

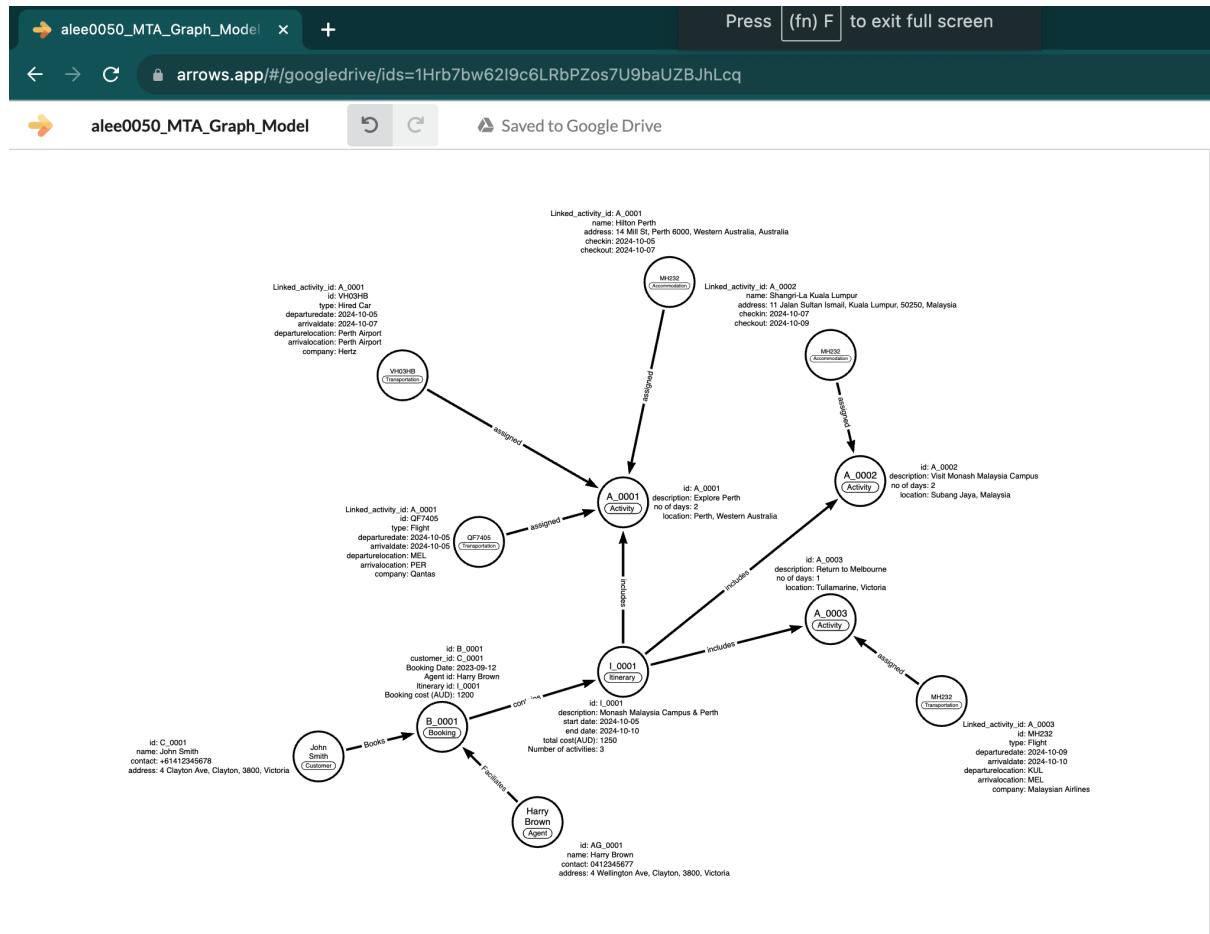
ASSESSMENT COVER SHEET

Student ID number 30864941	Unit Name and Code: FIT3176 Advanced Database Design Campus: Clayton Assignment Title: FIT3176 Individual Assignment - Sem 2/2023 (Weight: 30%) Name of Lecturer: Farah Kabir Name of Tutor: Farah Kabir Tutorial Day and Time: Tuesday 5PM-7PM Phone Number: 0405245648 Email Address: Alee0050@student.monash.edu		
Has any part of this assignment been previously submitted as part of another unit/course? <input type="checkbox"/> Yes <input checked="" type="checkbox"/> No			
Due Date: 20th of Oct Date Submitted: 29th of Oct			
All work must be submitted by the due date. If an extension of work is granted this must be specified with the signature of the lecturer/tutor. Extension granted until (date) _____ 29 th of Oct _____ _____ Monash Connect _____ Signature of lecturer/tutor _____			
Please note that it is your responsibility to retain copies of your assessments.			
<i>Intentional plagiarism or collusion amounts to cheating under Part 7 of the Monash University (Council) Regulations</i>			
Plagiarism: Plagiarism means taking and using another person's ideas or manner of expressing them and passing them off as one's own. For example, by failing to give appropriate acknowledgement. The material used can be from any source (staff, students or the internet, published and unpublished works).			
Collusion: Collusion means unauthorised collaboration with another person on assessable written, oral or practical work and includes paying another person to complete all or part of the work.			
Where there are reasonable grounds for believing that intentional plagiarism or collusion has occurred, this will be reported to the Associate Dean (Education) or delegate, who may disallow the work concerned by prohibiting assessment or refer the matter to the Faculty Discipline Panel for a hearing.			
Student Statement: <ul style="list-style-type: none"> • I have read the university's Student Academic Integrity Policy and Procedures. • I understand the consequences of engaging in plagiarism and collusion as described in Part 7 of the Monash University (Council) Regulations http://adm.monash.edu/legal/legislation/statutes • have taken proper care to safeguard this work and made all reasonable efforts to ensure it could not be copied. • No part of this assignment has been previously submitted as part of another unit/course. • I acknowledge and agree that the assessor of this assignment may for the purposes of assessment, reproduce the assignment and: <ul style="list-style-type: none"> i. provide to another member of faculty and any external marker; and/or ii. submit it to a text matching software; and/or iii. submit it to a text matching software which may then retain a copy of the assignment on its database for the purpose of future plagiarism checking. • I certify that I have not plagiarised the work of others or participated in unauthorised collaboration when preparing this assignment. SignatureAndrew Lee..... Date.....29/10/2023..... * delete (iii) if not applicable			

The information on this form is collected for the primary purpose of assessing your assignment and ensuring the academic integrity requirements of the University are met. Other purposes of collection include recording your plagiarism and collusion declaration, attending to course and administrative matters and statistical analyses. If you choose not to complete all the questions on this form it may not be possible for Monash University to assess your assignment. You have a right to access personal information that Monash University holds about you, subject to any exceptions in relevant legislation. If you wish to seek access to your personal information or inquire about the handling of your personal information, please contact the University Privacy Officer: privacyofficer@adm.monash.edu.au

A1:

<https://arrows.app/#/googledrive/ids=1Hrb7bw62I9c6LRbPZos7U9baUZBJhLcq>



A1 results:

```

1 // Node creation
2 MERGE (john_smith:Customer {id: "c_001", name: "John Smith", contact: "+61412567890", address: "4
  Clayton Ave, Clayton, 3800, Victoria"})
3 MERGE (harry_brown:Agent {id: "AG_001", name: "Harry Brown", contact: "0412345678", address: "4
  Wellington Ave, Clayton, 3800, Victoria"})
4 MERGE (b:Booking {id: "B_0001", booking_date: "2023-09-12", cost: 1200})
5 MERGE (itinerary: Itinerary {id: "l_0001", description: "Monash Malaysia Campus &
  Perth", start_date: "2024-10-05", end_date: "2024-10-10", total_cost: 1250, number_of_activities: 3})
6 MERGE (activity_1:Activity {id: "A_0001", description: "Explore Perth", number_of_days: 2, location:
  "Perth, Western Australia"})
7 MERGE (activity_2:Activity {id: "A_0002", description: "Visit Monash Malaysia
  Campus", number_of_days: 2, location: "Subang Jaya, Malaysia"})
    
```

Added 12 labels, created 12 nodes, set 63 properties, created 11 relationships, completed after 211 ms.



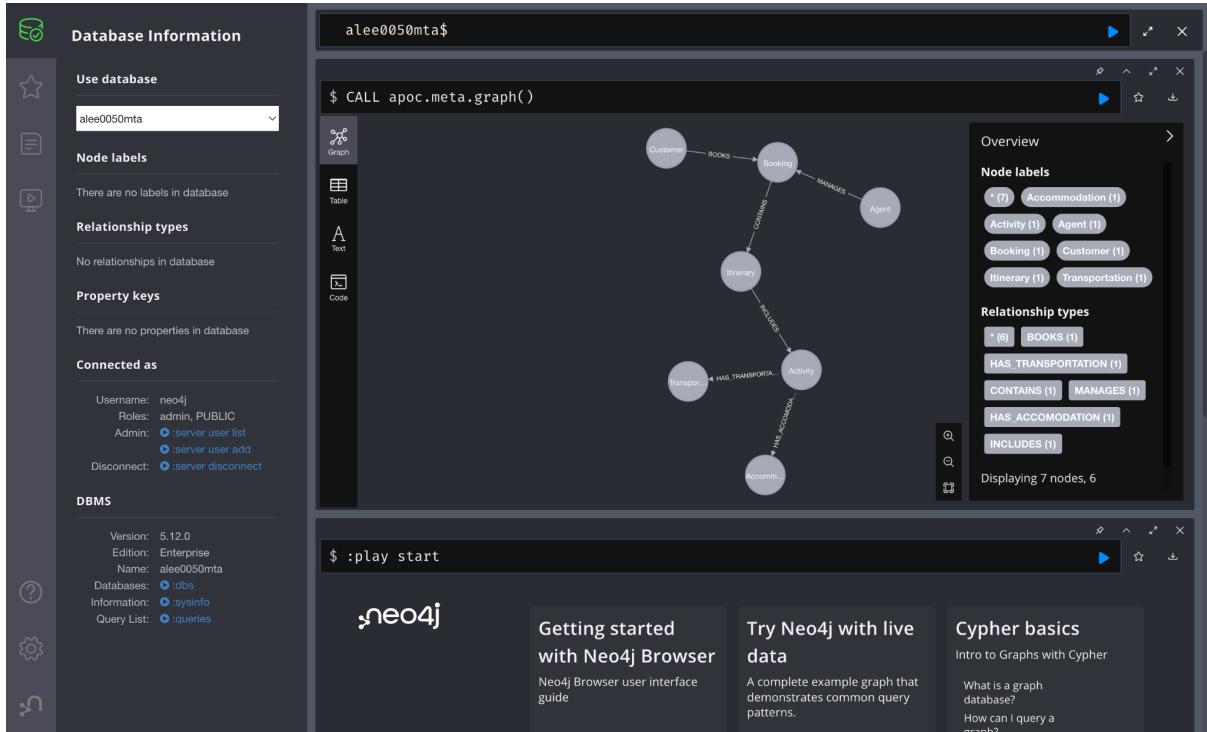
Table



Code

Added 12 labels, created 12 nodes, set 63 properties, created 11 relationships, completed after 211 ms.

APOC calls:



B.1. Creating the Database:

The screenshot shows the Neo4j Browser interface with a Cypher script in the editor. The script is used to load data from a CSV file into a Neo4j database. It starts by loading a CSV file named 'species_a2.csv' into a variable 'row'. It then iterates through each row, trimming leading and trailing whitespace and checking if the 'Species ID' is not null. If it's not null, it merges the species into the database. It then processes common names by splitting them at commas and merging them into individual nodes. Finally, it handles scientific names by splitting them at commas and merging them into individual nodes. The script ends with a note about missing fields.

```

1 //LOADING CSV from species_a2
2 LOAD CSV WITH HEADERS FROM "file:///species_a2.csv" AS row
3 //WITH row is to reference the value, the WITH statement is essentially calling a variable that exists and that has the latest
4 //value from previous operations, as cypher queries cannot reference a variable like a normal programming language, WITH will take the previous
5 //LOAD csv statement AS row, and it reads CSV row by row when it is iterating through the csv file.
6 WITH row
7
8 WHERE row.`Species ID` IS NOT NULL AND TRIM(row.`Species ID`) <> ""
9 MERGE (species:Species {species_id: row.`Species ID`})
10
11
12 //this ensures that our species ID is not a incomplete record and we trim any trailing white spaces
13 //we have different edge cases for an example a species ID can be : " " which is just white space, but it counts as a character still
14 //we don't want that, and we use the inequality comparison operator for that, if there is a white space then we trim it
15 //merging is just creating something that does not exist, if it does that this does nothing, it is a form of update as update queries
16 //can make new if something does not exist, or it simply leaves it alone if it does exist with the same record
17
18
19 WITH row //current row
20 UNWIND split(COALESCE(row.`Common Names`,"N/A"),',') AS new
21 MERGE (c:CommonName {common_name: trim(new)})
22
23 //in this case we are using UNWIND to iterate through the Common names in our CSV file, if there is an empty or null value
24 //in our common names, we want to split that and call the empty/null val next, because we don't want to use that as our value
25 //we will then call split to split that into individual common names, trim just removes anything that is next
26 //for an example our input can be "Eastern Coyote, Coyote SCAT", we want to seperate those grouping Eastern Coyote
27 //and Coyote SCAT individually from the our rows
28
29 WITH row
30 WHERE row.`Scientific Name` IS NOT NULL
31 AND row.Occurrence IS NOT NULL
32 AND row.ABundance IS NOT NULL
33 AND row.Seasonality IS NOT NULL
34 AND row.`Conservation Status` IS NOT NULL
35 AND row.Nativeness IS NOT NULL
36 AND row.Record_Status IS NOT NULL
37 AND row.Family IS NOT NULL
38 AND row.Order IS NOT NULL
39 AND row.Category IS NOT NULL
40
41 //we need to do this because we are not sure if any of the fields can be missing, if you manually check there are actually missing fields
42 //that is obviously an incomplete record, it is something that we don't want when we are doing data cleaning

```

```
alee0050_task_B.cypher
Users > hellachastrimx > alee0050_task_B.cypher
44 MERGE(Scientific: ScientificName {scientific: row.`Scientific Name`})
45 MERGE(occurrence: Occurrence{o:row.Occurrence})
46 MERGE(abundance: Abundance{a:row.Abundance})
47 MERGE(seasonality: Seasonality{s:row.Seasonality})
48 MERGE(conservationStatus: conservationStatus{cs:row.`Conservation Status`})
49 MERGE(native: Native{n:row.Nativity})
50 MERGE(recordStatus:RecordStatus {r: row.`Record Status`})
51 MERGE(family:Family{f:row.Family})
52 MERGE(order:Order{ord:row.Order})
53 MERGE(category:Category{cat:row.Category});
54
55 //MERGE statements is just creating reference nodes taken from species_a2, we are creating a node for every single column
56 //if we do a MATCH statement using n, n is a simply an abstract representation of all the nodes, and we return it we will be able to see
57 //every single node that has been created using the CSV file, we can also call apoc.meta.graph() to display a high
58
59
60
61 // Load parks CSV
62 LOAD CSV WITH HEADERS FROM "file:///parks_a2.csv" AS park_row
63 WITH park_row
64 WHERE park_row.`Park Name` IS NOT NULL
65 AND park_row.`Park Code` IS NOT NULL
66 AND park_row.`State Code` IS NOT NULL
67 AND park_row.`Area in Acres` IS NOT NULL
68 AND park_row.Latitude IS NOT NULL
69 AND park_row.Longitude IS NOT NULL
70 AND park_row.`State Name` IS NOT NULL
71 AND park_row.`Est Day` IS NOT NULL
72 AND park_row.`Est Month` IS NOT NULL
73 AND park_row.`Est Year` IS NOT NULL
74
75
76 MERGE (state:State {state_code: park_row.`State Code`}, state_name: park_row.`State Name`)
77 MERGE (park:Park {
78     park_code: park_row.`Park Code`,
79     park_name: park_row.`Park Name`,
80     area_in_acres: toInteger(park_row.Area in Acres`),
81     longitude: toFloat(park_row.Longitude),
82     latitude: toFloat(park_row.Latitude),
83     est_date: date(
84         year: toInteger(park_row.`Est Year`),
85         month: toInteger(park_row.`Est Month`),
```

The screenshot shows a code editor with multiple tabs open, displaying Cypher queries. The queries are used to merge park and state data from CSV files into a graph database. The code includes merging nodes for parks and states, setting properties for each node (like park_code, state_code, longitude, latitude, area_in_acres, est_date, year, month, day), and creating relationships between parks and states based on their location.

```
Users > velliecharxtrum > alee0050_task_B.cypher ● TaskC1.cypher ● alee0050_task_C1.cypher ● alee0050_task_C2.py

72 AND park_row.'Est Month' IS NOT NULL
73 AND park_row.'Est Year' IS NOT NULL
74
75
76 MERGE (state:State {state_code: park_row.'State Code', state_name: park_row.'State Name'})
77 MERGE (park:Park {
78   park_code: park_row.'Park Code',
79   park_name: park_row.'Park Name',
80   area_in_acres: toInteger(park_row.'Area in Acres'),
81   longitude:toFloat(park_row.Longitude),
82   latitude:toFloat(park_row.Latitude),
83   est_date: date(
84     year: toInteger(park_row.'Est Year'),
85     month: toInteger(park_row.'Est Month'),
86     day: toInteger(park_row.'Est Day')
87   })
88 });
89
90
91 //relationships
92
93
94
95 LOAD CSV WITH HEADERS FROM "file:///parks_a2.csv" AS row
96 WITH row
97 WHERE row.'Park Code' IS NOT NULL
98 AND row.'State Code' IS NOT NULL
99 AND row.'Area in Acres' IS NOT NULL
100
101
102 MATCH (park: Park {park_code: row.'Park Code'}),
103 (state: State {state_code: row.'State Code'})
104 MERGE (park)-[:LOCATED_IN {area_covered: toInteger(row.'Area in Acres')}]->(state);
105
106 //B3
107 MERGE (parkDEVA:Park {park_code: "DEVA"})-[:LOCATED_IN {area_covered: 790152}]->(stateCA:State {state_code: "CA"})
108 MERGE (parkDEVA)-[:LOCATED_IN {area_covered: 3950760}]->(stateNV:State {state_code: "NV"})
109
110 MERGE (parkYELL:Park {park_code: "YELL"})-[:LOCATED_IN {area_covered: 521490}]->(stateW:State {state_code: "WY"})
111 MERGE (parkYELL)-[:LOCATED_IN {area_covered: 521490}]->(stateMT:State {state_code: "MT"})
112 MERGE (parkYELL)-[:LOCATED_IN {area_covered: 521490}]->(stateID:State {state_code: "ID"})
```

```

    alee0050_task_B.cypher • TaskC1.cypher • alee0050_task_C1.cypher • alee0050_task_C2.py
Users > vellichastrixism > alee0050_task_B.cypher
117 MERGE (:parkYELL)-[:LOCATED_IN {area_covered: 521490}]->(stateMT:State {state_code: "MT"})
118 MERGE (:parkYELL)-[:LOCATED_IN {area_covered: 521490}]->(stateID:State {state_code: "ID"})
119
120 MERGE (:parkGRSM:Park {park_code: "GRSM"})-[:LOCATED_IN {area_covered: 510390}]->(stateTN:State {state_code: "TN"})
121 MERGE (:parkGRSM)-[:LOCATED_IN {area_covered: 111000}]->(stateNC:State {state_code: "NC"})
122
123 LOAD CSV WITH HEADERS FROM "file:///species_a2.csv" AS row
124
125 WITH row
126 MATCH (species:Species {species_id: row.'Species ID'})
127 MATCH (Scientific:ScientificName {scientific: row.'Scientific Name'})
128 MERGE (species)-[:BELONGS_TO]->(Scientific)
129 WITH row
130 MATCH (species:Species {species_id: row.'Species ID'})
131 MATCH (occurrence:Occurrence {o:row.Occurrence})
132 MERGE (species)-[:HAS]->(occurrence)
133 WITH row
134 MATCH (species:Species {species_id: row.'Species ID'})
135 MATCH (recordStatus:RecordStatus {rs: row.'Record Status'})
136 MERGE (species)-[:HAS]->(recordStatus)
137 WITH row
138 MATCH (species:Species {species_id: row.'Species ID'})
139 MATCH (conservationStatus:ConservationStatus {cs: row.'Conservation Status'})
140 MERGE (species)-[:HAS]->(conservationStatus)
141 WITH row
142 MATCH (species:Species {species_id: row.'Species ID'})
143 MATCH (seasonality:Seasonality {s: row.Seasonality})
144 MERGE (species)-[:HAS]->(seasonality)
145 WITH row
146 MATCH (seasonality:Seasonality {s: row.Seasonality}), (park:Park {park_name: row.'Park Name'})
147 MERGE (seasonality)-[:IN]->(park)
148 WITH row
149 MATCH (species:Species {species_id: row.'Species ID'})
150 MATCH (abundance:Abundance {a: row.Abandance})
151 MERGE (species)-[:HAS]->(abundance)
152 WITH row
153 MATCH (species:Species {species_id: row.'Species ID'})
154 MATCH (native:Native {n: row.Nativeness})
155 MERGE (species)-[:HAS]->(native)
156 WITH row
157 MATCH (Scientific:ScientificName {scientific: row.'Scientific Name'})
158 MATCH (family:Family {f: row.Family})

```



```

    alee0050_task_B.cypher
Users > vellichastrixism > alee0050_task_B.cypher
163 MERGE (:family)-[:BELONGS_TO]->(order)
164 WITH row
165 MATCH (order:Order {ord: row.Order})
166 MATCH (category:Category {cat: row.Category})
167 MERGE (order)-[:BELONGS_TO]->(category)
168 WITH row
169 MATCH (native:Native {n: row.Nativeness})
170 MATCH (park:Park {park_name: row.'Park Name'})
171 MERGE (native)-[:IN]->(park)
172 WITH row
173 MATCH (abundance:Abundance {a: row.Abandance})
174 MATCH (park:Park {park_name: row.'Park Name'})
175 MERGE (abundance)-[:IN]->(park)
176 WITH row
177 MATCH (occurrence:Occurrence {o: row.Occurrence})
178 MATCH (park:Park {park_name: row.'Park Name'})
179 MERGE (occurrence)-[:IN]->(park)
180 WITH row
181 MATCH (recordStatus:RecordStatus {rs: row.'Record Status'})
182 MATCH (park:Park {park_name: row.'Park Name'})
183 MERGE (recordStatus)-[:IN]->(park)
184 WITH row
185 MATCH (conservationStatus:ConservationStatus {cs: row.'Conservation Status'})
186 MATCH (park:Park {park_name: row.'Park Name'})
187 MERGE (conservationStatus)-[:IN]->(park)
188 WITH row
189 MATCH (species:Species {species_id: row.'Species ID'})
190 MATCH (park:Park {park_name: row.'Park Name'})
191 MERGE (species)-[:FOUND_IN {population: toInteger(row.'Species Population')}]->(park)
192 WITH row
193 MATCH (species:Species {species_id: row.'Species ID'})
194 MERGE (common_name:CommonName {common_name: row.'Common Names'})
195 MERGE (species)-[:KNOWN_AS]->(common_name);
196
197

```

Results:

The screenshot shows the Neo4j browser interface with the following details:

- Top Bar:** RecordStatus, ScientificName, Seasonality, Species, State, conservationStatus.
- Relationship types:** *{25,693} BELONGS_TO, FOUND_IN HAS IN, KNOWN_AS LOCATED_IN.
- Property keys:** a area_covered, area_in_acres, cat, col_name, common_name, cs, est_date, f, latitude, longitude, n, o, ord, park_code, park_name, population, rs, s, scientific, species_id, state_code, state_name, totalDist.
- Connected as:** Username: neo4j, Roles: admin, PUBLIC, Admin: server user list, server user add, Disconnect: server disconnect.
- DBMS:** Version: 5.12.0, Edition: Enterprise, Name: aleee0050mnc.

Command History:

```

alee0050mnc$ //LOADING CSV from species_a2 LOAD CSV WITH HEADERS FROM "file:///species_a2.csv" AS row WITH row WHERE ...
alee0050mnc$ LOAD CSV WITH HEADERS FROM "file:///species_a2.csv" AS row WITH row WHERE ...
alee0050mnc$ LOAD CSV WITH HEADERS FROM "file:///parks_a2.csv" AS park_row WITH park_r...
alee0050mnc$ LOAD CSV WITH HEADERS FROM "file:///parks_a2.csv" AS row WITH row WHERE r...
alee0050mnc$ MERGE (parkDEVA:Park {park_code: "DEVA"})-[:LOCATED_IN {area_covered: 790...
alee0050mnc$ LOAD CSV WITH HEADERS FROM "file:///species_a2.csv" AS row WITH row WITH ...
alee0050mnc$ MATCH (s:Species)-[r:KNOWN_AS]-(c:CommonName) MATCH (s)-[f:FOUND_IN]-(p...
alee0050mnc$ MATCH (s:Species)-[:HAS]-(c:conservationStatus{cs:"Endangered"}) MATCH ...
alee0050mnc$ MATCH (p:Park)-[r:LOCATED_IN]-(s:State) WITH s.state_code AS `State Code...

```

Visualisation of graph nodes and relationships when we call apoc.

The screenshot shows the Neo4j browser interface with the following details:

- Top Bar:** RecordStatus, ScientificName, Seasonality, Species, State, conservationStatus.
- Relationship types:** *{25,693} BELONGS_TO, FOUND_IN HAS IN, KNOWN_AS LOCATED_IN.
- Property keys:** a area_covered, area_in_acres, cat, col_name, common_name, cs, est_date, f, latitude, longitude, n, o, ord, park_code, park_name, population, rs, s, scientific, species_id, state_code, state_name, totalDist.
- Connected as:** Username: neo4j, Roles: admin, PUBLIC, Admin: server user list, server user add, Disconnect: server disconnect.
- DBMS:** Version: 5.12.0, Edition: Enterprise, Name: aleee0050mnc.

Graph Visualization: The interface displays a complex graph structure with various nodes (e.g., Park, Species, State, Category) and their relationships. A sidebar provides an "Overview" of node labels and relationship types.

Node Labels Overview:

- (14) Abundance (1)
- Category (1) CommonName (1)
- Family (1) Native (1)
- Occurrence (1) Order (1)
- Park (1) RecordStatus (1)
- ScientificName (1)
- Seasonality (1) Species (1)
- State (1)
- conservationStatus (1)

Relationship types Overview:

- (19) HAS (6)

Command History:

```

alee0050mnc$ CALL apoc.meta.graph()
alee0050mnc$ //LOADING CSV from species_a2 LOAD CSV WITH HEADERS FROM "file:///species_a2.csv" AS ...
alee0050mnc$ LOAD CSV WITH HEADERS FROM "file:///species_a2.csv" AS row WITH row WHERE ...
alee0050mnc$ LOAD CSV WITH HEADERS FROM "file:///parks_a2.csv" AS park_row WITH park_r...
alee0050mnc$ LOAD CSV WITH HEADERS FROM "file:///parks_a2.csv" AS row WITH row WHERE r...
alee0050mnc$ MERGE (parkDEVA:Park {park_code: "DEVA"})-[:LOCATED_IN {area_covered: 790...

```

B.2. Querying the Database:

(i)

Code:

```
// MATCH all the species with the relationship known_as
MATCH (s:Species)-[r:KNOWN_AS]->(c:CommonName)
// match species found_in park relationship
MATCH (s)-[f:FOUND_IN]->(p:Park)
// common name should contain Coyote
WHERE c.common_name CONTAINS 'Coyote'

WITH p, c.common_name AS comName, SUM(f.population) AS tot
// Filter out single names by checking for the existence of a comma
WHERE comName CONTAINS ','
RETURN p.park_name AS `Park Name`, collect(comName) AS `Coyote Species`, tot AS `Coyote Population`
ORDER BY tot DESC;
```

Code explanation:

```
// First, we want to extract every single instance that is dictated by a general graph structure of our database
// species has a one to many relationship to common names, as one species can be known by multiple commonnames
// we then want to match the species that are found in specific parks
// which species of which park they belong to etc, this will set up our query to find distinct parks using MATCH
// MATCH is a form of grep, pattern matching algorithm that is used for both nodes and relationships
// We want to use MATCH Neo4j iterates through rows one by one, as dictated in our LOAD CSV statement.
// Our next step, we will tell neo4j to iterate through each row and lookfor common names that contain 'Coyote'.
// Coyote species known as "Eastern Coyote" and also known as "Coyote Scat," both of these names
// using a WHERE clause, we can filters out all the rows that contain "Coyote."
// Understanding WITH is a important part of the operation, as variables need to use WITH clause in order to reference the variables that were accessed
// from the previous part of the query. We want to sum up all the populations of Coyotes for each park,
// and then we label the common_name AS `comName`. We will change that shortly when we return it, for readability we use a name instead of referencing c.common_name
// we will use another WHERE clause to filter out if comName contains a comma, this will indicate that it is known by two names rather than one
// for an example, a species can go by "Eastern Coyote, Coyote SCAT", there is a comma to indicate that it is more than one name
// Now we want to return our results for Neo4j to display with our set name, using AS statement
// then just order by descending order starting with population, this will give us the highest population first, second highest etc
```

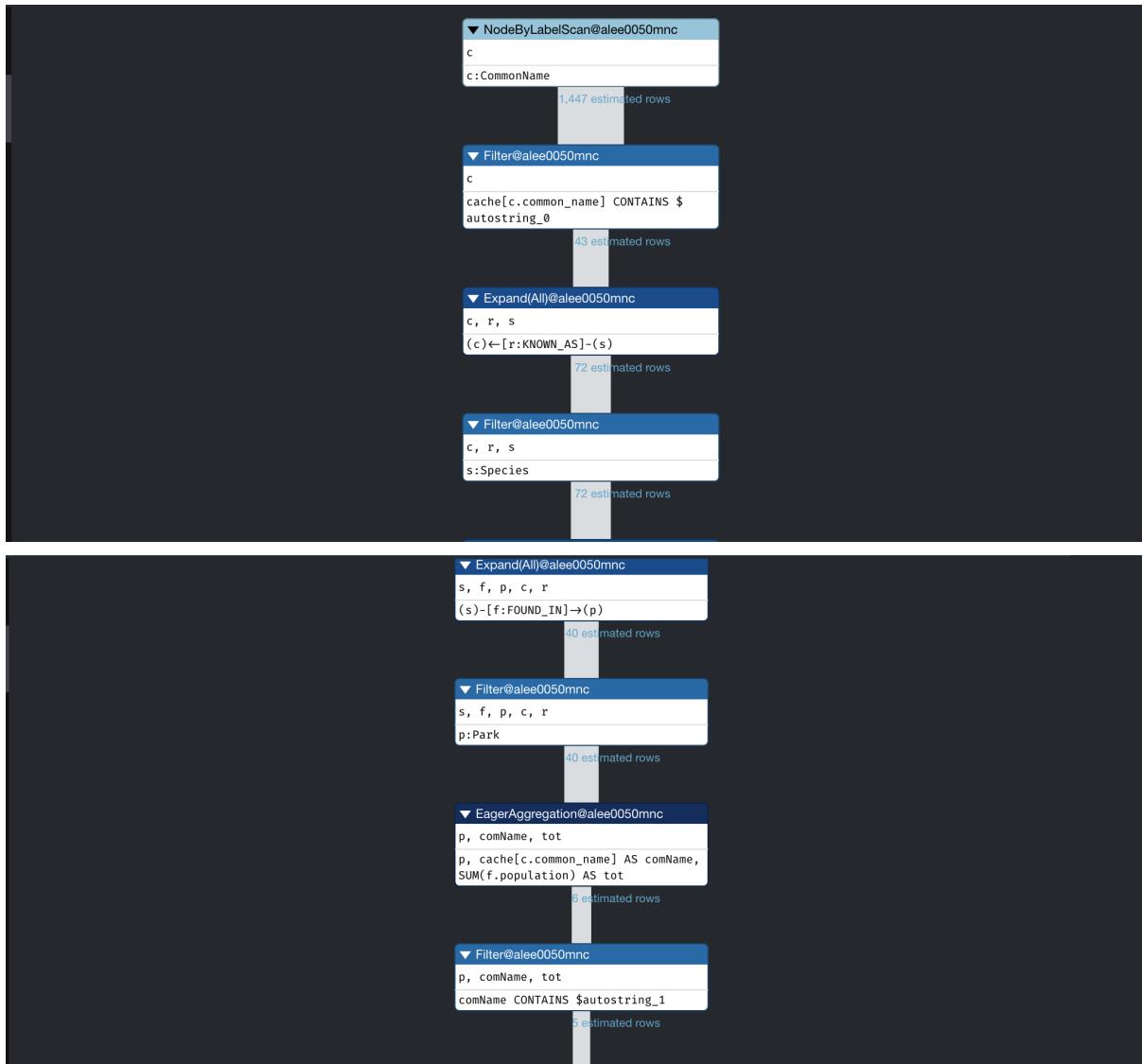
Results:

The screenshot shows the Neo4j browser interface with the following details:

- Relationship types:** A sidebar listing relationship types: *{25,829} BELONGS_TO, CONNECTED, FOUND_IN, HAS, IN KNOWN_AS, LOCATED_IN.
- Property keys:** A sidebar listing property keys: a, area_covered, area_in_acres, cat, col_name, common_name, cs, est_date, f, latitude, longitude, n, o, ord, park_code, park_name, population, rs, s, scientific, species_id, state_code, state_name, totalDist.
- Connected as:** A sidebar showing connection information: Username: neo4j, Roles: admin, PUBLIC, Admin: .server user list, .server user add, Disconnect: .server disconnect.
- DBMS:** A sidebar showing DBMS details: Version: 5.12.0, Edition: Enterprise, Name: ale0050mnc, Databases: dbs, Information: sysinfo, Query List: queries.
- Query Results:** The main pane displays the results of the query. The title bar says "alee0050mnc\$". The code input field shows the query: "alee0050mnc\$ // MATCH all the species with the relationship known_as MATCH (s:Species)-[...". The results are presented in a table:

Park Name	Coyote Species	Coyote Population
"Acadia National Park"	[{"Coyote, Eastern Coyote"}]	45
"Badlands National Park"	[{"Coyote, Coyote Scat"}]	36
"Wind Cave National Park"	[{"Coyote, Coyotes"}]	27

Execution plan:





(ii)

Code:

```

MATCH (s:Species)-[:HAS]->(c:conservationStatus{cs:"Endangered"})
MATCH (s)-[f:FOUND_IN]->(p:Park)

WITH COUNT(s) AS maxSpeciesCount,p
ORDER BY maxSpeciesCount DESC, p.area_in_acres DESC LIMIT 1

MATCH (nearby_park: Park)
WHERE p <> nearby_park

WITH
  maxSpeciesCount,
  p,
  nearby_park,
  point({latitude: p.latitude, longitude: p.longitude}) AS p1,
  point({latitude: nearby_park.latitude,longitude: nearby_park.longitude}) AS p2

WITH p, nearby_park, maxSpeciesCount, point.distance(p1, p2) AS dist
ORDER BY dist ASC
return p.park_name AS `Top parks with endangered species`, nearby_park.park_name AS `Other Parks`, round(dist/1000,2) AS `Distance(KM)`
LIMIT 3;

```

Code explanation:

```

//For this question, we have to understand what exactly we are returning from our query, which is 3 columns/variables
//As long as we understand that, we can find those 3 variables from our CSV
//we need to find parks that contains endangered species, the closest parks near parks of endangered species and the distance
//So we will use a match statement again, if we look into our CSV we can find that there is a column species of concern
//which is a form of enumeration type that you cast on a variable to group, the concept feels the same here.
//we have a group of endangered species that is labeled through the column species of concern that we can access with CSV
//we want to output the parks with the most species of that enumeration, this is not to have the population, but the specific
//this can be found using the relationship between species and conservationStatus, in order for us to access those variables
//then we want to know what species of endangered is found in what parks
//we use another relationship for species and parks, in which then we can use a COUNT clause
//to count the number of species that is endangered, this representation will then be called maxSpeciesCount
//again, WITH clause is used to access the variables, in this case all the columns from parks_a2.csv
//we will then order that first by descending with the area_in_acres, this area_in_acres = row.`Areas in Acre`
//we can then begin to gather parks that is not the current park, and we will call it nearby_park
//a point consists of latitude and longitude, to which one point to the other would be the distance
//from the parks, the longitude and latitude is described as the center point, so we can assume
//that the points are where the parks are located, and the distance between two points will be used for
//point distance calculation, of total distance then we refer that back using the WITH clause after we have
//completed all of our operations and simply return them As a specific name, to top 3 parks with LIMIT clause

```

Results:

The screenshot shows the Neo4j Browser interface. On the left, there's a sidebar with various icons and sections like 'Relationship types' and 'Property keys'. The main area displays a query result table.

Query:

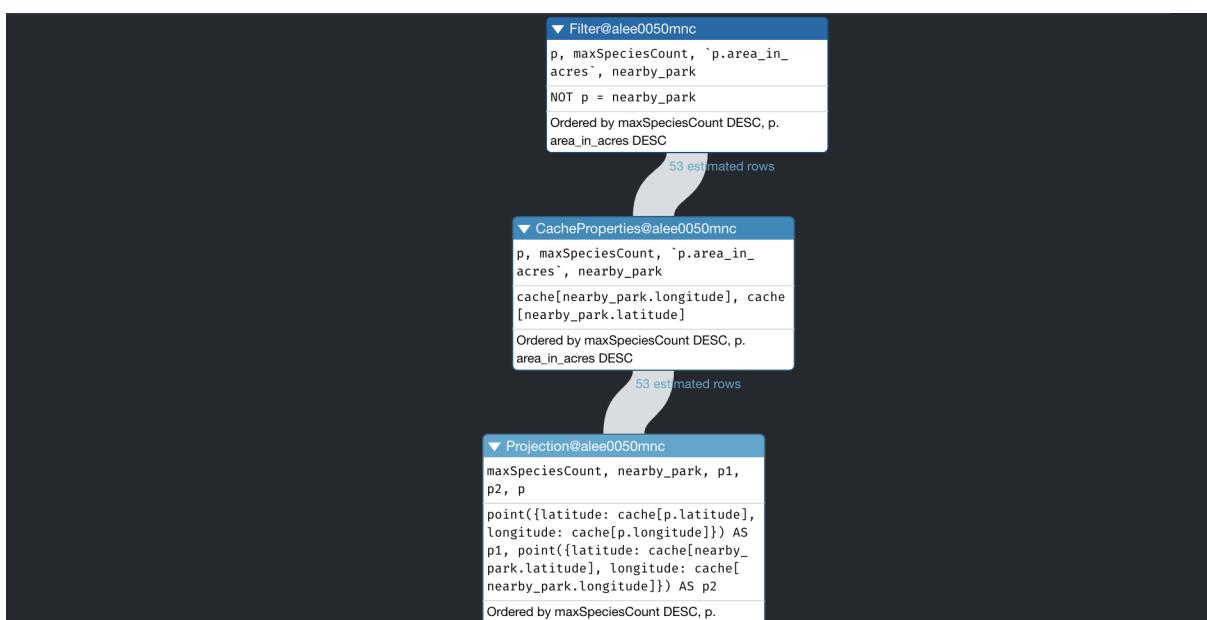
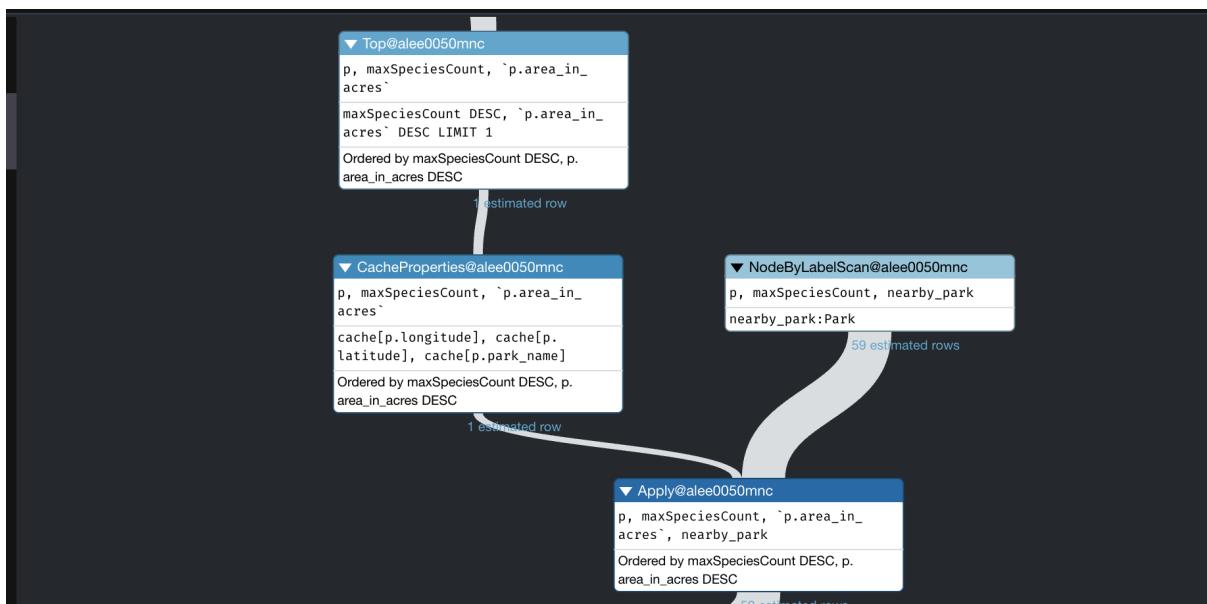
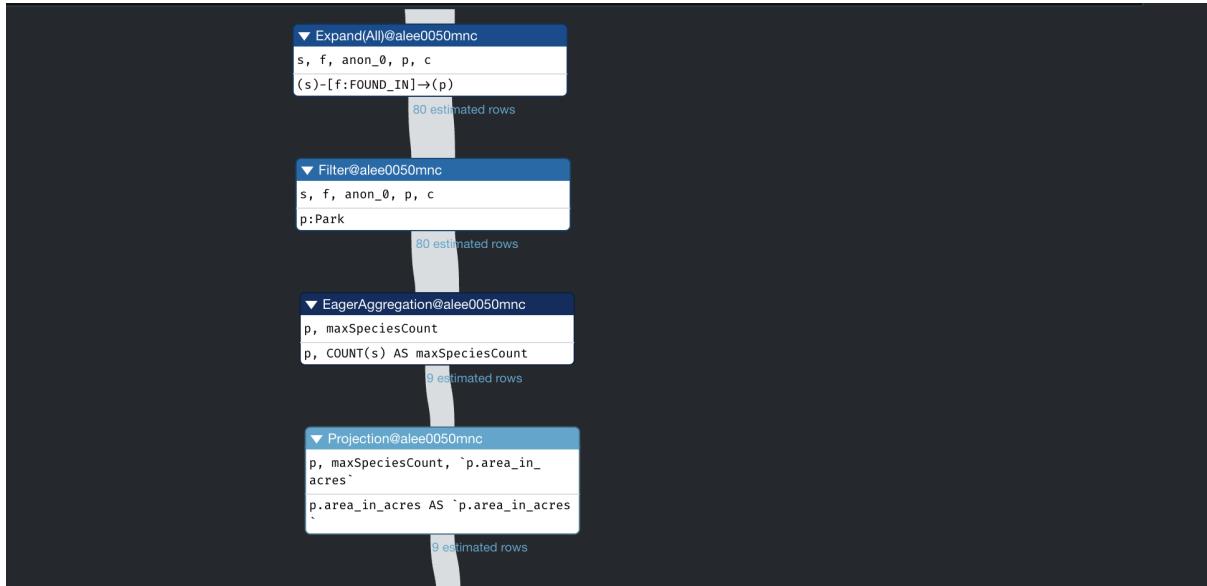
```
alee0050mnc$ MATCH (s:Species)-[:HAS]→(c:conservationStatus{cs:"Endangered"}) MATCH (s)...
```

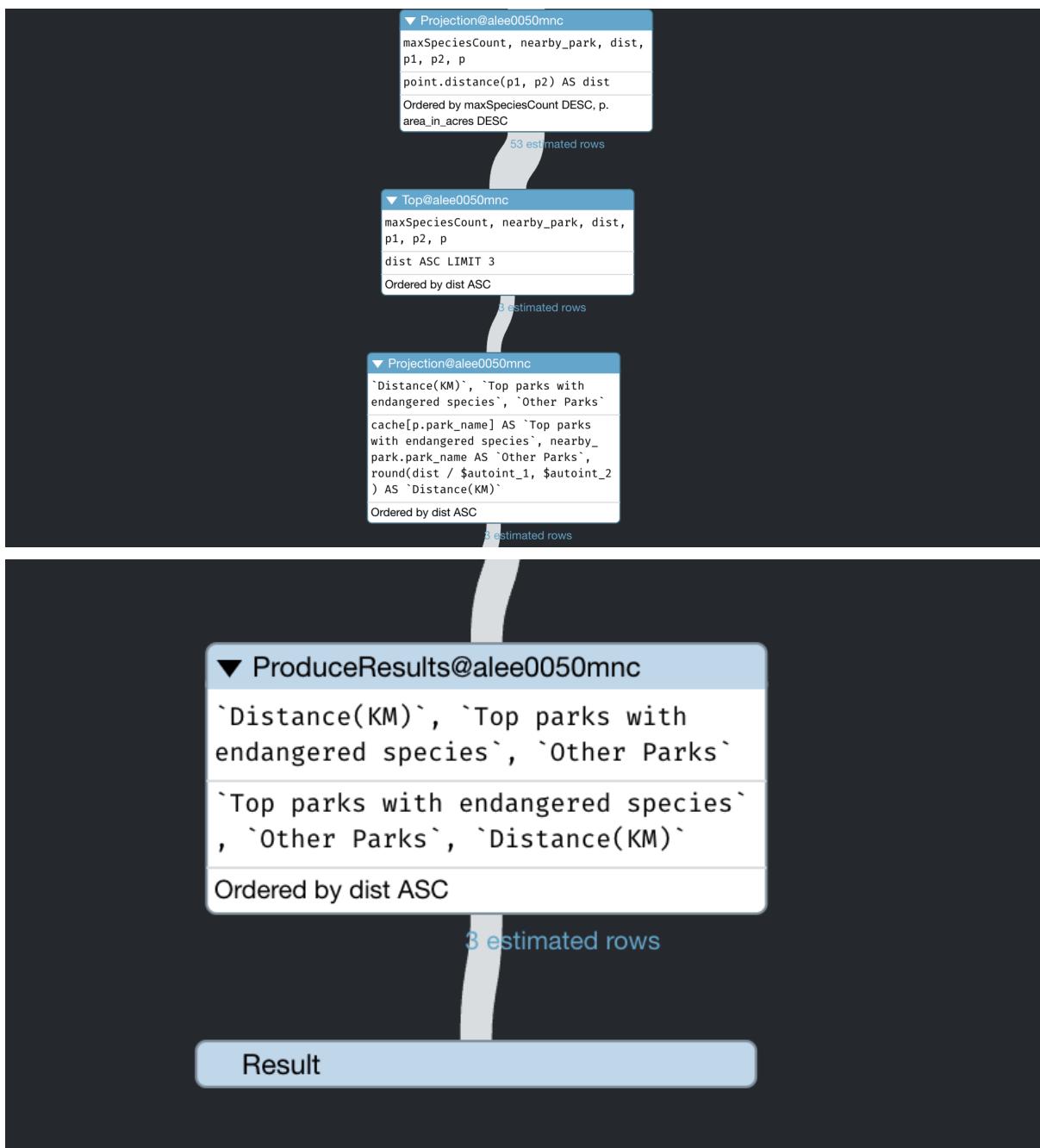
Table Results:

Top parks with endangered species	Other Parks	Distance (KM)
"Channel Islands National Park"	"Sequoia and Kings Canyon National Parks"	277.67
"Channel Islands National Park"	"Pinnacles National Park"	317.2
"Channel Islands National Park"	"Joshua Tree National Park"	326.14

MAX COLUMN WIDTH: [Slider]

Execution plan:





B.3. Modifying the Database:

State Code	Total Area Covered (acres)	Park Codes
"AK"	31159251	[{"WRST", "DENA", "GAAR", "LACL", "KOVA", "KEFJ", "KATM", "GLBA"}]
"CA"	15305852	[{"DEVA", "DEVA", "YOSE", "CHIS", "SEKI", "REDW", "PINN", "LAVO", "JOTR", "DEVA", "YOSE", "SEKI", "LAVO", "CHIS", "REDW", "JOTR", "DEVA", "PINN", "DEVA"]]
"NY"	5059572	[{"YELL", "YELL", "GRTE", "YELL", "GRT", "YELL", "YELL"}]

C1: Plugins (Shortest Path A* algorithm)

Code:

```

LOAD CSV WITH HEADERS FROM "file:///park_connections_a2.csv" AS park_row
MATCH (p:Park {park_code: park_row.Park Code}), (nearby_park:Park {park_code: park_row.Connected Park Code})
WHERE p.latitude IS NOT NULL AND p.longitude IS NOT NULL AND nearby_park.latitude IS NOT NULL AND nearby_park.longitude IS NOT NULL
WITH p, nearby_park,
    point.distance(
        point({latitude:toFloat(p.latitude), longitude:toFloat(p.longitude)}),
        point({latitude:toFloat(nearby_park.latitude), longitude:toFloat(nearby_park.longitude)}))
    ) AS totalDist
MERGE (p)-[:CONNECTED {totalDist: totalDist}]->(nearby_park)
MERGE (nearby_park)-[:CONNECTED {totalDist: totalDist}]->(p);

CALL gds.graph.project(
    'myGraph',
    'Park',
    'CONNECTED',
    {
        relationshipProperties: 'totalDist'
    }
);

MATCH path = (source:Park {park_code: 'PEFO'})-[:CONNECTED*]-(target:Park {park_code: 'GUMO'})
RETURN path LIMIT 1;

MATCH (source:Park {park_code: 'PEFO'}), (target:Park {park_code: 'GUMO'})
CALL gds.shortestPath.dijkstra.stream('myGraph', {
    sourceNode: source,
    targetNode: target,
    relationshipWeightProperty: 'totalDist'
})
YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path
UNWIND nodeIds AS nodeId
MATCH (parkNode) WHERE id(parkNode) = nodeId
RETURN
    gds.util.asNode(sourceNode).park_name AS `Start Park`,
    gds.util.asNode(targetNode).park_name AS `End Park`,
    totalCost AS `Total Travel Distance`,
    collect(parkNode.park_name) AS `Other Parks`;

```

Code Explanation:

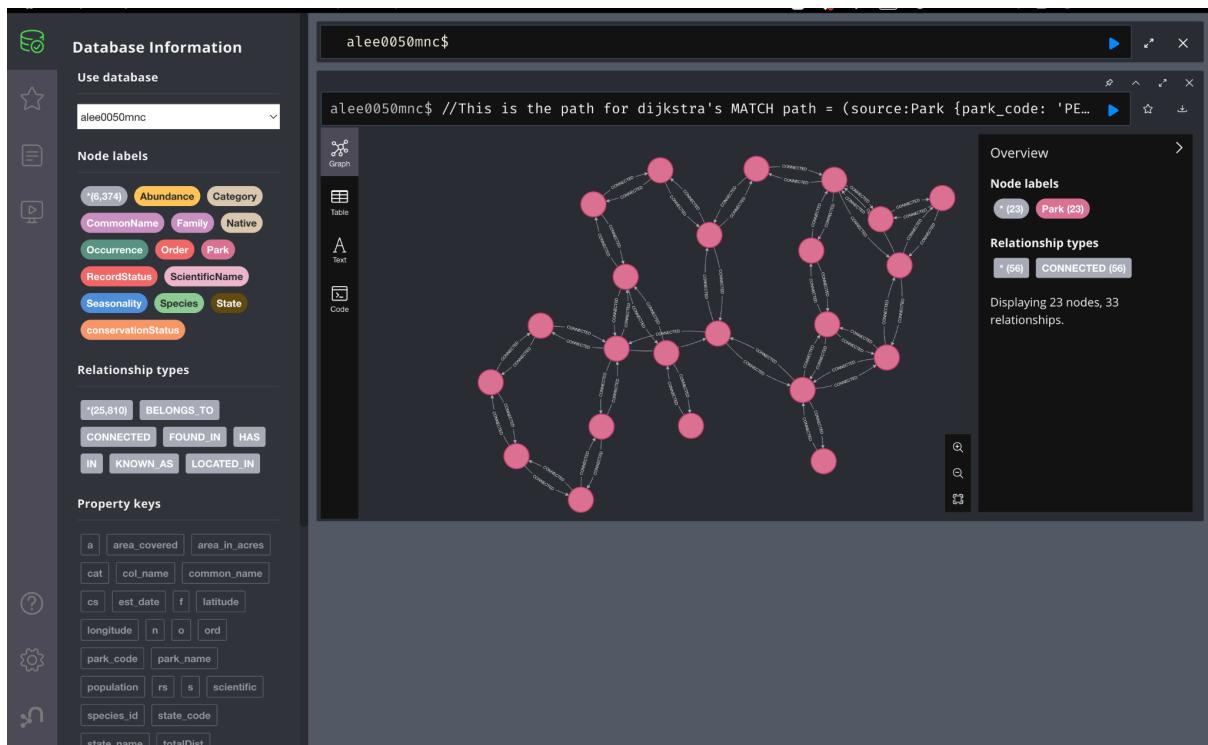
```

//In order to start with this question, a basic understanding of graph algorithm is required
//we already know that each node is a location in this case it is a park, where vertex is considered the weight
//distance between one node to the other. However, we cannot traverse from that point if a node is not Connected
//essentially a node has to be connected with another node, and the total distance is necessary for calculation from that one node
//to the other, we have to use point again with point distance.

//we also need to perform a check to see that the longitude and latitude is not null, because of incomplete records.
//we don't need to be extensive about everything not being null because in this case it is just park to park
//then we use totalDist which is the point.distance calculation from current park to nearby_park
//put that in :CONNECTED precisely because we want to calculate the total distance or the vertexes between each node
//from start to end and we will use that later on to calculate the total distance

//we can do a match statement to gain a visualisation of all the possibilities that you can go starting from PEFO to GUMO
//this is before we implement shortestpath dijkstra's algorithm
//how dijkstra's algorithm work, it is a greedy algorithm that will always go for the minimum weight to achieve going from one point
//to the other, which makes it very efficient, the complexity of dijkstra's is O(V + ElogV) where V = Vertex and E = Edges
//so it uses a form of priority queue to prioritise min using a heap data structure, we don't need to implement that because our library
//already takes care of it.
//our source node and target node is just start point and end point where totalCost is considered the weight, in this case the Distance
//that it takes to transverse from one point to the other.
//when dijkstra's algorithm is successfully executed, the path is only one because there is only one shortest path, so the path node
//is that one path with minimum distance that it has transversed.
//it also has to be bidirectional so you can come back from endpark to startpark as well

```



Results:

The screenshot shows two windows from the Neo4j desktop application. On the left is the 'Database Information' window, which includes sections for 'Use database' (set to 'alee0050mnc'), 'Node labels' (listing various categories like Abundance, Category, CommonName, Family, Native, Occurrence, Order, Park, RecordStatus, ScientificName, Seasonality, Species, State, and conservationStatus), 'Relationship types' (listing BELONGS_TO, CONNECTED, FOUND_IN, HAS, IN, KNOWN_AS, and LOCATED_IN), and 'Property keys' (listing area_covered, area_in_acres, cat, col_name, common_name, cs, est_date, f, latitude, longitude, n, o, ord, park_code, park_name, population, rs, s, scientific, species_id, state_code, state_name, and totalDist). On the right is the 'alee0050mnc\$' browser window displaying the results of a Cypher query:

```
alee0050mnc$ MATCH (source:Park {park_code: 'PEFO'}), (target:Park {park_code: 'GUMO'}) ...
```

The results table has columns: Start Park, End Park, Total Travel Distance, and Other Parks. One row is shown:

Start Park	End Park	Total Travel Distance	Other Parks
"Petrified Forest National Park"	"Guadalupe Mountains National Park"	15569492.695301317	["Petrified Forest National Park", "Great Basin National Park", "Pinnacles National Park", "Denali National Park and Preserve", "Biscayne National Park", "Wind Cave National Park", "Guadalupe Mountains National Park"]

C.2. Neo4j Drivers

Code:

```

41 WITH COUNT(s) AS maxSpeciesCount,
42 ORDER BY maxSpeciesCount DESC, p.area_in_acres DESC LIMIT 1
43
44 MATCH (nearby_park: Park)
45 WHERE p <-> nearby_park
46
47 WITH
48   maxSpeciesCount,
49   p,
50   nearby_park,
51   point({latitude: p.latitude, longitude: p.longitude}) AS p1,
52   point({latitude: nearby_park.latitude,longitude: nearby_park.longitude}) AS p2
53
54 WITH p, nearby_park, maxSpeciesCount, point.distance(p1, p2) AS dist
55 ORDER BY dist ASC
56 return p.park_name AS `Top parks with endangered species`, nearby_park.park_name AS `Other Parks`, round(dist/1000,2) AS `Distance(KM)`
57 LIMIT 3;
58 """
59
60 for record in records:
61     print(record)
62
63 print("\n")
64 print("B3\n")
65
66 #see explanation above.
67 records, summary, keys = driver.execute_query("""
68 MATCH (p:Park)-[r:LOCATED_IN]->(s:State)
69 WITH s.state_code AS `State Code`,
70      SUM(p.area_in_acres) AS `Total Area Covered (acres)` ,
71      collect(p.park_code) AS `Park Codes`
72 ORDER BY `Total Area Covered (acres)` DESC
73 RETURN `State Code`, `Total Area Covered (acres)`, `Park Codes`
74 LIMIT 3;
75 """
76
77 for record in records:
78     print(record)

```

Results:

```

1  from neo4j import GraphDatabase
2
3  #this is the connection port of which the server access is obtained through :SERVER STATUS
4  URI = "bolt://localhost:7689"
5  #credentials, as we are using the default user neo4j <username> and Pta59ypt123 <password>
6  AUTH = ("neo4j", "Pta59ypt123")
7  driver = GraphDatabase.driver(URI, auth=AUTH)
8  #this is initializing the driver object using neo4j import
9
10
11 #simple print statement to indicate which question it is
12 print("B1\n")
13
14 #using the to be fair we only need records, but this is saved for future implementation where a developer would want to measure
15 #the efficiency through summary and the keys is just used to return default object
16 records, summary, keys = driver.execute_query("""
17     // MATCH all the species with the relationship known_as
18     MATCH (s:Species)-[r:KNOWN_AS]->(c:CommonName)
19     // match species found_in park relationship
20     MATCH (s)-[f:FOUND_IN]->(p:Park)
21     // common name should contain Coyote
22     WHERE c.common_name CONTAINS 'Coyote'
23
24     WITH p, c.common_name AS comName, SUM(f.population) AS tot
25     // Filter out single names by checking for the existence of a comma
26     WHERE comName CONTAINS ','
27     RETURN p.park_name AS 'Park Name', collect(comName) AS 'Coyote Species', tot AS 'Coyote Population'
28     ORDER BY tot DESC;
29     | """
30     )
31     #this is just used to print all the records
32     for record in records:
33         print(record)
34
35     print("\n")
36     print("B2\n")
37
38     #this is repetitive, see explanation above
39     records, summary, keys = driver.execute_query("""
40         MATCH (s)-[f:FOUND_IN]->(p:Park)
41
42         WITH COUNT(s) AS maxSpeciesCount,p

```

The screenshot shows a code editor with two tabs open: `alee0050_task_B.cypher` and `alee0050_task_C2.py`. The `C2.py` tab contains Python code using the `neo4j` library to interact with a Neo4j database. The code performs two main queries. The first query finds parks that contain the common name 'Coyote' and calculates the total population of coyotes in those parks. The second query finds parks that have an 'ENDANGERED' conservation status and counts the number of species found in each park. The `PROBLEMS` panel shows that there are no problems detected in the workspace.