Data Warehousing

Lecture 10 – Advanced Cypher and APOC

CITS3401
CITS5504

Dr. Sirui Li

Computer Science and Software Engineering

School of Maths, Physics and Computing

Acknowledgement: The lecture slides are adopted from online sources.

Outline of This Week's Lectures

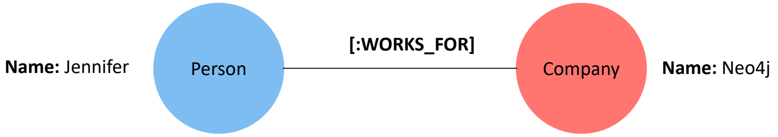
- Cypher recap
- Aggregation and other useful Cypher clauses
- Awesome Procedures on Cypher (APOC)
- Importing data in APOC
- APOC text functions
- Path Expansion in Cypher & APOC
- Virtual Graph

Outline of This Week's Lectures

- Cypher recap
- Aggregation and other useful Cypher clauses
- Awesome Procedures on Cypher (APOC)
- Importing data in APOC
- APOC text functions
- Path Expansion in Cypher & APOC
- Virtual Graph

Cypher Recap

```
(Person { name: "Jennifer" }) -
[:WORKS_FOR] ->
(Company { name: "Neo4j" })
```



Cypher Recap

- So far we have looked at several Cypher clauses for Creating, Reading, Updating and Deleting data:
  - **CREATE**, **MERGE**, and **LOAD CSV**
  - **MATCH** and **RETURN**, with **WHERE**
  - **SET** and **REMOVE**
  - **DELETE**
  - We have also briefly looked at **WITH**

Cypher Recap

- The Cypher queries we have looked at so far tend to either return nodes, relationships or properties, e.g.

```
MATCH   (Person {name: 'Jim'})-[:KNOWS]->(b:Person)
RETURN   b

MATCH   (a:Person {name: 'Jim'})-[:KNOWS]->(b:Person)
        -[:KNOWS]->(c:Person),
        (a)-[:KNOWS]->(c)
RETURN   b.name, c.name
```

Aggregation

- Sometimes we need to group records together to answer certain questions. For example:
  - For each person, list **the number of friends** they have.
  - For each movie, list the names of **all people** who watched that movie.
  - For each person named “Jack”, list the **name of all of their pets**.
- To be able to answer these questions we need to use **aggregation.**

Aggregation

For example, for “For each person, list **the number of friends** they have”, we would like our result to look as follows:

Person	Number of friends
Jack	5
Freddy	12
Jane	7
Wanda	16

Note we do not care about returning the friends of each person individually – we are only interested in the **total number**.

Aggregating by Count

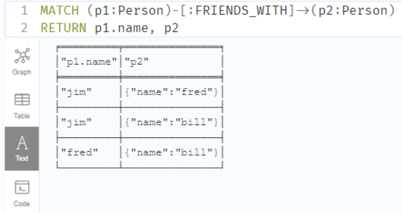
- The simplest aggregation is to aggregate by **count**. This simply returns the count of the results found in the database, rather than returning the objects themselves.
- To do this, we can use the **COUNT** function in Cypher:

```
MATCH   (p1:Person)-[:FRIENDS_WITH]->(p2:Person)
RETURN   p1.name, COUNT(p2)
```

- This will count the number of occurrences of `p2` for each distinct value of `p1.name` (which is the **grouping key**).

Aggregating by Count Example

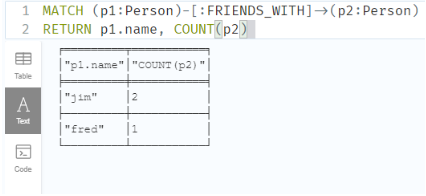
Without COUNT:



```
MATCH (p1:Person)-[:FRIENDS_WITH]→(p2:Person)
RETURN p1.name, p2
```

```
╒═══════════╤═════════════════╕
│"p1.name"  │"p2"             │
╞═══════════╪═════════════════╡
│"jim"      │{"name":"fred"}  │
├───────────┼─────────────────┤
│"jim"      │{"name":"bill"}  │
├───────────┼─────────────────┤
│"fred"     │{"name":"bill"}  │
└───────────┴─────────────────┘
```

With COUNT:

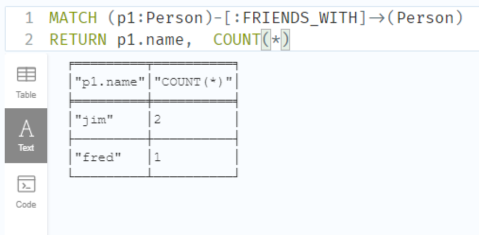


```
MATCH (p1:Person)-[:FRIENDS_WITH]→(p2:Person)
RETURN p1.name, COUNT(p2)
```

```
╒═══════════╤═════════════╕
│"p1.name"  │"COUNT(p2)"  │
╞═══════════╪═════════════╡
│"jim"      │2            │
├───────────┼─────────────┤
│"fred"     │1            │
└───────────┴─────────────┘
```

Aggregating by Count Example

- We can also use the asterisk (*) inside a COUNT function to count the total number of rows for each grouping key, for example:



```
MATCH (p1:Person)-[:FRIENDS_WITH]→(Person)
RETURN p1.name,  COUNT(*)
```

```
╒═══════════╤════════════╕
│"p1.name"  │"COUNT(*)"  │
╞═══════════╪════════════╡
│"jim"      │2           │
├───────────┼────────────┤
│"fred"     │1           │
└───────────┴────────────┘
```

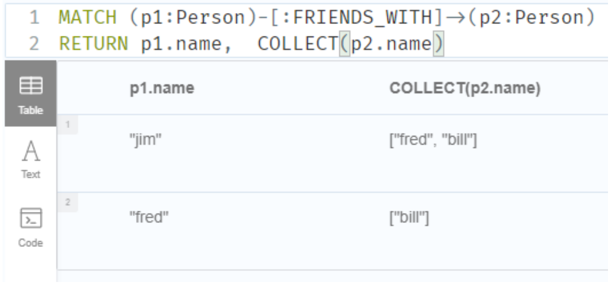
Aggregating by Value

- Sometimes we need to return lists of values – for example for listing all of the friends of a person.
- To do this, we can use the **COLLECT** function in Cypher:

```
MATCH   (p1:Person)-[:FRIENDS_WITH]->(p2:Person)
RETURN   p1.name, COLLECT(p2.name)
```

- All values of p2.name for each grouping key will be stored in a list.

Aggregating by Value Example



```
MATCH (p1:Person)-[:FRIENDS_WITH]->(p2:Person)
RETURN p1.name,  COLLECT(p2.name)
```

	p1.name	COLLECT(p2.name)
1	"jim"	["fred", "bill"]
2	"fred"	["bill"]

Other useful aggregation functions

The following aggregation functions work on numeric values:

- **AVG –** Calculates the average value
- **MAX –** Calculates the maximum value
- **MIN –** Calculates the minimum value
- **SUM** – Calculates the total sum (also works on durations)
- **stDev –** Standard deviation

https://neo4j.com/docs/cypher-manual/current/functions/aggregating

Data Profiling with Cypher

- Counting all nodes that exist in the graph:

```
MATCH (n) RETURN COUNT(n)
```

- Counting all relationships that exist in the graph:

```
MATCH (n)-[r]-(m) RETURN COUNT(r)
```

- Finding all distinct labels that exist in the graph:

```
1. MATCH (n) RETURN DISTINCT LABELS(n)

2. CALL db.labels()
```

- Finding all distinct relationship types:

```
1. MATCH n-[r]-() RETURN DISTINCT TYPE(r)

2. CALL db.relationshipTypes()
```

Data Profiling with Cypher

- Finding all nodes that are disjoint, i.e. nodes that do not have any relationship with the other nodes:

```
MATCH (n) WHERE NOT (n)-[]-() RETURN n
```

- Finding all nodes that have some specific property:

```
MATCH (n) WHERE HAS(n.name) RETURN n
```

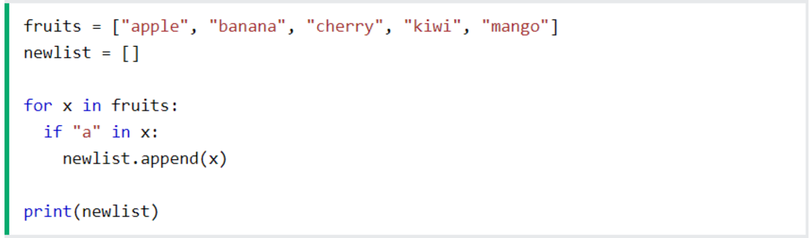
- Finding all nodes that have some specific relationship, regardless of the direction:

```
MATCH (n)-[:ORIGIN]-() RETURN DISTINCT n
```

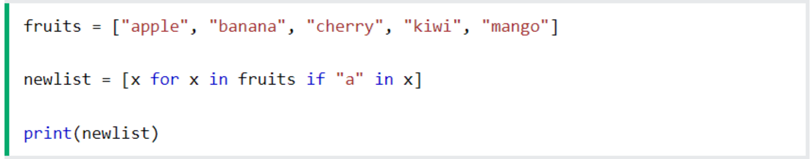
- Show **metagraph** (or **schema graph**) to illustrate what is related, and how

```
CALL db.schema.visualization()
```

List Comprehension in Python (recall)

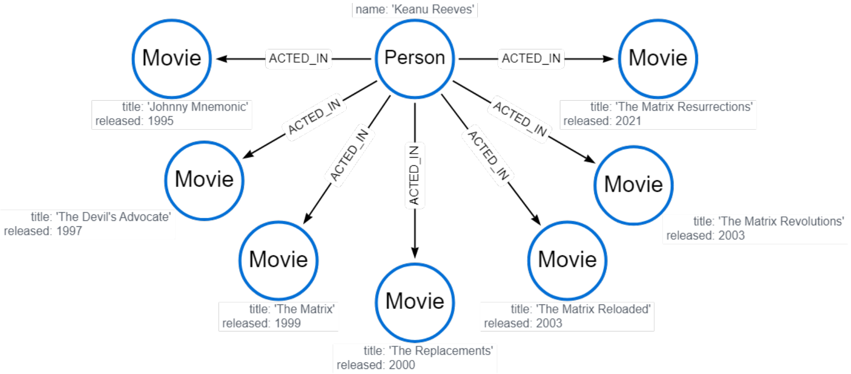


```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []

for x in fruits:
  if "a" in x:
    newlist.append(x)

print(newlist)
```



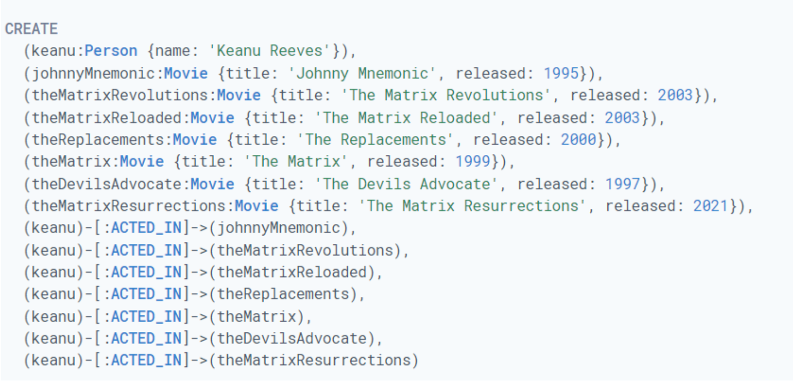
```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]

newlist = [x for x in fruits if "a" in x]

print(newlist)
```

Pattern and List Comprehension in Cypher



https://neo4j.com/docs/cypher-manual/current/values-and-types/lists/#cypher-pattern-comprehension

Pattern and List Comprehension in Cypher

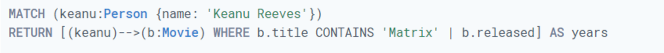


```
CREATE
  (keanu:Person {name: 'Keanu Reeves'}),
  (johnnyMnemonic:Movie {title: 'Johnny Mnemonic', released: 1995}),
  (theMatrixRevolutions:Movie {title: 'The Matrix Revolutions', released: 2003}),
  (theMatrixReloaded:Movie {title: 'The Matrix Reloaded', released: 2003}),
  (theReplacements:Movie {title: 'The Replacements', released: 2000}),
  (theMatrix:Movie {title: 'The Matrix', released: 1999}),
  (theDevilsAdvocate:Movie {title: 'The Devils Advocate', released: 1997}),
  (theMatrixResurrections:Movie {title: 'The Matrix Resurrections', released: 2021}),
  (keanu)-[:ACTED_IN]->(johnnyMnemonic),
  (keanu)-[:ACTED_IN]->(theMatrixRevolutions),
  (keanu)-[:ACTED_IN]->(theMatrixReloaded),
  (keanu)-[:ACTED_IN]->(theReplacements),
  (keanu)-[:ACTED_IN]->(theMatrix),
  (keanu)-[:ACTED_IN]->(theDevilsAdvocate),
  (keanu)-[:ACTED_IN]->(theMatrixResurrections)
```

https://neo4j.com/docs/cypher-manual/current/values-and-types/lists/#cypher-pattern-comprehension

Pattern Comprehension in Cypher

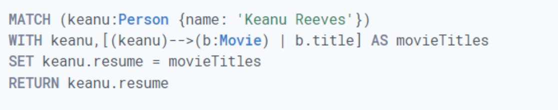
A syntactic construct in Cypher for creating a list based on matchings of a pattern.



```
MATCH (keanu:Person {name: 'Keanu Reeves'})
RETURN [(keanu)-->(b:Movie) WHERE b.title CONTAINS 'Matrix' | b.released] AS years
```

A pattern comprehension

- matches the specified pattern like a normal MATCH clause,
- with predicates like a normal WHERE clause, but yields a custom projection as specified.

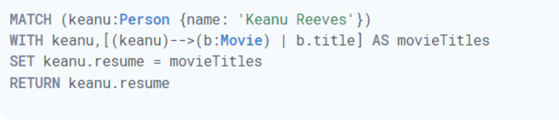


```
MATCH (keanu:Person {name: 'Keanu Reeves'})
WITH keanu,[(keanu)-->(b:Movie) | b.title] AS movieTitles
SET keanu.resume = movieTitles
RETURN keanu.resume
```

https://neo4j.com/docs/cypher-manual/current/values-and-types/lists/#cypher-pattern-comprehension

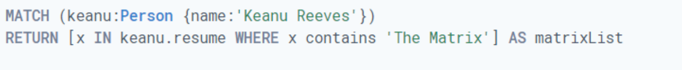
List Comprehension in Cypher

A syntactic construct in Cypher for creating a list based on an existing list.



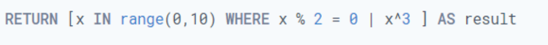
```
MATCH (keanu:Person {name: 'Keanu Reeves'})
WITH keanu,[(keanu)-->(b:Movie) | b.title] AS movieTitles
SET keanu.resume = movieTitles
RETURN keanu.resume
```

`keanu.resume` is an existing list, and we could do

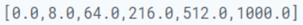


```
MATCH (keanu:Person {name:'Keanu Reeves'})
RETURN [x IN keanu.resume WHERE x contains 'The Matrix'] AS matrixList
```

What will the following return?



```
RETURN [x IN range(0,10) WHERE x % 2 = 0 | x^3 ] AS result
```



```
[0.0,8.0,64.0,216.0,512.0,1000.0]
```

https://neo4j.com/docs/cypher-manual/current/values-and-types/lists/#cypher-pattern-comprehension

Summary

- We have looked at **aggregation** in Cypher, which allows us to **group similar records together** and optionally run calculations on them rather than returning records individually.
- There are several aggregation functions available in Neo4j, notably **COUNT** and **COLLECT**.
- **Pattern** and **list comprehension**, which creating a list based on matchings of a pattern or an existing list.

Outline of This Week's Lectures

- Cypher recap
- Aggregation and other useful Cypher clauses
- Awesome Procedures on Cypher (APOC)
- Importing data in APOC
- APOC text functions
- Path Expansion in Cypher & APOC
- Virtual Graph

Awesome Procedures on Cypher (APOC)

- The APOC library is a massive library for Neo4j that contains over **450** standard procedures and functions.
- It provides a wide range of functionality, such as:
  - More powerful tools for importing/exporting data
  - Text parsing functions
  - Path expansion
  - Virtual graphs

https://neo4j.com/developer/neo4j-apoc/

Awesome Procedures on Cypher (APOC)

- APOC contains both procedures and functions:
  - **Procedures** generally return a list (of nodes, etc) that are unwound.
  - **Functions** are simpler, and typically return a single value i.e. a map, list, string or number.
- APOC often replaces the need to write your own functions in other scripting languages e.g. Python etc, and can therefore be a huge timesaver if you familiarise yourself with the functionality it offers.

https://neo4j.com/developer/neo4j-apoc/

Awesome Procedures on Cypher (APOC)

- Before using APOC we need to **enable it from the plugins window** (just like with GDS).
- APOC functions can be run using CALL
- You can get a list of available APOC procedures and functions using:

```
CALL apoc.help("")
```

- You can search by keywords:

```
CALL apoc.help("import")
```

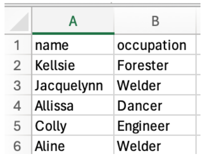
https://neo4j.com/developer/neo4j-apoc/

Importing data with APOC

- So far we have looked at **LOAD CSV**, which allows us to import data from a CSV file.
- APOC has a much **wider range of functionality for importing data** – it can load CSV, JSON files, excel spreadsheets, web APIs, etc.
- In this lecture we will focus on loading CSVs using APOC, but feel free to check out the other functionality in the Neo4j documentation.

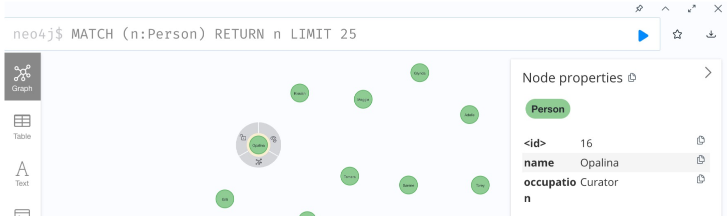
Importing CSVs - nodes

people.csv



	A	B
1	name	occupation
2	Kellsie	Forester
3	Jacquelynn	Welder
4	Allissa	Dancer
5	Colly	Engineer
6	Aline	Welder

```
CALL apoc.import.csv([{fileName: 'file:/people.csv',

labels: ['Person']}], [], {})
```



```
neo4j$ MATCH (n:Person) RETURN n LIMIT 25
```

Node properties

Person

<id>	16
name	Opalina
occupation	Curator

Importing CSVs – nodes and relationships

- APOC's CSV import will import nodes and relationships from a CSV file directly into your Neo4j database:

```
apoc.import.csv(<nodes>, <relationships>, <config>)
```

- Each argument is a list, where each element is a map of the filename and labels. For example:

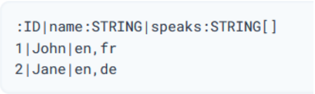
```
apoc.import.csv([
    {filename: "file:/people.csv", labels: ["Person"]},
    …
```

Importing CSVs – nodes and relationships

A complete import CSV call might look as follows:

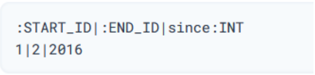
```
CALL apoc.import.csv(
  [{fileName: "file:/persons.csv", labels: ["Person"]}],
  [{fileName: "file:/knows.csv", type: "KNOWS"}],
  { delimiter: "|", arrayDelimiter: ",", stringIds:
    false } )
```

persons.csv



```
:ID|name:STRING|speaks:STRING[]
1|John|en,fr
2|Jane|en,de
```

knows.csv



```
:START_ID|:END_ID|since:INT
1|2|2016
```

https://neo4j.com/labs/apoc/4.3/import/import-csv/#_usage

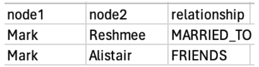
Creating dynamic relationships

- An interesting use case for APOC is creating **dynamic relationship types** from a single CSV file.
- For example, imagine you had many **different types of relationships** between people – 'knows', 'friends_with', 'enemies_of', 'brother_of', 'sister_of'… etc.
- Without APOC, the simplest way to do this would be to create a CSV for each type of relationship in your data:
  - people.csv
  - knows.csv
  - friends_with.csv
  - enemies_of.csv…

Creating dynamic relationships

- APOC has a function called `apoc.create.relationship` that creates a relationship of a given type between two nodes.
- You can combine this with standard Cypher's **LOAD CSV** to create relationship types based on the value stored in a column:

```
LOAD CSV WITH HEADERS from "file:///people.csv" AS row
MERGE (p1:Person {name: row.node1})
MERGE (p2:Person {name: row.node2}) WITH p1, p2, row
CALL apoc.create.relationship(p1, row.relationship, {}, p2)
YIELD rel RETURN rel
```



node1	node2	relationship
Mark	Reshmee	MARRIED_TO
Mark	Alistair	FRIENDS

https://www.markhneedham.com/blog/2016/10/30/neo4j-create-dynamic-relationship-type/

APOC's Text Functions

APOC features a substantial number of **text functions** which you can use on strings in your database. Some notable ones include:

- `apoc.text.levenshteinSimilarity` – measure the distance between two strings.
- `apoc.text.clean` – strip the given string of everything except alpha numeric characters and convert it to lower case.
- `apoc.text.fuzzyMatch` – check if two words can be matched in a fuzzy way.
- `apoc.text.indexOf` – find first occurrence of the lookup string in the text.

https://neo4j.com/labs/apoc/4.3/overview/apoc.text/

Summary

- We have looked at **APOC** (Awesome Procedures for Cypher), which offers a range of useful functions and procedures to extend Cypher.
- APOC has functions for **loading/importing data** that can be much quicker to write than many individual Cypher statements.
- It can also **create dynamic relationships** based on a value inside a column of a CSV file.
- We have also looked at some of APOC's **text functions**.

Outline of This Week's Lectures

- Cypher recap
- Aggregation and other useful Cypher clauses
- Awesome Procedures on Cypher (APOC)
- Importing data in APOC
- APOC text functions
- Path Expansion in Cypher & APOC
- Virtual Graph

Path Expansion in Cypher

- Sometimes it can be useful to describe a pattern containing a **long path**. For example, we might write:

  `(a)-->()-->(b)`

  … which is a path of length 2.

- With longer paths, though, the Cypher can become long-winded…

  `(a)-->()-->()-->()-->()-->(b)`

  `(a)-->()-->()-->()-->()-->()-->()-->(b)`…

Path Expansion in Cypher

Cypher allows us to specify a **variable path length**:

`(a)-[*5]->(b)`	Paths of length 5
`(a)-[*..5]->(b)`	Paths of length ranging from 0 to 5 inclusive
`(a)-[*5..]->(b)`	Paths of length 5 or more
`(a)-[*3..5]->(b)`	Paths from length 3 to 5 inclusive

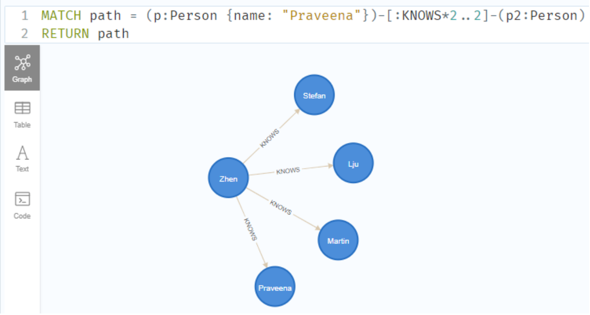
Path Expansion in Cypher

This can be useful for many applications, such as querying for friends as well as friends of friends, e.g.:

```
MATCH (me)-[:KNOWS*1..2]-(remote_friend)
WHERE me.name = "Filipa"
RETURN remote_friend.name
```

Path Expansion in Cypher

We can also capture the path as a variable and return it – that way Neo4j will visualise the entire path.



```
MATCH path = (p:Person {name: "Praveena"})-[:KNOWS*2..2]-(p2:Person)
RETURN path
```

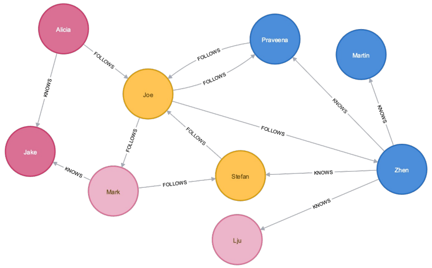
Path Expansion in APOC

- APOC contains functions and procedures for working with paths as part of its **Path Expander** module.
- Unlike regular Cypher, APOC allows for:
  - The direction of the relationship to be specified per relationship type.
  - Whitelist/blacklisted label types.
  - Specification of end nodes for the expansion.

https://neo4j.com/labs/apoc/4.1/graph-querying/path-expander/

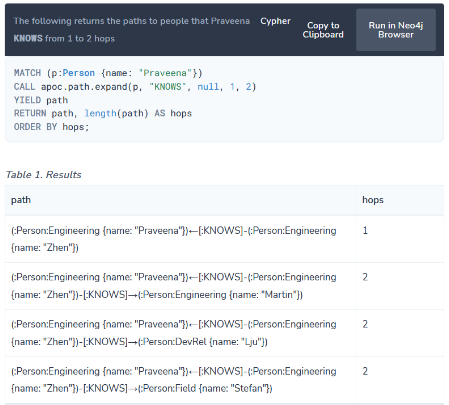
Path Expansion in APOC

- The most simple function is `apoc.path.expand`, which allows us to traverse paths based on relationship filters or node filters.
- We specify the start node, relationship filter, label filter, and path length.
- To demonstrate this we will use the graph on the right as an example.



https://neo4j.com/labs/apoc/4.1/graph-querying/expand-paths/

Path Expansion in APOC

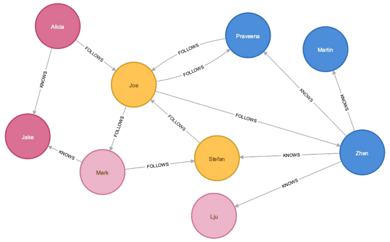


The following returns the paths to people that Praveena **KNOWS** from 1 to 2 hops

```
MATCH (p:Person {name: "Praveena"})
CALL apoc.path.expand(p, "KNOWS", null, 1, 2)
YIELD path
RETURN path, length(path) AS hops
ORDER BY hops;
```

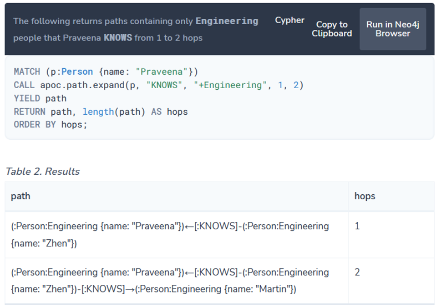
Table 1. Results

path	hops
(:Person:Engineering {name: "Praveena"})←[:KNOWS]-(:Person:Engineering {name: "Zhen"})	1
(:Person:Engineering {name: "Praveena"})←[:KNOWS]-(:Person:Engineering {name: "Zhen"})-[:KNOWS]→(:Person:Engineering {name: "Martin"})	2
(:Person:Engineering {name: "Praveena"})←[:KNOWS]-(:Person:Engineering {name: "Zhen"})-[:KNOWS]→(:Person:DevRel {name: "Lju"})	2
(:Person:Engineering {name: "Praveena"})←[:KNOWS]-(:Person:Engineering {name: "Zhen"})-[:KNOWS]→(:Person:Field {name: "Stefan"})	2



https://neo4j.com/labs/apoc/4.1/graph-querying/expand-paths/

Path Expansion in APOC

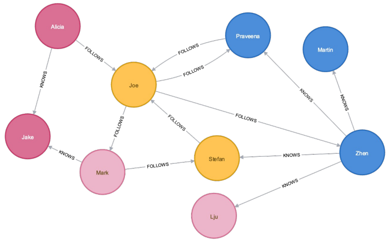


The following returns paths containing only **Engineering** people that Praveena **KNOWS** from 1 to 2 hops

```
MATCH (p:Person {name: "Praveena"})
CALL apoc.path.expand(p, "KNOWS", "+Engineering", 1, 2)
YIELD path
RETURN path, length(path) AS hops
ORDER BY hops;
```

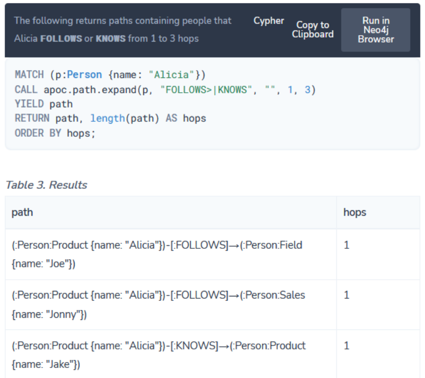
Table 2. Results

path	hops
(:Person:Engineering {name: "Praveena"})←[:KNOWS]-(:Person:Engineering {name: "Zhen"})	1
(:Person:Engineering {name: "Praveena"})←[:KNOWS]-(:Person:Engineering {name: "Zhen"})-[:KNOWS]→(:Person:Engineering {name: "Martin"})	2



https://neo4j.com/labs/apoc/4.1/graph-querying/expand-paths/

Path Expansion in APOC

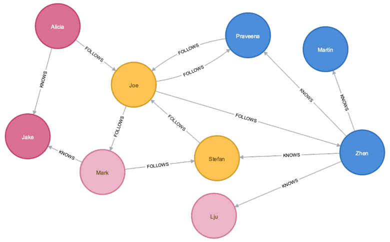


The following returns paths containing people that Alicia **FOLLOWS** or **KNOWS** from 1 to 3 hops

```
MATCH (p:Person {name: "Alicia"})
CALL apoc.path.expand(p, "FOLLOWS>|KNOWS", "", 1, 3)
YIELD path
RETURN path, length(path) AS hops
ORDER BY hops;
```

Table 3. Results

path	hops
(:Person:Product {name: "Alicia"})-[:FOLLOWS]→(:Person:Field {name: "Joe"})	1
(:Person:Product {name: "Alicia"})-[:FOLLOWS]→(:Person:Sales {name: "Jonny"})	1
(:Person:Product {name: "Alicia"})-[:KNOWS]→(:Person:Product {name: "Jake"})	1

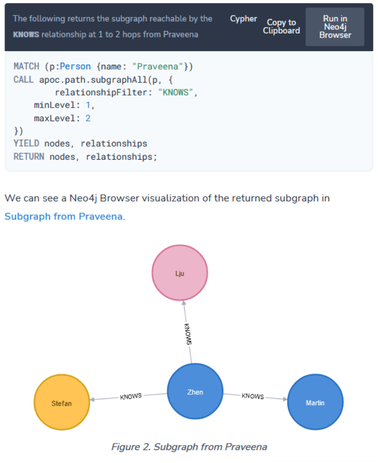


https://neo4j.com/labs/apoc/4.1/graph-querying/expand-paths/

Expanding to Subgraph

APOC also has a procedure for **expanding to a subgraph.**

The procedure returns **nodes** and **relationships**, excluding the starting node.



The following returns the subgraph reachable by the **KNOWS** relationship at 1 to 2 hops from Praveena

```
MATCH (p:Person {name: "Praveena"})
CALL apoc.path.subgraphAll(p, {
        relationshipFilter: "KNOWS",
    minLevel: 1,
    maxLevel: 2
})
YIELD nodes, relationships
RETURN nodes, relationships;
```

We can see a Neo4j Browser visualization of the returned subgraph in Subgraph from Praveena.

Figure 2. Subgraph from Praveena

https://neo4j.com/labs/apoc/4.1/graph-querying/expand-subgraph/

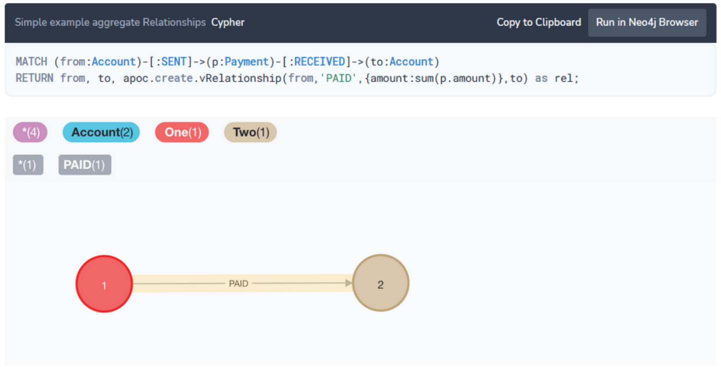
Virtual Nodes & Relationships

- APOC allows us to create **virtual nodes and relationships**, which don't exist in the graph – they are **stored in memory** and **returned to the UI**.
- A good use case for this is representing **transitive relationships** between nodes – rather than visualising the entire graph from point A to point B, we could draw a link between A and B with a property set to the total distance between them.

https://neo4j.com/labs/apoc/4.1/virtual/virtual-nodes-rels/

Virtual Nodes & Relationships

For example, we can create a **virtual relationship** between two accounts where some money has been sent from one account to another:



Simple example aggregate Relationships

```
MATCH (from:Account)-[:SENT]->(p:Payment)-[:RECEIVED]->(to:Account)
RETURN from, to, apoc.create.vRelationship(from,'PAID',{amount:sum(p.amount)},to) as rel;
```

https://neo4j.com/labs/apoc/4.1/virtual/virtual-nodes-rels/

Summary

- In this lecture we have looked at describing **variable length paths** in Cypher.
- We have also seen APOC's Path Expander module, which allows more control over path expansion.
- APOC also provides the ability to create **virtual nodes and relationships,** giving us a lot more control over our Neo4j visualisations.

Copyright Notice

Copyright Notice