

# Cypher Refresher

(Just in Case)



# Resources for learning Cypher



- Cypher Reference Card  
[neo4j.com/docs/cypher-refcard/](https://neo4j.com/docs/cypher-refcard/)
- Cypher Railroad Diagrams  
[bit.ly/cypher-railroad](https://bit.ly/cypher-railroad)
- Neo4j Developer Pages  
[neo4j.com/developer/cypher](https://neo4j.com/developer/cypher)
- Neo4j Documentation  
[neo4j.com/docs](https://neo4j.com/docs)



# Cypher Query Structure



**MATCH** pattern

**WHERE** predicate

**RETURN/WITH** expression AS alias ...

**ORDER BY** expression

**SKIP** ... **LIMIT** ...



# Case sensitivity



- Cypher keywords/clauses are mostly case insensitive
- But, several things in the datastore are case sensitive:
  - Labels
  - Relationship types
  - Property names (keys)
  - Variables you use in Cypher

# Labels



- labels are like type tags for nodes
- label-based indexes and constraints

**CREATE** (p:Person) *// create Labeled node*

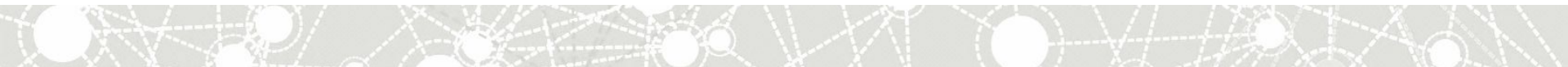
**SET** p:Person *// set Label on node*

**REMOVE** p:Person *// remove Label*

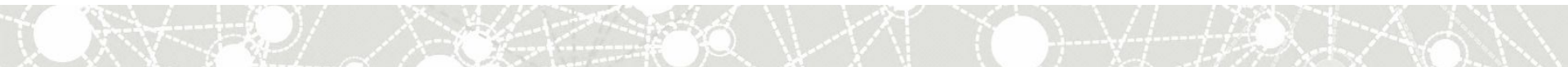
**MATCH** (p:Person) *// match nodes with Label*

**WHERE** p:Person *// Label predicate*

**RETURN** labels(p) *// get Label collection*



# MATCH



# MATCH patterns



- pattern matching - describe your traversal in a pattern
- use labels in your pattern to give your query starting points
- create new variables as the query matches the pattern



## MATCH examples

*// a simple pattern, with RELTYPE*

**MATCH** (n)-[:LINK]-(m)

*// match a complex pattern*

**MATCH** (n)-->(m)<--(o), (p)-->(m)

*// match a variable-length path*

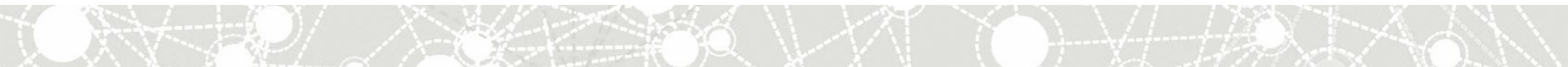
**MATCH** p=(n)-[:LINKED\*]-()

*// use a specialized matcher*

**MATCH** p=**shortestPath**((n)-[\*]-(o))



# WHERE



# WHERE



- use MATCH to find patterns, and WHERE to filter them
- use patterns as predicates in WHERE  
e.g. `WHERE NOT EXISTS ((n)-->())`
- can't create new identifiers in WHERE - only predicates on existing identifiers defined in MATCH or WITH  
e.g. we can't do `WHERE NOT EXISTS ((n)-->(newIdentifier))`

## WHERE examples

*// filter on a property value*

**WHERE** n.name = "Andrés"

*// filter with predicate patterns*

**WHERE NOT** (n)<--(m)

*// filter on path/collection length*

**WHERE** size(p) > 3

*// filter on multiple predicates*

**WHERE** n.born < 1980 **AND** n.name =~ "A.\*"

# RETURN



# RETURN



- like SQL's SELECT: specify the projection you want to see in results
- alias results with AS
- calculate expressions as they're returned (math, etc.)
- aggregations: collect, count, statistical

## RETURN examples

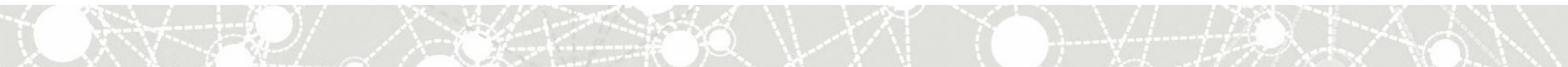


*// implicit group by n, count(\*)*

**RETURN** n, count(\*) **AS** count

*// collect things into a collection*

**RETURN** n, collect(r)

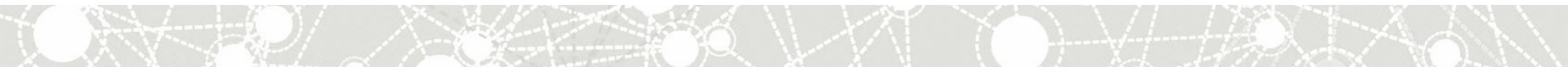


# ORDER BY/LIMIT/SKIP

# ORDER BY/SKIP/LIMIT



- Cypher doesn't guarantee ordering unless you ORDER BY
- LIMIT, SKIP let us restrict the results returned
- things you order by must be in the RETURN/WITH clause
- all optional clauses (e.g. you can do LIMIT without ORDER BY)





## ORDER BY/SKIP/LIMIT examples

*// descending sort, limit 5*

**RETURN** n, count(\*) **AS** count

**ORDER BY** count **DESC**

**LIMIT** 5

*// get the next 5*

**RETURN** n, count(\*) **as** count

**ORDER BY** count **DESC**

**SKIP** 5

**LIMIT** 5

# Data flow using WITH



# WITH



- separates query-parts and controls data flow
- WITH is like RETURN
  - it can aggregate, project, order, paginate, distinct
  - filter with WHERE (WITH + WHERE = "HAVING")
- needs variable (alias) for each expression
- controls visibility of variables in the next part of the query
- You can use as many WITHs as you need

## WITH examples



*// intermediate projection, ordering, pagination*

**WITH** a.name **as** name, a.born **as** born

**ORDER BY** born **DESC**

**LIMIT** 5

...

*// aggregation + filter*

**WITH** m, count(\*) **as** cast, collect(a.name) **as** actors

**WHERE** cast > 5

**RETURN** m.title, actors

# Exercise: Translate English to Cypher



## Task: Complex Graph Operation

**Find** all Actors and Movies they acted in

**Whose** name contains the letter "a"

**Aggregate** the frequency and movie titles

**Filter by** who acted in more than 5 movies

**Return** their name, birth year and movie titles

**Ordered by** number of movies

**Limited** to top 10



## SQL version of the query



```
SELECT a.name, a.born,  
       group_concat(m.title) AS movies,  
       count(*) AS cnt  
FROM   actors AS a JOIN actor_movie  ON (a.id = actor_movie.actor_id)  
JOIN   movies AS m  
      ON (actor_movie.movie_id = m.id)  
WHERE  a.name LIKE "%a%"  
GROUP BY a.name, a.born  
HAVING cnt > 5  
ORDER BY cnt DESC
```

## Exercise: Write and execute the query

**Find** all Actors and Movies they acted in

**Whose** name contains letter "a"

**Aggregate** the frequency and movie titles

**Filter by** who acted in more than 5 movies

**Return** their name, birth year and movie titles

**Ordered by** number of movies in **descending** order

**Limited** to top 10

Write the query one step at a time and don't forget to use the Cypher refcard if you get

stuck: [neo4j.com/docs/cypher-refcard](https://neo4j.com/docs/cypher-refcard) :Play movies ← load data



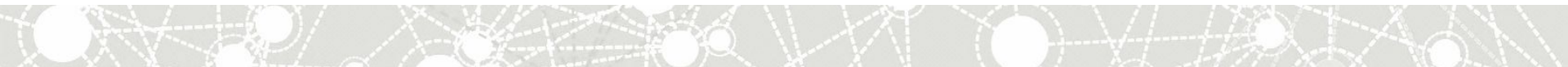
**Solution on next slide**

## Solution: Complex Graph Query



1. find people (click :Person in browser)
2. add limit
3. find people whose name contains the letter "a"
4. find people whose name contains the letter "a", who acted in a movie
5. return aggregation
6. add ordering
7. introduce WITH for in-between filter

# Breakdown



**MATCH**  
describes the pattern



# MATCH the pattern



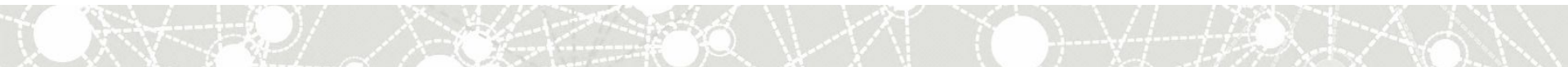
```
MATCH (a:Person)
```

```
RETURN a
```

```
LIMIT 10
```



**WHERE**  
**filters the result set**



## Filter using WHERE



```
MATCH (a:Person)
WHERE a.name CONTAINS "a"
RETURN a
LIMIT 10
```



**MATCH**  
describes the pattern





## MATCH the pattern



```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)
WHERE a.name CONTAINS "a"
RETURN a
LIMIT 10
```



**RETURN**  
returns the results



## RETURN the results



```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)
WHERE a.name CONTAINS "a"
RETURN a.name, a.born
LIMIT 10
```



# Aggregation with auto grouping



# Aggregation



```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)
WHERE a.name CONTAINS "a"
```

```
RETURN a.name,
        a.born,
        count(m) AS cnt,
        collect(m.title) AS movies
```

```
LIMIT 10
```



# ORDER BY / LIMIT / SKIP

## Sort and paginate

## ORDER BY LIMIT - Paginate



```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)
WHERE a.name CONTAINS "a"
RETURN a.name,
       a.born,
       count(m) AS cnt,
       collect(m.title) AS movies
ORDER BY size(movies) DESC
LIMIT 10
```

**WITH + WHERE**  
**computes intermediate**  
**results + filter**





## WITH + WHERE - filter



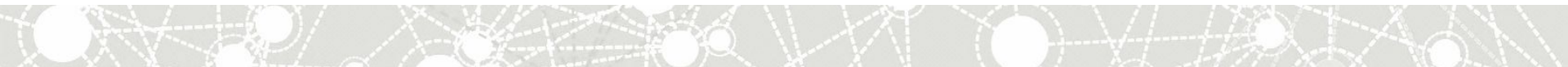
```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)
WHERE a.name CONTAINS "a"
WITH a,
      count(m) AS cnt,
      collect(m.title) AS movies
WHERE cnt > 5
RETURN a.name, a.born, movies
ORDER BY cnt DESC
LIMIT 10
```

## Solution



```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)
WHERE a.name CONTAINS "a"
WITH a,
      count(m) AS cnt,
      collect(m.title) AS movies
WHERE cnt > 5
RETURN a.name, a.born, movies
ORDER BY size(movies) DESC
LIMIT 10
```

# Quick Review of Update Operations



# **CREATE**

**creates nodes,  
relationships, and patterns**



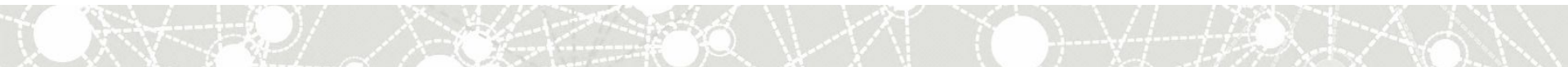
# CREATE nodes, relationships, structures



```
CREATE (m:Movie {title:"The Matrix", released:1999})  
WITH m  
UNWIND ["Lilly Wachowski","Lana Wachowski"] AS name  
MERGE (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,";")
```

```
{
  "movieId": 1,
  "personId": 42,
  "movieData": {
    "title": "Something Famous"
  },
  "personData": {
    "name": "Someone Famous"
  },
  "roles": "Cool Hand Luke;Dirty
Dave"
}
```

## Dense node merging and matching

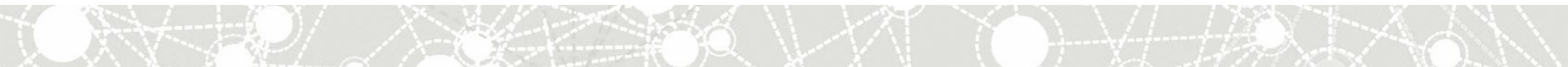


- The Cypher compiler picks the side of smallest cardinality when MERGEing relationships
- This is particularly noticeable when you have a dense node follower pattern.  
e.g. `(:Movie)-[:HAS_GENRE]->(comedy)`



# SET, REMOVE

## update attributes and labels



# SET



```
MATCH (a:Person)
```

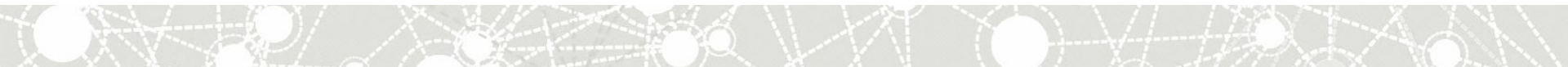
```
WHERE (a)-[:ACTED_IN]->()
```

```
SET a:Actor
```

```
MATCH (m:Movie)
```

```
WHERE exists(m.movieId)
```

```
SET m.id = m.movieId
```



# REMOVE



```
MATCH (m:Movie)
```

```
WHERE exists(m.movieId)
```

```
REMOVE m.movieId
```

## SET + REMOVE

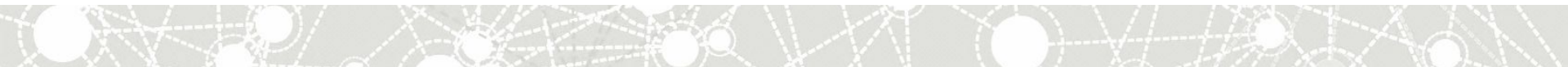


```
MATCH (m:Movie)
```

```
WHERE exists(m.movieId)
```

```
SET m.id = m.movieId
```

```
REMOVE m.movieId
```



# **DELETE**

## **remove nodes and 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

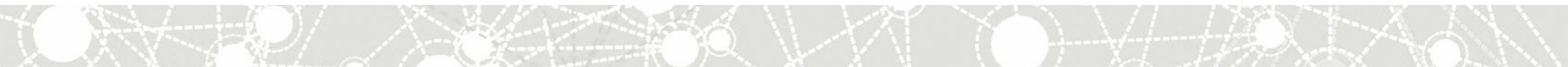


- Delete node + relationships attached to it

*// will delete Tom Hanks and all his  
// relationships*

**MATCH** (p:Person {name: "Tom Hanks"})

**DETACH DELETE** p



# Delete everything in the database



- Delete node + relationships attached to it

*// will delete everything in db*

**MATCH** (n)

**DETACH DELETE** n

Be careful when doing this with datasets > 1m nodes - all the nodes get loaded into memory before being deleted!





# End of Cypher Refresher

Questions?

