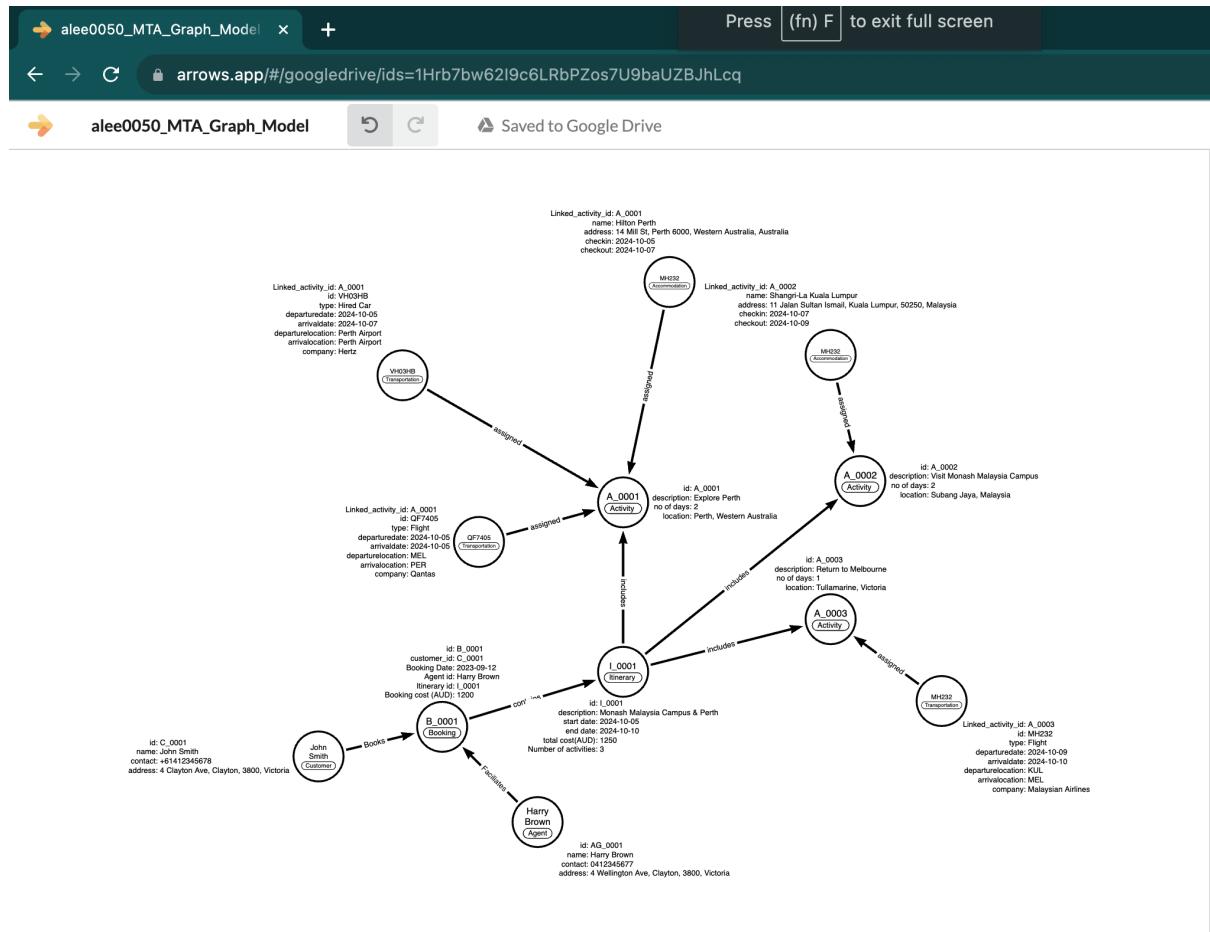


A1:

<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.



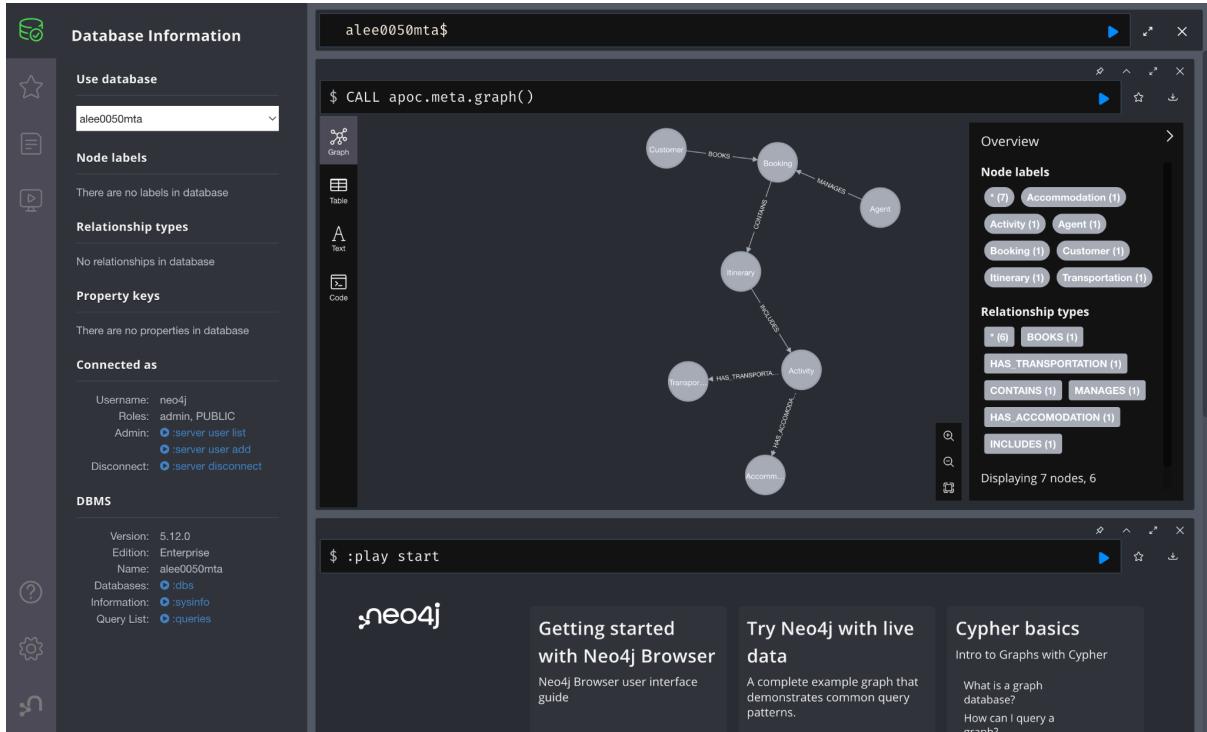
Table



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. If the 'Species ID' is not null, it merges the species into the database. If it is null, it creates a new node with the common name. The script then processes the 'Common Names' field by splitting it into individual names and merging them into a single node. Finally, it iterates through the remaining fields (Scientific Name, Occurrence, Abundance, Seasonality, Conservation Status, Nativeness, Record Status, Family, Order, and Category) and merges them into the same node. 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

```

```
alée0050_task_B.cypher • TaskC1.cypher • alée0050_task_C1.cypher • alée0050_task_C2.py
Users > vellichastrixism > alée0050_task_B.cypher
1 Users > vellichastrixism > alée0050_task_B.cypher
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
288
289
289
290
291
292
293
294
295
296
297
298
299
299
300
301
302
303
304
305
306
307
308
309
309
310
311
312
313
314
315
316
317
318
319
319
320
321
322
323
324
325
326
327
328
329
329
330
331
332
333
334
335
336
337
338
339
339
340
341
342
343
344
345
346
347
348
349
349
350
351
352
353
354
355
356
357
358
359
359
360
361
362
363
364
365
366
367
368
369
369
370
371
372
373
374
375
376
377
378
379
379
380
381
382
383
384
385
386
387
388
389
389
390
391
392
393
394
395
396
397
398
399
399
400
401
402
403
404
405
406
407
408
409
409
410
411
412
413
414
415
416
417
418
419
419
420
421
422
423
424
425
426
427
428
429
429
430
431
432
433
434
435
436
437
438
439
439
440
441
442
443
444
445
446
447
448
449
449
450
451
452
453
454
455
456
457
458
459
459
460
461
462
463
464
465
466
467
468
469
469
470
471
472
473
474
475
476
477
478
479
479
480
481
482
483
484
485
486
487
488
489
489
490
491
492
493
494
495
496
497
498
499
499
500
501
502
503
504
505
506
507
508
509
509
510
511
512
513
514
515
516
517
518
519
519
520
521
522
523
524
525
526
527
528
529
529
530
531
532
533
534
535
536
537
538
539
539
540
541
542
543
544
545
546
547
548
549
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
688
689
689
690
691
692
693
694
695
696
697
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
789
789
790
791
792
793
794
795
796
797
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
879
879
880
881
882
883
884
885
886
887
888
889
889
890
891
892
893
894
895
896
897
897
898
899
899
900
901
902
903
904
905
906
907
908
909
909
910
911
912
913
914
915
916
917
918
919
919
920
921
922
923
924
925
926
927
928
929
929
930
931
932
933
934
935
936
937
938
939
939
940
941
942
943
944
945
946
947
948
949
949
950
951
952
953
954
955
956
957
958
959
959
960
961
962
963
964
965
966
967
968
969
969
970
971
972
973
974
975
976
977
978
979
979
980
981
982
983
984
985
986
987
987
988
989
989
990
991
992
993
994
995
996
997
998
999
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1097
1098
1099
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1187
1188
1189
1189
1190
1191
1192
1193
1194
1195
1196
1196
1197
1198
1199
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1297
1298
1299
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1397
1398
1399
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1497
1498
1499
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1597
1598
1599
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1697
1698
1699
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1788
1789
1789
1790
1791
1792
1793
1794
1795
1796
1797
1797
1798
1799
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1888
1889
1889
1890
1891
1892
1893
1894
1895
1896
1897
1897
1898
1899
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1988
1989
1989
1990
1991
1992
1993
1994
1995
1996
1997
1997
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2088
2089
2089
2090
2091
2092
2093
2094
2095
2096
2097
2097
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2188
2189
2189
2190
2191
2192
2193
2194
2195
2196
2197
2197
2198
2199
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2229
2
```

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
130 WITH row
131 MATCH (species:Species {species_id: row.'Species ID'})
132 MATCH (occurrence:Occurrence {o:row.Occurrence})
133 MERGE (species)-[:HAS]->(occurrence)
134
135 WITH row
136 MATCH (species:Species {species_id: row.'Species ID'})
137 MATCH (recordStatus:RecordStatus {rs: row.'Record Status'})
138 MERGE (species)-[:HAS]->(recordStatus)
139
140 WITH row
141 MATCH (species:Species {species_id: row.'Species ID'})
142 MATCH (conservationStatus:ConservationStatus {cs: row.'Conservation Status'})
143 MERGE (species)-[:HAS]->(conservationStatus)
144
145 WITH row
146 MATCH (seasonality:Seasonality {s: row.Seasonality})
147 MERGE (species)-[:HAS]->(seasonality)
148
149 WITH row
150 MATCH (species:Species {species_id: row.'Species ID'})
151 MATCH (abundance:Abundance {a: row.Abandance})
152 MERGE (species)-[:HAS]->(abundance)
153
154 WITH row
155 MATCH (native:Native {n: row.Nativeness})
156 MERGE (species)-[:HAS]->(native)
157
158 WITH row
159 MATCH (Scientific:ScientificName {scientific: row.'Scientific Name'})
160 MATCH (family:Family {f: row.Family})

```

alee0050_task_B.cypher

```

163 MERGE (:family)-[:BELONGS_TO]->(order)
164
165 WITH row
166 MATCH (order:Order {ord: row.Order})
167 MATCH (category:Category {cat: row.Category})
168 MERGE (order)-[:BELONGS_TO]->(category)
169
170 WITH row
171 MATCH (native:Native {n: row.Nativeness})
172 MATCH (park:Park {park_name: row.'Park Name'})
173 MERGE (native)-[:IN]->(park)
174
175 WITH row
176 MATCH (abundance:Abundance {a: row.Abandance})
177 MATCH (park:Park {park_name: row.'Park Name'})
178 MERGE (abundance)-[:IN]->(park)
179
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
185 WITH row
186 MATCH (park:Park {park_name: row.'Park Name'})
187 MERGE (conservationStatus)-[:IN]->(park)
188
189 WITH row
190 MATCH (species:Species {species_id: row.'Species ID'})
191 MATCH (park:Park {park_name: row.'Park Name'})
192 MERGE (species)-[:FOUND_IN {population: toInteger(row.'Species Population')}]->(park)
193
194 WITH row
195 MATCH (species:Species {species_id: row.'Species ID'})
196 MERGE (common_name:CommonName {common_name: row.'Common Names'})
197 MERGE (species)-[:KNOWN_AS]->(common_name)

```

Results:

```

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 ...
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 WHERE ...
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 graph visualization window shows a network of nodes and relationships. Nodes include Species (green), State (brown), Park (pink), and Category (yellow). Relationships are labeled with terms like 'Belongs To', 'Has', 'Known As', 'Located In', and 'Occurs In'. The 'Overview' panel on the right lists node labels and relationship types.

```

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 ...
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
```

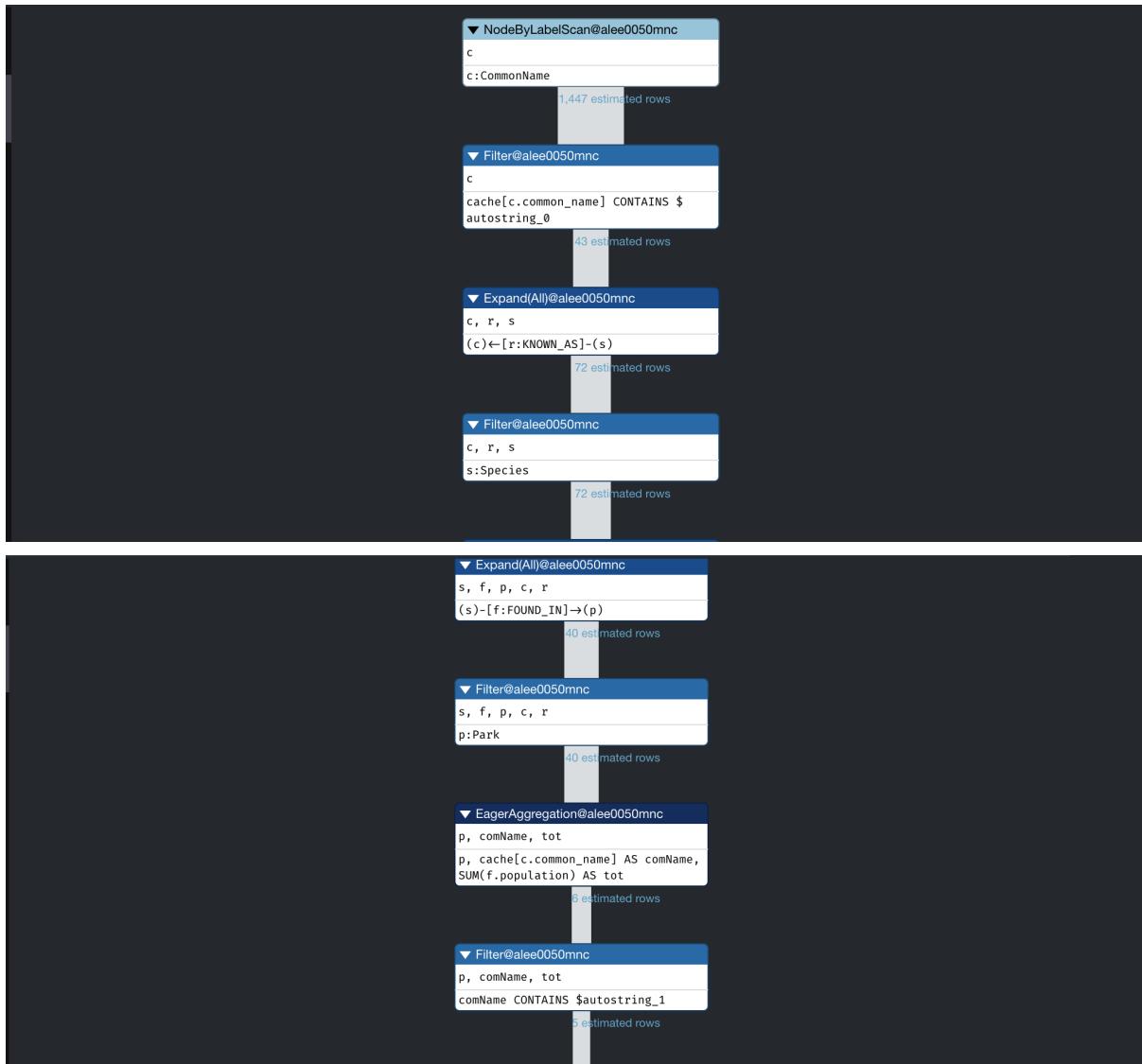
Results:

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:

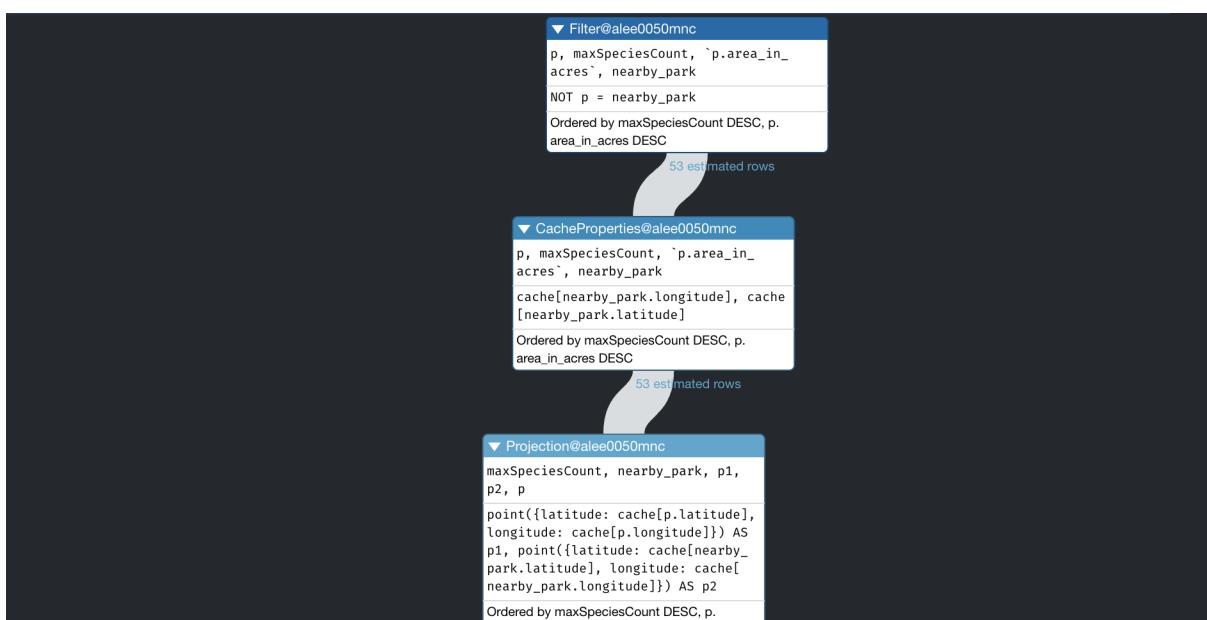
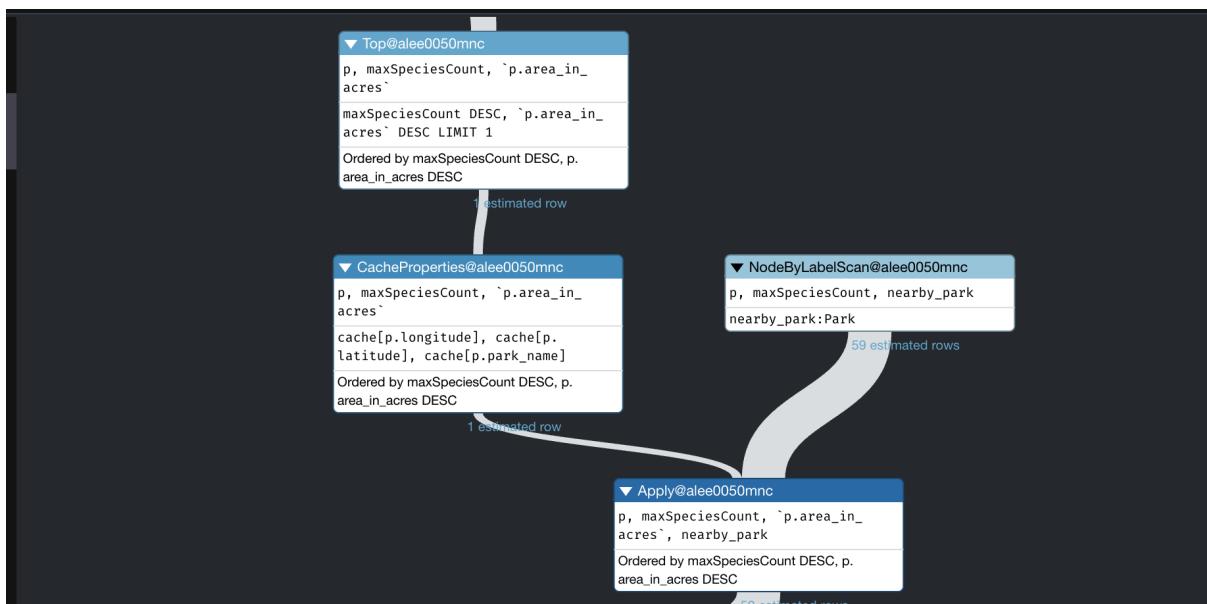
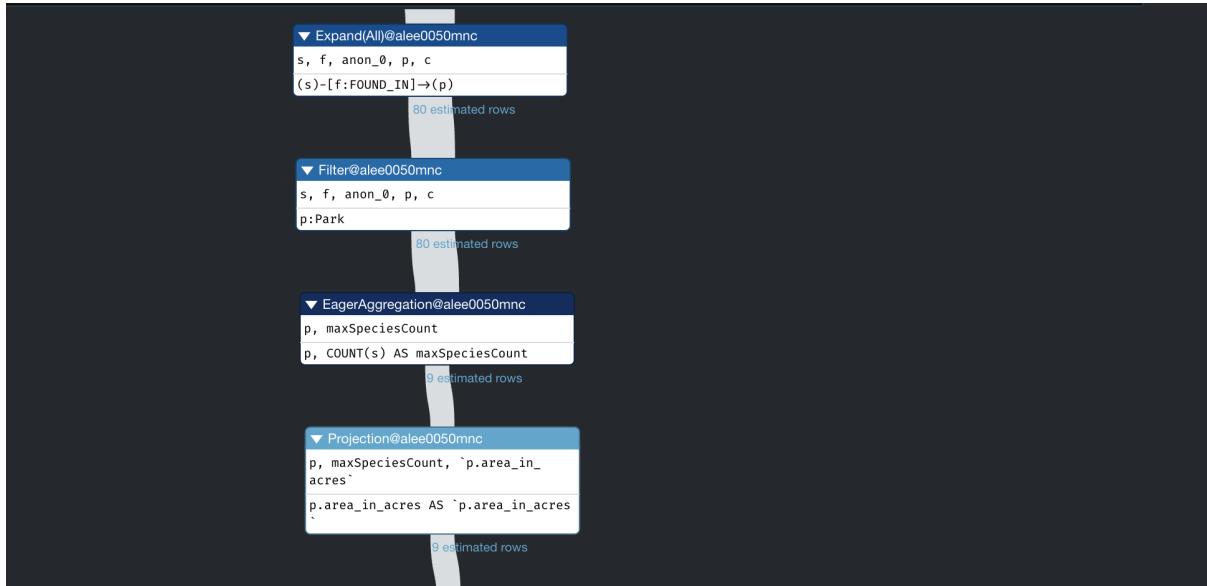
The screenshot shows the Neo4j Browser interface. On the left, the sidebar displays relationship types like 'BEONGS_TO', 'CONNECTED', 'FOUND_IN', 'HAS', 'KNOWN_AS', and 'LOCATED_IN'. Below that is a section for 'Property keys' containing various database columns such as 'area_covered', 'area_in_acres', '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'. Under 'Connected as', it shows the current user session with 'Username: neo4j', 'Roles: admin, PUBLIC', 'Admin: .server user list', 'Disconnect: .server disconnect', and 'DBMS' information including 'Version: 5.12.0', 'Edition: Enterprise', 'Name: ale0050mnc', 'Databases: .dbs', 'Information: .sysinfo', and 'Query List: .queries'.

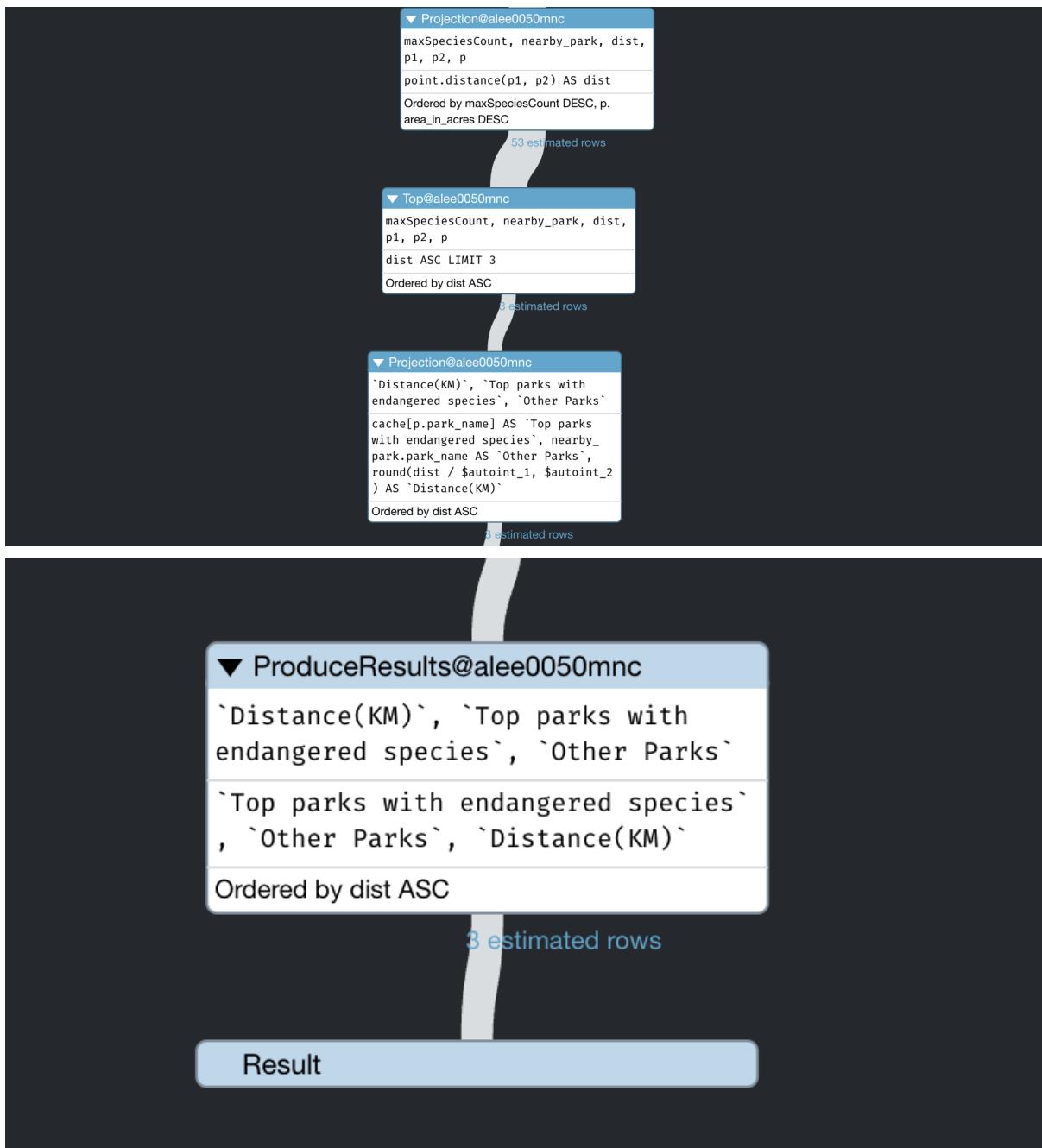
On the right, the main window shows a query result table titled 'Top parks with endangered species'. The table has three columns: 'Top parks with endangered species', 'Other Parks', and 'Distance (KM)'. The data is as follows:

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

Below the table is a 'MAX COLUMN WIDTH:' slider set to its maximum value.

Execution plan:





B.3. Modifying the Database:

The screenshot shows the Neo4j Browser interface. On the left, there's a sidebar with various configuration options like 'Relationship types' (e.g., 'BELONGS_TO'), 'Property keys', and 'Connected as'. On the right, the main window displays a table titled 'alee0050mnc\$ //B.3. Modifying the Database (After Modification Query): MATCH (p:Park)-[r...'. The table has three columns: 'State Code', 'Total Area Covered (acres)', and 'Park Codes'. The data includes rows for AK, CA, and WY.

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"]
"WY"	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:

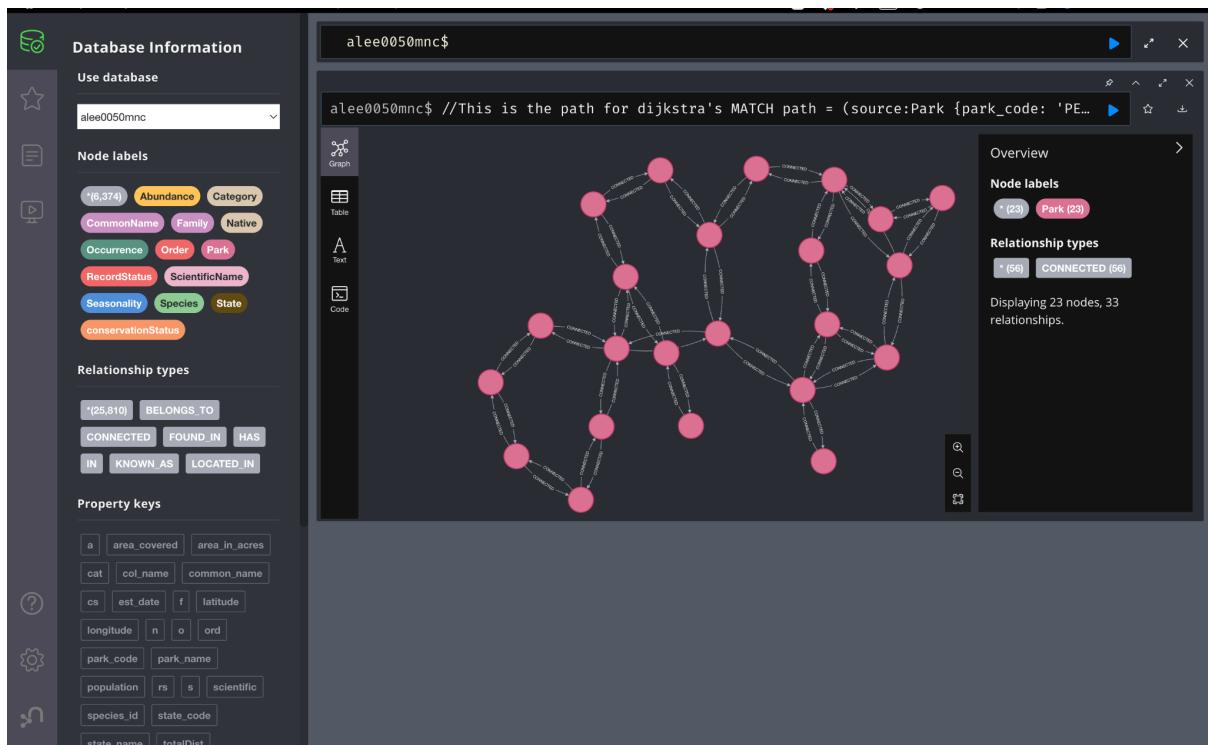
```

//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.

Database Information Window:

- Use database:** aleee0050mnc
- Node labels:** Abundance, Category, CommonName, Family, Native, Occurrence, Order, Park, RecordStatus, ScientificName, Seasonality, Species, State, conservationStatus
- Relationship types:** BELONGS_TO, CONNECTED, 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

Neo4j Browser Window:

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

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 development environment with the following components:

- EXPLORER:** Shows a folder structure with 'NO FOLDER OPENED' and a timeline entry for 'alee0050_task_C2.py'.
- OUTLINE:** Displays the outline of the current file, including sections like URI, AUTH, driver, records, summary, and keys.
- PROBLEMS:** Shows no problems detected in the workspace.
- TERMINAL:** Displays the output of running the Python script, showing results for parks containing coyotes and their populations.

```

9
10
11     #simple print statement to indicate which question it is
12     print("B1\n")
13
14     #records, summary, keys = driver.execute_query("""
15     // MATCH all the species with the relationship known_as
16     MATCH (s:Species)-[r:KNOWN_AS]->(c:CommonName)
17     // match species found_in park relationship
18     MATCH (s)-[f:FOUND_IN]->(p:Park)
19     // common name should contain Coyote
20     WHERE c.common_name CONTAINS 'Coyote'
21
22     WITH p, c.common_name AS comName, SUM(f.population) AS tot
23     // Filter out single names by checking for the existence of a comma
24     WHERE comName CONTAINS ','
25     RETURN p.park_name AS 'Park Name', collect(comName) AS 'Coyote Species', tot AS 'Coyote Population'
26     ORDER BY tot DESC;
27     | """
28     )
29     #this is just used to print all the records
30     for record in records:
31         print(record)
32
33     print("\n")
34     print("B2\n")
35
36     #this is repetitive, see explanation above
37     records, summary, keys = driver.execute_query("""
38         MATCH (s)-[f:FOUND_IN]->(p:Park)
39
40         WITH COUNT(s) AS maxSpeciesCount,p

```