Emre Doğan
May 26, 2019

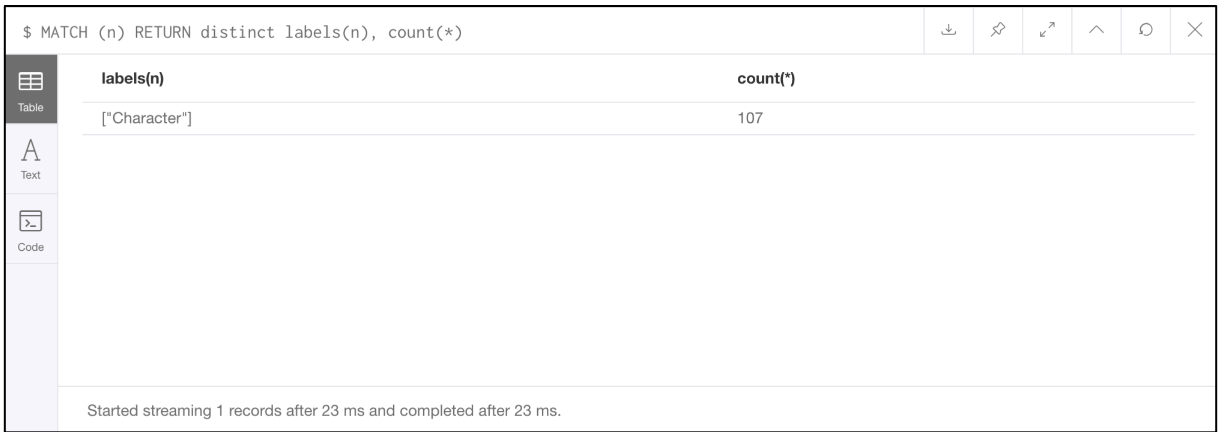
CENG790 BIG DATA ANALYTICS
Assignment 4: Introduction to Neo4j

1. According to the official documentation of Neo4j[1], unique property constraints ensure that property values are unique for all nodes with a specific label. Although these constraints are running slowly, it is important to have them not to have duplicate nodes for the same label. The reason the constraints work so slow is that they check all the database whether it is necessary to apply this constraint to the data.

2. The number of characters in the graph is found by the following Cypher query:

```
MATCH (n) RETURN distinct labels(n), count(*)
```

The total number of characters is 107. The related screenshot from Neo4j Desktop is available in Figure 1.


```
$ MATCH (n) RETURN distinct labels(n), count(*)
```

labels(n)	count(*)
["Character"]	107

Started streaming 1 records after 23 ms and completed after 23 ms.

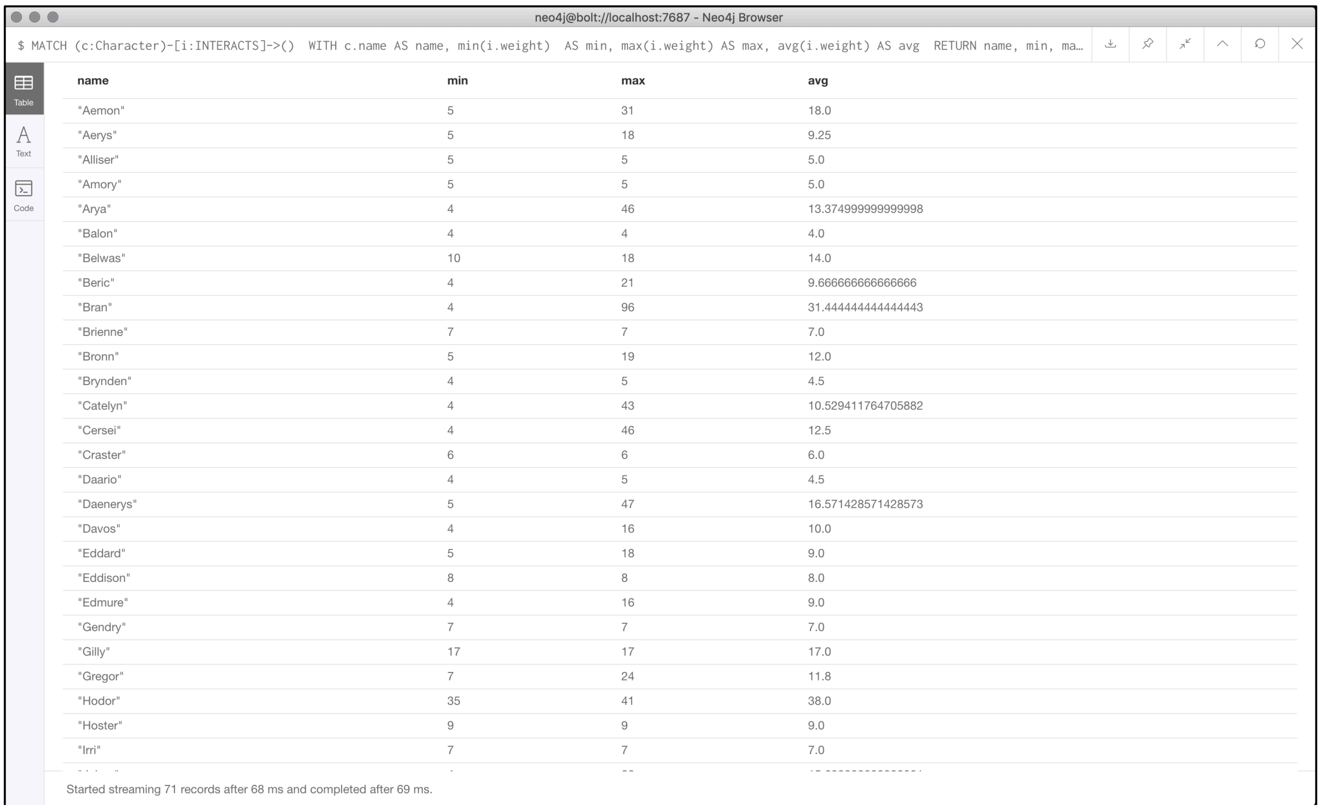
Figure 1. Screenshot from Neo4j showing the number of total characters in the graph.

3. The summary statistics for each character is achieved by the following query:

```
MATCH (c:Character)-[i:INTERACTS]->()
WITH c.name AS name, min(i.weight)
AS min, max(i.weight) AS max, avg(i.weight) AS avg
RETURN name, min, max, avg
```

The related screenshot of the result is given in Figure 2.

[1] https://neo4j.com/docs/cypher-manual/current/schema/constraints/



neo4j@bolt://localhost:7687 - Neo4j Browser

```
$ MATCH (c:Character)-[i:INTERACTS]->()  WITH c.name AS name, min(i.weight)  AS min, max(i.weight) AS max, avg(i.weight) AS avg  RETURN name, min, ma…
```

name	min	max	avg
"Aemon"	5	31	18.0
"Aerys"	5	18	9.25
"Alliser"	5	5	5.0
"Amory"	5	5	5.0
"Arya"	4	46	13.374999999999998
"Balon"	4	4	4.0
"Belwas"	10	18	14.0
"Beric"	4	21	9.666666666666666
"Bran"	4	96	31.444444444444443
"Brienne"	7	7	7.0
"Bronn"	5	19	12.0
"Brynden"	4	5	4.5
"Catelyn"	4	43	10.529411764705882
"Cersei"	4	46	12.5
"Craster"	6	6	6.0
"Daario"	4	5	4.5
"Daenerys"	5	47	16.571428571428573
"Davos"	4	16	10.0
"Eddard"	5	18	9.0
"Eddison"	8	8	8.0
"Edmure"	4	16	9.0
"Gendry"	7	7	7.0
"Gilly"	17	17	17.0
"Gregor"	7	24	11.8
"Hodor"	35	41	38.0
"Hoster"	9	9	9.0
"Irri"	7	7	7.0

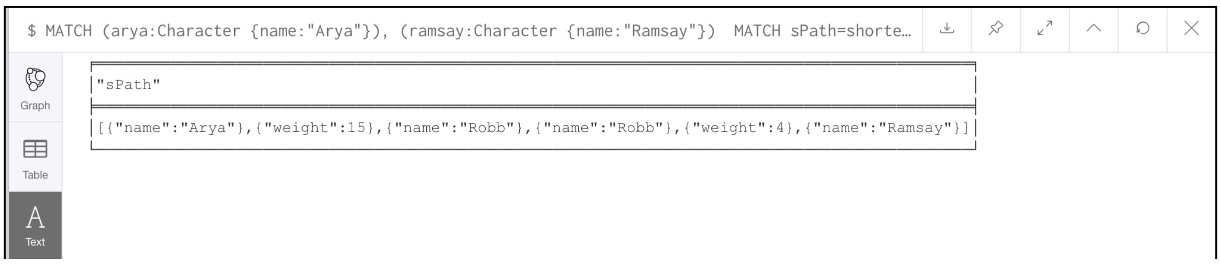
Started streaming 71 records after 68 ms and completed after 69 ms.

Figure 2. Summary statistics for the minimum, maximum and average number of characters each character has interacted with.

4. In order to find the shortest path between two characters (from Arya to Ramsay for our case), I tried the default `shortestPath` algorithm defined in the Neo4j. But this function does not take into account the weight property and calculate the distance as the steps between source and target nodes. For our case (Arya -> Ramsay), the shortest path results with 2 steps between nodes. But when considering the weight, the shortest path is 19.

The related query and its screenshot are given below in Figure 3.

```
// Shortest path from Arya to Ramsay –default neo4j implementation!
MATCH (arya:Character {name:"Arya"}), (ramsay:Character {name:"Ramsay"})
MATCH sPath=shortestPath((arya)-[:INTERACTS*]-(ramsay))
RETURN sPath
```



```
$ MATCH (arya:Character {name:"Arya"}), (ramsay:Character {name:"Ramsay"})  MATCH sPath=shorte…
```

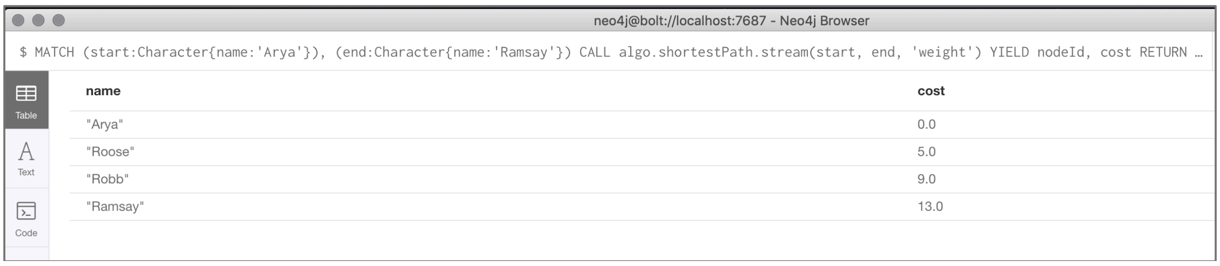
```
"sPath"
[{"name":"Arya"},{"weight":15},{"name":"Robb"},{"name":"Robb"},{"weight":4},{"name":"Ramsay"}]
```

Figure 3. The result with the default shortest path implementation of Neo4j.
Notice that shortest path is 19 in this case (15 + 4).

Then, I installed the plugin of `Graph Algorithms`[2] implemented for Neo4j and used the shortest path algorithm within this library. The related query is given below:

```
// Shortest path from Arya to Ramsay – Graph Algorithms implementation
MATCH (start:Character{name:'Arya'}), (end:Character{name:'Ramsay'})
CALL algo.shortestPath.stream(start, end, 'weight')
YIELD nodeId, cost
RETURN algo.asNode(nodeId).name AS name, cost
```

The resulting screenshot is available in Figure 4.



neo4j@bolt://localhost:7687 - Neo4j Browser

```
$ MATCH (start:Character{name:'Arya'}), (end:Character{name:'Ramsay'}) CALL algo.shortestPath.stream(start, end, 'weight') YIELD nodeId, cost RETURN …
```

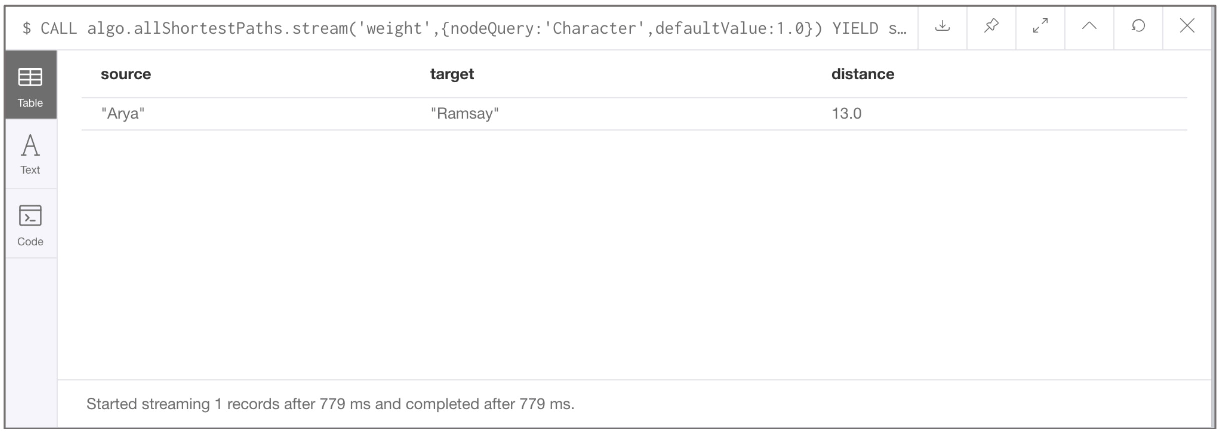
name	cost
"Arya"	0.0
"Roose"	5.0
"Robb"	9.0
"Ramsay"	13.0

Figure 4. Result with the Graph Algorithms Shortest Path implementation. Notice that the shortest path is less than the previous one (13<19) although the number of steps between nodes is greater (3>2).

Notice that the shortest path between these characters is 13.

To check whether there are other shortest paths with length 13 between Arya and Ramsay, I used the following query and resulted with the screenshot in Figure 5.

```
// ALL Shortest Paths--proof of the single shortest path
CALL algo.allShortestPaths.stream('weight',{nodeQuery:'Character',defaultValue:1.0})
YIELD sourceNodeId, targetNodeId, distance
WITH sourceNodeId, targetNodeId, distance
WHERE algo.isFinite(distance) = true
MATCH (source:Character{name:'Arya'}) WHERE id(source) = sourceNodeId
MATCH (target:Character{name:'Ramsay'}) WHERE id(target) = targetNodeId
WITH source, target, distance WHERE source <> target
RETURN source.name AS source, target.name AS target, distance
ORDER BY distance DESC
```



```
$ CALL algo.allShortestPaths.stream('weight',{nodeQuery:'Character',defaultValue:1.0}) YIELD s…
```

source	target	distance
"Arya"	"Ramsay"	13.0

Started streaming 1 records after 779 ms and completed after 779 ms.

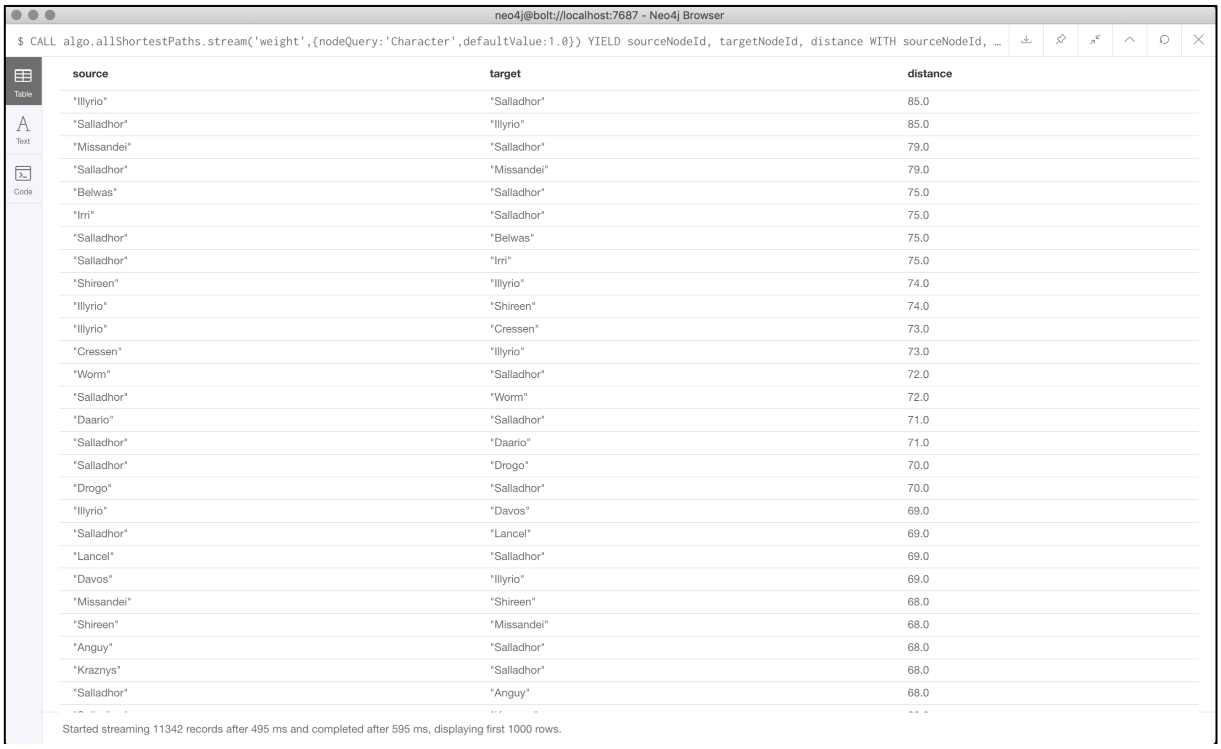
Figure 5. All shortest paths between Arya and Ramsay. Observe that there is only one shortest path available.

[2] https://github.com/neo4j-contrib/neo4j-graph-algorithms

5. To find the longest shortest path within the graph, I used the `algo.allShortestPaths.stream` function from the Graph Algorithms plugin. The query is in the following form:

```
// Finding the longest shortest path!
CALL algo.allShortestPaths.stream('weight',{nodeQuery:'Character',defaultValue:1.0})
YIELD sourceNodeId, targetNodeId, distance
WITH sourceNodeId, targetNodeId, distance
WHERE algo.isFinite(distance) = true
MATCH (source:Character) WHERE id(source) = sourceNodeId
MATCH (target:Character) WHERE id(target) = targetNodeId
WITH source, target, distance WHERE source <> target
RETURN source.name AS source, target.name AS target, distance
ORDER BY distance DESC
```

The longest shortest paths of the graph are listed in Figure 6.


neo4j@bolt://localhost:7687 - Neo4j Browser

```
$ CALL algo.allShortestPaths.stream('weight',{nodeQuery:'Character',defaultValue:1.0}) YIELD sourceNodeId, targetNodeId, distance WITH sourceNodeId, …
```

source	target	distance
"Illyrio"	"Salladhor"	85.0
"Salladhor"	"Illyrio"	85.0
"Missandei"	"Salladhor"	79.0
"Salladhor"	"Missandei"	79.0
"Belwas"	"Salladhor"	75.0
"Irri"	"Salladhor"	75.0
"Salladhor"	"Belwas"	75.0
"Salladhor"	"Irri"	75.0
"Shireen"	"Illyrio"	74.0
"Illyrio"	"Shireen"	74.0
"Illyrio"	"Cressen"	73.0
"Cressen"	"Illyrio"	73.0
"Worm"	"Salladhor"	72.0
"Salladhor"	"Worm"	72.0
"Daario"	"Salladhor"	71.0
"Salladhor"	"Daario"	71.0
"Salladhor"	"Drogo"	70.0
"Drogo"	"Salladhor"	70.0
"Illyrio"	"Davos"	69.0
"Salladhor"	"Lancel"	69.0
"Lancel"	"Salladhor"	69.0
"Davos"	"Illyrio"	69.0
"Missandei"	"Shireen"	68.0
"Shireen"	"Missandei"	68.0
"Anguy"	"Salladhor"	68.0
"Kraznys"	"Salladhor"	68.0
"Salladhor"	"Anguy"	68.0

Started streaming 11342 records after 495 ms and completed after 595 ms, displaying first 1000 rows.

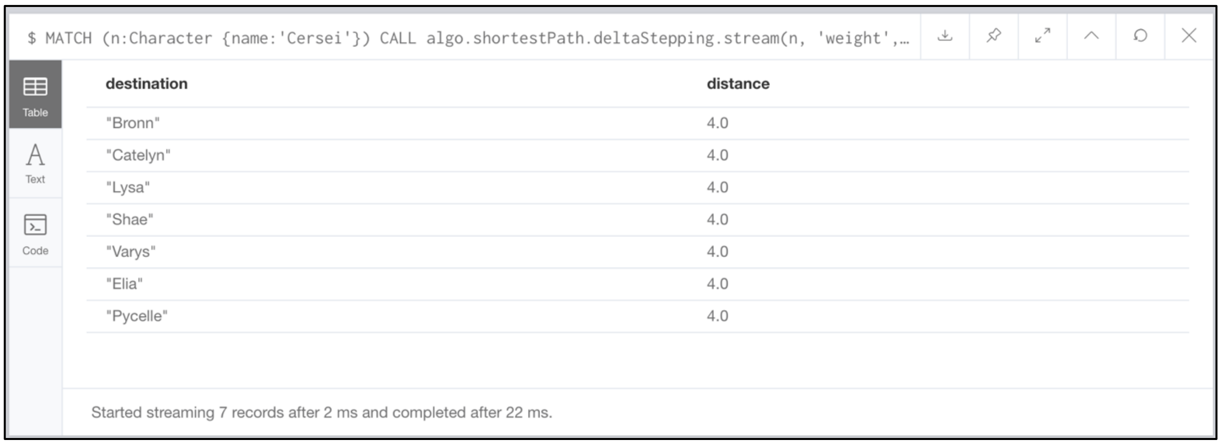
Figure 6. The Longest Shortest Path of the Graph.

Notice that the longest shortest path is between 'Illyrio' and 'Salladhor' with the distance of 85.

6. In order to find the characters with interaction distance 4 to Cersei Lannister, I used the following query:

```
// Characters with interaction distance 4 to Cersei
MATCH (n:Character {name:'Cersei'})
CALL algo.shortestPath.deltaStepping.stream(n, 'weight', 3.0)
YIELD nodeId, distance WHERE distance=4
RETURN algo.asNode(nodeId).name AS destination, distance
```

There are **7** characters with interaction distance 4 to Cersei Lannister. These characters are listed in the screenshot given in Figure 7.



```
$ MATCH (n:Character {name:'Cersei'}) CALL algo.shortestPath.deltaStepping.stream(n, 'weight',…
```

destination	distance
"Bronn"	4.0
"Catelyn"	4.0
"Lysa"	4.0
"Shae"	4.0
"Varys"	4.0
"Elia"	4.0
"Pycelle"	4.0

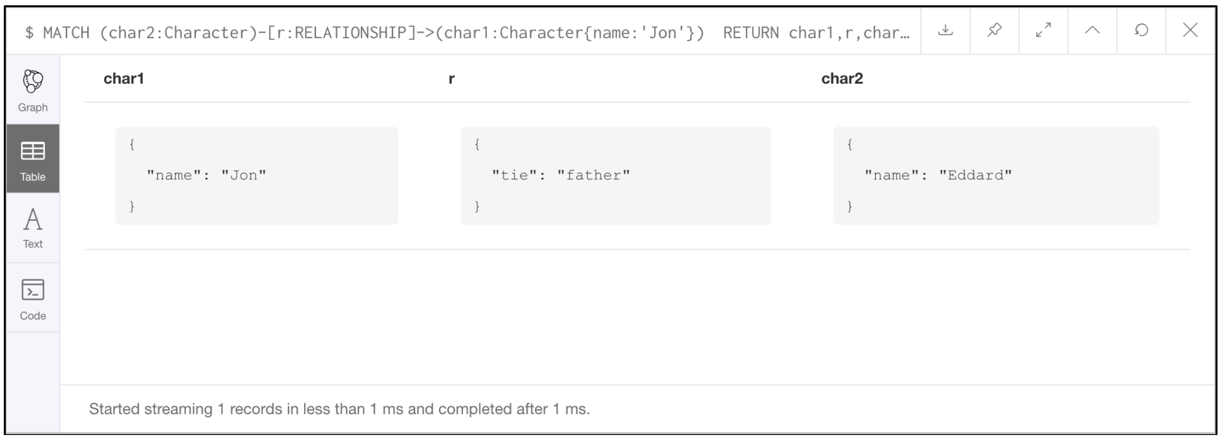
Started streaming 7 records after 2 ms and completed after 22 ms.

Figure 7. The characters with interaction distance 4 to Cersei Lannister

7. In order to find the parents of Jon Snow, the following query is used:

```
// finding the parents of Jon Snow
MATCH (char2:Character)-[r:RELATIONSHIP]->(char1:Character{name:'Jon'})
RETURN char1,r,char2;
```

The query returns the father connection of Jon Snow. There is no information on Jon's mother in the dataset. The screenshot is available in Figure 8.



```
$ MATCH (char2:Character)-[r:RELATIONSHIP]->(char1:Character{name:'Jon'})  RETURN char1,r,char…
```

char1	r	char2
{ "name": "Jon" }	{ "tie": "father" }	{ "name": "Eddard" }

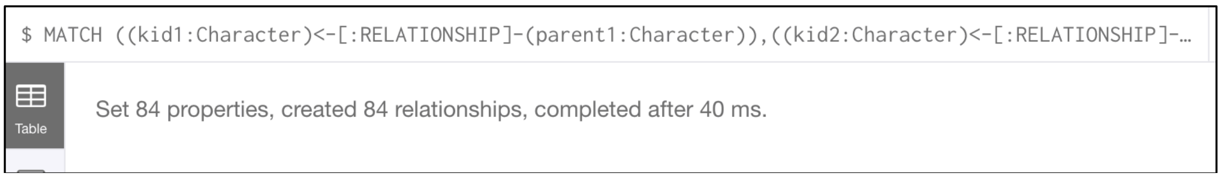
Started streaming 1 records in less than 1 ms and completed after 1 ms.

Figure 8. Parent Relations of Jon Snow.

8. For two characters to have a sibling relationship, it is assumed that one of their parents (mother or father) should be the same. The following query is used to create these relationships:

```
// Creating the sibling relationships.
MATCH ((kid1:Character)<-[:RELATIONSHIP]-(parent1:Character))
,((kid2:Character)<-[:RELATIONSHIP]-(parent2:Character))
WHERE parent1.name=parent2.name
CREATE (kid1)-[r:RELATIONSHIP{tie:'Sibling'}]->( kid2)
```

Figure 10 shows the creation process of 'Sibling' relationships.



```
$ MATCH ((kid1:Character)<-[:RELATIONSHIP]-(parent1:Character)),((kid2:Character)<-[:RELATIONSHIP]-…
```

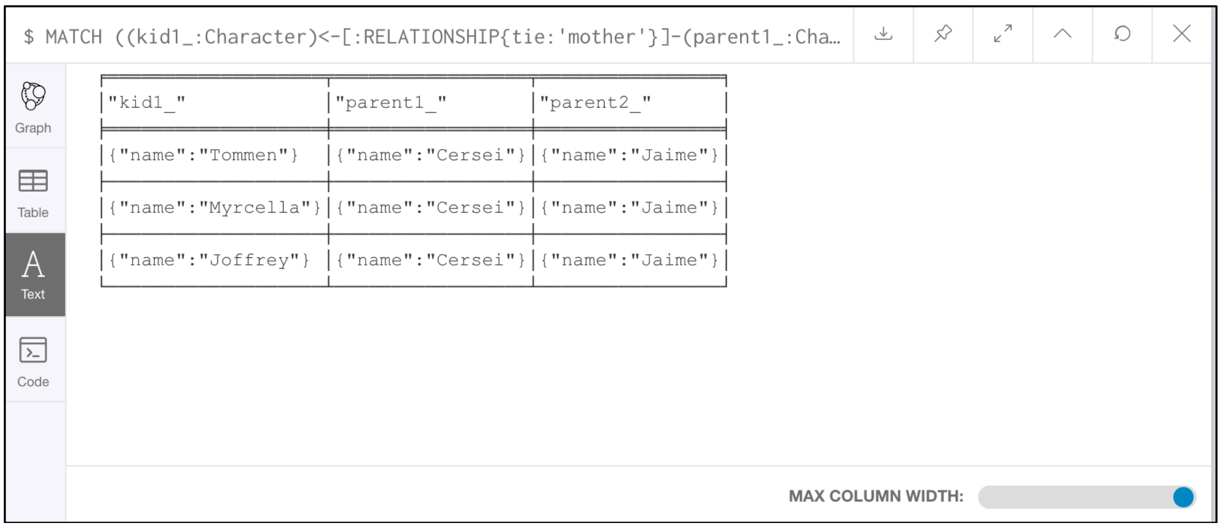
```
Set 84 properties, created 84 relationships, completed after 40 ms.
```

Figure 10. Creating the 'Sibling' Relationships.

9. To find the children of an incestuous relationship, I defined such a rule that if a character's mother and father are siblings, then this character is a child of an incestuous relationship. For this purpose, I used the following query:

```
// children of incestuous relationships!
MATCH ((kid1_:Character)<-[:RELATIONSHIP{tie:'mother'}]-(parent1_:Character))
,((kid1_:Character)<-[:RELATIONSHIP{tie:'father'}]-(parent2_:Character))
WHERE (parent1_:Character)<-[:RELATIONSHIP{tie:'Sibling'}]-(parent2_:Character)
RETURN kid1_,parent1_,parent2_
```

The children of an incestuous relationship are listed in Figure 11.



```
$ MATCH ((kid1_:Character)<-[:RELATIONSHIP{tie:'mother'}]-(parent1_:Cha…
```

```
╒════════════════════╤════════════════╤═══════════════╕
│"kid1_"             │"parent1_"      │"parent2_"     │
╞════════════════════╪════════════════╪═══════════════╡
│{"name":"Tommen"}   │{"name":"Cersei"}│{"name":"Jaime"}│
├────────────────────┼────────────────┼───────────────┤
│{"name":"Myrcella"} │{"name":"Cersei"}│{"name":"Jaime"}│
├────────────────────┼────────────────┼───────────────┤
│{"name":"Joffrey"}  │{"name":"Cersei"}│{"name":"Jaime"}│
└────────────────────┴────────────────┴───────────────┘
```

Figure 11. The list of children born from an incestuous relationship.

The final version of the graph is available in Figure 12:

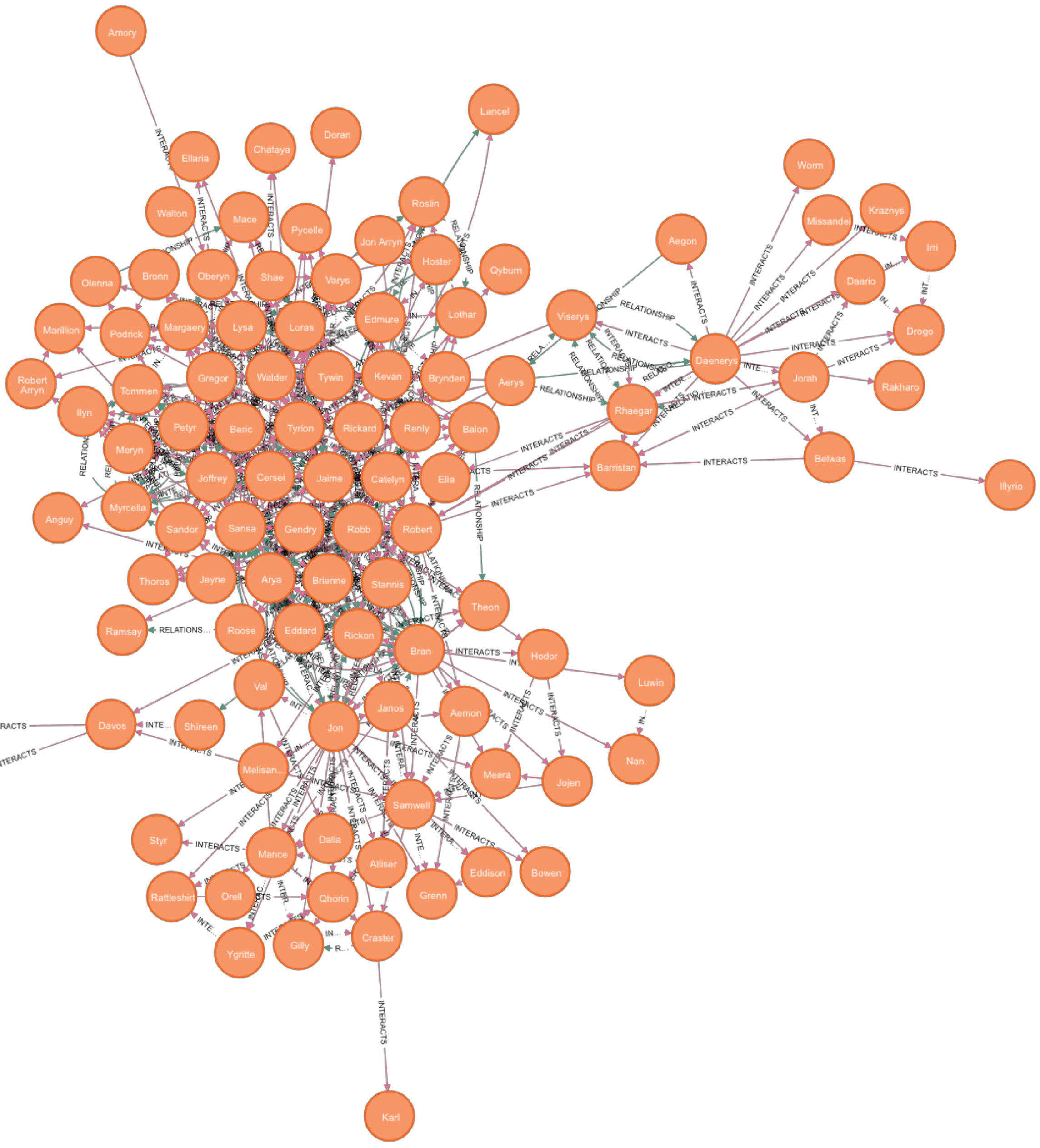


Figure 12. The final graph consisting of both INTERACTS and RELATIONSHIP interactions. RELATIONSHIP interactions (dataset + sibling relationships) are labeled in green where INTERACTS are pink.