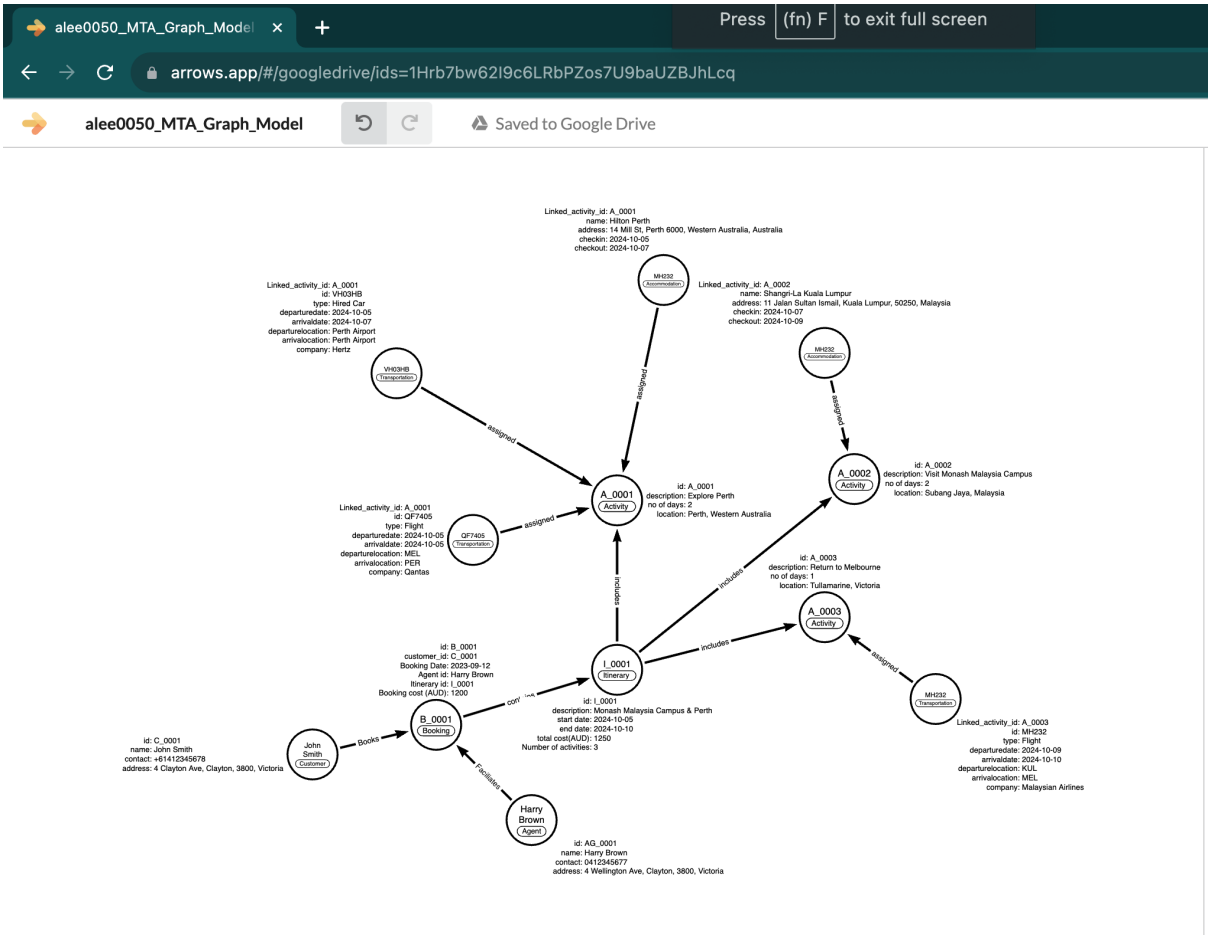
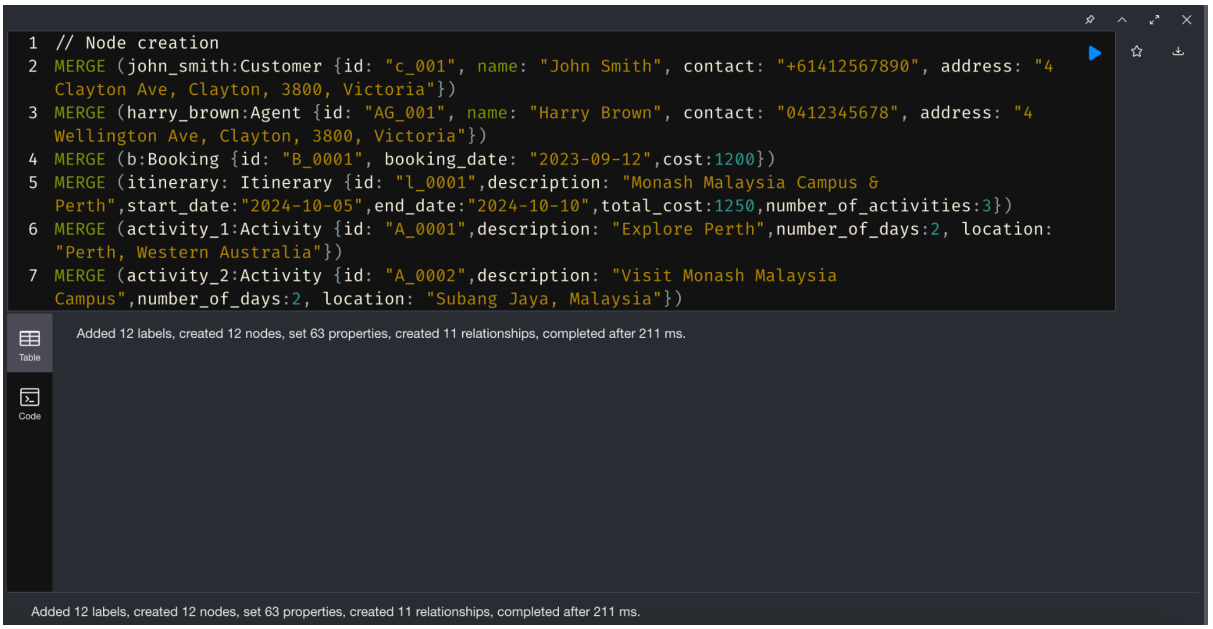
A1:

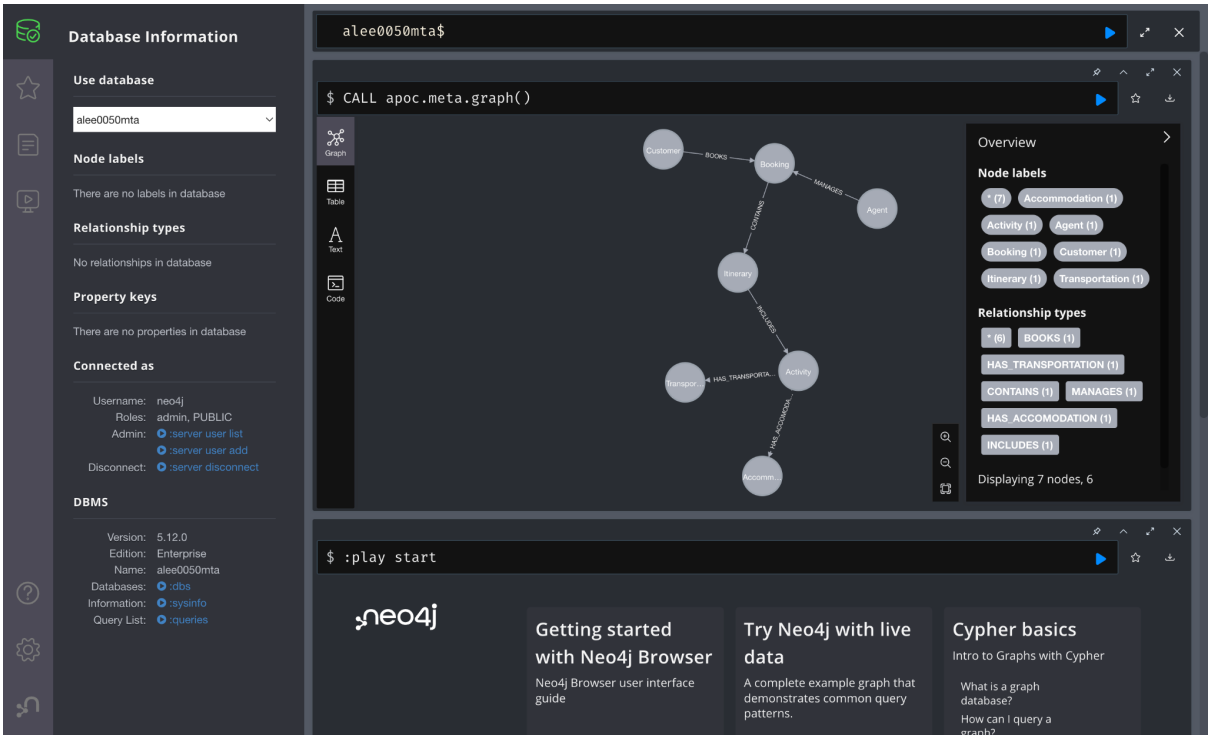
https://arrows.app/#/googledrive/ids=1Hrb7bw62l9c6LRbPZos7U9baUZBJhLcq



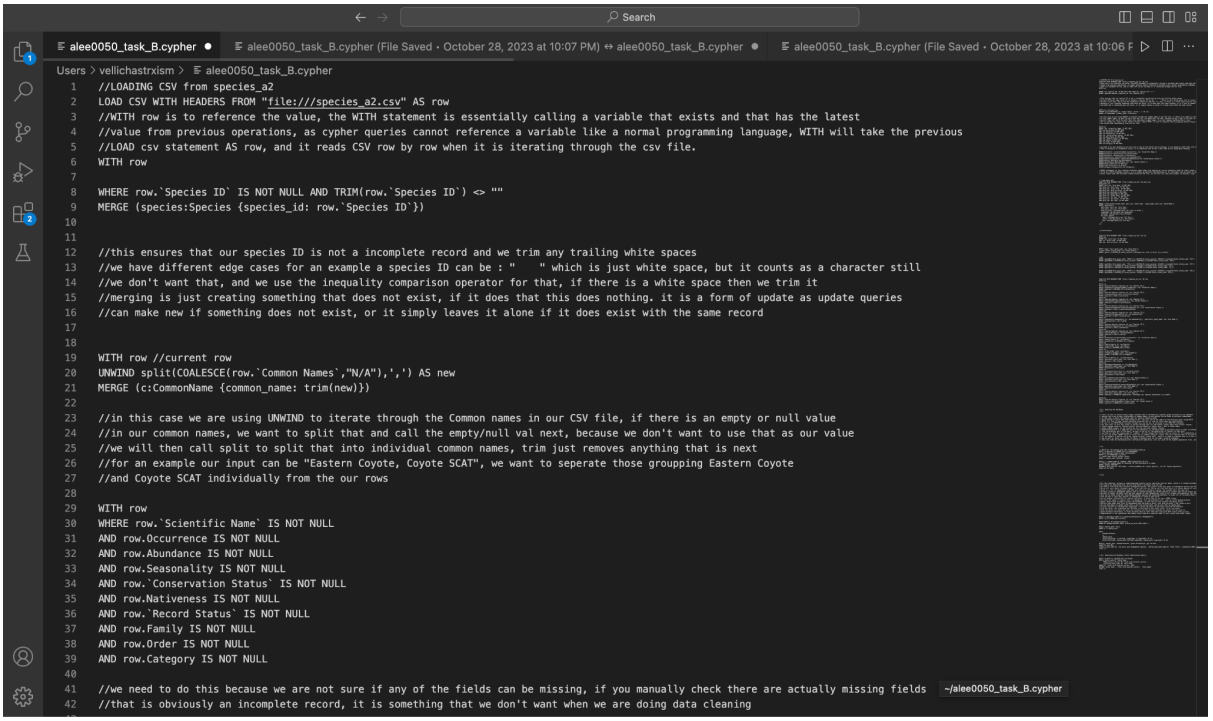
A1 results:

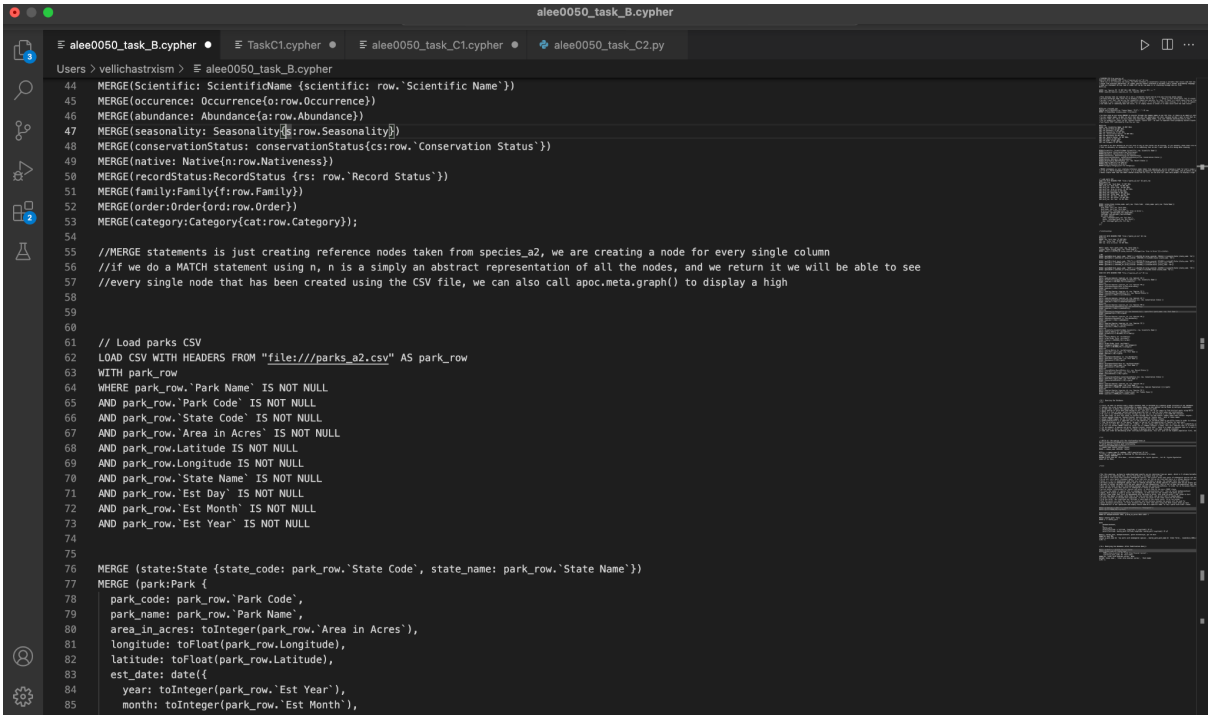


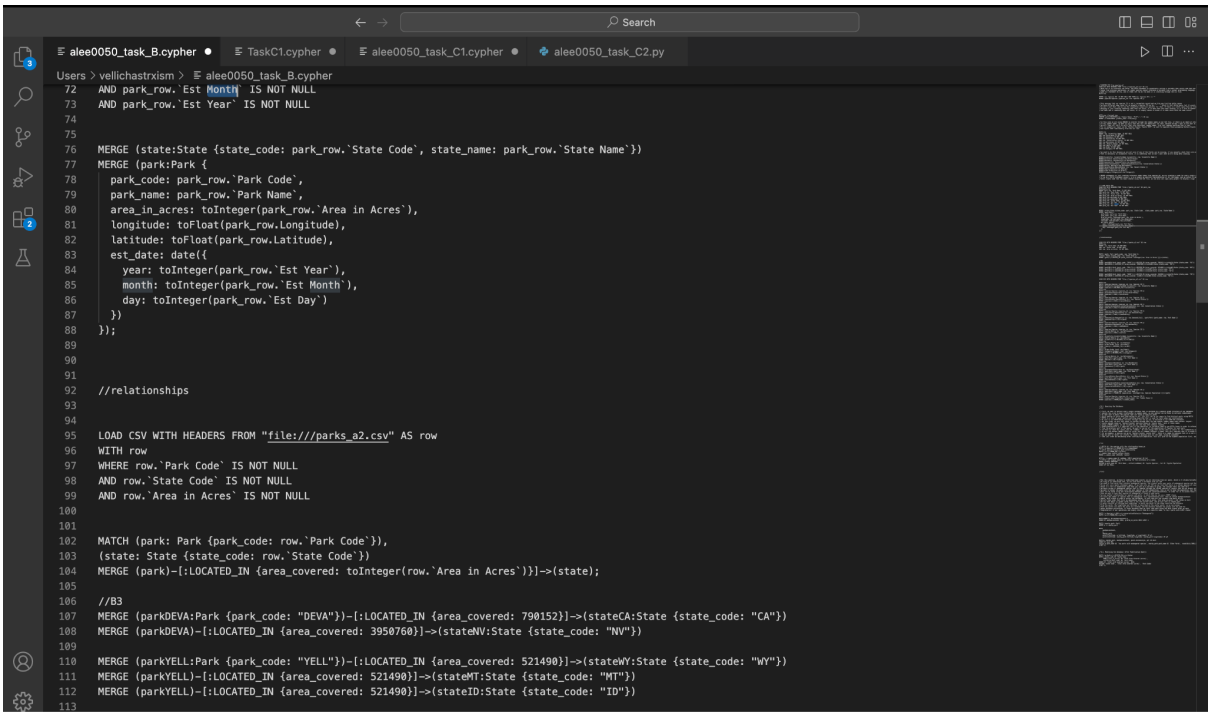
APOC calls:

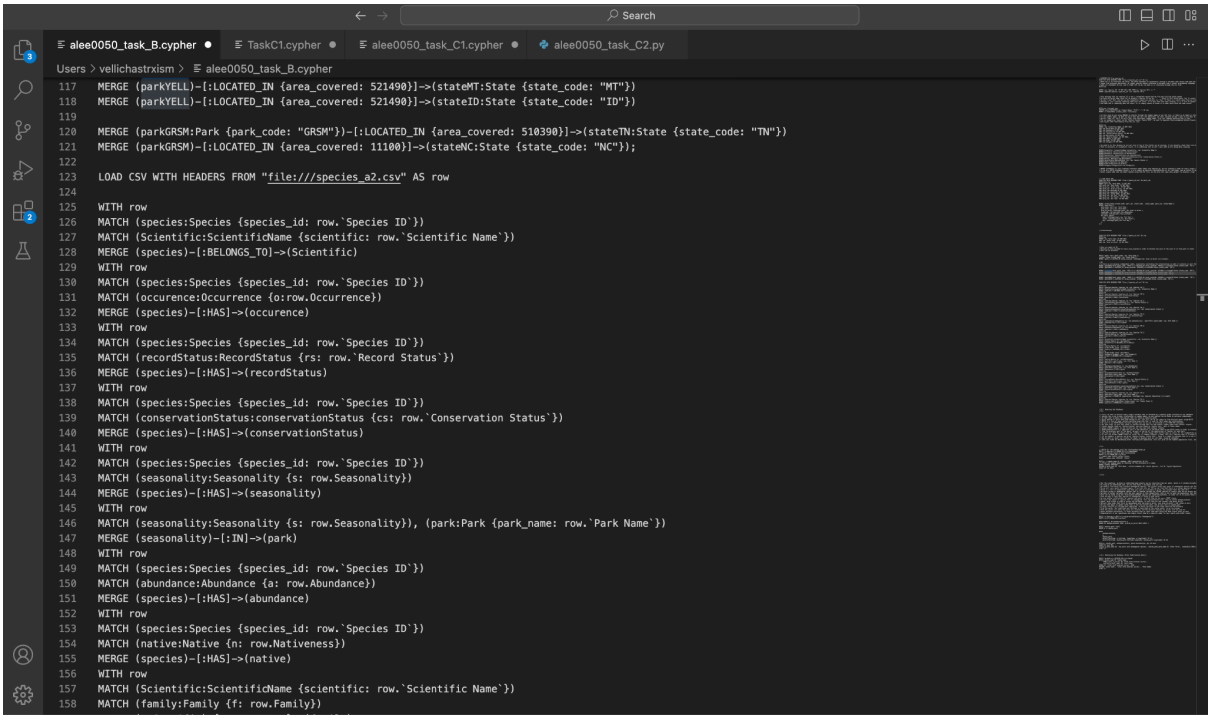


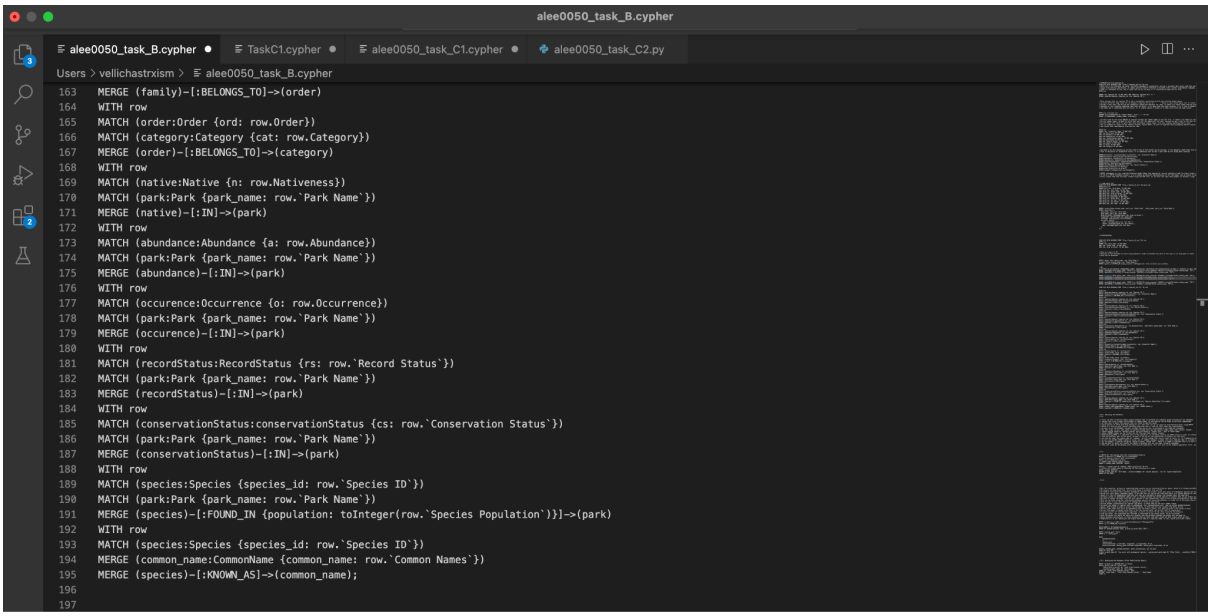
B.1. Creating the Database:



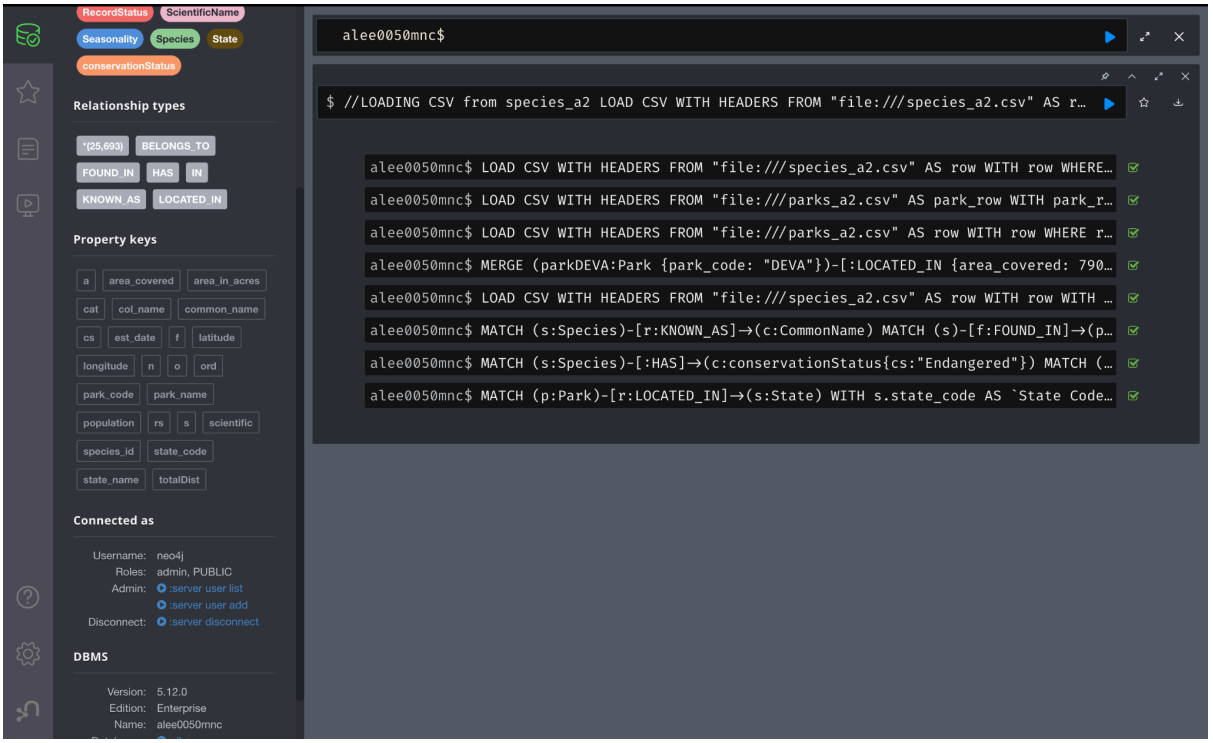

alee0050_task_B.cypher
alee0050_task_B.cypher
TaskC1.cypher
alee0050_task_C1.cypher
alee0050_task_C2.py
Users > vellichastrxism > alee0050_task_B.cypher
MERGE(Scientific: ScientificName {scientific: row.`Scientific Name`})
MERGE(occurence: Occurrence{o:row.Occurrence})
MERGE(abundance: Abundance{a:row.Abundance})
MERGE(seasonality: Seasonality{s:row.Seasonality})
MERGE(conservationStatus: conservationStatus{cs:row.`Conservation Status`})
MERGE(native: Native{n:row.Nativeness})
MERGE(recordStatus:RecordStatus {rs: row.`Record Status`})
MERGE(family:Family{f:row.Family})
MERGE(order:Order{ord:row.Order})
MERGE(category:Category{cat:row.Category});
//MERGE statements is just creating reference nodes taken from species_a2, we are creating a node for every single column
//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
//every single node that has been created using the CSV file, we can also call apoc.meta.graph() to display a high
// Load parks CSV
LOAD CSV WITH HEADERS FROM "file:///parks_a2.csv" AS park_row
WITH park_row
WHERE park_row.`Park Name` IS NOT NULL
AND park_row.`Park Code` IS NOT NULL
AND park_row.`State Code` IS NOT NULL
AND park_row.`Area in Acres` IS NOT NULL
AND park_row.Latitude IS NOT NULL
AND park_row.Longitude IS NOT NULL
AND park_row.`State Name` IS NOT NULL
AND park_row.`Est Day` IS NOT NULL
AND park_row.`Est Month` IS NOT NULL
AND park_row.`Est Year` IS NOT NULL
MERGE (state:State {state_code: park_row.`State Code`, state_name: park_row.`State Name`})
MERGE (park:Park {
park_code: park_row.`Park Code`,
park_name: park_row.`Park Name`,
area_in_acres: toInteger(park_row.`Area in Acres`),
longitude: toFloat(park_row.Longitude),
latitude: toFloat(park_row.Latitude),
est_date: date({
year: toInteger(park_row.`Est Year`),
month: toInteger(park_row.`Est Month`),


Search
alee0050_task_B.cypher
TaskC1.cypher
alee0050_task_C1.cypher
alee0050_task_C2.py
Users > vellichastrxism > alee0050_task_B.cypher
AND park_row.`Est Month` IS NOT NULL
AND park_row.`Est Year` IS NOT NULL
MERGE (state:State {state_code: park_row.`State Code`, state_name: park_row.`State Name`})
MERGE (park:Park {
park_code: park_row.`Park Code`,
park_name: park_row.`Park Name`,
area_in_acres: toInteger(park_row.`Area in Acres`),
longitude: toFloat(park_row.Longitude),
latitude: toFloat(park_row.Latitude),
est_date: date({
year: toInteger(park_row.`Est Year`),
month: toInteger(park_row.`Est Month`),
day: toInteger(park_row.`Est Day`)
})
});
//relationships
LOAD CSV WITH HEADERS FROM "file:///parks_a2.csv" AS row
WITH row
WHERE row.`Park Code` IS NOT NULL
AND row.`State Code` IS NOT NULL
AND row.`Area in Acres` IS NOT NULL
MATCH (park: Park {park_code: row.`Park Code`}),
(state: State {state_code: row.`State Code`})
MERGE (park)-[:LOCATED_IN {area_covered: toInteger(row.`Area in Acres`)}]->(state);
//B3
MERGE (parkDEVA:Park {park_code: "DEVA"})-[:LOCATED_IN {area_covered: 790152}]->(stateCA:State {state_code: "CA"})
MERGE (parkDEVA)-[:LOCATED_IN {area_covered: 3950760}]->(stateNV:State {state_code: "NV"})
MERGE (parkYELL:Park {park_code: "YELL"})-[:LOCATED_IN {area_covered: 521490}]->(stateWY:State {state_code: "WY"})
MERGE (parkYELL)-[:LOCATED_IN {area_covered: 521490}]->(stateMT:State {state_code: "MT"})
MERGE (parkYELL)-[:LOCATED_IN {area_covered: 521490}]->(stateID:State {state_code: "ID"})

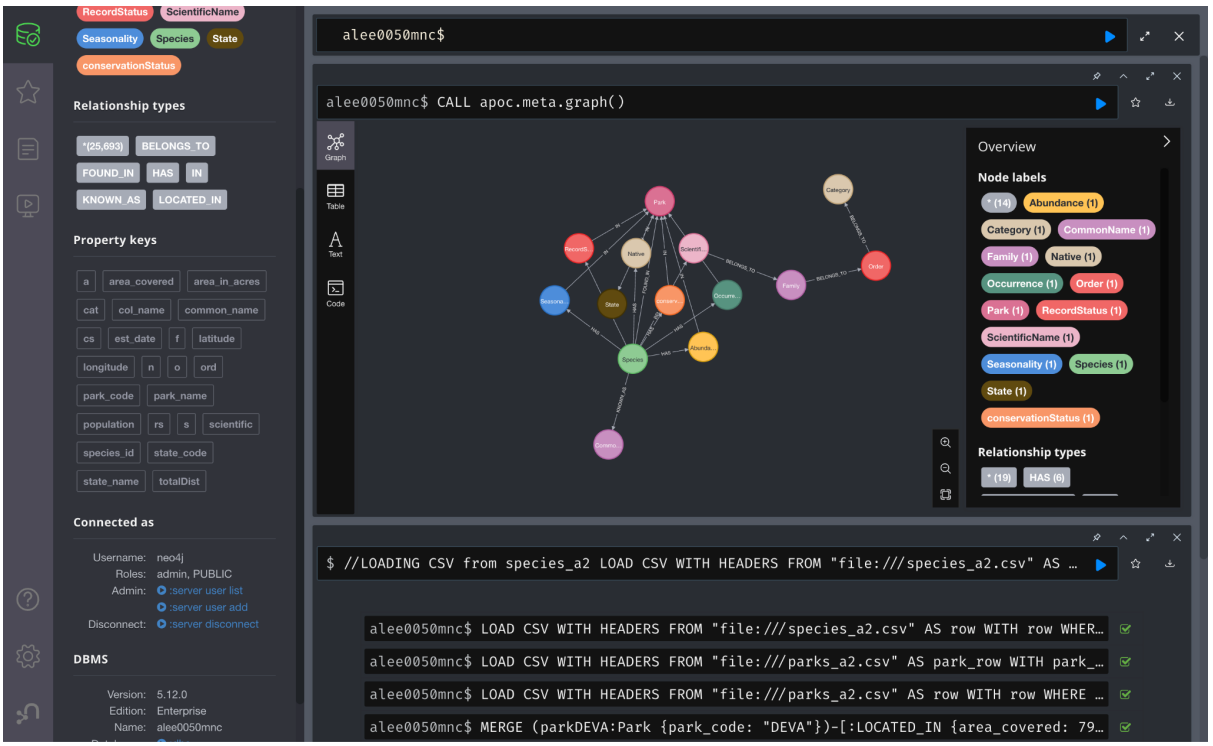

Search
alee0050_task_B.cypher
TaskC1.cypher
alee0050_task_C1.cypher
alee0050_task_C2.py
Users > vellichastrxism > alee0050_task_B.cypher
MERGE (parkYELL)-[:LOCATED_IN {area_covered: 521490}]->(stateMT:State {state_code: "MT"})
MERGE (parkYELL)-[:LOCATED_IN {area_covered: 521490}]->(stateID:State {state_code: "ID"})
MERGE (parkGRSM:Park {park_code: "GRSM"})-[:LOCATED_IN {area_covered: 510390}]->(stateTN:State {state_code: "TN"})
MERGE (parkGRSM)-[:LOCATED_IN {area_covered: 11100}]->(stateNC:State {state_code: "NC"});
LOAD CSV WITH HEADERS FROM "file:///species_a2.csv" AS row
WITH row
MATCH (species:Species {species_id: row.`Species ID`})
MATCH (Scientific:ScientificName {scientific: row.`Scientific Name`})
MERGE (species)-[:BELONGS_TO]->(Scientific)
WITH row
MATCH (species:Species {species_id: row.`Species ID`})
MATCH (occurence:Occurrence {o:row.Occurrence})
MERGE (species)-[:HAS]->(occurence)
WITH row
MATCH (species:Species {species_id: row.`Species ID`})
MATCH (recordStatus:RecordStatus {rs: row.`Record Status`})
MERGE (species)-[:HAS]->(recordStatus)
WITH row
MATCH (species:Species {species_id: row.`Species ID`})
MATCH (conservationStatus:conservationStatus {cs: row.`Conservation Status`})
MERGE (species)-[:HAS]->(conservationStatus)
WITH row
MATCH (species:Species {species_id: row.`Species ID`})
MATCH (seasonality:Seasonality {s: row.Seasonality})
MERGE (species)-[:HAS]->(seasonality)
WITH row
MATCH (seasonality:Seasonality {s: row.Seasonality}), (park:Park {park_name: row.`Park Name`})
MERGE (seasonality)-[:IN]->(park)
WITH row
MATCH (species:Species {species_id: row.`Species ID`})
MATCH (abundance:Abundance {a: row.Abundance})
MERGE (species)-[:HAS]->(abundance)
WITH row
MATCH (species:Species {species_id: row.`Species ID`})
MATCH (native:Native {n: row.Nativeness})
MERGE (species)-[:HAS]->(native)
WITH row
MATCH (Scientific:ScientificName {scientific: row.`Scientific Name`})
MATCH (family:Family {f: row.Family})


alee0050_task_B.cypher
alee0050_task_B.cypher
TaskC1.cypher
alee0050_task_C1.cypher
alee0050_task_C2.py
Users > vellichastrxism > alee0050_task_B.cypher
MERGE (family)-[:BELONGS_TO]->(order)
WITH row
MATCH (order:Order {ord: row.Order})
MATCH (category:Category {cat: row.Category})
MERGE (order)-[:BELONGS_TO]->(category)
WITH row
MATCH (native:Native {n: row.Nativeness})
MATCH (park:Park {park_name: row.`Park Name`})
MERGE (native)-[:IN]->(park)
WITH row
MATCH (abundance:Abundance {a: row.Abundance})
MATCH (park:Park {park_name: row.`Park Name`})
MERGE (abundance)-[:IN]->(park)
WITH row
MATCH (occurence:Occurrence {o: row.Occurrence})
MATCH (park:Park {park_name: row.`Park Name`})
MERGE (occurence)-[:IN]->(park)
WITH row
MATCH (recordStatus:RecordStatus {rs: row.`Record Status`})
MATCH (park:Park {park_name: row.`Park Name`})
MERGE (recordStatus)-[:IN]->(park)
WITH row
MATCH (conservationStatus:conservationStatus {cs: row.`Conservation Status`})
MATCH (park:Park {park_name: row.`Park Name`})
MERGE (conservationStatus)-[:IN]->(park)
WITH row
MATCH (species:Species {species_id: row.`Species ID`})
MATCH (park:Park {park_name: row.`Park Name`})
MERGE (species)-[:FOUND_IN {population: toInteger(row.`Species Population`)}]->(park)
WITH row
MATCH (species:Species {species_id: row.`Species ID`})
MERGE (common_name:CommonName {common_name: row.`Common Names`})
MERGE (species)-[:KNOWN_AS]->(common_name);

Results:



Visualisation of graph nodes and relationships when we call apoc.



B.2. Querying the Database:

(i)

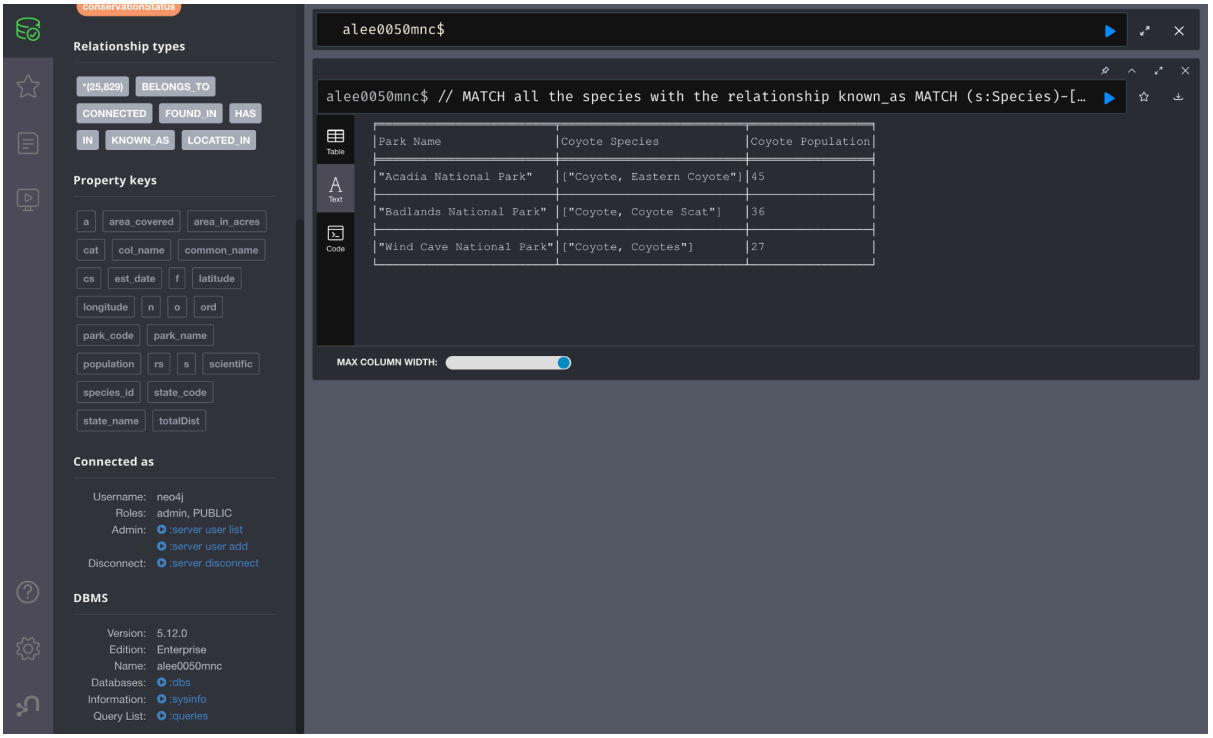
Code:

```
// 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;
```

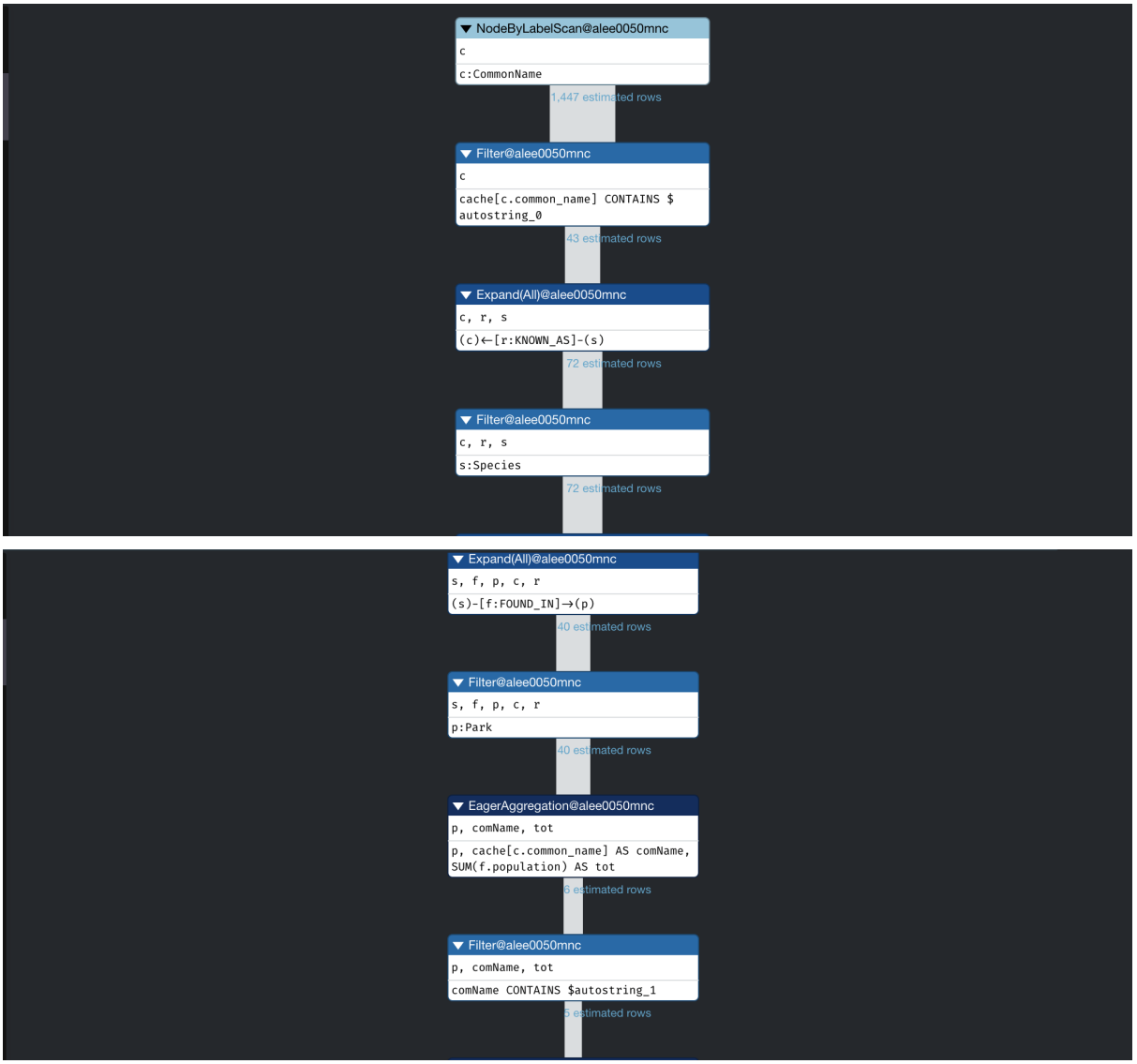
Code explanation:

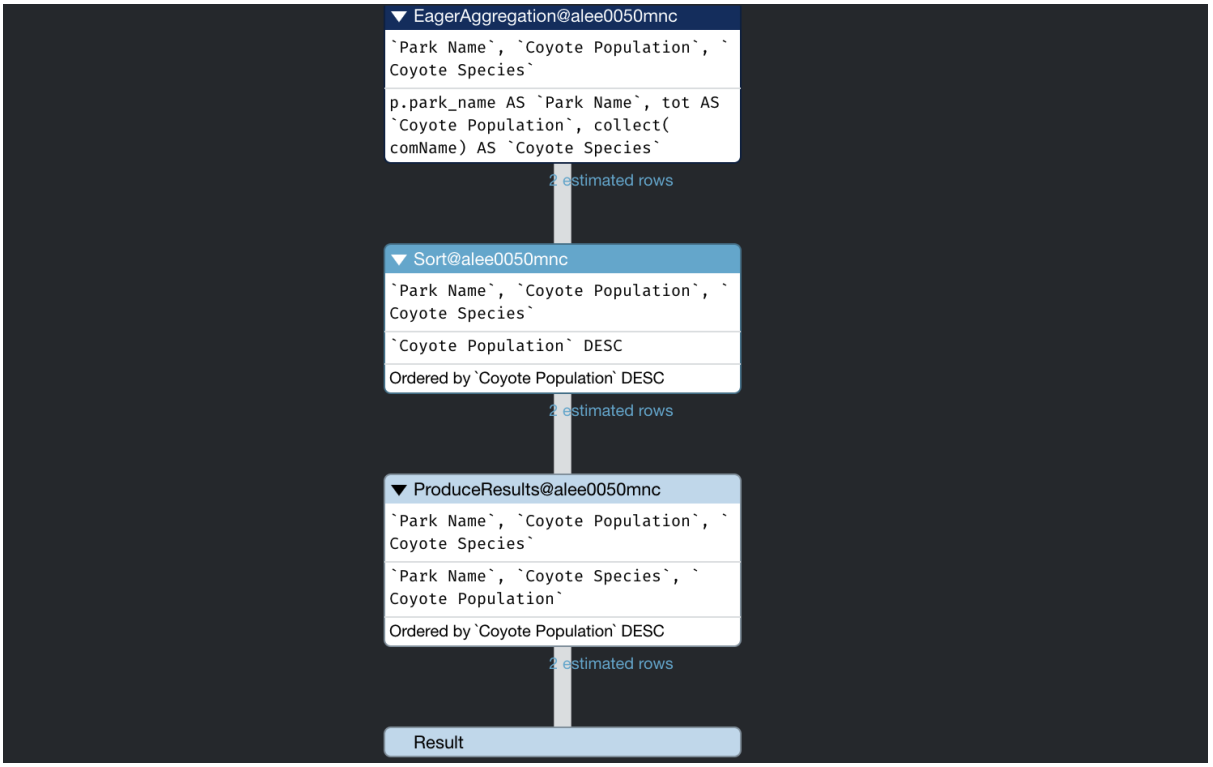
```
// First, we want to extract every single instance that is dictated by a general graph structure of our database
// species has a one to many relationship to common names, as one species can be known by multiple commonnames
// we then want to match the species that are found in specific parks
// 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:



Execution plan:





(ii)

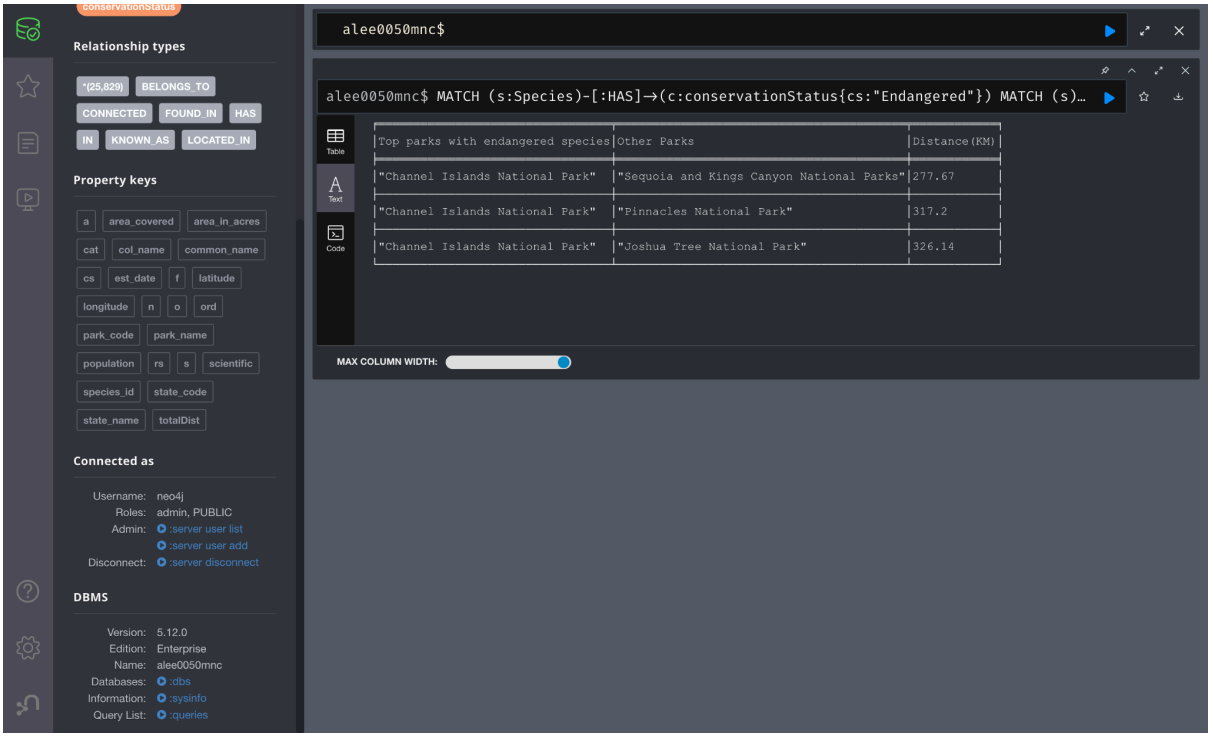
Code:

```
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 longtitude, to which one point to the other would be the distance
//from the parks, the longtitude 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
```

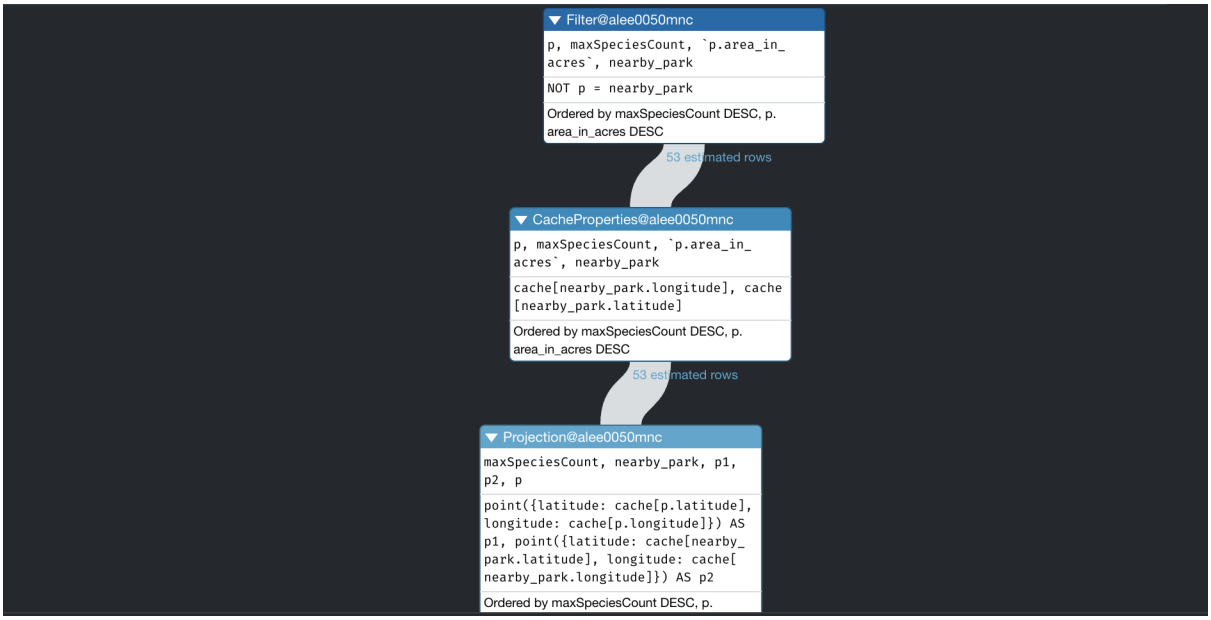
Results:

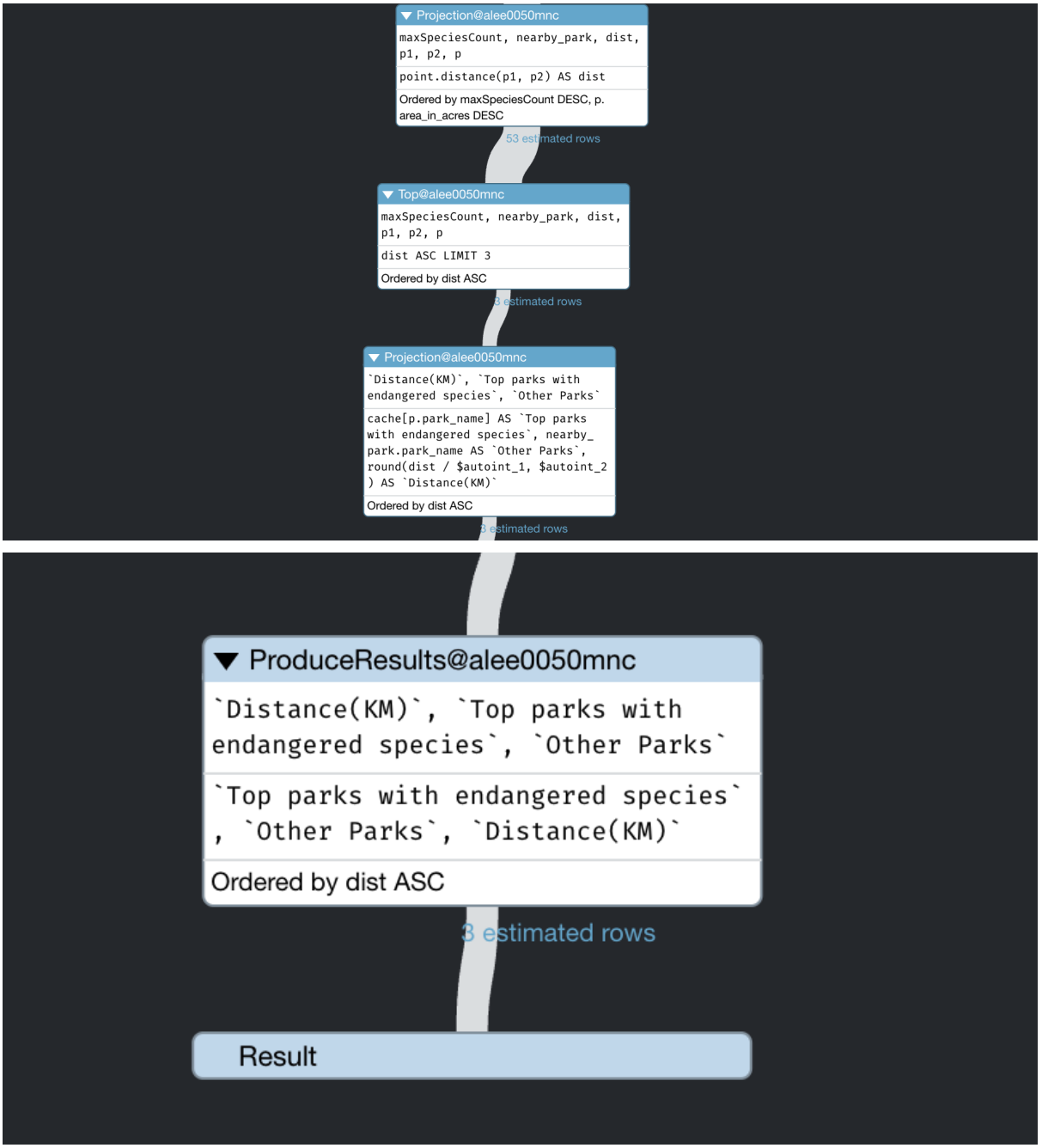


Execution plan:

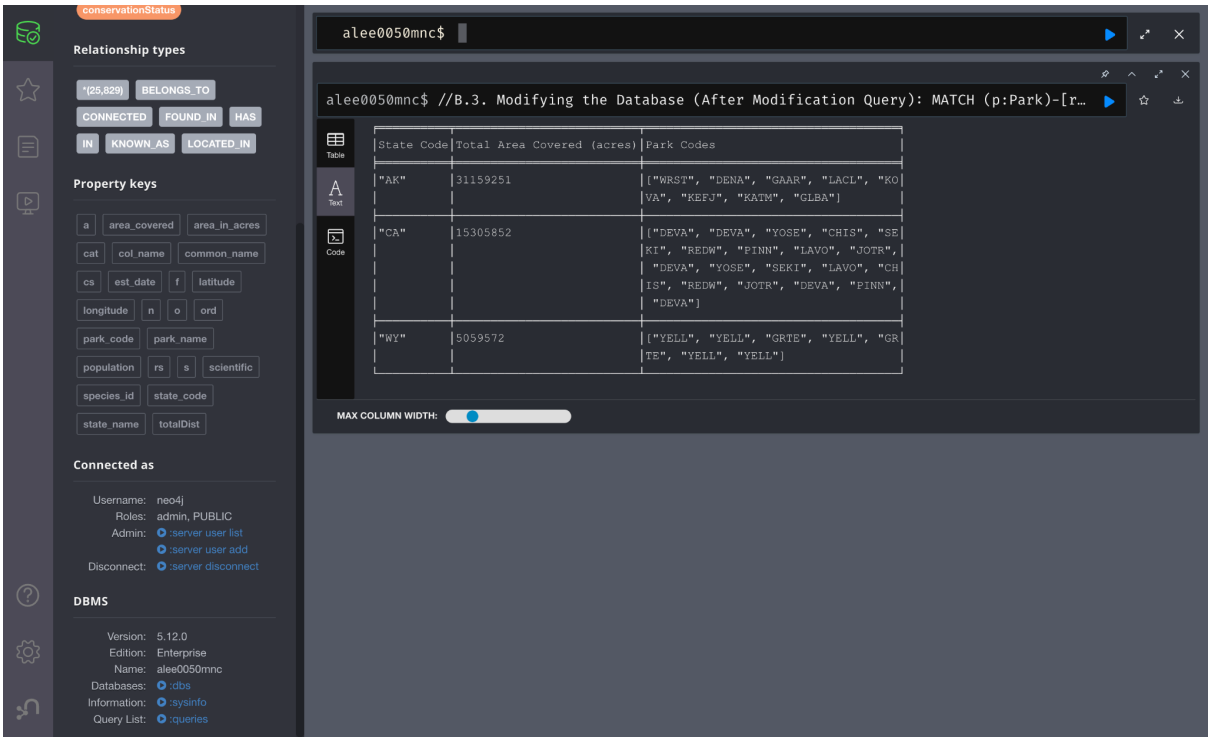

Expand(All)@alee0050mnc
s, f, anon_0, p, c
(s)-[f:FOUND_IN]→(p)
80 estimated rows
Filter@alee0050mnc
s, f, anon_0, p, c
p:Park
80 estimated rows
EagerAggregation@alee0050mnc
p, maxSpeciesCount
p, COUNT(s) AS maxSpeciesCount
9 estimated rows
Projection@alee0050mnc
p, maxSpeciesCount, `p.area_in_acres`
p.area_in_acres AS `p.area_in_acres`
9 estimated rows


Top@alee0050mnc
p, maxSpeciesCount, `p.area_in_acres`
maxSpeciesCount DESC, `p.area_in_acres` DESC LIMIT 1
Ordered by maxSpeciesCount DESC, p.area_in_acres DESC
1 estimated row
CacheProperties@alee0050mnc
p, maxSpeciesCount, `p.area_in_acres`
cache[p.longitude], cache[p.latitude], cache[p.park_name]
Ordered by maxSpeciesCount DESC, p.area_in_acres DESC
1 estimated row
NodeByLabelScan@alee0050mnc
p, maxSpeciesCount, nearby_park
nearby_park:Park
59 estimated rows
Apply@alee0050mnc
p, maxSpeciesCount, `p.area_in_acres`, nearby_park
Ordered by maxSpeciesCount DESC, p.area_in_acres DESC
59 estimated rows


Filter@alee0050mnc
p, maxSpeciesCount, `p.area_in_acres`, nearby_park
NOT p = nearby_park
Ordered by maxSpeciesCount DESC, p.area_in_acres DESC
53 estimated rows
CacheProperties@alee0050mnc
p, maxSpeciesCount, `p.area_in_acres`, nearby_park
cache[nearby_park.longitude], cache[nearby_park.latitude]
Ordered by maxSpeciesCount DESC, p.area_in_acres DESC
53 estimated rows
Projection@alee0050mnc
maxSpeciesCount, nearby_park, p1, p2, p
point({latitude: cache[p.latitude], longitude: cache[p.longitude]}) AS p1, point({latitude: cache[nearby_park.latitude], longitude: cache[nearby_park.longitude]}) AS p2
Ordered by maxSpeciesCount DESC, p.


Projection@alee0050mnc
maxSpeciesCount, nearby_park, dist, p1, p2, p
point.distance(p1, p2) AS dist
Ordered by maxSpeciesCount DESC, p.area_in_acres DESC
53 estimated rows
Top@alee0050mnc
maxSpeciesCount, nearby_park, dist, p1, p2, p
dist ASC LIMIT 3
Ordered by dist ASC
3 estimated rows
Projection@alee0050mnc
`Distance(KM)`, `Top parks with endangered species`, `Other Parks`
cache[p.park_name] AS `Top parks with endangered species`, nearby_park.park_name AS `Other Parks`, round(dist / $autoint_1, $autoint_2) AS `Distance(KM)`
Ordered by dist ASC
3 estimated rows
ProduceResults@alee0050mnc
`Distance(KM)`, `Top parks with endangered species`, `Other Parks`
`Top parks with endangered species`, `Other Parks`, `Distance(KM)`
Ordered by dist ASC
3 estimated rows
Result

B.3. Modifying the Database:



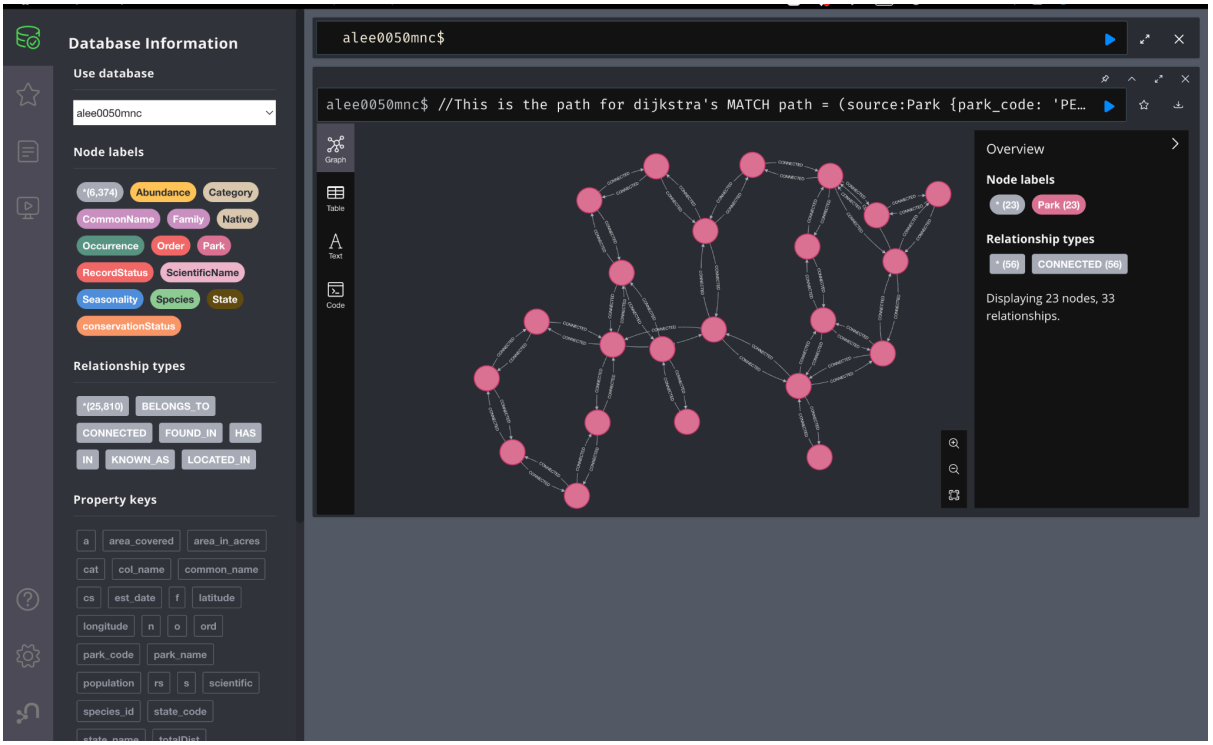
C1: Plugins (Shortest Path A* algorithm)

Code:

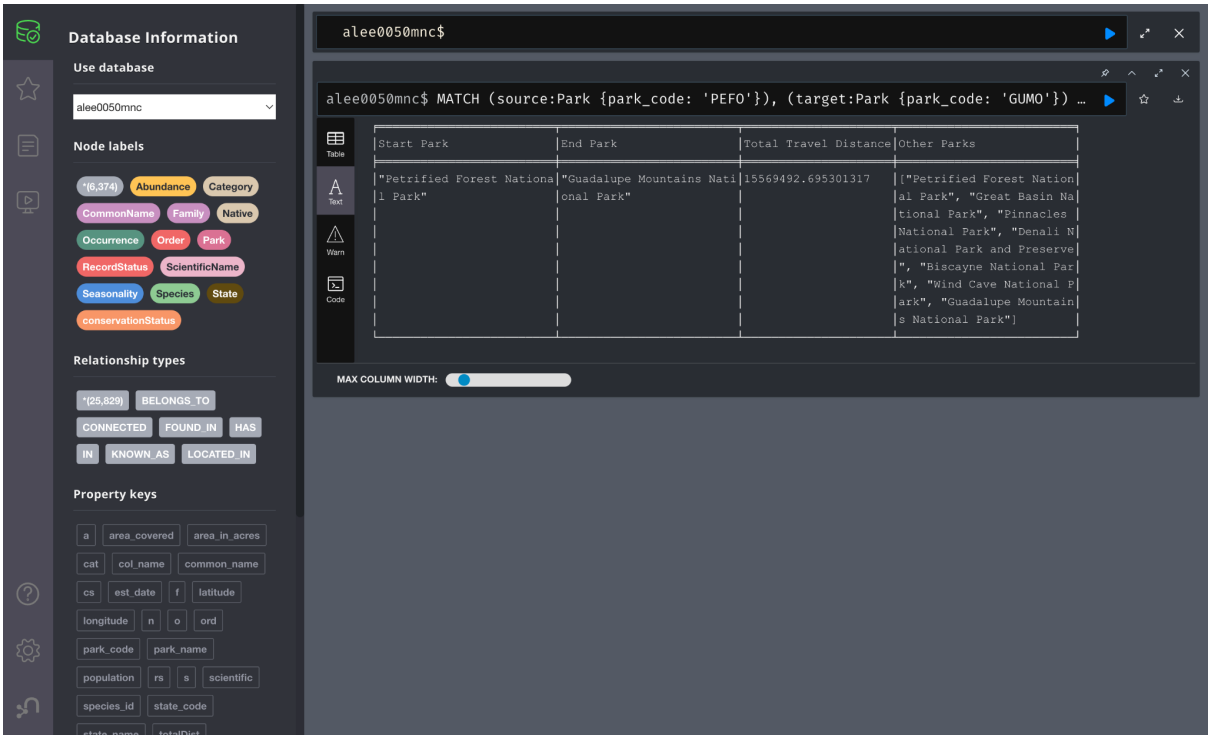
```
LOAD CSV WITH HEADERS FROM "file:///park_connections_a2.csv" AS park_row
MATCH (p:Park {park_code: park_row.`Park Code`}), (nearby_park:Park {park_code: park_row.`Connected Park Code`})
WHERE p.latitude IS NOT NULL AND p.longitude IS NOT NULL AND nearby_park.latitude IS NOT NULL AND nearby_park.longitude 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 tranverse 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 dijistra'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:



C.2. Neo4j Drivers

Code:

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

Results:

```python
from neo4j import GraphDatabase

#this is the connection port of which the server access is obtained through :SERVER STATUS
URI = "bolt://localhost:7689"
#credentials, as we are using the default user neo4j <username> and Pta59ypt123 <password>
AUTH = ("neo4j", "Pta59ypt123")
driver = GraphDatabase.driver(URI, auth=AUTH)
#this is initializing the driver object using neo4j import


#simple print statement to indicate which question it is
print("B1\n")

#using the to be fair we only need records, but this is saved for future implementation where a developer would want to measure
#the efficiency through summary and the keys is just used to return default object
records, summary, keys = driver.execute_query("""
// 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;
    """
)
#this is just used to print all the records
for record in records:
    print(record)

print("\n")
print("B2\n")

#this is repetitive, see explanation above
records, summary, keys = driver.execute_query("""MATCH (s:Species)-[:HAS]->(c:conservationStatus{cs:"Endangered"})
MATCH (s)-[f:FOUND_IN]->(p:Park)

WITH COUNT(s) AS maxSpeciesCount,p
```