Advanced Cypher Concepts

UNION

UNION

- **`UNION`** two or more full Cypher queries together
- aliases in **`RETURN`** must be exactly the same
- **`UNION ALL`** if you don't want to remove duplicates

UNION Example

```
// note that aliases are the same
MATCH (a:Actor)
RETURN a.name as name
UNION
MATCH (d:Director)
RETURN d.name as name
```

CASE WHEN

CASE/WHEN

- just like most **`SQL CASE/WHEN`** implementations
- adapt your result set to change values
- adapt your result set for easier grouping
- use for predicates in **`WHERE`**
- can be in both forms:
  - **`CASE val WHEN 1 THEN ... END`**
  - **`CASE WHEN val = 1 THEN ... END`**

CASE/WHEN example

```
... // group by age-range
RETURN CASE
  WHEN p.age < 20 THEN 'under 20'
  WHEN p.age < 30 THEN 'twenties'
  ...
  END AS age_group
```

Collections

powerful datastructure handling

Cypher Collections

- first class citizens in Cypher's type system
- nested collections in Cypher (not in properties)
- collection predicates: **`IN, SOME, ALL, SINGLE`**
- collection operations: **`extract,filter,reduce,`**

  `[x IN list WHERE predicate(x) | expression(x)]`
- slice notation **[1..3]**, **map[key]** access
- clauses: **UNWIND**, **FOREACH**

Collections

```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)
WHERE a.name STARTS WITH "T"
WITH a, count(m) AS cnt,
   collect(m.title) AS movies
WHERE cnt > 5
RETURN {name: a.name, movies: movies} as data
ORDER BY length(data["movies"]) DESC
LIMIT 10
```

Exercise: Collection Basics

1. get the first element of **`[1,2,3,4]`**
2. get the last element of **`[1,2,3,4]`**
3. get the elements of **`[1,2,3,4]`** that are above **2**
4. find the sum of **`[1,2,3,4]`**
5. get the actors for the top **5** rated movies
6. get the movies for the top actors (from the previous query)

Answers: Collections Basics

```
1.  // sum of items in [1,2,3,4]
    RETURN reduce(acc=0, x in [1,2,3,4] | acc + x)
2.  // first element in [1,2,3,4]
    RETURN [1,2,3,4][0]
3.  // last element in [1,2,3,4]
    RETURN [1,2,3,4][-1]
4.  // get the elements that are above 2
    RETURN [x in [1,2,3,4] WHERE x > 2]
```

Fun with Collections

```
WITH range(1,9) AS list
WHERE all(x IN list WHERE x < 10)
  AND any(x in [1,3,5] WHERE x IN list)
WITH [x IN list WHERE x % 2 = 0 | x*x ] as squares
UNWIND squares AS s
RETURN s
```

Dynamic property lookup

- for maps, nodes, relationships
- keys(map)
- properties(map)
- map[key]

```
WITH "title" AS key
MATCH (m:Movie)
RETURN m[key]
```

Dynamic property lookup

```
MATCH (movie:Movie)
UNWIND keys(movie) as key
WITH movie, key
WHERE key ENDS WITH "_score"
RETURN avg(movie[key])
```

FOREACH

- lets you iterate over a collection and update the graph (`CREATE`, `MERGE`, `DELETE`)
- delete nodes/rels from a collection (or a path) without `UNWIND`
- consider (and test) `FOREACH` vs `UNWIND`, one or the other may be somewhat faster

FOREACH example

```
// we'll create some nodes
// from properties in a collection
WITH ["Drama","Action",...] AS genres
FOREACH(name in genres|
  CREATE (:Genre {name:name})
)
```

UNWIND

- `UNWIND` lets you transform a collection into rows
- very useful for massaging collections, sorting, etc.
- allows collecting a set of nodes to avoid requerying, during aggregation

UNWIND Example

```
MATCH (m:Movie)<-[:ACTED_IN]-(p)
WITH collect(p) AS actors,
     count(p) AS actorCount, m
UNWIND actors AS actor
RETURN m, actorCount, actor
```

UNWIND Example:
Post-UNION Processing

```
MATCH (a:Actor)
RETURN a.name AS name
UNION
MATCH (d:Director)
RETURN d.name AS name
// no means for sort / limit
```

https://neo4j.com/blog/cypher-union-query-using-collect-clause/

Another UNWIND Example
Post-UNION Processing

```
MATCH (a:Actor)
WITH collect(a.name) AS actors
MATCH (d:Director)
WITH actors, collect(d.name) AS directors
UNWIND (actors + directors) AS name
RETURN DISTINCT name
ORDER BY name ASC LIMIT 10
```

Quick Review of Update Operations

CREATE

creates nodes, relationships and patterns

CREATE

nodes, relationships, structures

```
CREATE (m:Movie {title:"The Matrix", released:1999})
UNWIND ["Lilly Wachowski","Lana Wachowski"] AS name
MATCH  (d:Director {name:name})
CREATE (d)-[:DIRECTED]->(m)
```

MERGE

matches or creates

MERGE

get or create

```
UNWIND {data} AS pair
MERGE (m:Movie {id:pair.movieId})
  ON CREATE SET m += pair.movieData
  ON MATCH  SET m.updated = timestamp()
MERGE (p:Person {id:pair.personId})
  ON CREATE SET p += pair.personData
MERGE (p)-[r:ACTED_IN]->(m)
  ON CREATE SET r.roles = split(pair.roles,";")
```

Dense node merging + matching

- Picks the side of smallest cardinality when **`MERGE`**ing relationships
- Particularly noticeable when you have a dense node follower pattern, for example, **`(:Movie)-[:HAS_GENRE]->(comedy)`**

SET, REMOVE

update attributes and labels

SET, REMOVE

```
MATCH (a:Person)
WHERE (a)-[:ACTED_IN]->()
SET a:Actor

MATCH (m:Movie) WHERE exists(m.movieId)
SET m.id = m.movieId
REMOVE m.movieId
```

DELETE

remove nodes & relationships

Delete

- DELETE node or relationships
- Must delete all relationships before deleting node

```
// will delete Tom Hanks if no
// relationships exists
MATCH (p:Person {name: "Tom Hanks"})
DELETE p
```

Detach Delete

- Delete node + relationships attached to it

```
// will delete Tom Hanks and all his
// relationships
MATCH (p:Person {name: "Tom Hanks"})
DETACH DELETE p
```

Delete Everything in Database

- Delete node + relationships attached to it

```
// will delete everything in db
MATCH (n)
DETACH DELETE n;
```

- `Watch out for bulk updates (>1M records)`

INDEXes, CONSTRAINTs

represent optional

schema

Indexes Overview

- based on labels
- can be hinted
- used for exact lookup, text and range queries
- automatic

Index Example

```
// create and drop an index
CREATE INDEX ON :Director(name);
DROP INDEX ON :Director(name);
```

Index Example

```
// use an index for a lookup
MATCH (p:Person)
WHERE p.name="Clint Eastwood"
RETURN p;
```

Range queries

- Index supported range queries
- For numbers and strings
- Pythonic expression syntax

Range queries

```
MATCH (p:Person)
WHERE p.born > 1980 RETURN p;
```

```
MATCH (m:Movie)
WHERE 2000 <= m.released < 2010 RETURN m;
```

```
MATCH (p:Person)
WHERE p.name >= "John"
RETURN p;
```

Text Search

- **STARTS WITH**
- **ENDS WITH**
- **CONTAINS**
- are index supported

Text Search

```
MATCH (p:Person)
WHERE p.name STARTS WITH "John" RETURN p;

MATCH (p:Person)
WHERE p.name CONTAINS "Wachowski" RETURN p;

MATCH (m:Movie)
WHERE m.title CONTAINS "Matrix" RETURN m;
```

Index Hints USING SCAN

- syntax: **`USING INDEX`** `m:Movie(title)`
- you can force a label scan on lower cardinality labels: **`USING SCAN`** `m:Comedy`

```
MATCH (a:Actor)-->(m:Movie:Comedy)
RETURN count(distinct a);

vs

MATCH (a:Actor)-->(m:Movie:Comedy)
USING SCAN m:Comedy
RETURN count(distinct a);
```

Constraints

- Constraints on label, property combinations
- **UNIQUE** constraints available
- **EXIST**ence constraints in enterprise version for properties on nodes and relationships
- creates accompanying index automatically

```
CREATE CONSTRAINT ON (p:Person)
ASSERT p.id IS UNIQUE
```

CONSTRAINTs

```
CREATE CONSTRAINT ON (p:Person)
ASSERT p.id IS UNIQUE
CREATE CONSTRAINT ON (p:Person)
    ASSERT exists(p.name)
CREATE CONSTRAINT ON (:Person)-[r:ACTED_IN]->(:Movie)
    ASSERT exists(r.roles)
```

MORE Cypher

Map Projections,

Pattern Comprehension

Map Projections (Neo4j 3.1)

```
MATCH (m:Movie)
RETURN m { .title, .genres } AS movie

MATCH (m:Movie)<-[:ACTED_IN]-(p:Person)
WITH collect(p) AS people
RETURN m { .title, .genres,  cast: [p in people |
 p.name] } AS movie
```

Pattern Comprehensions (Neo4j 3.1)

```
MATCH (m:Movie)
RETURN m.title, [ (m)<-[:ACTED_IN]-(p:Person) | p.name ] AS
 cast


MATCH (m:Movie)
RETURN m { .title, .genres,
   cast: [ (m)<-[r:ACTED_IN]-(p:Person) | {name: p.name,
 roles: r.roles} ] }
AS movie
```

End of Module
Advanced Cypher Concept

Questions?

neo4j