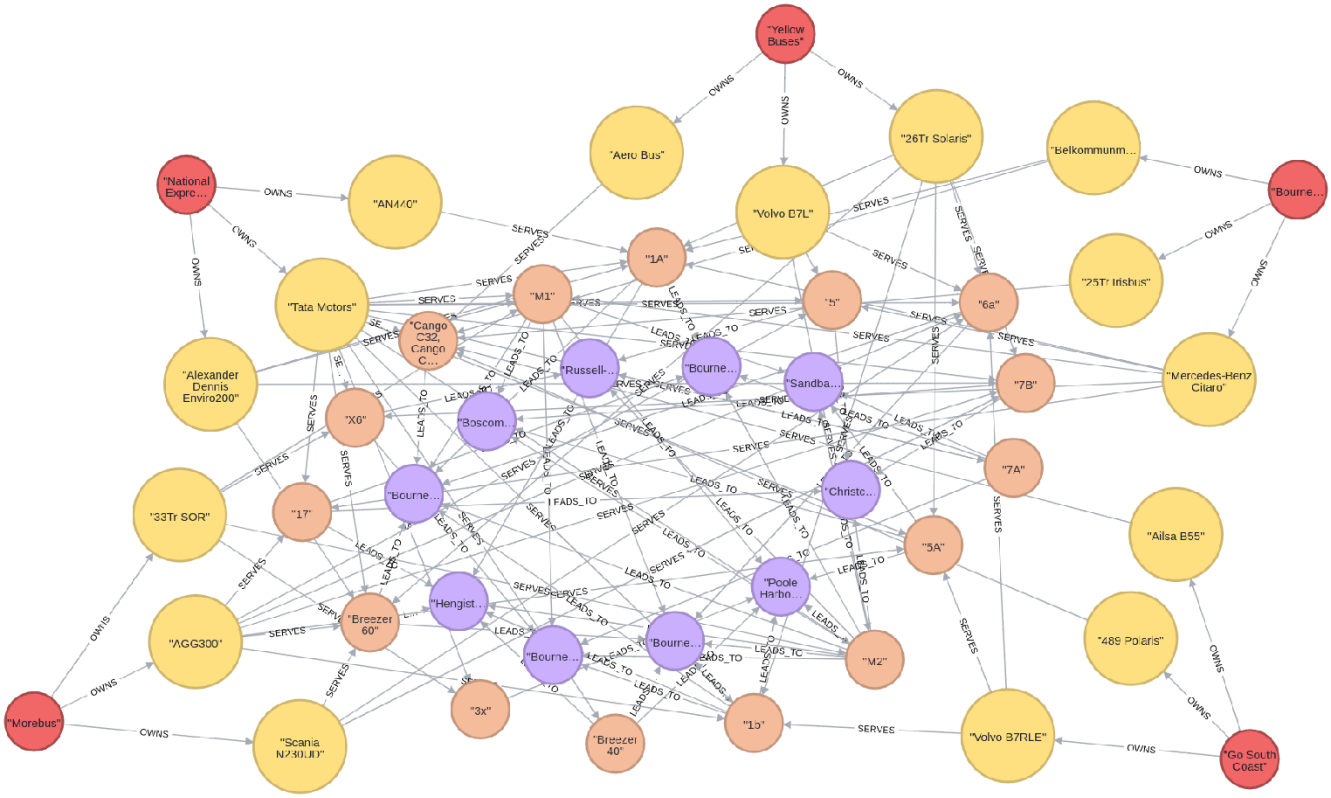


Figure 55: Graphical representation of full bus network

3.3. MongoDB

We are using entities Passenger, Discount, Address, and Trip for our Mongo DB. database

Mongo DB is a flexible data model that allows users to store data very flexibly, unlike traditional relational databases. In Mongo DB. Storing data in a table format consisting of rows and columns is unnecessary. So, in this database data of four entities are stored in JSON documents.

- The address has a one-to-one relation with the Passenger, meaning there is only one address for each passenger
- Discount has a one-to-one relation with passenger, meaning there a passenger may have a discount.
- Passenger has one too many relationships with the trip, meaning each passenger may have multiple trip data
- Phone number is a multivalued attribute, meaning a passenger has one or many phone numbers.

Taking the above relations into account the data below are plotted

3.3.1. Data Entry picture 1

Figure 56: Passenger 1 data

3.3.2. Data Entry picture 2

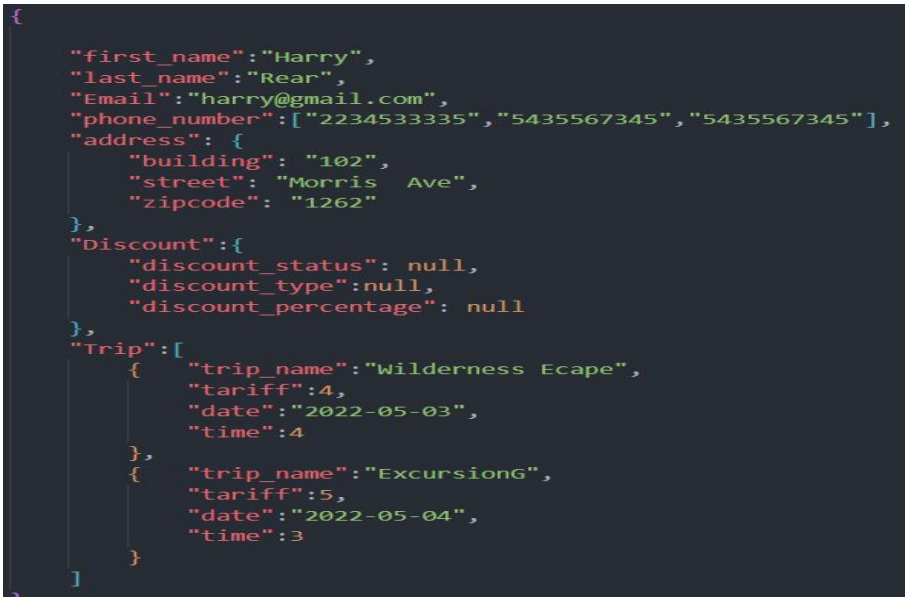


Figure 57: Passenger 2 data

3.3.3. Data Entry picture 3

Figure 58: Passenger 3 data

3.3.4. Data Entry picture 4

Figure 59: Passenger 4 data

3.3.5. Data Entry picture 5

Figure 60: Passenger 5 data

3.3.6. Data Entry Process

At first, we connect to mongo db cluster so that we can enter our data there

```
PS C:\Users\Asus> mongosh "mongodb+srv://cluster0.69z4wry.mongodb.net/myFirstDatabase"
--apiVersion 1 --username Biraj
Enter password: *********
Current Mongosh Log ID: 644ed3ce27190a2aeb2d64f1
Connecting to:          mongodb+srv://<credentials>@cluster0.69z4wry.mongodb.net/myFirs
tDatabase?appName=mongosh+1.8.1
Using MongoDB:          6.0.5 (API Version 1)
Using Mongosh:          1.8.1

For mongosh info see: https://docs.mongodb.com/mongodb-shell/

Atlas atlas-c2t7ml-shard-0 [primary] myFirstDatabase>
```

Figure 61: Data entry process 1

Secondly, we locate our data passenger.json and parse that data using JSON.parse() function

```
Atlas atlas-c2t7ml-shard-0 [primary] myFirstDatabase> var file = cat('C:/Users/Asus/Doc
uments/mangodb/passengers.json');

Atlas atlas-c2t7ml-shard-0 [primary] myFirstDatabase> var o = JSON.parse(file)
```

Figure 62: Data entry process 2

Finally, we store the JSON data in our collection passenger using the command db. passenger.insert() command

```
Atlas atlas-c2t7ml-shard-0 [primary] test> db.passenger.insert(o)
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId("6457f65cde54ff16aa1632f4"),
    '1': ObjectId("6457f65cde54ff16aa1632f5"),
    '2': ObjectId("6457f65cde54ff16aa1632f6"),
    '3': ObjectId("6457f65cde54ff16aa1632f7"),
    '4': ObjectId("6457f65cde54ff16aa1632f8")
  }
}
```

Figure 63: Data entry process 3

4. Reasons for Particular Entity:

In the case of SQL, data is organized in table form and different entities relate to each other by establishing relations like by using primary keys and foreign keys. SQL works better if we develop good relation between entities so, we choose three entities from our entity relation diagram they are Bus_company, Bus and Route. The reason behind choosing those entities is that we can easily build good relation between these entities. If we take real case scenario Bus_company have one to many or one to one relationship with bus and route and bus need a route to run and route serves bus and bus company, so they are directly depended on each other. The one more reason behind choosing these entities is that using these entities we can also create good test case.

In the case of Neo4J, data is organized as nodes and edges and this database models works for complex relationships. Here we wanted to plot relationship between Bus_company and Trip but in SQL it will be little complex to show because we must go through each structure relation using multiple tables but in Neo4J we can using different nodes and edges to reach out the destination. Neo4J is easy to use and can solve very complex data with minimal time and effort. The main reason behind choosing Neo4J for attributes like Bus_company, Bus, Route and Trip is that, if we have many attributes, we don't have to create tables for each entity, just node and edged will work and coding in Neo4J is easier than coding in SQL.

The entered data consists of data of 4 entities Discount (discount_status, discount_type), Address (building, street, zipcode), Passenger (First_name, Last_name, email, Phone_number (Multi Valued). The reason we have chosen these four entities is that it is well linked with Passenger real-time information making it easir to understand mongo db whoever views this document. Ultimately, these four entities are the kind of data normally used for social media posts, weblogs, and user preferences which is the most suitable for NoSQL databases like MongoDB.

5. Test Cases

5.1. SQL

5.1.1. Test Case 1:

```
SELECT bcompany.name, bus.model AS total_bus
FROM BusCompany bcompany
INNER JOIN Bus bus ON bcompany.company_id = bus.company_id
WHERE bcompany.company_id = 'bc3'
```

Figure 64: Creating SQL test case 1

The code above retrieves data from two tables, "BusCompany" and "Bus". We utilised INNER JOIN operation based on a "company_id" to join two tables.

The query we used above chooses the "name" column from the "BusCompany" table and the bus model from the "Bus" table, and we gave it the alias total_bus, and it also filters using "company_id" which is "bc3".

The result is shown below, where the company name and its total bus models are clearly represented.

Outcome:

```
SQL> SELECT bcompany.name, bus.model AS total_bus -- this command will select the company name and total mileage of all buses of that specific company
  2  FROM BusCompany bcompany -- from the BusCompany table, alias it as 'bcompany'
  3  INNER JOIN Bus bus ON bcompany.company_id = bus.company_id -- this code will join with the Bus table using the company_id column, alias it as 'bus'
  4  WHERE bcompany.company_id = 'bc3' -- it will filter for records where the company_id is bc3
```

Select all rows | Save as: CSV

NAME	TOTAL_BUS
Morebus	33Tr SOR
Morebus	AGG300
Morebus	Scania N230UD

Page 1 of 1 (1-3 of 3 rows)

Figure 65: Output of SQL test case 1

5.1.2. Test Case 2:

```
SELECT bcompany.name, b.model, r.name
FROM BUSCOMPANY bcompany
JOIN Bus b on bcompany.company_id = b.company_id
JOIN SERVES s ON b.bus_id = s.bus_id
JOIN ROUTE r on s.route_id = r.route_id
WHERE bcompany.name = 'Yellow Buses';
```

Figure 66: Creating SQL test case 2

The code above retrieves data from all tables, "BusCompany", “Bus”, “Route” and "Serves". We utilised multiple JOIN operations based on a BusCompany "name" to join tables.

The first JOIN operation is used to join “BusCompany” and “Bus” tables, using “company_id column as a key. The second JOIN operation is used to join “Bus” and “Serves” tables, using “bus_id” column as a key. The third JOIN operation is used to join “Serves” and “Route” tables, using the “route_id” column as a key.

Finally, we filtered the code using WHERE operation to retrieve the details about “Yellow Buses”.

Outcome:

```
SQL> SELECT bcompany.name, b.model, r.name
  2  FROM BUSCOMPANY bcompany
  3  JOIN Bus b on bcompany.company_id = b.company_id
  4  JOIN SERVES s ON b.bus_id = s.bus_id
  5  JOIN ROUTE r on s.route_id = r.route_id
  6  WHERE bcompany.name = 'Yellow Buses';
```

Select all rows | Save as: CSV

	NAME	MODEL	NAME
☑	Yellow Buses	26Tr Solaris	M2
☑	Yellow Buses	26Tr Solaris	3x
☑	Yellow Buses	Aero Bus	7B
☑	Yellow Buses	Aero Bus	5
☑	Yellow Buses	Volvo B7L	Breezer 60
☑	Yellow Buses	Volvo B7L	5

Page 1 of 1 (1-6 of 6 rows)

Figure 67: Output of SQL test case 2

5.1.3. Test Case 3:

```
SELECT bcompany.name, SUM(b.mileage) AS total_mileage
FROM BusCompany bcompany
INNER JOIN Bus b ON bcompany.company_id = b.company_id
WHERE bcompany.company_id = 'bc5'
GROUP BY bcompany.name;
```

Figure 68: Creating SQL test case 3

The code above retrieves data from two tables, "BusCompany" and “Bus”. We utilised INNER JOIN operations based to combine tables based on their shared keys.

It will select “name” from the “BusCompany” table and will show the total mileage of the buses with company_id “bc5”. With the SELECT operation we have used SUM function to calculate the total sum of mileage. Finally, the GROUP BY is used to group the results by company name. It’s important to use GROUP BY operation when we use aggregate function like SUM, this is because it will return the single value for each group of rows.

Using this query, it will return the single line output with company name along with the total sum of buses that belongs to the company.

Outcome:

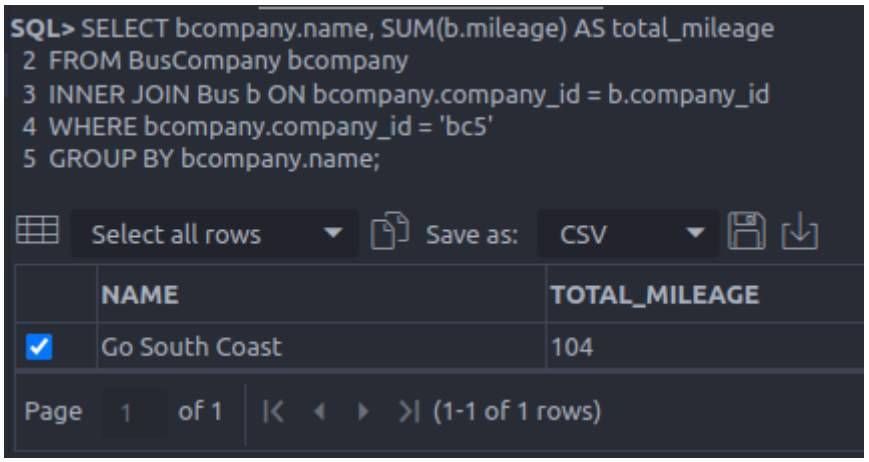


Figure 69: Output of SQL test case 3

5.1.4. Test Case 4:

```
SELECT name, (SELECT AVG(mileage) FROM Bus
WHERE company_id = bcompany.company_id) AS average_mileage
From BUSCOMPANY bcompany
WHERE name = 'Bournemouth Transport Ltd';
```

Figure 70: Creating SQL test case 4

The code above retrieves data from two tables, "BusCompany" and “Bus”. The above query used for extracting bus company name and average mileage of its buses.

The first line of the query will choose the name of a company from the "BusCompany" table, and the average mileage is calculated using a subquery that will select the average mileage from the "Bus" table based on a "company_id".

Finally, the outer part of the query will filter based on company name.

Outcome:

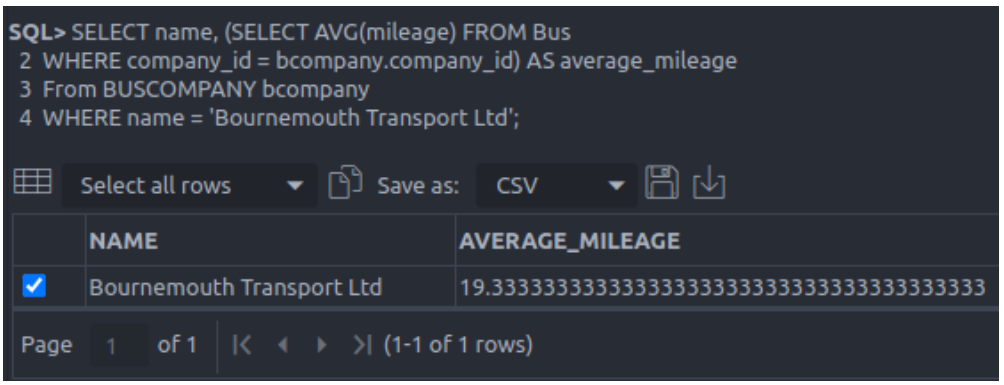


Figure 71: Output of test case 4

5.1.5. Test Case 5:

```
SELECT b.model AS Bus_Model, b.year AS Manufacture_Year, r.name AS Route_Name, bc.name AS BusCompany_Name
FROM Bus b
JOIN BUSCOMPANY bc ON b.company_id = bc.company_id
JOIN SERVES s ON b.bus_id = s.bus_id
JOIN Route r on s.route_id = r.route_id
WHERE b.year < 2015;
```

Figure 72: Creating SQL test case 5

The code above retrieves data from four tables, "BusCompany", “Bus”, “Route” and “Serves”. The above query is mainly used for extracting bus model that were manufactured before 2015.

This query uses multiple JOIN operation but mainly it joins “Bus” table with “BusCompany” using “company_id” column and after it joins the table with “Serves” table using the “bus_id” column. Finally, everything is joined with “Route” table using “route_id”. Outer layer of this query filters using WHERE operation where the year is less than 2015.

Outcome:

```
SQL> SELECT b.model AS Bus_Model, b.year AS Manufacture_Year, r.name AS Route_Name, bc.name AS BusCompany_Name
  2  FROM Bus b
  3  JOIN BUSCOMPANY bc ON b.company_id = bc.company_id
  4  JOIN SERVES s ON b.bus_id = s.bus_id
  5  JOIN Route r on s.route_id = r.route_id
  6  WHERE b.year < 2015;
```

Select all rows | Save as: CSV

BUS_MODEL	MANUFACTURE_YEAR	ROUTE_NAME	BUSCOMPANY_NAME
26Tr Solaris	1995	M2	Yellow Buses
33Tr SOR	2011	1b	Morebus
26Tr Solaris	1995	3x	Yellow Buses
Tata Motors	2011	6a	Go South Coast
25Tr Irisbus	1990	6a	Bournemouth Transport Ltd
Aero Bus	2011	7B	Yellow Buses
Volvo B7RLE	2011	7B	Wilts & Dorset
Scania N230UD	2013	Breezer 40	Morebus
Ailsa B55	2011	Breezer 60	Wilts & Dorset
Volvo B7L	2010	Breezer 60	Yellow Buses

Page 1 of 2 (1-10 of 17 rows)

Figure 73: Output of test case 5

5.2. Neo4J

5.2.1. Test Case 1:

```
1 MATCH (b:Bus)-[:SERVES]→(:Route)
2 WITH b, count(*) AS sr
3 WHERE sr > 5
4 RETURN b.model, sr
5 ORDER BY b.model
```

Table RAW

b.model	sr
"25Tr Irisbus"	8
"26Tr Solaris"	6
"489 Polaris"	10
"AGG300"	7
"Ailsa B55"	8
"Tata Motors"	15

Showing **1-6** of **6** results

Figure 74: Neo4j test case 1

Above code is picking all "Bus" nodes that have a "SERVES" link to a "Route" node. The results are then grouped by Bus and the number of routes they serve is counted and saved as the variable "sr." It limits the results to buses that serve on more than two routes (sr > 5).

Finally, it outputs the model's name of each Bus as well as the number of routes on which it operates as shown in the above picture.

5.2.2. Test Case 2:

```
1 MATCH (b:Bus)
2 OPTIONAL MATCH (b:Bus)-[s:SERVES]→(:Route)
3 RETURN b.model, count(s)
```

Table RAW

b.model	count(s)
"Mercedes-Benz Citaro"	5
"Volvo B7L"	3
"Scania N230UD"	3
"Volvo B7RLE"	3
"Alexander Dennis Enviro200"	4
"25Tr Irisbus"	8

Showing **1-15** of **15** results

Figure 75: Neo4j test case 2

Above code first match all the nodes of “Bus” and counts the total number of relationships labelled as “SERVES” that each “Bus” node has with a “Route” node. As we can see in the above figure it returns two columns i.e., model of bus and total number of buses that serves in a particular route.

5.2.3. Test Case 3:

```
1 MATCH (r:Route)
2 WITH r, toInteger(r.length) as lengthValue
3 RETURN r.name, sum(lengthValue) as total_length
```

Table RAW

r.name	total_length
"1A"	5
"5"	7
"M1"	10
"17"	14
"X6"	52
"5A"	66
"M2"	32
"1b"	96

Showing **1-15** of **15** results

Figure 76: Neo4j test case 3

Above code first match with the “Route” node using “MATCH” function then in the second line we are using “WITH” clause to pass the match nodes to next stage where we are converting string length to integer using inbuilt function called “toInteger()”. Then after out length get converted to integer to return the route name and total length of the route as total_length.

5.2.4. Test Case 4:

```
MATCH (bc:Bus_company)-[:OWNS]→(b:Bus)
WITH bc, b
MATCH (b)-[:SERVES]→(r:Route)
WITH bc, b, r
MATCH (r)-[:LEADS_TO]→(t:Trip)
RETURN bc.name, b.model, r.name, t.name
```

Table | RAW

bc.name	b.model	r.name	t.name
"Bournemouth Transport Ltd"	"Belkommunmash"	"1A"	"Hengistbury Head"
"Bournemouth Transport Ltd"	"Belkommunmash"	"1A"	"Bournemouth Gardens"
"Bournemouth Transport Ltd"	"Belkommunmash"	"1A"	"Bournemouth Beach"
"Bournemouth Transport Ltd"	"Belkommunmash"	"Cango C32, Cango C33"	"Bournemouth Beach"
"Bournemouth Transport Ltd"	"25Tr Irisbus"	"5"	"Russell-Cotes Art Gallery and Museum"

Showing **1-50** of **264** results 1 2 ... 6 Show 50

Figure 77: Neo4j test case 4

Above code first match with the “Bus_company” then through “OWNS” relation node it matches with the “Bus” node. Then “Bus” nodes match with the “Route” node using “SERVES” relation node. Finally, “Route” node matches with “Trip” node using their relation node “LEADS_TO”. Then the query retrieves information of bus company name and bus they own and route that serves those buses and finally trip name the route takes.

5.2.5. Test Case 5:

```
1 MATCH (bc:Bus_company{company_id:"bc3"})-[:OWNS]→(b:Bus)-[:SERVES]→(r:Route)
2 RETURN bc.name, b.model, r.name
```

Table | RAW

bc.name	b.model	r.name
"Morebus"	"AGG300"	"6a"
"Morebus"	"AGG300"	"5"
"Morebus"	"AGG300"	"1b"
"Morebus"	"AGG300"	"5A"
"Morebus"	"AGG300"	"Breezer 60"
"Morebus"	"AGG300"	"7B"
"Morebus"	"AGG300"	"17"
"Morebus"	"33Tr SOR"	"M1"

Showing **1-14** of **14** results

Show 50

Figure 78: Neo4j test case 5

Above code first match the “Bus_company” node that has company_id of “bc3” and using “OWNS” relation node it will match all the buses “bc3” owns and again using “SERVES” relation node it will match “Route” node were those buses runs. Finally, it will print information about bus company name, bus model name that company owns and route name that serves those buses.

5.3. MongoDB

5.3.1. Test case 1

Before applying Query:

```
Atlas atlas-c2t7ml-shard-0 [primary] test> db.passenger.findOne( { "first_name": "Alia", "last_name": "Sing" });
{
  _id: ObjectId("6457f65cde54ff16aa1632f8"),
  first_name: 'Alia',
  last_name: 'Sing',
  Email: 'Alia@gmail.com',
  phone_number: [ '2234039345', '2234510225', '2344407345' ],
  address: { building: '19', street: 'Road 3', zipcode: '4633' },
  Discount: {
    discount_status: 'Applied',
    discount_type: 'child',
    discount_percentage: 50
  },
  Trip: [
    {
      trip_name: 'Urban Excursion',
      tariff: 4,
      date: '2022-05-09',
      time: 2
    },
    {
      tariff: 5,
      trip_name: 'Excursion 3',
      date: '2022-05-10',
      time: 5
    }
  ]
}
```

Figure 79: Before MangoDB test case 1

Query Used:

```
db.passenger.updateOne(
{
    "first_name": "Alia",
    "Trip.trip_name": "Urban Excursion"
},
{
    $set: { "Trip.$.trip_name": "Updated Trip" }
}
);
```

Figure 80: MangoDB test case 1

This query updateOne() finds the First name Alia and the Trip name Urban Excursion and uses $set operator then set the value of the trip name to an updated Trip for the user Alia. The trip_name changes into Updated Trip after applying Query.

```
Atlas atlas-c2t7ml-shard-0 [primary] test> db.passenger.updateOne(
... {
...     "first_name": "Alia",
...     "Trip.trip_name": "Urban Excursion"
... },
... {
...     $set: { "Trip.$.trip_name": "Updated Trip" }
... }
... );
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
Atlas atlas-c2t7ml-shard-0 [primary] test> db.passenger.findOne( { "first_name": "Alia", "last_name": "Sing" });
{
  _id: ObjectId("6457f65cde54ff16aa1632f8"),
  first_name: 'Alia',
  last_name: 'Sing',
  Email: 'Alia@gmail.com',
  phone_number: [ '2234039345', '2234510225', '2344407345' ],
  address: { building: '19', street: 'Road 3', zipcode: '4633' },
  Discount: {
    discount_status: 'Applied',
    discount_type: 'child',
    discount_percentage: 50
  },
  Trip: [
    {
      trip_name: 'Updated Trip',
      tariff: 4,
      date: '2022-05-09',
      time: 2
    },
    {
      tariff: 5,
      trip_name: 'Excursion 3',
      date: '2022-05-10',
      time: 5
    }
  ]
}
```

Figure 81: Output of mangodb test case 1

5.3.2. Test Cases 2:

Before applying the query:

```
Atlas atlas-c2t7ml-shard-0 [primary] test> db.passenger.find({}, { "Discount.discount_status": 1, _id: 0 })
[
  { Discount: { discount_status: 'Applied' } },
  { Discount: { discount_status: null } },
  { Discount: { discount_status: 'Applied' } },
  { Discount: { discount_status: 'Applied' } },
  { Discount: { discount_status: 'Applied' } }
]
```

Figure 82: Before MangoDB test case 2

Query Used:

```
db.passenger.updateMany( { "Discount.discount_status": "Applied" }, { $set: { "Discount.discount_status": "Expired" } })
```

Figure 83: MangoDB test case 2

This queryuses updateMany() to update all the values in the document with Discount Status Applied to Expired using the $set parameter.

The results show all 5 data have been changed to Expired

```
Atlas atlas-c2t7ml-shard-0 [primary] test> db.passenger.updateMany( { "Discount.discount_status": "Applied" }, { $set: { "Discount.discount_status": "Expired" } })
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 4,
  modifiedCount: 4,
  upsertedCount: 0
}
Atlas atlas-c2t7ml-shard-0 [primary] test> db.passenger.find({}, { "Discount.discount_status": 1, _id: 0 })
[
  { Discount: { discount_status: 'Expired' } },
  { Discount: { discount_status: null } },
  { Discount: { discount_status: 'Expired' } },
  { Discount: { discount_status: 'Expired' } },
  { Discount: { discount_status: 'Expired' } }
]
```

Figure 84: Output of mangodb test case 2

5.3.3. Test cases 3:

Query Used:

```
db.passenger.aggregate([
    {$unwind: "$Trip"},{ $group:{_id:null,averageTariff: {$avg: "$Trip.tariff"}}}
])
```

Figure 85: MangoDB test case 3

This query basically calculates the average tariff of all the data present in the document.

Query uses:

- $unwind operator to flatten the data in a Trip array
- $group operator gathers all these data into a single group
- $avg calculates the average tariff of all data in the documents

The results show the average tariff of trips for all data in the document

```
Atlas atlas-c2t7ml-shard-0 [primary] test> db.passenger.aggregate([
...     {$unwind: "$Trip"},{ $group:{_id:null,averageTariff: {$avg: "$Trip.tariff"}}}
... ])
[ { _id: null, averageTariff: 4.2 } ]
```

Figure 86: Output of mangodb test case 3

5.3.4. Test cases 4

Query Used:

```
db.passenger.aggregate([
    { $unwind: "$Trip" },{ $sort: { "Trip.time": 1 } },
    { $group:
        {
        _id: "$Trip.trip_name",
        time: { $first: "$Trip.time" },
        tariff:{$first:"$Trip.tariff", }
        }
    },
        { $sort: { time: 1 } },{ $project: { _id: 0, trip_name: "$_id", time: 1, tariff:1 } }
    ])
```

Figure 87: MangoDB test case 4

This query calculates the shortest time with its tariff values for each trip_name from the collection passenger and gives the output in ascending order with time tariff and trip_name. Query uses:

- $Unwind flattens the array
- $sort sorts data in ascending order
- $group groups the data by trip_name field within Trip and measures the first-time duration and its tariff value for every group.
- $sort sorts the data in ascending manner
- $project reshapes the document and gives output with time, tariff, and trip_name only and it excludes _id field.

The result shows the shortest duration time with their tariff and trip_name

```
Atlas atlas-c2t7ml-shard-0 [primary] test> db.passenger.aggregate([
...     { $unwind: "$Trip" },{ $sort: { "Trip.time": 1 } },
...     { $group:
...             {
...             _id: "$Trip.trip_name",
...             time: { $first: "$Trip.time" },
...             tariff:{$first:"$Trip.tariff", }
...             }
...     },
...             { $sort: { time: 1 } },{ $project: { _id: 0, trip_name: "$_id", time: 1, tariff:1 } }
...     ])
[
  { time: 1, tariff: 5, trip_name: 'ExcursionA' },
  { time: 1, tariff: 4, trip_name: 'Cultural Expedition' },
  { time: 2, tariff: 2, trip_name: 'Hudden Delight' },
  { time: 2, tariff: 4, trip_name: 'Updated Trip' },
  { time: 3, tariff: 4, trip_name: 'Lakeside Retreat' },
  { time: 3, tariff: 4, trip_name: 'ExcursionF' },
  { time: 3, tariff: 5, trip_name: 'ExcursionG' },
  { time: 4, tariff: 4, trip_name: 'Wilderness Ecape' },
  { time: 5, tariff: 5, trip_name: 'ExcursionB' },
  { time: 5, tariff: 5, trip_name: 'Excursion 3' }
]
```

Figure 88: Output of mangodb test case 4

5.3.5. Test cases 5:

Query Used:

```
db.passenger.updateMany({}, [
  { $set: { Length_of_phoneNumbers: { $size: "$phone_number" } } }
])
```

Figure 89: MangoDB test case 5

This query creates the Length_of_phoneNumbers field in a document which consists of the total number of phone numbers a user has for each data in the document.

Query uses:

- $size to return the number of elements in phone_number
- $set to set the new value for every data in the document
- $updateMany to update every data in the passenger's collection

Results show the new parameter Length_of_phoneNumbers which contains a total number of phone numbers for all the data in the document.

```
Atlas atlas-c2t7ml-shard-0 [primary] test> db.passenger.find({}, { Length_of_phoneNumbers: 1, _id: 0 })
[
  { Length_of_phoneNumbers: 3 },
  { Length_of_phoneNumbers: 3 },
  { Length_of_phoneNumbers: 3 },
  { Length_of_phoneNumbers: 3 },
  { Length_of_phoneNumbers: 3 }
]
```

Figure 90: Output of mangodb test case 5

Part B

In an era of growing data complexity and volume, feature selection and construction techniques play a key role in understanding our data in helping reduce the dimensionality and improve learnability in data analytics problems. Both for data and big data processing and analytics feature selection techniques are important for reducing the time required to build machine learning models and improving the performance of these algorithms. Moreover, principal component analysis is an important algorithm used in data and big data processing for the purpose of data visualisation, as well as for dimensionality reduction and for gaining insight in the knowledge hidden in the data.

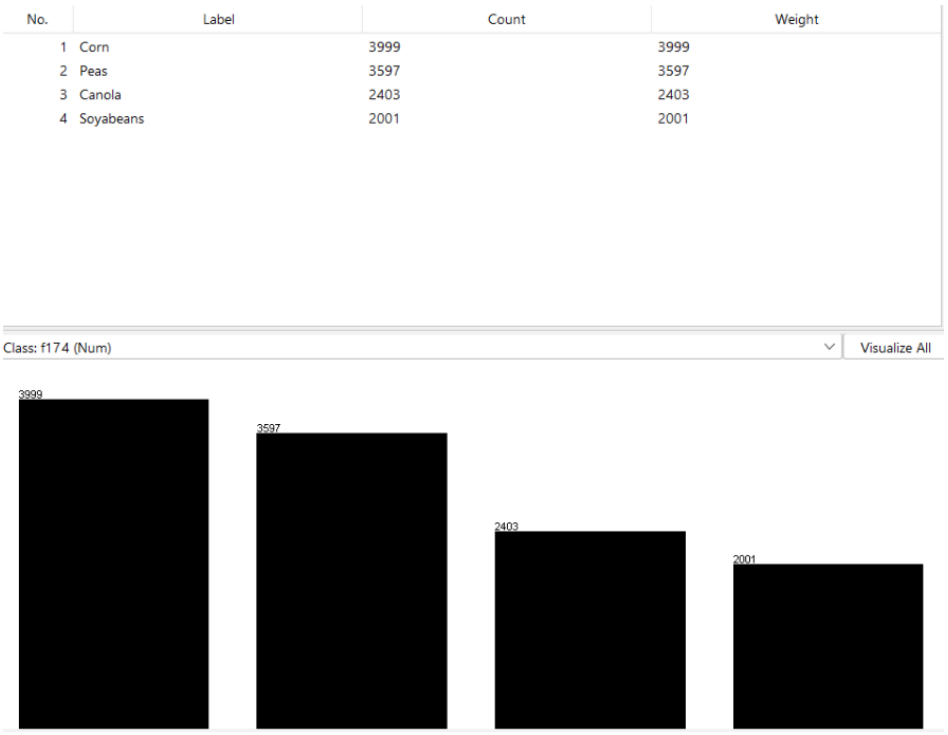
For the submission, you are given 3 datasets (below this section) and you are asked to define the classification problem of your choice, select one dataset and perform the following tasks:

1. Define the training and testing set for your dataset. 2. Implement neural network and one other classification algorithm of your choice and compare the performance for the dataset you choose. 3. Apply Principal Component Analysis to the dataset and explain its outcome. How does the number of principal components affect the percentage of variance covered for this dataset? 4. Apply any feature selection method of your choice and compare the performance using any one algorithm used in question 2 before and after applying the feature selection algorithm. 5. Discuss the challenges and implications regarding the time required to build the required models. Compare the times with and without feature selection method.

Part B:

6. Training and Testing

For this task, we are given a dataset for “Crop mapping using fused optical radar”, in which there are 4 crop type classes with their respective instances i.e. – Corn (3999), Peas (3597), Canola (2403), and Soybeans (2001) - totalling up to 12000.



No.	Label	Count	Weight
1	Corn	3999	3999
2	Peas	3597	3597
3	Canola	2403	2403
4	Soyabeans	2001	2001

Figure 91: Dataset category

Regarding the Attribute Information. It has a total of 175 attributes including:

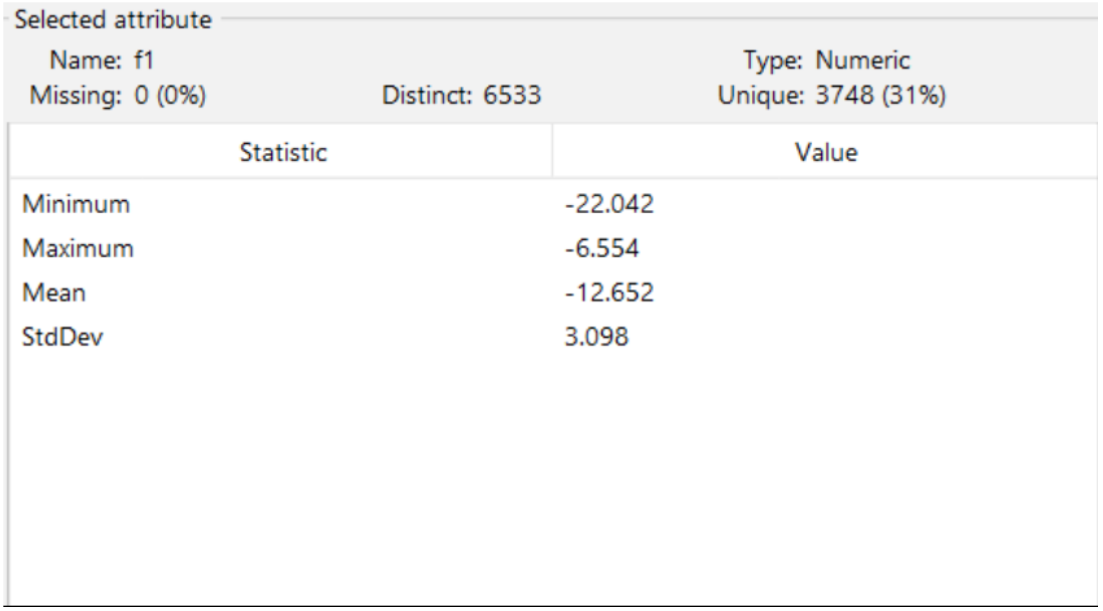
- Class.
- f1 to f49: Polarimetric features on 05 July 2012.
- f50 to f98: Polarimetric features on 14 July 2012.
- f99 to f136: Optical features on 05 July 2012.
- f137 to f174: Optical features on 14 July 2012.

The selected attribute is f1 which has a minimum value of -22.042 which indicates the smallest observed value for the attribute within the dataset and a Maximum value of -6.554 which indicates the largest observed value for the attribute within the dataset.

By inspecting an attribute's Minimum and Maximum values, we can gain insights into the range or spread of the values.

The Mean value of an attribute represents the average value of that attribute within a dataset and in this attribute, the Mean value is -12.652.

StdDev or Standard Deviation is a statistical measure that quantifies the amount of variation or dispersion in a set of values. For f1 attribute the StdDev value is 3.098.


Selected attribute

Name: f1 Type: Numeric
Missing: 0 (0%) Distinct: 6533 Unique: 3748 (31%)

Statistic	Value
Minimum	-22.042
Maximum	-6.554
Mean	-12.652
StdDev	3.098

Figure 92: Minimum and Maximum values

Here’s the graphical representation of the attribute’s data.

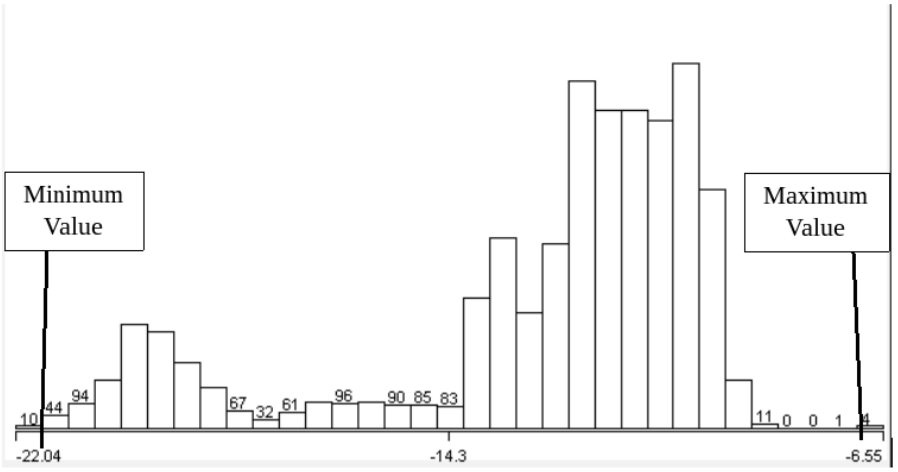


Figure 93: Graphical representation of the attribute’s data

Another Example we can see is for an attribute f49: -

Selected attribute

Name: f49 Type: Numeric
Missing: 0 (0%) Distinct: 9146 Unique: 7963 (66%)

Statistic	Value
Minimum	0.006
Maximum	0.134
Mean	0.052
StdDev	0.038

Figure 94: MinMax value for f49

Here, the Minimum and Maximum value ranges from 0.006 to 0.134 respectively. The Mean value of the attribute is 0.052. And the StdDev value is 0.038.

The data is graphically represented below:

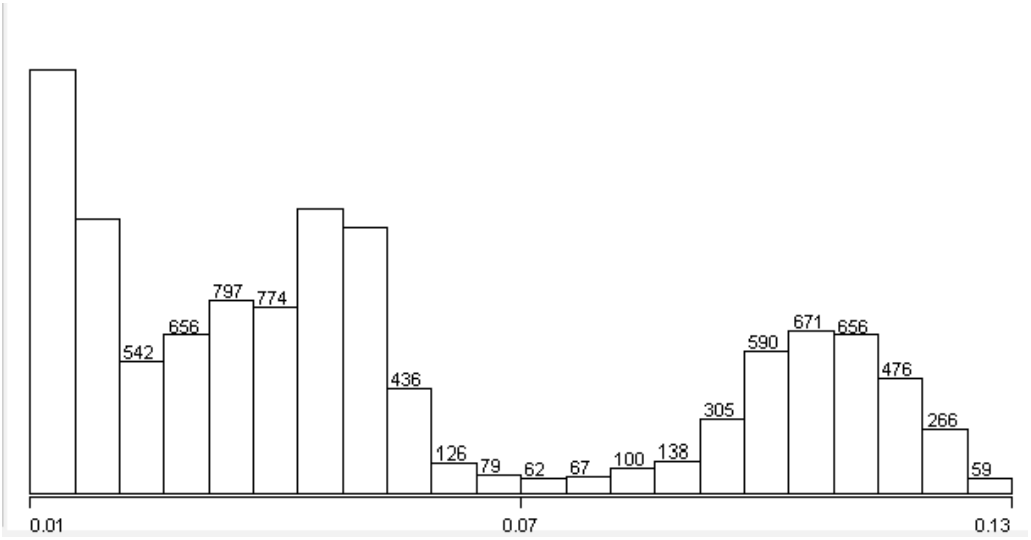


Figure 95: Graphical representation of value ranges from 0.006 to 0.134

7. Implementation of Neural Network and Classification Algorithm

7.1. Implementation of Neural Network (60% training and 40% testing)

After running the data on Neural Network with a split of 60% training and 40% testing we received an output that it took a total of 1200.65 seconds (approx. 20 minutes) to build the model and 0.46 seconds to test the model on the test split.

Classifier output

```
=== Run information ===

Scheme:       weka.classifiers.functions.MultilayerPerceptron -L 0.3 -M 0.2 -N 500 -V 0 -S 0 -E 20 -H a
Relation:     WinnipegDataset
Instances:    12000
Attributes:   175
              [list of attributes omitted]
Test mode:    split 60.0% train, remainder test
```

Figure 96: Classifier output

The summary for this test indicates that the Correctly Classified Instances i.e., 4800 were classified correctly by the neural network model, which corresponds to a 100% accuracy rate.

```
Time taken to build model: 1200.65 seconds

=== Evaluation on test split ===

Time taken to test model on test split: 0.46 seconds

=== Summary ===

Correctly Classified Instances        4800               100      %
Incorrectly Classified Instances         0                 0      %
Kappa statistic                          1
Mean absolute error                      0.0002
Root mean squared error                  0.005
Relative absolute error                  0.0582 %
Root relative squared error              1.1784 %
Total Number of Instances             4800

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 1.000    0.000    1.000      1.000    1.000      1.000    1.000     1.000     Corn
                 1.000    0.000    1.000      1.000    1.000      1.000    1.000     1.000     Peas
                 1.000    0.000    1.000      1.000    1.000      1.000    1.000     1.000     Canola
                 1.000    0.000    1.000      1.000    1.000      1.000    1.000     1.000     Soyabeans
Weighted Avg.    1.000    0.000    1.000      1.000    1.000      1.000    1.000     1.000

=== Confusion Matrix ===

    a    b    c    d   <-- classified as
 1619    0    0    0 |    a = Corn
    0 1464    0    0 |    b = Peas
    0    0  960    0 |    c = Canola
    0    0    0  757 |    d = Soyabeans
```

Figure 97: Accuracy output

The Incorrectly Classified Instances have a value of 0. This represents the number of instances that were misclassified by the neural network model during the evaluation, which means the model achieved a perfect classification performance without making any misclassifications.

The Kappa statistic is a measure of inter-rated agreement that quantifies the agreement between the predicted class labels and the actual labels in a classification task. In this output, the Kappa statistic value is 1, which means that it signifies a perfect agreement between the predicted class labels and the actual class labels.

The Mean Absolute Error (MAE) has a value of 0.0002. This measures the average absolute difference between the predicted values and the actual values of the target variable. Having the value of 0.0002 indicates that the neural network model's predictions are very close to the actual values of the target variable.

The Root Mean Squared Error or RMSE measures the square root of the average of the squared differences between the predicted values and the actual values of the target variable. The value 0.005 indicates the neural network model's predictions have a small error when compared to the actual values of the target variable.

The Relative Absolute Error is reported to be 0.0582% meaning that the neural network model's prediction has an average absolute error that is approximately 0.0582% of the mean absolute error of the actual values. A lower Relative Absolute Error indicates a better performance, as it signifies a smaller relative difference between the predicted and the actual values.

Root Relative Squared Error, also known as RRSE measures the square root of the average of the squared differences between the predicted values and the actual values of the target variable, normalized by the squared mean absolute error of the actual values. Overall, the value 1.1784% signifies that the neural network model has a reasonably accurate level of performance in its regression predictions.

Neural Network		
Correctly Classified Instances	4800	100% Accuracy
Incorrectly Classified Instances	0	0
Kappa statistic	1	
Mean Absolute Error	0.0002	
Root Mean Squared Error	0.005	
Relative Absolute Error	0.0582%	
Root Relative Squared Error	1.1784%	
Total Number of Instances	4800	

Table 1: Summery in the table

In the end, the Total Number of Instances i.e., 4800 gives an understanding of the sample size on which the model's performance is based.

Detailed Accuracy by Class – This table shows the performance measures for each class in the classification task.

```
=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 1.000    0.000    1.000      1.000    1.000      1.000    1.000     1.000     Corn
                 1.000    0.000    1.000      1.000    1.000      1.000    1.000     1.000     Peas
                 1.000    0.000    1.000      1.000    1.000      1.000    1.000     1.000     Canola
                 1.000    0.000    1.000      1.000    1.000      1.000    1.000     1.000     Soyabeans
Weighted Avg.    1.000    0.000    1.000      1.000    1.000      1.000    1.000     1.000
```

Figure 98: Classification report 1

In this case, the classes are Corn, Peas, Canola, and Soyabean. And the column represents the following:

TP Rate (True Positive Rate): This indicates the instances that are correctly classified in each class. A value of 1.000 means that all the instances belonging to that class were correctly classified.

FP Rate (False Positive Rate): This indicates the instances that are falsely classified in each class. A value of 0.000 indicates that there were no instances that were misclassified as belonging to the current class.

Precision: Precision is the number of correct positive predictions (TP) divided by the sum of true positives and false positives. A precision of 1.000 means that all positive predictions for the class were correct.

Recall: It is the ability of a model to find all the relevant cases within a dataset. It measures the proportion of actual positives that are correctly identified. A recall of 1.000 means that all instances of the class were correctly identified as positive.

F-Measure: It combines precision and recall into a single metric. The formula to calculate this is:

$$\text{Precision} = t\_p / (t\_p + f\_p)$$

$$\text{Recall} = t\_p / (t\_p + f\_n)$$

$$\text{F-Score} = 2 * \text{Precision} * \text{Recall} / (\text{Precision} + \text{Recall})$$

t_p is the number of True Positives, f_p is the number of false positives and f_n is the number of false negatives. A value of 1.000 means that the classifier achieved perfect precision and recall.

MCC (Matthews's correlation Coefficient): It is a correlation between predicted and observed classifications. A value of 1.000 represents a perfect prediction.

ROC Area (Receiver Operating Characteristic): ROC curve is a graphical plot that measures the usefulness of a test in general. A value of 1.000 indicates perfect performance.

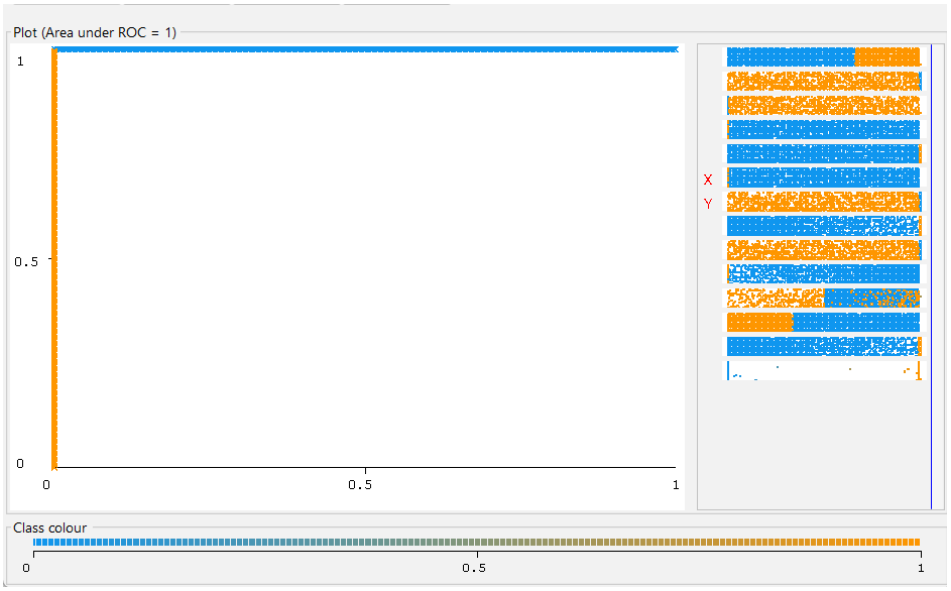


Figure 99: ROC curve

PRC Area: PRC (Precision-Recall Curve) area is the curve that was obtained by combining precision (PPV) and sensitivity (TPR). A value of 1.000 indicates perfect performance.

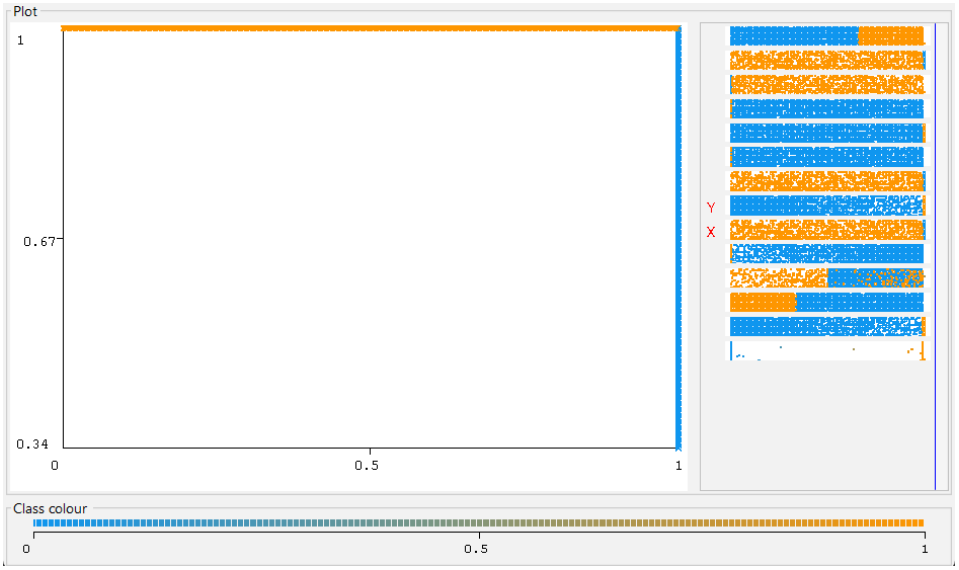


Figure 100: PRC Curve

Confusion Matrix: This table shows the result of the neural network classification model that was applied to the dataset with four classes: Corn, Peas, Canola, and Soybeans.

```
=== Confusion Matrix ===

    a    b    c    d   <-- classified as
 1619    0    0    0 |    a = Corn
    0 1464    0    0 |    b = Peas
    0    0  960    0 |    c = Canola
    0    0    0  757 |    d = Soyabeans
```

Figure 101: Confusion matrix report 1

From the above picture:

- Column (a) of the first row indicates that all 1619 instances were correctly classified as corn. There were no instances that were misclassified as other classes.
- Column (b) of the second row represents all the 1464 instances that were correctly classified as peas. There were no instances that were misclassified as other classes.
- Column (c) of the third row indicates all the 960 instances that were correctly classified as canola. There were no instances that were misclassified as other classes.
- The fourth column (d) of the fourth row shows that all the 757 instances were correctly classified as soybeans. There were no instances that were misclassified as other classes.

To conclude, the confusion matrix demonstrates that the neural network classification model, when implemented to the dataset, has flawless classification accuracy.

7.2. Implementing Classification method (i.e., J48 using 60% training and 40% testing)

Now, after implementing another classification method i.e., J48 to the same dataset with the same split of 60% Training and 40% Testing, to compare their performance, we got the following results:

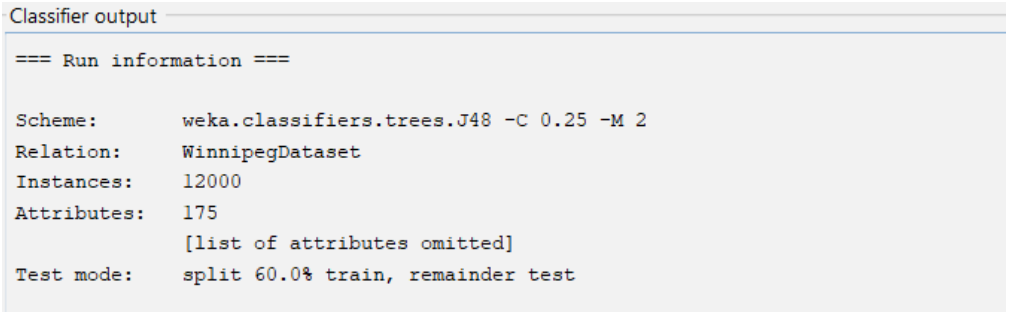


Figure 102: Data splitting

It took 1.69 seconds to build the model and 0.05 seconds to test the model on test split which is significantly faster compared to the neural network which took 1200.65 seconds to build and 0.46 seconds to test.

```
Time taken to build model: 1.69 seconds

=== Evaluation on test split ===

Time taken to test model on test split: 0.05 seconds

=== Summary ===

Correctly Classified Instances        4781               99.6042 %
Incorrectly Classified Instances        19                0.3958 %
Kappa statistic                          0.9946
Mean absolute error                      0.0021
Root mean squared error                  0.0442
Relative absolute error                  0.5779 %
Root relative squared error             10.3462 %
Total Number of Instances             4800

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 0.998    0.002    0.997      0.998    0.998      0.996    0.998     0.996     Corn
                 0.999    0.003    0.993      0.999    0.996      0.994    0.998     0.992     Peas
                 0.995    0.001    0.998      0.995    0.996      0.995    0.997     0.994     Canola
                 0.988    0.000    0.999      0.988    0.993      0.992    0.995     0.989     Soyabeans
Weighted Avg.    0.996    0.002    0.996      0.996    0.996      0.995    0.997     0.993

=== Confusion Matrix ===

    a    b    c    d   <-- classified as
 1616    3    0    0 |    a = Corn
    1 1462    0    1 |    b = Peas
    3    2  955    0 |    c = Canola
    1    6    2  748 |    d = Soyabeans
```

Figure 103: Accuracy output

The Summary for this test indicates that the Correctly Classified Instances i.e., 4781 were classified correctly out of 4800 achieving an accuracy rate of 99.6042%, whereas the neural network test indicated an accuracy rate of 100% meaning that it may be even more effective for this task.

The Incorrectly Classified Instances have a value of 19. This represents the number of instances that were misclassified by the J48 model during the evaluation, which means out of 4800 instances 19 instances were classified incorrectly by the model.

The Kappa Statistic measures the inter-rated agreement that quantifies the agreement between the predicted class labels and the actual labels in a classification task. We received a value of 0.9946 which indicates a high level of agreement however, it is still lower than the value obtained by the kappa statistic implemented by the neural network.

The Mean Absolute Error (MAE) has a value of 0.0021. This measures the average absolute difference between the predicted values and the actual values of the target variable. Compared to this neural network has a better result with 0.0002.

J48		
Correctly Classified Instances	4781	99.6042% Accuracy
Incorrectly Classified Instances	19	0.3958
Kappa statistic	0.9946	
Mean Absolute Error	0.0021	
Root Mean Squared Error	0.0442	
Relative Absolute Error	0.5779%	
Root Relative Squared Error	10.3462%	
Total Number of Instances	4800	

Table 2: J48 result table

The Root Mean Square Error or RMSE measures the square root of the average differences between the predicted values and the actual values of the target variable. The value 0.0442 indicates a smaller value, however, compared to the neural network, it has a better performance result with 0.005 meaning that the values are closer to the actual values.

The Relative Absolute Error is reported to be 0.5779% meaning that on average the J48 classifier's predictions differ from the actual values by 0.5779% of the average actual value. A lower Relative Absolute Error indicates better performance, as it signifies a smaller relative difference between the predicted and the actual values. Therefore, a Relative Absolute Error of 0.0582% in the case of neural network implies a better performance than the J48 classifier, which had a higher error rate of 0.5779%.

Root Relative Squared Error, also known as RRSE measures the square root of the average of squared differences between the predicted values and the actual values of the target variable, normalized by the squared mean absolute error of the actual values. The value 10.3462% signifies that the J48 classifier's predicted values differ from the actual values by 10.3462% of the range of the target variable. Whereas the neural network had a value of only 1.1784% and a higher Root Relative Squared Error indicates worse performance.

In the end, the Total Number of Instances i.e., 4781 shows the sample size on which the model's performance is based.

```
=== Detailed Accuracy By Class ===

                TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                0.998    0.002    0.997      0.998    0.998      0.996    0.998     0.996     Corn
                0.999    0.003    0.993      0.999    0.996      0.994    0.998     0.992     Peas
                0.995    0.001    0.998      0.995    0.996      0.995    0.997     0.994     Canola
                0.988    0.000    0.999      0.988    0.993      0.992    0.995     0.989     Soyabeans
Weighted Avg.   0.996    0.002    0.996      0.996    0.996      0.995    0.997     0.993
```

Figure 104: Classification report 2

TP Rate (True Positive Rate): This indicates the instances that are correctly classified in each class. A value of 0.998 indicates that the J48 model is good at correctly identifying positive instances. However, the neural network has a slightly better performance with a perfect TP Rate of 1.000.

FP Rate (False Positive Rate): This indicates the instances that are falsely classified in each class. A value of 0.002 indicates that only 0.2% of the negative instances were incorrectly predicted as positive. However, the Neural Network had an even lower FP Rate of 0.000, meaning it made even fewer false positive predictions than J48.

Precision: Precision is the number of correct positive predictions (TP) divided by the sum of true positives and false positives. A value of 0.997 means that out of all the positive predictions made, 99.7% were true positives. On the other hand, a precision value of 1.000 in the neural network model indicates that all positive predictions made by the model were true positives.

Recall: It is the ability of a model to find all the relevant cases within a dataset. It measures the proportion of actual positives that are correctly identified. A recall of 0.998 means that 99.8% of the positive instances in the dataset were correctly identified by the J48 classifier. Whereas, Neural Network had a recall of 100%.

F-Measure: It combines precision and recall into a single metric. The formula to calculate this is

$$\text{Precision} = t\_p / (t\_p + f\_p)$$

$$\text{Recall} = t\_p / (t\_p + f\_n)$$

$$\text{F-Score} = 2 * \text{Precision} * \text{Recall} / (\text{Precision} + \text{Recall})$$

t_p is the number of True Positives, f_p is the number of false positives and f_n is the number of false negatives. A value of 0.998 indicates the classifier has achieved high precision and recall, with a relatively low number of false positives and false negatives. However, the Neural network had a value of 1.000 which indicates that the classifier has achieved perfect precision and recall.

MCC (Matthews Correlation Coefficient): It is a correlation between predicted and observed classifications. A value of 0.996 indicates a high level of agreement between the predicted

and actual labels. However, compared to Neural Network value of 1.000, the J48 model is slightly less accurate.

ROC Area (Receiver Operation Characteristic): ROC Curve is a graphical plot that measures the usefulness of a test in general. A value of 0.998 indicates that it has a high true positive rate with a low false positive rate. However, the Neural Network had a perfect ROC value of 1.000, indicating an excellent performance in correctly classifying the positive and negative instances.

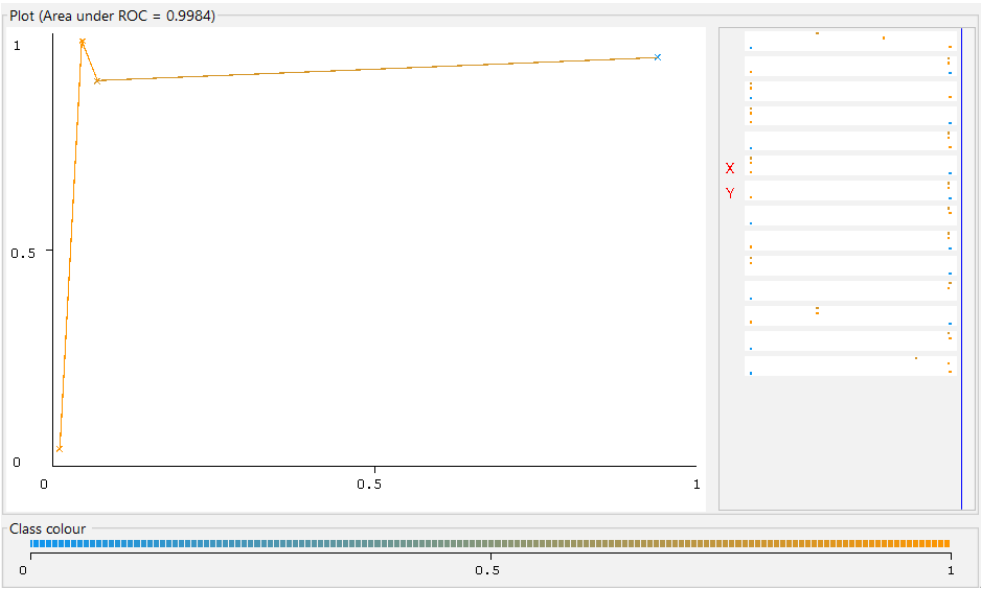


Figure 105: ROC Curve 2

PRC Area: PRC (Precision-Recall Curve) area is the curve that was obtained by combining precision (PPV) and sensitivity (TPR). A value of 0.996 indicates a high level of precision and recall for the classifier's predictions, but it is slightly lower than the perfect value of 1.000 achieved by Neural Network.

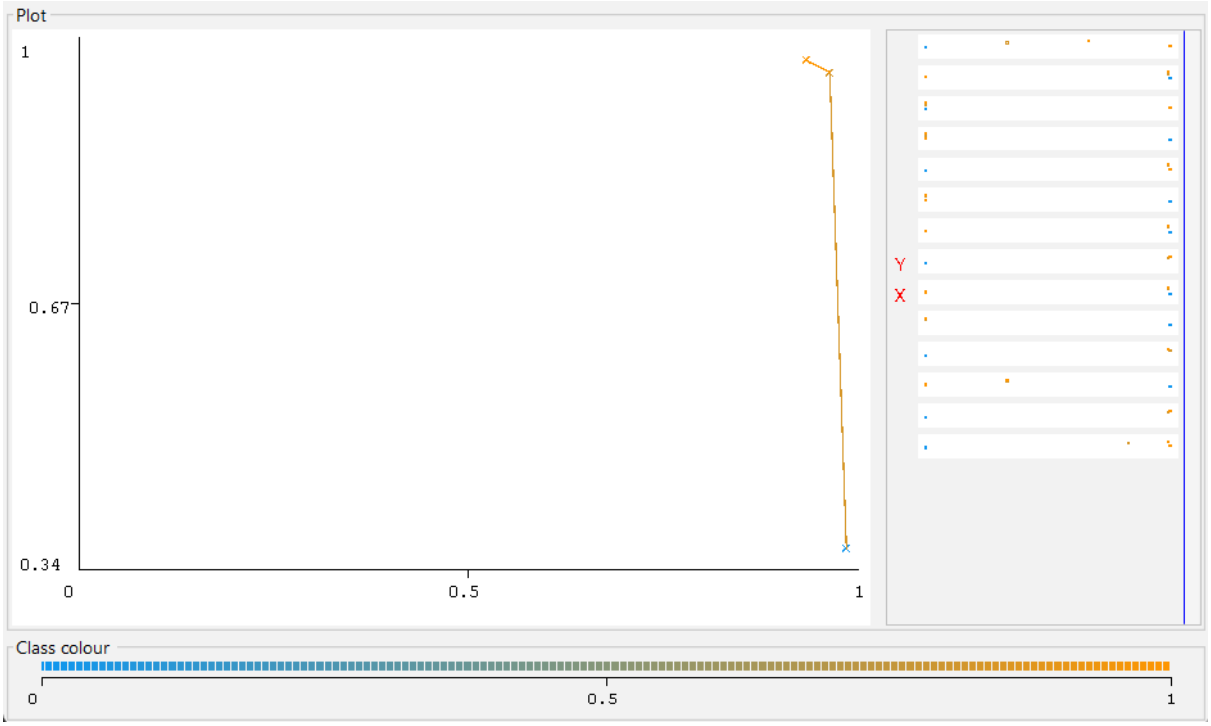


Figure 106: Precision-Recall Curve 2

Confusion Matrix: This table shows the result of the neural network classification model that was applied to the dataset with four classes – Corn, Peas, Canola, and Soybeans.

```
=== Confusion Matrix ===

    a    b    c    d   <-- classified as
 1616    3    0    0 |    a = Corn
    1 1462    0    1 |    b = Peas
    3    2  955    0 |    c = Canola
    1    6    2  748 |    d = Soyabeans
```

Figure 107: Confusion matrix report 2

- Column (a) of the first row indicates that all 1616 instances were correctly classified as corn. There were 3 instances that were misclassified as other classes.
- Column (b) of the second row represents all the 1462 instances that were correctly classified as peas. There were 2 instances that were misclassified as other classes.

- Column (c) of the third row indicates all the 955 instances that were correctly classified as canola. There were 5 instances that were misclassified as other classes.
- The fourth column (d) of the fourth row shows that all the 748 instances were correctly classified as soybeans. There were 9 instances that were misclassified as other classes.

	J48 with 60% Training & 40% Testing	**Neural Network with 60% Training & 40% Testing**
Correctly Classified Instances	4781	4800
Incorrectly Classified Instances	19	0
Kappa statistic	0.9946	1
Mean Absolute Error	0.0021	0.0002
Root Mean Squared Error	0.0442	0.005
Relative Absolute Error	0.5779%	0.0582%
Root Relative Squared Error	10.3462%	1.1784%
Total Number of Instances	4800	4800

Table 3: Comparison classifier 60/40 with nn 60/40

To conclude, the confusion matrix suggests that the classifier performance was well, however, when compared with the confusion matrix of Neural Network, Neural Network is a better classifier than J48.

7.3. Implementation of Neural Network (75% training and 25% testing)

After running the data on Neural Network with a split of 75% training and 25% testing we received an output that it took a total of 1537.65 seconds to build the model and 0.69 seconds to test the model on the test split.

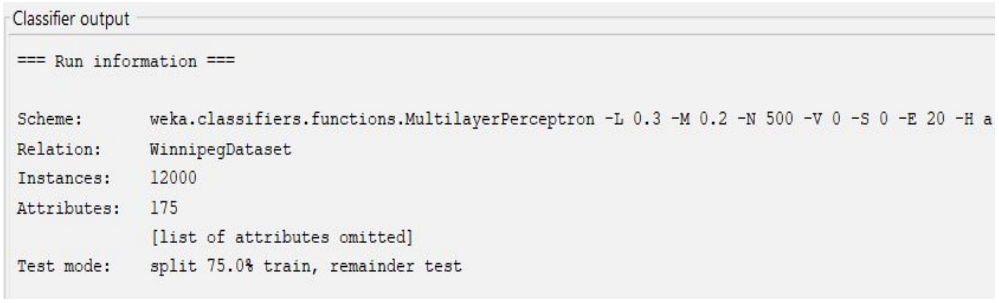


Figure 108: Classifier report

The summary for this test indicates that the Correctly Classified Instances i.e., 3000 were classified correctly by the neural network model, which corresponds to a 100% accuracy rate.

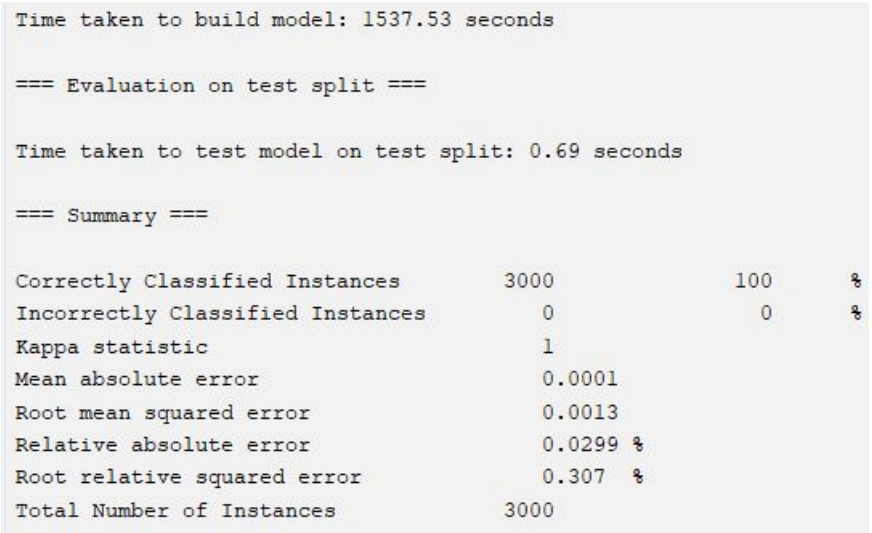


Figure 109: Accuracy report from nn

	Neural Network with 75% training and 25% testing	Neural Network with 60% training and 40% testing
Correctly Classified Instances	3000	4800
Incorrectly Classified Instances	0	0
Kappa statistic	1	1
Mean Absolute Error	0.0001	0.0002
Root Mean Squared Error	0.0013	0.005
Relative Absolute Error	0.0299%	0.0582%
Root Relative Squared Error	0.307 %	1.1784%
Total Number of Instances	3000	4800

Table 4: Comparison of nn between 75/25 and 60/40

Detailed accuracy by class: This table shows the performance measures for each class in the classification task.

```
=== Detailed Accuracy By Class ===

                TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                1.000    0.000    1.000      1.000    1.000      1.000    1.000     1.000     Corn
                1.000    0.000    1.000      1.000    1.000      1.000    1.000     1.000     Peas
                1.000    0.000    1.000      1.000    1.000      1.000    1.000     1.000     Canola
                1.000    0.000    1.000      1.000    1.000      1.000    1.000     1.000     Soyabeans
Weighted Avg.   1.000    0.000    1.000      1.000    1.000      1.000    1.000     1.000
```

Figure 110: Classification report for nn

Confusion Matrix: This table shows the result of the neural network classification model that was applied to the dataset with four classes: Corn, Peas, Canola, and Soybeans.

```
=== Confusion Matrix ===

    a    b    c    d   <-- classified as
 1036    0    0    0 |    a = Corn
    0  896    0    0 |    b = Peas
    0    0  600    0 |    c = Canola
    0    0    0  468 |    d = Soyabeans
```

Figure 111: Confusion matrix report for nn

Overall, after comparing the results of the Neural Network using a training split of 75% and 25% testing, it can be concluded that this configuration outperforms the Neural Network with a training split of 60% and 40% testing across all the metrics.

7.4. Implementing Classification method (i.e., J48 using 75% training and 25% testing)

Now, after implementing another classification method i.e., J48 to the same dataset with the same split of 75% Training and 25% Testing, to compare their performance, we got the following results:

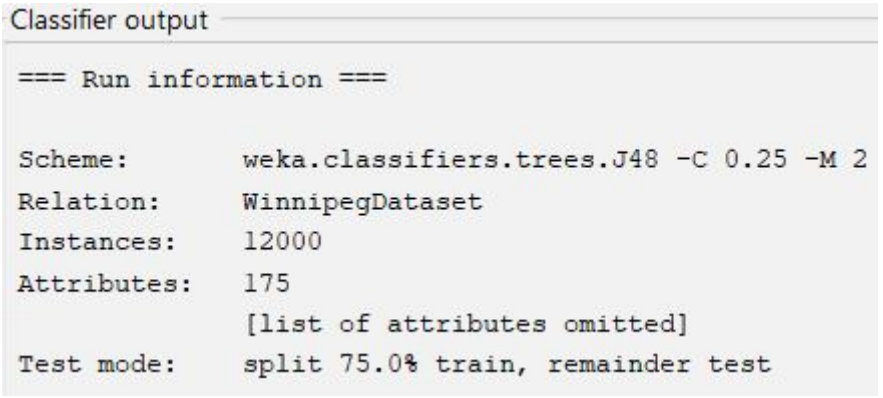


Figure 112: Classification output

It took 1.25 seconds to build the model and 0 seconds to test the model on test split which is significantly faster compared to the neural network which took 1200.65 seconds to build and 0.46 seconds to test.

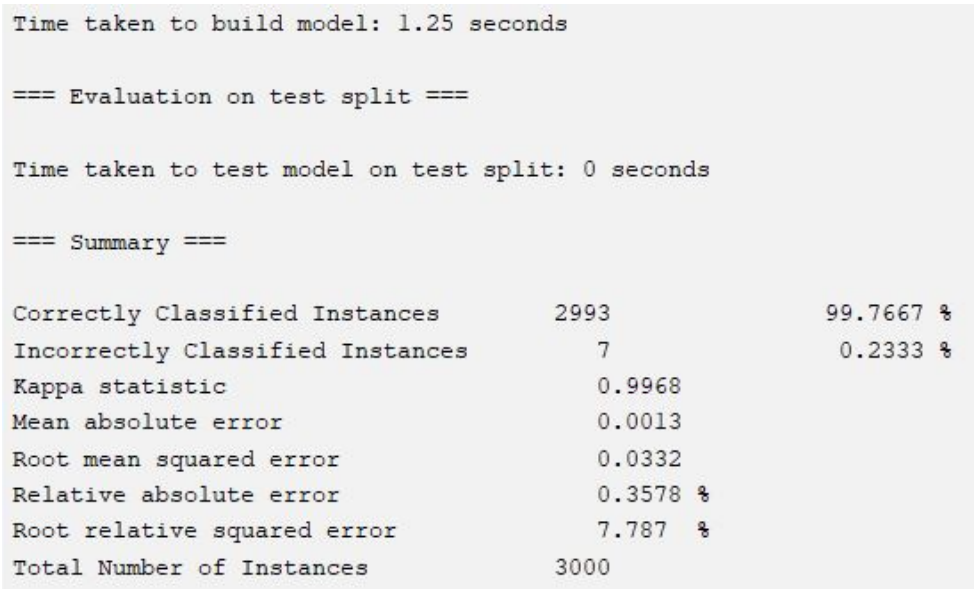


Figure 113: Accuracy report

	J48 with 75% training and 25% testing	**J48 with 60% training and 40% testing**
Correctly Classified Instances	2993	4781
Incorrectly Classified Instances	7	19
Kappa statistic	0.9968	0.9946
Mean Absolute Error	0.0013	0.0021
Root Mean Squared Error	0.0332	0.0442
Relative Absolute Error	0.3578%	0.5779%
Root Relative Squared Error	7.787%	10.3462%
Total Number of Instances	3000	4800

Table 5: Comparison of J48 between 75/25 and 60/40

Detailed accuracy by class: This table shows the performance measures for each class in the classification task.

```
=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 1.000    0.000    1.000      1.000    1.000      1.000    1.000     1.000     Corn
                 1.000    0.000    1.000      1.000    1.000      1.000    1.000     1.000     Peas
                 1.000    0.000    1.000      1.000    1.000      1.000    1.000     1.000     Canola
                 1.000    0.000    1.000      1.000    1.000      1.000    1.000     1.000     Soyabeans
Weighted Avg.    1.000    0.000    1.000      1.000    1.000      1.000    1.000     1.000
```

Figure 114: Classification report

Confusion Matrix: This table shows the result of the classification model that was applied to the dataset with four classes: Corn, Peas, Canola, and Soybeans.

```
=== Confusion Matrix ===

   a    b    c    d   <-- classified as
1036    0    0    0 |    a = Corn
   0  896    0    0 |    b = Peas
   0    0  600    0 |    c = Canola
   0    0    0  468 |    d = Soyabeans
```

Figure 115: Confusion matrix

Overall, after comparing the results of the J48 using a training split of 75% and 25% testing, it can be concluded that this configuration outperforms the J48 with a training split of 60% and 40% testing across all the metrics. However, even after getting better results, the J48 classification still lags behind the Neural Network in terms of performance.

8. Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a way of identifying patterns in data and expressing the data in such a way as to highlight their similarities and differences.

Attribute selection output

```
=== Run information ===

Evaluator:     weka.attributeSelection.PrincipalComponents -R 0.95 -A 5
Search:        weka.attributeSelection.Ranker -T -1.7976931348623157E308 -N -1
Relation:      WinnipegDataset
Instances:     12000
Attributes:    175
               [list of attributes omitted]
Evaluation mode:    evaluate on all training data
```

Figure 116: Attribute selection in PCA

After performing the PCA algorithm in Weka we got the following output: -

Correlation Matrix: The output depicts the correlation matrix with each cell representing the correlation between the two variables. The diagonal values are all 1, as each variable is perfectly correlated with themselves, and the off-diagonal values show the correlation between pairs of variables.

Correlation matrix

1	0.95	0.81	0.91	0.96	0.93	0.21	0.56	0.63	0.26	0.27	0.41	0.13	0.64	-0.3	-0.2	0.35	-0.45	-0.03	-0.06	-0.36	-0.16	-0.54	0	0.82	0.83	0.69
0.95	1	0.87	0.94	0.96	0.95	0.03	0.79	0.64	0.35	0.19	0.36	-0.08	0.79	-0.15	-0.12	0.29	-0.42	-0.19	-0.24	-0.5	-0.26	-0.68	-0.18	0.82	0.91	0.83
0.81	0.87	1	0.96	0.93	0.95	-0.39	0.72	0.19	0.54	-0.01	0.24	-0.46	0.48	0.3	0.1	0.13	-0.34	0.05	-0	-0.19	0.02	-0.35	0.02	0.95	0.79	0.62
0.91	0.94	0.96	1	0.95	0.99	-0.17	0.69	0.38	0.51	-0.04	0.19	-0.25	0.57	0.08	0.13	0.08	-0.29	0.05	-0.01	-0.25	-0	-0.41	0.04	0.93	0.83	0.66
0.96	0.96	0.93	0.95	1	0.96	-0.03	0.67	0.48	0.39	0.27	0.46	-0.12	0.61	-0.07	-0.17	0.38	-0.52	-0.03	-0.07	-0.34	-0.12	-0.52	-0.02	0.91	0.85	0.71
0.93	0.95	0.95	0.99	0.96	1	-0.11	0.68	0.42	0.4	0.01	0.19	-0.2	0.59	0.02	0.06	0.11	-0.27	0.04	-0.02	-0.26	-0.03	-0.44	0.03	0.92	0.84	0.68
0.21	0.03	-0.39	-0.17	-0.03	-0.11	1	-0.33	0.68	-0.49	0.44	0.24	0.99	0.22	-0.99	-0.49	0.35	-0.15	-0.13	-0.1	-0.25	-0.29	-0.28	-0.03	-0.3	-0.02	0.05
0.56	0.79	0.72	0.69	0.67	0.68	-0.33	1	0.47	0.4	-0.02	0.17	-0.45	0.84	0.19	0.08	0.08	-0.23	-0.43	-0.5	-0.61	-0.38	-0.72	-0.47	0.58	0.78	0.84
0.63	0.64	0.19	0.38	0.48	0.42	0.68	0.47	1	-0.15	0.4	0.35	0.58	0.85	-0.78	-0.4	0.39	-0.32	-0.45	-0.47	-0.7	-0.57	-0.82	-0.4	0.17	0.58	0.7
0.26	0.35	0.54	0.51	0.39	0.4	-0.49	0.4	-0.15	1	-0.34	0.1	-0.51	0.1	0.45	0.5	-0.13	-0.28	0.14	0.1	-0.01	0.21	-0	0.07	0.46	0.27	0.18
0.27	0.19	-0.01	-0.04	0.27	0.01	0.44	-0.02	0.4	-0.34	1	0.9	0.43	0.23	-0.46	-0.98	0.97	-0.8	-0.26	-0.2	-0.35	-0.41	-0.42	-0.17	0.03	0.17	0.25
0.41	0.36	0.24	0.19	0.46	0.19	0.24	0.17	0.35	0.1	0.9	1	0.22	0.29	-0.28	-0.81	0.97	-0.98	-0.21	-0.16	-0.37	-0.34	-0.45	-0.15	0.25	0.31	0.34
0.13	-0.08	-0.46	-0.25	-0.12	-0.2	0.99	-0.45	0.58	-0.51	0.43	0.22	1	0.08	-0.96	-0.48	0.33	-0.12	-0.05	-0	-0.14	-0.21	-0.15	0.05	-0.36	-0.13	-0.07
0.64	0.79	0.48	0.57	0.61	0.59	0.22	0.84	0.85	0.1	0.23	0.29	0.08	1	-0.36	-0.2	0.27	-0.29	-0.61	-0.65	-0.83	-0.63	-0.94	-0.61	0.38	0.8	0.93
-0.3	-0.15	0.3	0.08	-0.07	0.02	-0.99	0.19	-0.78	0.45	-0.46	-0.28	-0.96	-0.36	1	0.5	-0.38	0.2	0.22	0.19	0.36	0.38	0.41	0.12	0.23	-0.1	-0.19
-0.2	-0.12	0.1	0.13	-0.17	0.06	-0.49	0.08	-0.4	0.5	-0.98	-0.81	-0.48	-0.2	0.5	1	-0.92	0.69	0.28	0.22	0.33	0.43	0.4	0.18	0.05	-0.12	-0.21
0.35	0.29	0.13	0.08	0.38	0.11	0.35	0.08	0.39	-0.13	0.97	0.97	0.33	0.27	-0.38	-0.92	1	-0.92	-0.24	-0.19	-0.37	-0.38	-0.45	-0.17	0.15	0.25	0.31
-0.45	-0.42	-0.34	-0.29	-0.52	-0.27	-0.15	-0.23	-0.32	-0.28	-0.8	-0.98	-0.12	-0.29	0.2	0.69	-0.92	1	0.17	0.13	0.36	0.28	0.43	0.12	-0.33	-0.35	-0.36
-0.03	-0.19	0.05	0.05	-0.03	0.04	-0.13	-0.43	-0.45	0.14	-0.26	-0.21	-0.05	-0.61	0.22	0.28	-0.24	0.17	1	0.8	0.79	0.9	0.75	0.95	0.12	-0.46	-0.63
-0.06	-0.24	-0	-0.01	-0.07	-0.02	-0.1	-0.5	-0.47	0.1	-0.2	-0.16	-0	-0.65	0.19	0.22	-0.19	0.13	0.8	1	0.81	0.7	0.76	0.84	0.09	-0.45	-0.65
-0.36	-0.5	-0.19	-0.25	-0.34	-0.26	-0.25	-0.61	-0.7	-0.01	-0.35	-0.37	-0.14	-0.83	0.36	0.33	-0.37	0.36	0.79	0.81	1	0.77	0.92	0.79	-0.12	-0.64	-0.82
-0.16	-0.26	0.02	-0	-0.12	-0.03	-0.29	-0.38	-0.57	0.21	-0.41	-0.34	-0.21	-0.63	0.38	0.43	-0.38	0.28	0.9	0.7	0.77	1	0.77	0.77	0.08	-0.48	-0.64
-0.54	-0.68	-0.35	-0.41	-0.52	-0.44	-0.28	-0.72	-0.82	-0	-0.42	-0.45	-0.15	-0.94	0.41	0.4	-0.45	0.43	0.75	0.76	0.92	0.77	1	0.73	-0.27	-0.76	-0.92
0	-0.18	0.02	0.04	-0.02	0.03	-0.03	-0.47	-0.4	0.07	-0.17	-0.15	0.05	-0.61	0.12	0.18	-0.17	0.12	0.95	0.84	0.79	0.77	0.73	1	0.11	-0.45	-0.63
0.82	0.82	0.95	0.93	0.91	0.92	-0.3	0.58	0.17	0.46	0.03	0.25	-0.36	0.38	0.23	0.05	0.15	-0.33	0.12	0.09	-0.12	0.08	-0.27	0.11	1	0.75	0.55
0.83	0.91	0.79	0.83	0.85	0.84	-0.02	0.78	0.58	0.27	0.17	0.31	-0.13	0.8	-0.1	-0.12	0.25	-0.35	-0.46	-0.45	-0.64	-0.48	-0.76	-0.45	0.75	1	0.93
0.69	0.83	0.62	0.66	0.71	0.68	0.05	0.84	0.7	0.18	0.25	0.34	-0.07	0.93	-0.19	-0.21	0.31	-0.36	-0.63	-0.65	-0.82	-0.64	-0.92	-0.63	0.55	0.93	1
0.17	0.3	-0.06	0.02	0.09	0.05	0.37	0.46	0.71	-0.21	0.23	0.15	0.27	0.77	-0.47	-0.26	0.2	-0.1	-0.87	-0.85	-0.86	-0.83	-0.84	-0.85	-0.16	0.51	0.7
-0.73	-0.84	-0.53	-0.62	-0.7	-0.64	-0.27	-0.79	-0.87	-0.12	-0.34	-0.42	-0.14	-0.96	0.4	0.3	-0.4	0.43	0.48	0.57	0.79	0.54	0.92	0.47	-0.47	-0.79	-0.9
-0.36	-0.31	-0.15	-0.1	-0.39	-0.12	-0.31	-0.13	-0.39	0.11	-0.96	-0.97	-0.28	-0.31	0.35	0.9	-0.99	0.91	0.33	0.25	0.43	0.46	0.5	0.25	-0.16	-0.3	-0.37

Figure 117: Correlation matrix

Eigenvalue: After the correlation Matrix is calculated, Weka then calculates the Eigenvalue, Eigenvectors based on the correlation matrix. Eigenvalues are closely related to Eigenvectors; it is a measure of how much information is captured by each principal component and they are usually arranged in descending order.

eigenvalue	proportion	cumulative	
62.46585	0.359	0.359	-0.119f27-0.118f45-0.118f49-0.116f26-0.116f143...
34.95957	0.20092	0.55992	0.156f92+0.156f95+0.155f96+0.151f93+0.145f74...
16.84526	0.09681	0.65673	0.209f64-0.205f56-0.196f62+0.193f79-0.187f83...
11.05064	0.06351	0.72024	0.171f69+0.168f133+0.163f125+0.157f163-0.156f131...
5.80147	0.03334	0.75358	-0.237f12+0.237f30+0.235f18-0.227f17-0.206f11...
4.85861	0.02792	0.7815	0.189f139-0.189f146+0.158f115+0.156f101+0.15 f167...
4.33437	0.02491	0.80641	0.25 f171-0.246f169-0.245f174-0.243f173+0.241f172...
3.37672	0.01941	0.82582	0.322f135+0.31 f136-0.31f134+0.304f131-0.287f133...
3.07872	0.01769	0.84351	0.222f118+0.217f162+0.216f160+0.205f122+0.203f124...
2.37235	0.01363	0.85715	-0.26f42-0.251f48+0.207f16+0.202f10+0.185f61...
2.20165	0.01265	0.8698	0.261f97+0.181f91-0.181f10+0.179f118-0.172f120...
1.93678	0.01113	0.88093	0.325f165+0.31 f127-0.3f164-0.288f126+0.205f160...
1.7167	0.00987	0.8908	-0.387f130-0.354f132-0.28f168-0.247f136-0.213f170...
1.44611	0.00831	0.89911	0.307f127-0.278f126+0.203f123-0.192f166+0.163f100...
1.43537	0.00825	0.90736	-0.438f166-0.319f165+0.289f127-0.273f126+0.267f164...
1.39026	0.00799	0.91535	-0.469f48-0.462f42-0.143f18+0.136f13+0.132f7...
1.10743	0.00636	0.92171	0.8 f128+0.438f166+0.156f127+0.14 f164-0.108f165...
1.00503	0.00578	0.92749	-0.333f91-0.31f10-0.252f71-0.242f34-0.242f118...
0.9482	0.00545	0.93294	0.497f97-0.323f44+0.285f91+0.194f71-0.179f47...
0.884	0.00508	0.93802	0.392f48+0.326f97+0.253f44-0.235f43+0.187f10...
0.68646	0.00395	0.94196	-0.242f97+0.235f48+0.211f114+0.183f70+0.183f118...
0.6499	0.00374	0.9457	0.557f166-0.402f128-0.283f97+0.23 f71+0.215f164...
0.63297	0.00364	0.94934	0.367f128-0.361f166+0.274f99+0.274f71+0.226f115...
0.53991	0.0031	0.95244	-0.247f70-0.222f59-0.198f42+0.18 f43+0.176f130...

Figure 118: Eigenvalue

Here, the first eigenvalue is 62.46585, which means that it has the highest value i.e., 35.9% of the variance in the data. After that second eigenvalue is 34.95957, which indicates that it has the second highest value i.e., 20.092% of the variance in the data, and so on.

Eigenvectors: Eigenvectors indicate the direction of the principal components that have the largest variance. The Eigenvector for the 175 primary components is shown in the table. Each row in the table represents an eigenvector, and each column represents a variable. The contributions of each variable to the corresponding principal component are shown by the values in each column.

Eigenvectors

V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V15	V16	V17	V18	V19	V20	V21	V22	V23	V24	
-0.102	0.0629	-0.074	0.057	0.062	0.0588	-0.0373	0.0202	0.0075	-0.0519	-0.0009	-0.0259	-0.0358	0.0607	-0.0027	0.0265	0.008	-0.0035	-0.0779	0.0259	-0.0528	0.0267	0.0278	0.0718	f1
-0.1123	0.0581	-0.0168	0.0532	0.0361	0.0845	-0.0142	0.0061	0.0149	-0.014	0.0068	-0.0035	0.0097	-0.0111	0.0153	-0.0103	0.0043	-0.033	-0.0316	0.0015	-0.0586	0.0281	0.034	0.0054	f2
-0.0837	0.1182	0.0196	0.0417	-0.0252	0.0574	-0.0076	0.0023	0.0094	-0.0077	0.0045	-0.0325	-0.0126	0.0384	-0.024	-0.0535	-0.0027	-0.0324	-0.0014	-0.0675	-0.0507	0.0073	-0.0041	0.0554	f3
-0.0914	0.1046	-0.0094	0.0371	0.0551	0.0853	-0.0163	-0.0095	0.0012	0.0083	-0.0036	-0.0327	-0.0147	0.0422	-0.0126	-0.0322	0.0009	-0.0508	-0.0445	-0.0262	-0.0737	0.0263	0.0102	0.0609	f4
-0.101	0.083	-0.041	0.0684	-0.0089	0.0534	-0.0194	0.0219	0.0142	-0.0459	-0.0121	-0.0266	-0.0191	0.05	-0.0139	-0.0109	0.0006	-0.034	-0.0371	-0.0137	-0.0478	0.0176	0.0166	0.0591	f5
-0.0944	0.0983	-0.0191	0.0354	0.0647	0.0804	-0.0287	0.0025	-0.001	-0.0204	0.0224	-0.0243	-0.0089	0.0277	-0.0064	-0.051	0.004	-0.0092	-0.0458	-0.055	-0.0567	0.0161	0.0064	0.0764	f6
-0.0207	-0.0996	-0.1504	0.0202	0.1408	-0.0035	-0.0463	0.0282	-0.0039	-0.0694	-0.009	0.0137	-0.0355	0.0316	0.0362	0.1323	0.0173	0.0491	-0.1213	0.1549	0.0018	0.03	0.051	0.0205	f7
-0.0976	0.031	0.0981	0.0295	-0.0248	0.1079	0.0343	-0.0228	0.0244	0.0632	0.0195	0.0405	0.094	-0.1454	0.0453	-0.0778	-0.0042	-0.0796	0.0669	-0.0457	-0.0516	0.0221	0.0356	-0.1236	f8
-0.0949	-0.0692	-0.0649	0.0417	0.1125	0.0802	-0.0168	0.0087	0.0153	-0.016	0.0066	0.0441	0.0395	-0.0829	0.0689	0.0636	0.0129	-0.0156	-0.0617	0.1095	-0.0383	0.0451	0.0753	-0.0765	f9
-0.0213	0.0904	0.0625	0.0281	-0.0409	0.0726	0.0784	-0.087	0.0159	0.2015	-0.1809	-0.0728	-0.0468	0.1191	-0.0485	0.1152	-0.021	-0.3102	-0.0115	0.1868	-0.1511	0.0822	0.0312	-0.0797	f10
-0.0423	-0.0593	-0.1058	0.1085	-0.2061	-0.096	-0.0124	0.1032	0.0432	-0.179	-0.0284	0.0164	-0.0163	0.0308	-0.0057	0.067	-0.0008	0.05	0.0197	0.0385	0.0773	-0.0258	0.0225	0.0011	f11
-0.0547	-0.0211	-0.0833	0.128	-0.2375	-0.0683	0.0231	0.0692	0.0532	-0.0967	-0.1137	-0.0162	-0.0389	0.0877	-0.0284	0.1243	-0.0106	-0.0903	0.0156	0.1271	0.0122	0.0106	0.0383	-0.0356	f12
-0.0056	-0.096	-0.1594	0.0176	0.1332	-0.0204	-0.0497	0.029	-0.0072	-0.0731	-0.0123	0.0065	-0.051	0.0536	0.0262	0.1362	0.0155	0.0548	-0.1322	0.1583	0.0061	0.0286	0.0503	0.0362	f13
-0.1131	-0.0359	0.0348	0.031	0.0489	0.1006	0.0118	-0.0078	0.0186	0.0274	0.0263	0.0441	0.0644	-0.1036	0.0567	0.0132	0.0101	-0.0151	0.0204	0.0289	-0.0321	0.0214	0.0364	-0.089	f14
0.0372	0.1001	0.1396	-0.0253	-0.1386	-0.0093	0.0432	-0.0249	0.0015	0.0608	0.0041	-0.0186	0.0296	-0.0209	-0.0406	-0.1314	-0.0174	-0.0471	0.1182	-0.1566	0.0034	-0.0328	-0.0574	-0.0088	f15
0.0361	0.0722	0.1052	-0.0935	0.1826	0.1005	0.0243	-0.114	-0.0412	0.2065	-0.0072	-0.0259	0.0045	-0.0067	-0.0043	-0.0396	-0.0036	-0.1063	-0.0351	0.0057	-0.1032	0.0454	-0.0054	-0.0215	f16
-0.0504	-0.0411	-0.0962	0.1214	-0.2271	-0.084	0.0058	0.0889	0.0497	-0.1417	-0.0729	0.0002	-0.0274	0.0591	-0.0167	0.099	-0.0055	-0.0191	0.0194	0.0828	0.0463	-0.0082	0.0306	-0.0147	f17
0.0568	0.0024	0.071	-0.1298	0.235	0.053	-0.0358	-0.0485	-0.0501	0.0517	0.1429	0.0263	0.0462	-0.1031	0.0354	-0.1435	0.014	0.1446	-0.0002	-0.1598	0.0198	-0.0313	-0.0514	0.0493	f18
0.0646	0.0994	-0.1105	0.0131	0.0626	0.0178	0.0045	-0.0048	0.0278	0.0163	-0.0759	-0.0285	0.0587	-0.0596	0.0136	-0.0963	-0.0193	-0.1532	-0.0194	-0.0675	-0.0156	0.0391	0.0396	-0.0062	f19
0.0674	0.0949	-0.1071	0.0068	0.0385	-0.0379	-0.0056	0.0104	-0.0071	-0.0304	-0.0497	-0.0221	-0.0101	0.0048	0.0028	-0.0417	0.0008	-0.023	-0.0682	0.0491	-0.1078	0.0572	0.0919	-0.015	f20
0.0957	0.0806	-0.0612	-0.0264	0.0254	-0.0209	-0.0099	0.0067	0.0271	-0.0393	-0.0126	-0.0491	-0.0099	-0.0124	-0.0076	-0.0992	0.0053	0.0335	-0.0139	-0.0479	-0.0651	0.0109	0.0114	-0.0444	f21
0.0693	0.1003	-0.0618	-0.0073	0.0673	0.0016	0.0432	-0.0346	0.0314	0.0743	-0.0828	-0.0229	0.0733	-0.0488	0.0071	-0.0827	-0.0322	-0.1421	0.0195	-0.0731	0.1395	-0.0042	0.0056	0.0678	f22
0.111	0.0656	-0.0372	-0.038	0.0196	-0.0481	-0.001	-0.0074	0.0099	0.0102	-0.0151	-0.0415	-0.0225	0.0273	-0.0213	-0.0414	-0.0066	0.0257	-0.0094	-0.0178	0.0135	-0.0001	-0.0055	0.0422	f23
0.0635	0.0917	-0.128	0.0175	0.0527	0.0193	-0.0298	0.017	0.0243	-0.0307	-0.0585	-0.0282	0.0216	-0.0471	0.0101	-0.0801	-0.0066	-0.1149	-0.0498	-0.0487	-0.0909	0.0678	0.0819	-0.0787	f24
-0.0763	0.1203	-0.0067	0.0418	-0.0107	0.0513	-0.0214	0.0319	-0.0133	-0.0728	-0.0055	-0.0381	-0.019	0.0769	-0.0384	-0.0661	0.0084	-0.0263	-0.0191	-0.0839	0.0057	-0.0236	-0.0675	0.1594	f25
-0.1163	0.0341	0.0401	0.019	0.0174	0.0437	-0.0333	0.0124	-0.0308	-0.0437	0.0423	-0.006	-0.0634	0.0934	-0.0239	0.0251	0.0021	0.0801	-0.0444	-0.0376	-0.0105	-0.0134	-0.0277	0.0747	f26
-0.1186	-0.0119	0.0568	0.0231	-0.0051	0.0674	-0.008	0.0088	-0.0252	-0.0222	0.0323	0.0253	-0.0023	0.014	0.0031	0.0141	0.0053	0.014	0.001	-0.0424	-0.0168	-0.0277	-0.0308	0.0187	f27
-0.0772	-0.1126	0.0716	-0.0187	0.0422	0.0276	-0.014	-0.0073	0.0012	0.0101	0.0686	0.0282	-0.0324	-0.015	0.0319	0.1095	0.0116	0.1208	0.0186	0.042	0.0231	0.0134	0.004	-0.0852	f28
0.1143	0.0251	0.0023	-0.0567	-0.029	-0.1009	-0.0121	-0.0071	-0.0182	-0.0013	0.0092	-0.0456	-0.0728	0.1025	-0.0537	-0.0093	-0.0058	0.1005	-0.0217	-0.0314	0.0145	-0.0174	-0.0289	0.0924	f29
0.0559	0.0435	0.078	-0.1149	0.2369	0.0874	-0.002	-0.084	-0.0297	0.1366	0.0656	-0.0021	0.0419	-0.0767	0.0252	-0.1009	0.0032	0.0121	-0.0024	-0.0909	-0.0291	0.0207	-0.0181	0.0027	f30
0.0948	-0.0482	0.044	-0.086	-0.006	-0.1152	-0.0305	-0.0095	-0.0176	-0.0056	0.0602	-0.0348	-0.117	0.1091	-0.0384	0.0758	0.0003	0.2149	-0.021	0.0045	0.0333	-0.0068	-0.0202	0.0376	f31
-0.1127	-0.0513	0.0245	0.0387	0.0338	0.0896	0.0085	0.0005	0.0115	0.0106	0.0126	0.0412	0.0478	-0.0769	0.0461	0.0323	0.0081	-0.0439	0.026	0.0273	-0.0037	0.0137	0.0151	-0.0838	f32
0.0962	0.0932	-0.0421	-0.007	-0.0432	-0.052	0.0111	-0.0022	-0.0128	0.0036	-0.0471	-0.0417	-0.0022	0.0612	-0.0517	-0.0951	-0.0102	-0.0528	-0.0151	-0.0584	-0.0101	-0.0226	-0.0283	0.1185	f33

Figure 119: Eigenvectors

For e.g.: - The first principal component f1, shows the 1st variable V1 with the highest value of -0.102, and then the second principal component f2 shows the highest contribution in the 2nd variable V2 with a value of 0.0581, and so on.

Ranked attribute: The last part of the PCA output, provides a list of attributes or variables ranked in descending order of their importance i.e., 24 out of 174 attributes. Every feature is assigned a value or weight, known as “Eigenvalue”. The bigger the eigenvalue, the more important the attribute is in describing the variance in the data.

```
Ranked attributes:
 0.641     1 -0.119f27-0.118f45-0.118f49-0.116f26-0.116f143...
 0.4401    2 0.156f92+0.156f95+0.155f96+0.151f93+0.145f74...
 0.3433    3 0.209f64-0.205f56-0.196f62+0.193f79-0.187f83...
 0.2798    4 0.171f69+0.168f133+0.163f125+0.157f163-0.156f131...
 0.2464    5 -0.237f12+0.237f30+0.235f18-0.227f17-0.206f11...
 0.2185    6 0.189f139-0.189f146+0.158f115+0.156f101+0.15 f167...
 0.1936    7 0.25 f171-0.246f169-0.245f174-0.243f173+0.241f172...
 0.1742    8 0.322f135+0.31 f136-0.31f134+0.304f131-0.287f133...
 0.1565    9 0.222f118+0.217f162+0.216f160+0.205f122+0.203f124...
 0.1429   10 -0.26f42-0.251f48+0.207f16+0.202f10+0.185f61...
 0.1302   11 0.261f97+0.181f91-0.181f10+0.179f118-0.172f120...
 0.1191   12 0.325f165+0.31 f127-0.3f164-0.288f126+0.205f160...
 0.1092   13 -0.387f130-0.354f132-0.28f168-0.247f136-0.213f170...
 0.1009   14 0.307f127-0.278f126+0.203f123-0.192f166+0.163f100...
 0.0926   15 -0.438f166-0.319f165+0.289f127-0.273f126+0.267f164...
 0.0847   16 -0.469f48-0.462f42-0.143f18+0.136f13+0.132f7...
 0.0783   17 0.8  f128+0.438f166+0.156f127+0.14 f164-0.108f165...
 0.0725   18 -0.333f91-0.31f10-0.252f71-0.242f34-0.242f118...
 0.0671   19 0.497f97-0.323f44+0.285f91+0.194f71-0.179f47...
 0.062    20 0.392f48+0.326f97+0.253f44-0.235f43+0.187f10...
 0.058    21 -0.242f97+0.235f48+0.211f114+0.183f70+0.183f118...
 0.0543   22 0.557f166-0.402f128-0.283f97+0.23 f71+0.215f164...
 0.0507   23 0.367f128-0.361f166+0.274f99+0.274f71+0.226f115...
 0.0476   24 -0.247f70-0.222f59-0.198f42+0.18 f43+0.176f130...
```

Figure 120: Ranked attributes

For e.g., the first-ranked attribute has a **variance** of 0.641 which has the highest importance or weight, then the second value has the second highest variance i.e., 0.4401, and so on. And at the last 0.0476 which has the bare minimum variance.

Now, we will plot the data in the elbow curve, and take only the values that are above the elbow curve to get the best results.

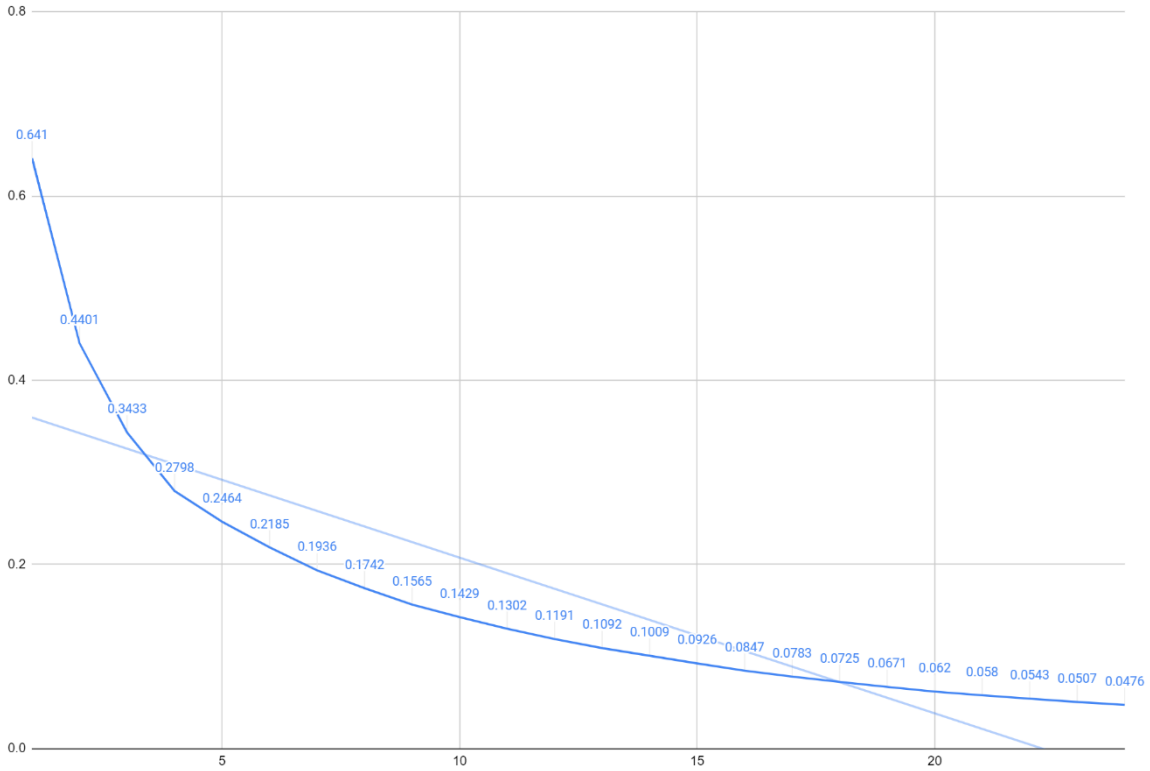


Figure 121: Elbow curve

The values that are above the curve are: -

X	Y
1	0.641
2	0.4401
3	0.3433

Table 6: Elbow value in table

Even after deleting the rest of the values and only keeping these 3 variances we got a better result.

```
Ranked attributes:
 0.667  1 0.9780.209f64-0.205f56-0.196f62+0.193f79-0.187f83...-0.180.156f92+0.156f95+0.155f96+0.151f93+0.145f74...-0.109-0.119f27-0.118f45-0.118f49-0.116f26-0.116f143...
 0.333  2 0.981-0.119f27-0.118f45-0.118f49-0.116f26-0.116f143...+0.1420.156f92+0.156f95+0.155f96+0.151f93+0.145f74...+0.1350.209f64-0.205f56-0.196f62+0.193f79-0.187f83...
 0      3 -0.9730.156f92+0.156f95+0.155f96+0.151f93+0.145f74...+0.164-0.119f27-0.118f45-0.118f49-0.116f26-0.116f143...-0.1610.209f64-0.205f56-0.196f62+0.193f79-0.187f83...
```

Figure 122: Result after feature elimination

9. Feature Selection Method

For feature selection, we will select the "Wrapper based" to be specific "WrapperSubsetEval" method evaluator which uses a decision tree classifier, J48.

After implementing this we got the following result:

```
=== Attribute Selection on all input data ===

Search Method:
        Best first.
        Start set: no attributes
        Search direction: forward
        Stale search after 5 node expansions
        Total number of subsets evaluated: 2352
        Merit of best subset found:    0.999
```

Figure 123: Attribute selection

A total of 2352 subsets were evaluated during the search and the merit of the best subset found is 0.999, which indicates that it has a high classification accuracy.

```
Selected attributes: 2,5,13,24,40,64,140,157,167 : 9
                     f2
                     f5
                     f13
                     f24
                     f40
                     f64
                     f140
                     f157
                     f167
```

Figure 124: Best subsets

The Weka then selects the best subset of features selected by the feature selection algorithm.

By selecting these attributes, we can reduce the dimensionality of the dataset and potentially improve the performance of our model.

Now, after keeping only the features that were selected by the feature selection model, we will run the J48 classification algorithm again to check if we will get better results or not.

```
=== Summary ===

Correctly Classified Instances        4799               99.9792 %
Incorrectly Classified Instances         1                0.0208 %
Kappa statistic                          0.9997
Mean absolute error                      0.0012
Root mean squared error                  0.0133
Relative absolute error                  0.3192 %
Root relative squared error              3.1254 %
Total Number of Instances             4800
```

Figure 125: Summary of accuracy

The result summary shows that the performance of the classifier J48 improved after applying the feature selection algorithm.

Before applying the feature selection, the J48 classifier got a classification accuracy of 99.6042% and 19 incorrectly classified instances. However, after applying the feature selection algorithm, the J48 classifier achieved a higher accuracy of 99.8333% and only 8 incorrectly classified instances.

	J48 Before Feature Extraction	J48 After Feature Extraction
Correctly Classified Instances	4781	4799
Incorrectly Classified Instances	19	1
Kappa statistic	0.9946	0.9997
Mean Absolute Error	0.0021	0.0012
Root Mean Squared Error	0.0442	0.0133
Relative Absolute Error	0.5779%	0.3192%
Root Relative Squared Error	10.3462%	3.1254
Total Number of Instances	4800	4800

Table 7: Comparison between before and after feature selection

Overall, we can conclude that the feature selection algorithm has helped to improve the performance of the J48 classifier by selecting only the most relevant features and it even helped to reduce the time taken to build and test the model on test split i.e., 0.1 seconds and 0.01 seconds respectively.

10. Links for Presentation Videos

- **Bidhan Pant:** *Presentation video link*
- **Biraj Pokharel:** *Presentation video link*
- **Pritpal Singh Nanray:** *Presentation video link*