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

- 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

Cypher collections

- first class citizens in Cypher's type system
- nested collections in Cypher (not in properties)
- collection predicates: `IN`, `ANY`, `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: Collections Basics

1. find the sum of [1,2,3,4]
2. get the first element of [1,2,3,4]
3. get the last element of [1,2,3,4]
4. get the elements of [1,2,3,4] that are above 2
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]

Dynamic property lookup

```
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

FOREACH

- iterate over a collection and update the graph

  (`CREATE`, `MERGE`, `DELETE`)
- delete nodes/rels from a collection (or a path)
- Try out `UNWIND` as well. One may be faster than the other.

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})
)
```

FOREACH example - conditional logic

```
FOREACH(ignoreMe IN CASE WHEN EXISTS(person.country)
                         THEN [1]
                         ELSE [] END |
  MERGE (country:Country {name: person.country})
  MERGE (person)-[:LIVES_IN]->(country)
);
```

UNWIND

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. Especially useful 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
```

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
```

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 ENDS WITH "Matrix"
RETURN m;
```

Index Hints: USING INDEX

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

```
USING SCAN m:Comedy
```

Index Hints: USING SCAN

```
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);
```

Composite Indexes

Neo4j doesn’t have composite indexes at the moment but we can create a "dummy" property to simulate one. (Since 3.4 we has it now)

```
CREATE(:Director {_id: ["id1", "id2", "id3"] });
CREATE INDEX ON :Director(_id);
```

Composite Indexes

Composite keys allow multiple properties to be indexed for a Label.

```
CREATE INDEX ON :Movie(title, tagline);
```

Constraints

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

Constraints

```
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)
```

Map Projections

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

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

Map Projections and Pattern Comprehensions

(>= Neo4j 3.1)

Pattern Comprehensions

```
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?