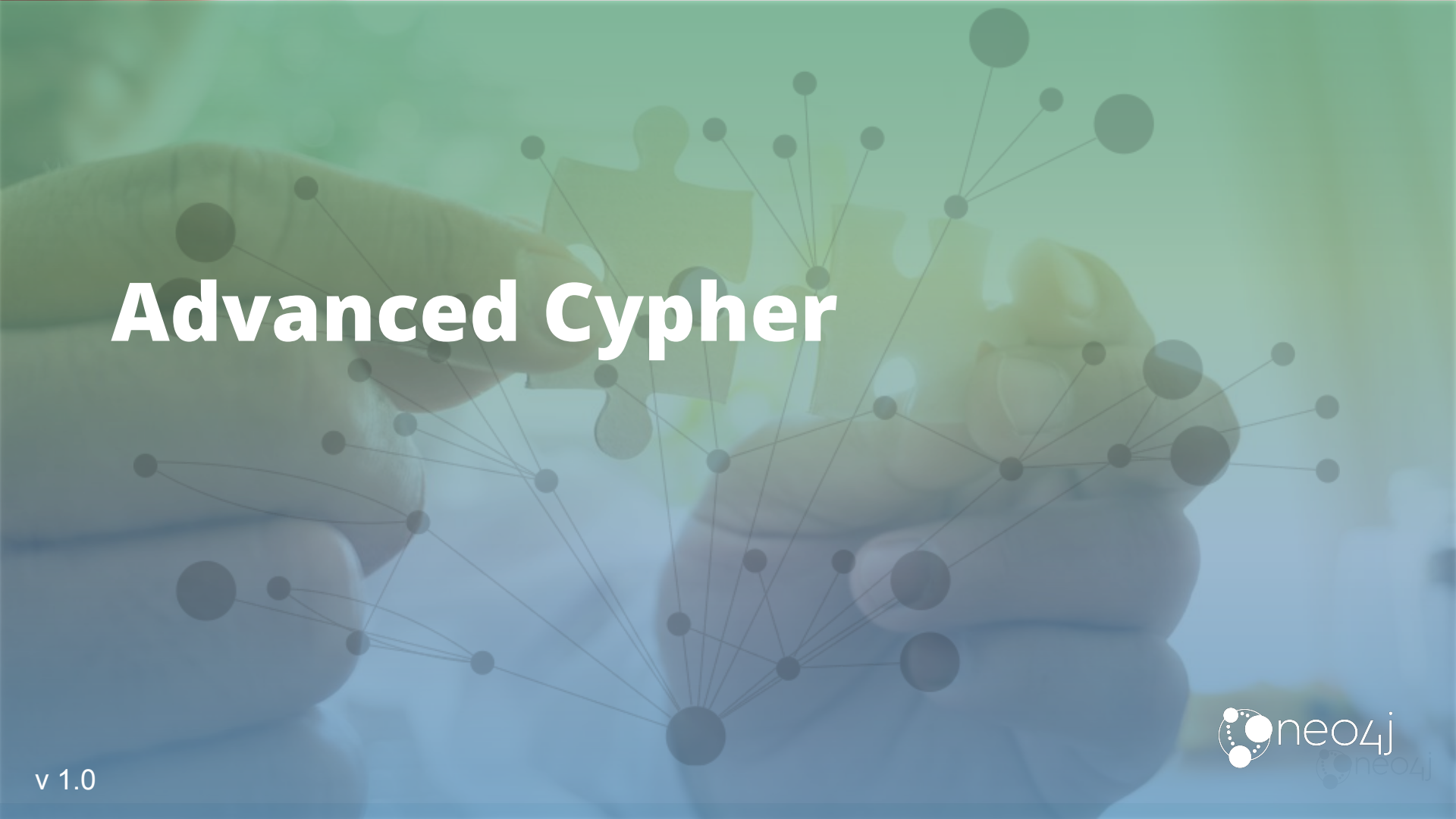
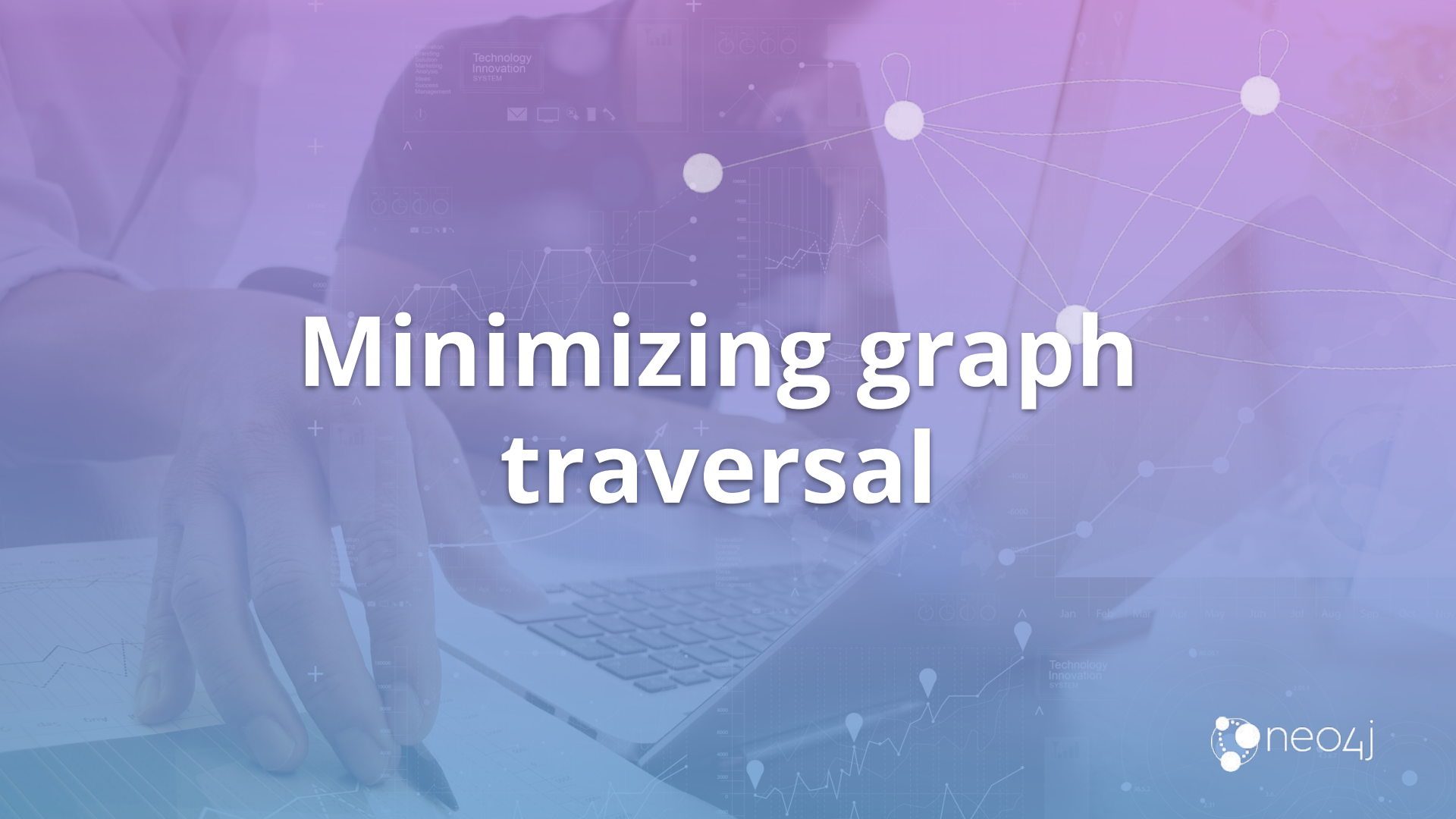

Advanced Cypher
v 1.0
neo4j


Getting More Out of Cypher
neo4j

Overview

At the end of this module, you should be able to:

- Minimize graph traversals for queries.
- Use APOC for graph traversal.
- Use UNWIND as well as pattern and list comprehension.
- Combine query results.
- Import normalized and denormalized data into the graph.
- Aggregate data and work with lists and maps.
- Iterate and perform conditional processing.



Minimizing graph traversal

neo4j

Minimizing graph traversal

A key to writing Cypher code that performs as efficiently as possible is to understand how nodes and relationships are traversed in a single execution.

We will explore these common graph traversal scenarios:

- Single path specified in the MATCH clause
- Multiple MATCH clauses
- Multiple paths specified in the MATCH clause
- Relationship path uniqueness
- Returning distinct nodes
- Using APOC's path expander procedures

Note: You should rely heavily on profiling your queries with the PROFILE keyword or using the Query Log Analyzer that you will learn about later in this module.

Review: Paths and MATCH clauses

Before we learn about some common graph traversal scenarios, let's review some basic graph traversal behavior using:

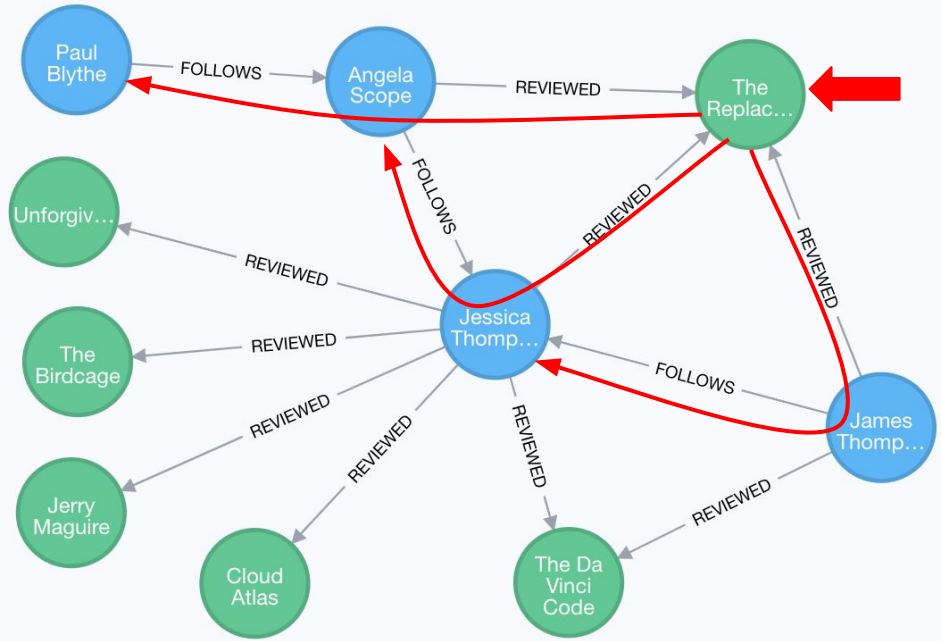
- Single path
- Multiple MATCH clauses
- Multiple paths in a MATCH clause

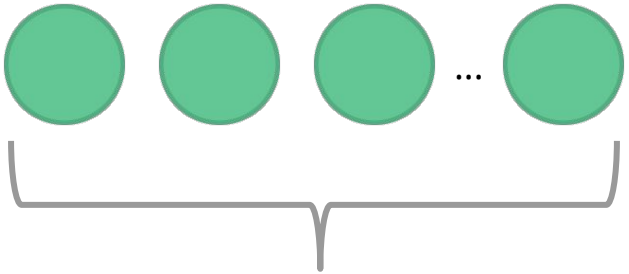
Review: One path specified in MATCH clause

Here is the query with multiple traversals at runtime:

```
MATCH (follower:Person)-[:FOLLOWS]->
(reviewer:Person)-[:REVIEWED]->(m:Movie)
WHERE m.title = "The Replacements"
RETURN follower.name, reviewer.name
```

follower.name	reviewer.name
"Paul Blythe"	"Angela Scope"
"Angela Scope"	"Jessica Thompson"
"James Thompson"	"Jessica Thompson"





If no index on :Movie(title), first traverse all Movie nodes to find the anchor node.

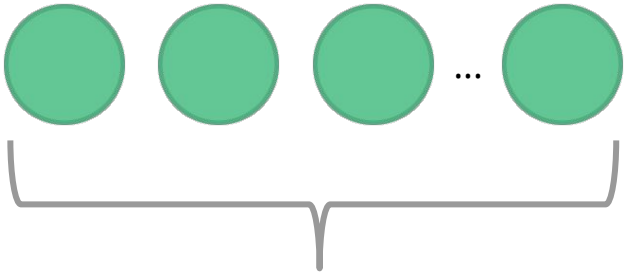
neo4j

Review: Multiple MATCH clauses

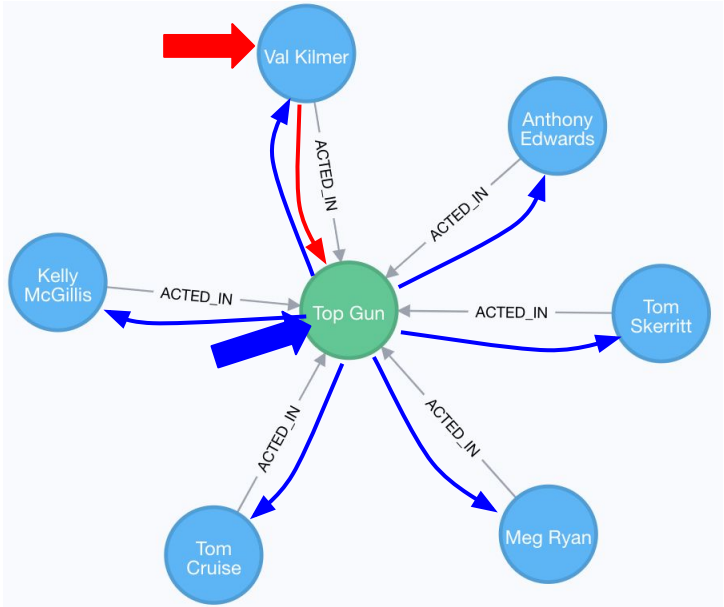
Here is a query with 2 separate MATCH clauses:

```
MATCH (a1:Person)-[:ACTED_IN]->(m:Movie)
MATCH (a2:Person)-[:ACTED_IN]->(m)
WHERE a1.name = 'Val Kilmer'
RETURN m.title as movie , collect(a2.name)
```

movie	collect(a2.name)
"Top Gun"	["Meg Ryan", "Tom Skerritt", "Anthony Edwards", "Val Kilmer", "Kelly McGillis", "Tom Cruise"]



If no index on :Movie(title), first traverse all Movie nodes.



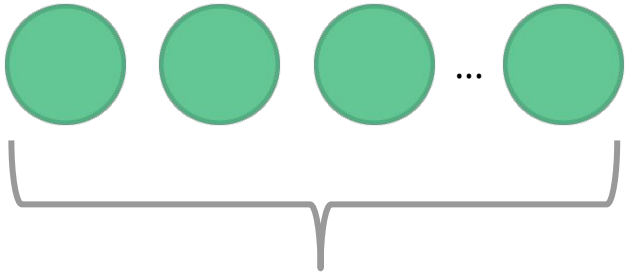
neo4j

Review: Multiple paths in a MATCH clause

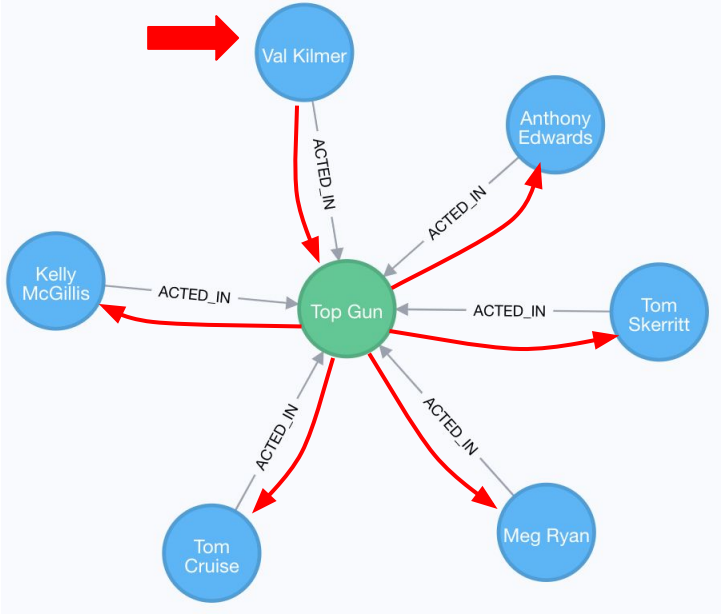
Here is the query with 2 paths specified in the MATCH clause:

```
MATCH (a1:Person)-[:ACTED_IN]->(m:Movie),
      (a2:Person)-[:ACTED_IN]->(m)
WHERE a1.name = 'Val Kilmer'
RETURN m.title as movie , collect(a2.name)
```

movie	collect(a2.name)
"Top Gun"	["Meg Ryan", "Tom Skerritt", "Anthony Edwards", "Kelly McGillis", "Tom Cruise"]



If no index on :Movie(title), first traverse all Movie nodes.



neo4j

Review: Using WITH - 1

- Intermediate results are used for further queries. (chaining results)
- WHERE, DISTINCT, ORDER BY, LIMIT, SKIP, UNWIND can be used with WITH, in order to change the result set before continuing.

```
MATCH (a:Person-[:ACTED_IN]->(m:Movie)
WITH m ORDER BY m.title
MATCH (m)<-[rv:REVIEWED]-(r:Person)
RETURN m.title, rv.rating, r.name
```

The Movie nodes are ordered when used for the second query

Review: Using WITH - 2

Person node, **a** is carried through to next query; **acted** is evaluated to filter results and also carried through

```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)
WITH a, count(m) AS acted
WHERE acted >= 2
WITH a, acted
MATCH (a)-[:DIRECTED]->(m:Movie)
RETURN a.name, acted, collect(m.title) AS directed
```

This is a different Movie node from the one returned in the first query because **m** was not carried through

```
$ MATCH (a:Person)-[:ACTED_IN]->(m:Movie) WITH a, count(m) AS acted WHERE acted >= 2 WITH a, acted MATCH (a)-
```

Table

a.name	acted	directed
"Tom Hanks"	12	["That Thing You Do"]
"Danny DeVito"	2	["Hoffa"]

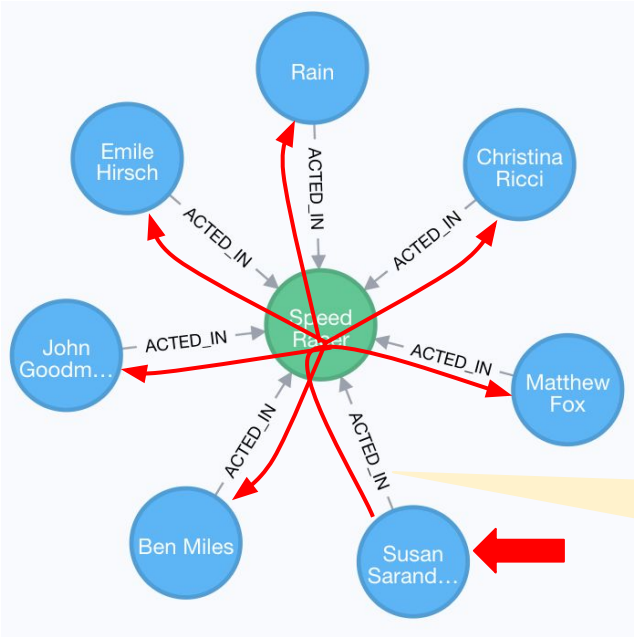
Relationship path uniqueness

During the execution of a single MATCH clause, Neo4j traverses relationships exactly once. This prevents infinite (circular) traversals.

Relationship path uniqueness - 1

Here is the query:

```
MATCH (actor:Person {name:'Susan Sarandon'})
      -[:ACTED_IN]->(m:Movie)<-[:ACTED_IN]-(coActor:Person)
RETURN m.title, coActor.name
```

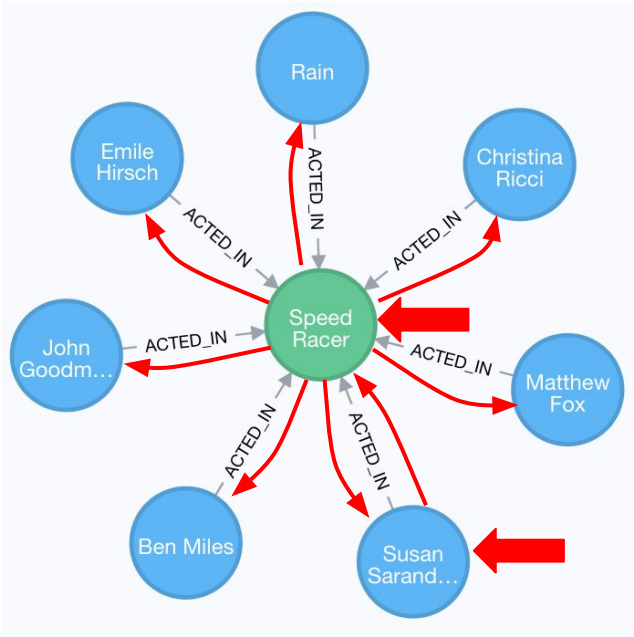


We can only traverse this relationship once per path. Since we used it to traverse from Susan Sarandon, we cannot use it to traverse back to her in the same path.

Relationship path uniqueness - 2

Here is the query:

```
MATCH (actor:Person {name:'Susan Sarandon'})-[:ACTED_IN]->(m:Movie)
MATCH  (m)<-[:ACTED_IN]-(coActor:Person)
RETURN m.title, coActor.name
```



Returning distinct nodes

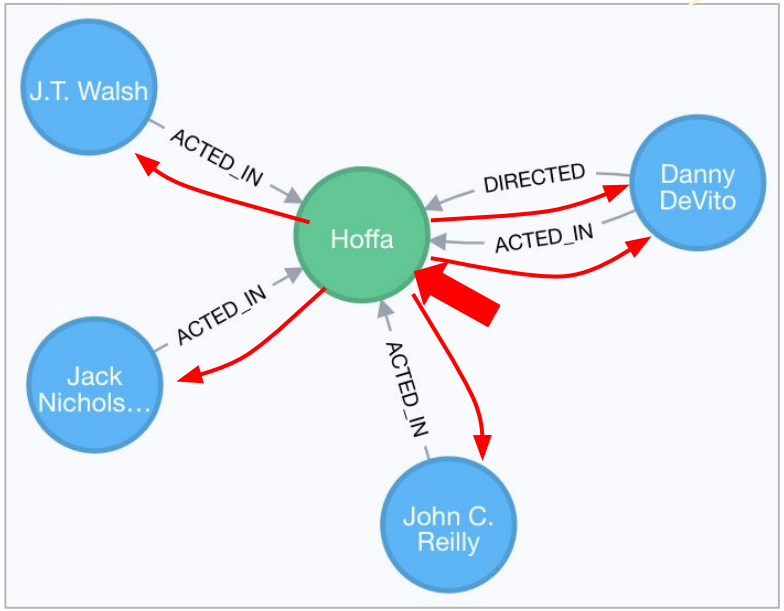
Cypher is all about finding all possible paths that match the given pattern. This is good for most queries, but not good for others, such as finding distinct connected nodes or distinct counts of connected nodes.

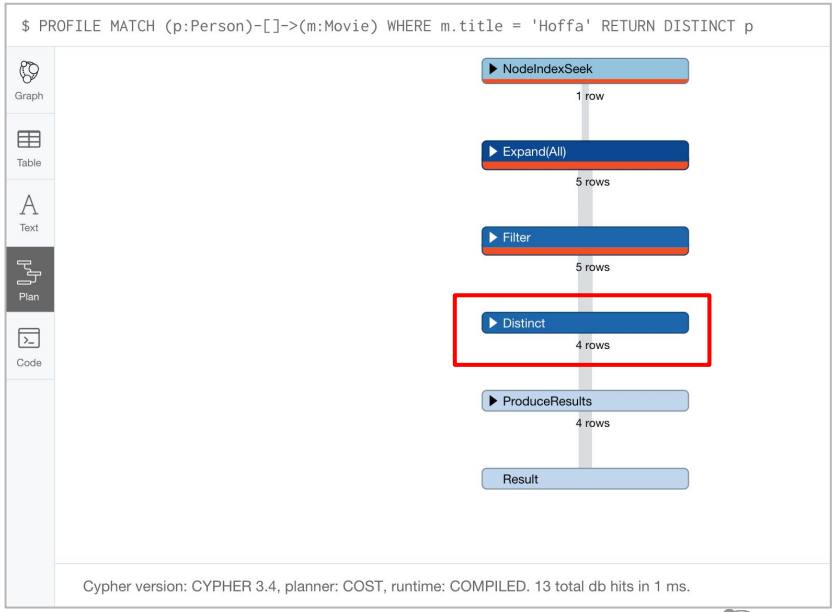
Returning distinct nodes - 1

Finding distinct nodes can be expensive because relationships are traversed, even though the result does not require them:

```
MATCH (p:Person)-[]->(m:Movie)
 WHERE m.title = 'Hoffa'
 RETURN DISTINCT p
```

Same nodes, relationships traversed with or without DISTINCT

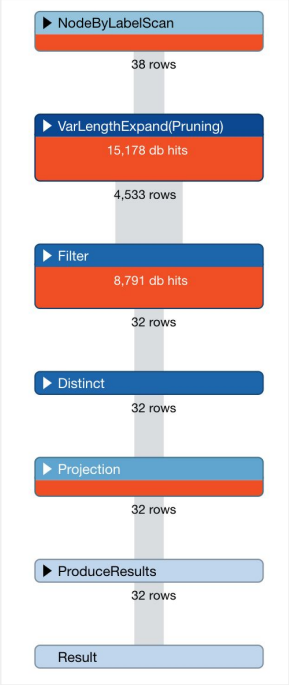




Returning distinct nodes - 2

Suppose we want to retrieve all movies that are connected to a particular person (by traversing Movie and Person nodes) that are up to 5 hops away.

```
// Do this, rather than RETURN DISTINCT m.title
// property access is expensive
MATCH (p:Person {name: 'Clint Eastwood'})-[*5]-(m:Movie)
WITH DISTINCT m
RETURN m.title
```




```
$ MATCH (p:Person {name: 'Clint Eastwood'})-[*5]-(m:Movie) WITH DISTINCT m RETURN m.title
```

Table | Text | Code

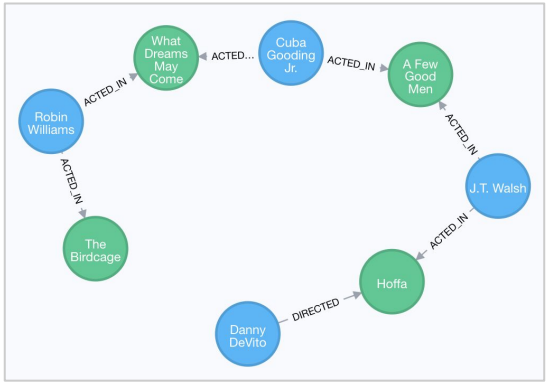
m.title
"The Matrix"
"The Matrix Reloaded"
"The Matrix Revolutions"
"The Devil's Advocate"
"A Few Good Men"
"Top Gun"
"Jerry Maguire"
"Stand By Me"
"As Good as It Gets"
"What Dreams May Come"
"You've Got Mail"
"Sleepless in Seattle"
"Joe Versus the Volcano"
"That Thing You Do"
"The Replacements"
"The Birdcage"

Started streaming 32 records after 10 ms and completed after 152 ms.

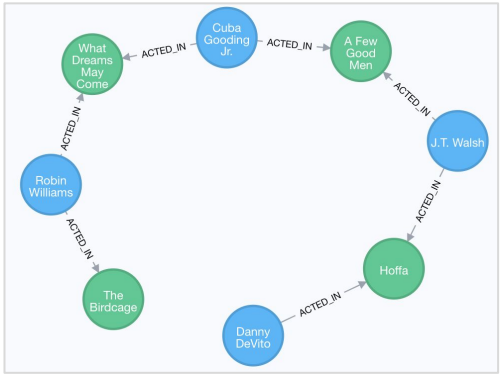
shortestPath

- **Bi-directional** breadth-first search
- Works with path predicates all() and none()
- If it cannot use bread-first (e.g. any()), it will fallback to depth-first search.

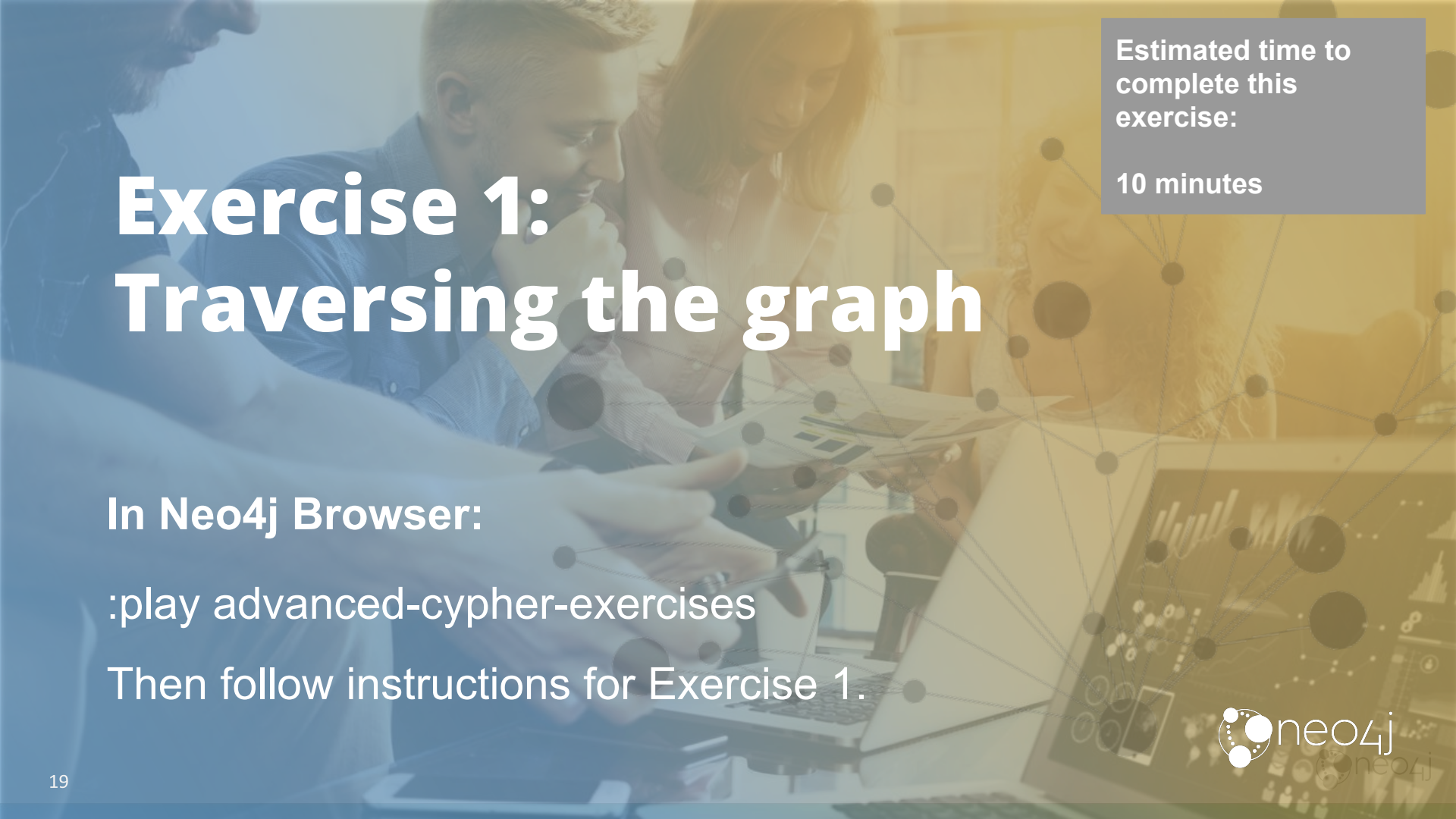
```
MATCH (p1:Person { name: 'Danny DeVito' }),
(m1:Movie { title: 'The Birdcage' }),
p = shortestPath((p1)-[*]-(m1))
RETURN p
```




```
MATCH (p1:Person { name: 'Danny DeVito' }),
(m1:Movie { title: 'The Birdcage' }),
p = shortestPath((p1)-[*]-(m1))
WHERE none(r IN relationships(p) WHERE type(r)=
'DIRECTED')
RETURN p
```



neo4j



Estimated time to complete this exercise:

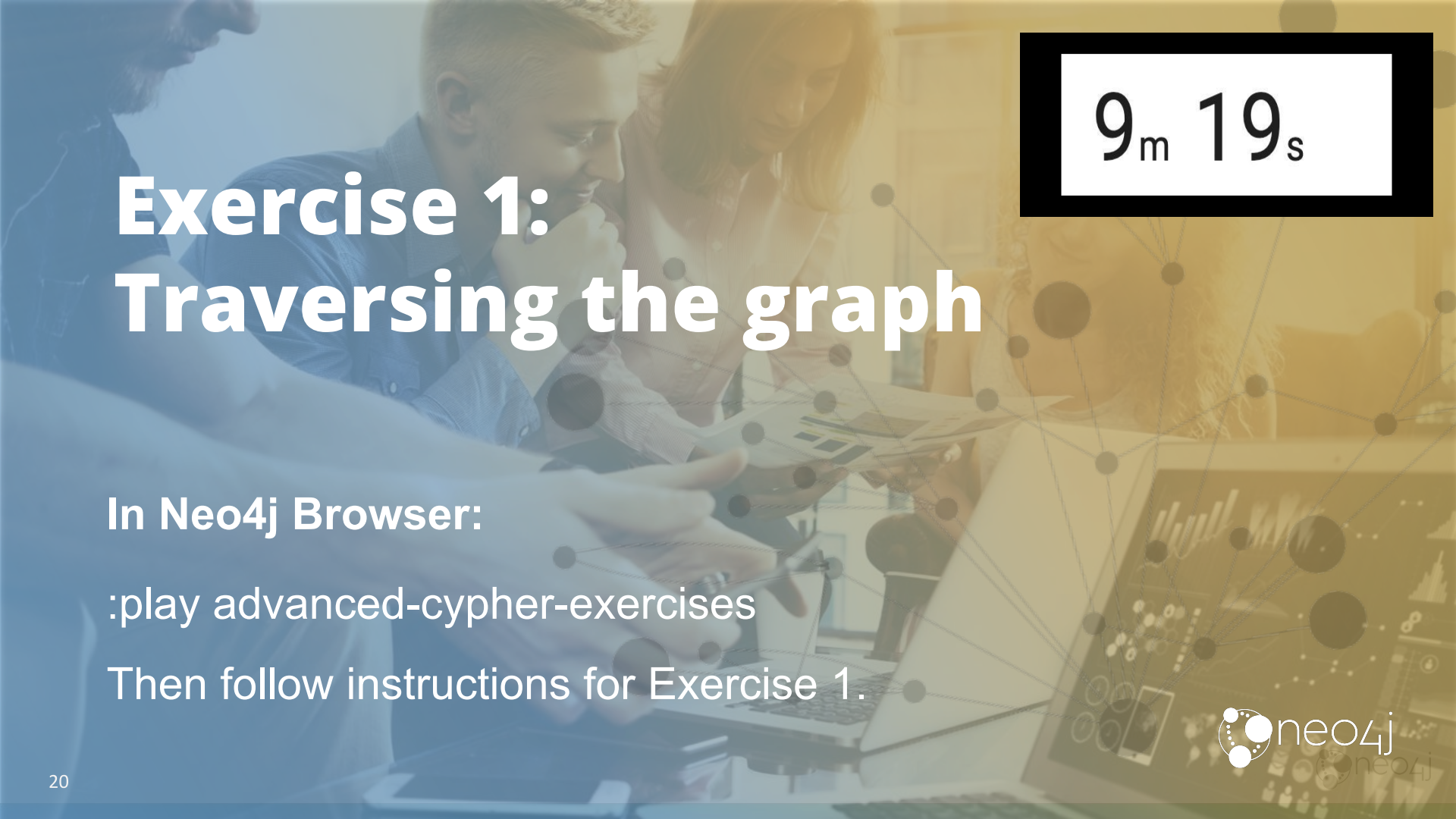
10 minutes

Exercise 1: Traversing the graph

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 1.

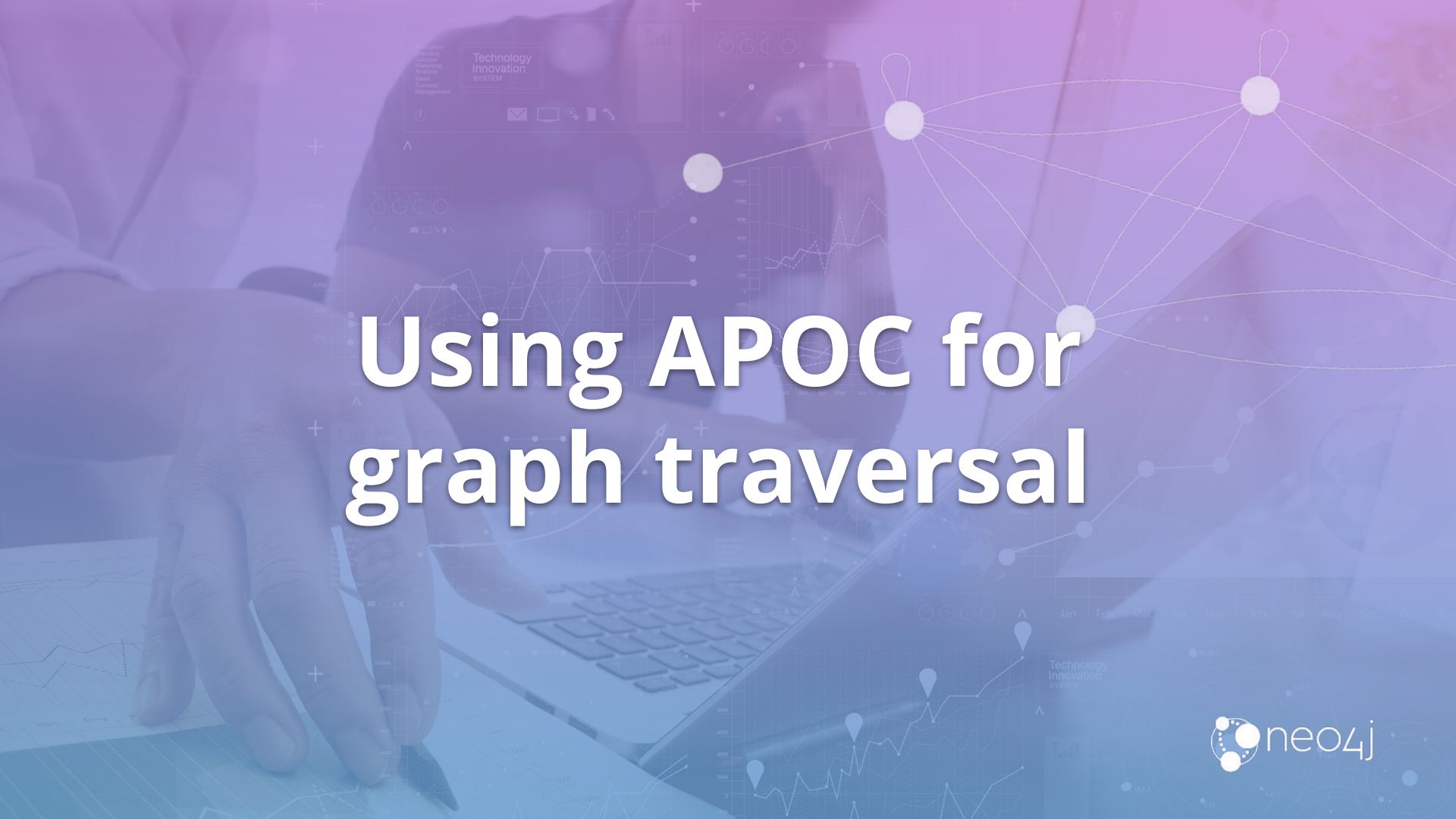


Exercise 1: Traversing the graph

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 1.



Using APOC for graph traversal

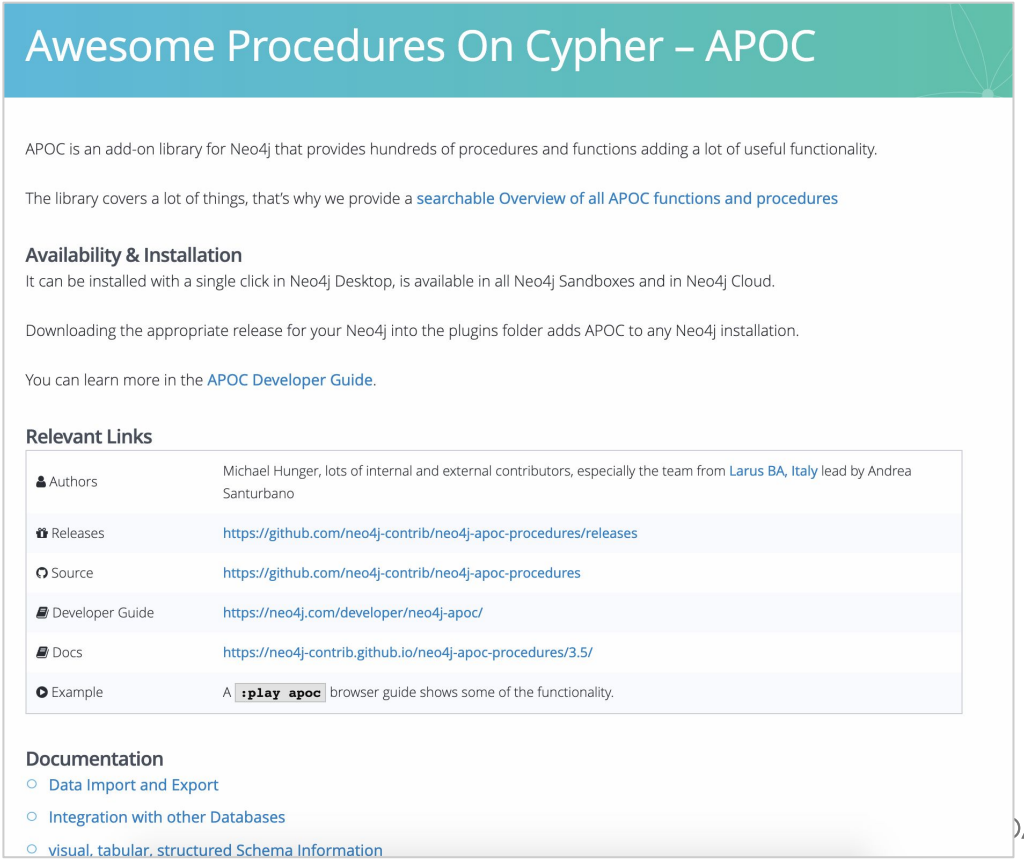
neo4j

About APOC

- Large standard library of utility functions and procedures
- Actively developed - many contributors
- Makes Cypher easier to use
- Enable some specific use-cases
- Plan is it migrate some of the functionality into the Neo4j product

APOC documentation

- Installation instructions
- Videos
- Searchable overview table
- Detailed explanation
- Examples



Awesome Procedures On Cypher – APOC

APOC is an add-on library for Neo4j that provides hundreds of procedures and functions adding a lot of useful functionality.

The library covers a lot of things, that's why we provide a searchable Overview of all APOC functions and procedures

Availability & Installation

It can be installed with a single click in Neo4j Desktop, is available in all Neo4j Sandboxes and in Neo4j Cloud.

Downloading the appropriate release for your Neo4j into the plugins folder adds APOC to any Neo4j installation.

You can learn more in the APOC Developer Guide.

Relevant Links

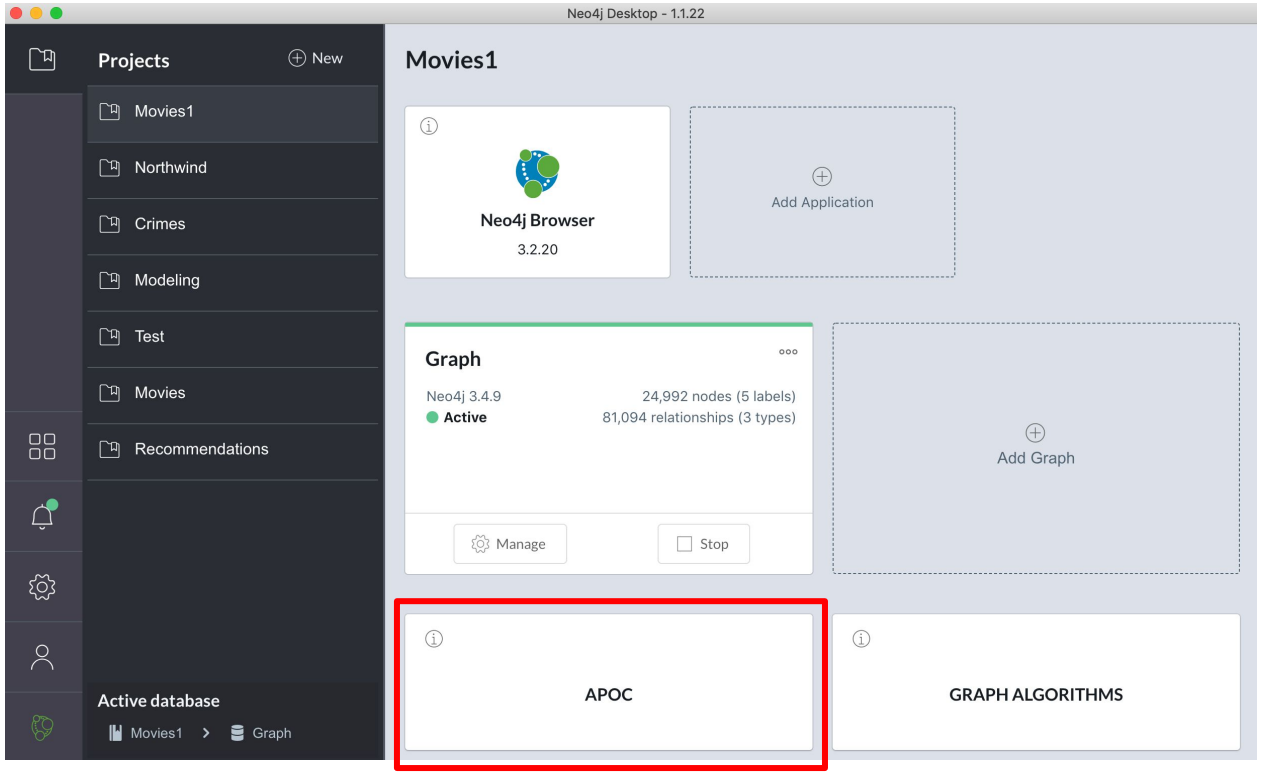
Authors	Michael Hunger, lots of internal and external contributors, especially the team from Larus BA, Italy lead by Andrea Santurbano
Releases	https://github.com/neo4j-contrib/neo4j-apoc-procedures/releases
Source	https://github.com/neo4j-contrib/neo4j-apoc-procedures
Developer Guide	https://neo4j.com/developer/neo4j-apoc/
Docs	https://neo4j-contrib.github.io/neo4j-apoc-procedures/3.5/
Example	A `:play apoc` browser guide shows some of the functionality.

Documentation

- Data Import and Export
- Integration with other Databases
- visual, tabular, structured Schema Information

Reference: https://neo4j.com/labs/apoc/

APOC installed in Neo4j Desktop project



APOC help

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

Table | Text | Code

type	name	text	signature
"procedure"	"apoc.load.csv"	"apoc.load.csv('url',{config}) YIELD lineNo, list, map - load CSV fom URL as stream of values, config contains any of: {skip:1,limit:5,header:false,sep:'TAB',ignore:['tmp'],nullValues:['na'],arraySep:';',mapping:{years:{type:'int',arraySep:'-',array:false,name:'age',ignore:false}}"	"apoc.load.csv(url :: STRING?, config = {} :: MAP?) :: (lineNo :: INTEGER?, list :: LIST? OF ANY?, strings :: LIST? OF STRING?, map :: MAP?, stringMap :: MAP?)"
"procedure"	"apoc.load.driver"	"apoc.load.driver('org.apache.derby.jdbc.EmbeddedDriver') register JDBC driver of source database"	"apoc.load.driver(driverClass :: STRING?) :: VOID"
"procedure"	"apoc.load.html"	"apoc.load.html('url',{name: jquery, name2: jquery}, config) YIELD value - Load Html page and return the result as a Map"	"apoc.load.html(url :: STRING?, query = {} :: MAP?, config = {} :: MAP?) :: (value :: MAP?)"
"procedure"	"apoc.load.jdbc"	"apoc.load.jdbc('key or url','table or statement', params, config) YIELD row - load from relational database, from a full table or a sql statement"	"apoc.load.jdbc(jdbc :: STRING?, tableOrSql :: STRING?, params = [] :: LIST? OF ANY?, config = {} :: MAP?) :: (row :: MAP?)"

APOC Path expanders

Customized path expansion from start node(s)

- More flexible than Cypher's pattern matching, fine-grained control possible
- Min/max traversals
- Limit number of results
- Optional (no rows removed if no results, if set to true)
- Choice of BFS/DFS expansion
- Custom uniqueness (restrictions on visitations of nodes/rels)
- Relationship and label filtering
  - Supports repeating sequences
- No property filtering/evaluation

Note: Only use when you really need to, not as a general solution.

Expander procedures: apoc.path.*

expand(*startNode(s), relationshipFilter, labelFilter, minLevel, maxLevel*) YIELD path

- The original, when you don't need much customization

expandConfig(*startNode(s), configMap*) YIELD path

- Most flexible, rich configuration map

subgraphNodes(*startNode(s), configMap*) YIELD node

- Only distinct nodes, don't care about paths

spanningTree(*startNode(s), configMap*) YIELD path

- Only one distinct path to each node

subgraphAll(*startNode(s), configMap*) YIELD nodes, relationships

- Only distinct nodes and all rels between them

Getting subgraph nodes from a starting node

Here is an APOC procedure that can help you to get a subgraph that contains unique nodes:

```
CALL apoc.path.subgraphNodes(<starting-node>,
                                {<option-list-for-traversal}
                             ) YIELD node
```

Some of the options for this procedure:

- labelFilter
- relationshipFilter
- maxLevel

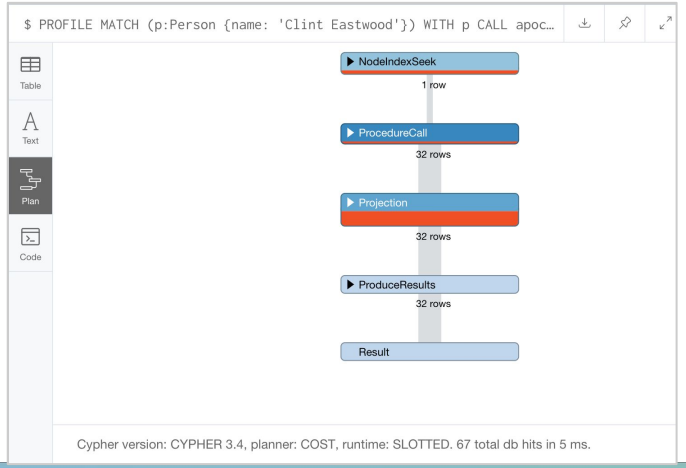
Example: Returning distinct nodes with APOC

Here again is the query we ran previously to return all movie titles that are at most 5 hops away from Clint Eastwood:

```
MATCH (p:Person {name: 'Clint Eastwood'})-[*5]-(m:Movie)
WITH  DISTINCT m
RETURN m.title
```

And here is the code:

```
MATCH (p:Person {name: 'Clint Eastwood'})
WITH p CALL apoc.path.subgraphNodes(p, {labelFilter:'>Movie',maxLevel:5}) YIELD node AS movie
RETURN  movie.title
```



neo4j

Getting a custom subgraph with APOC

With **apoc.path.expandConfig()** you can traverse variable length paths and provide custom filters. This procedure returns the paths, rather than the nodes:

```
CALL apoc.path.expandConfig(<starting-node>,
                                                {<option-list-for-traversal}
                                               ) YIELD path
```

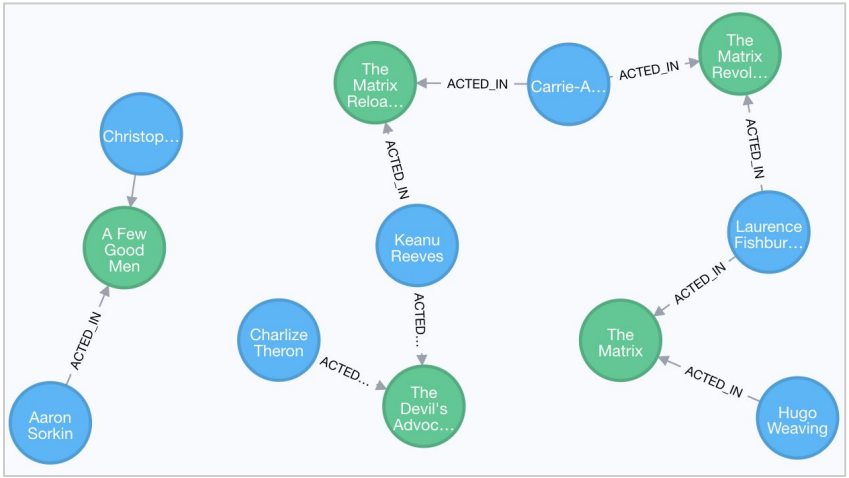
Some of the options for this procedure:

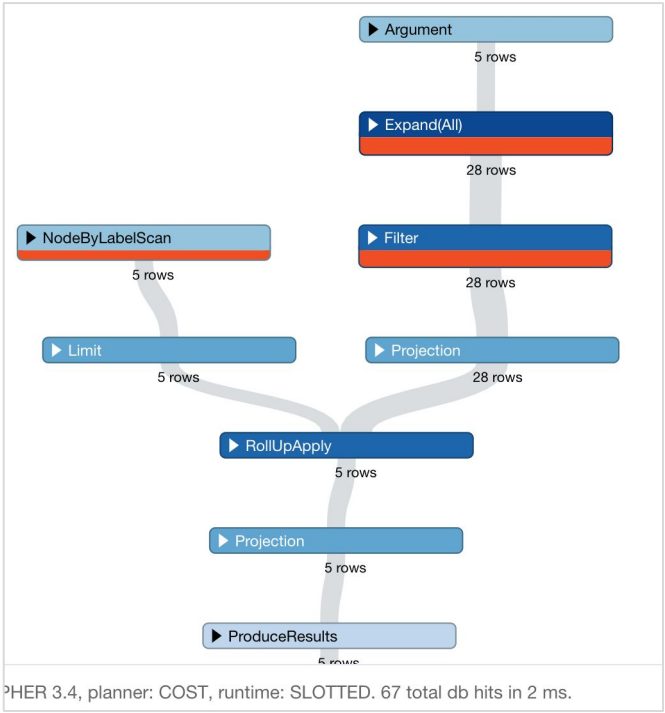
- labelFilter
- relationshipFilter
- minLevel
- maxLevel
- limit

Example:
Returning a custom subgraph with Cypher

Here is the Cypher query for:
Retrieve five movies and for each movie, return the :ACTED_IN path to at most two actors.

```
MATCH (m:Movie) WITH m LIMIT 5
RETURN m, [path = (m)<-[:ACTED_IN]-(:Person)
| path][0..2] as paths
```

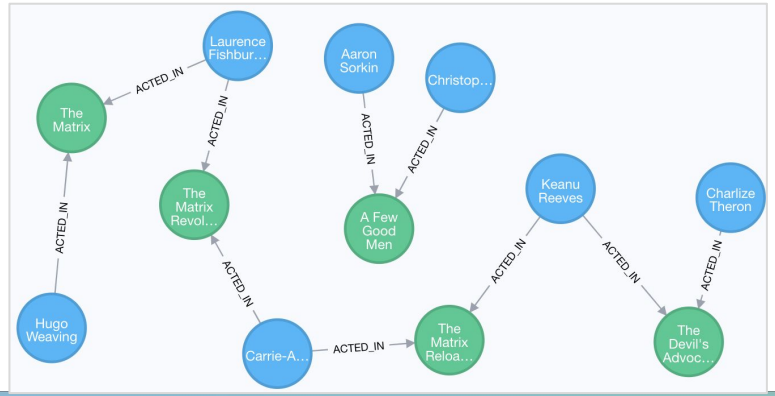


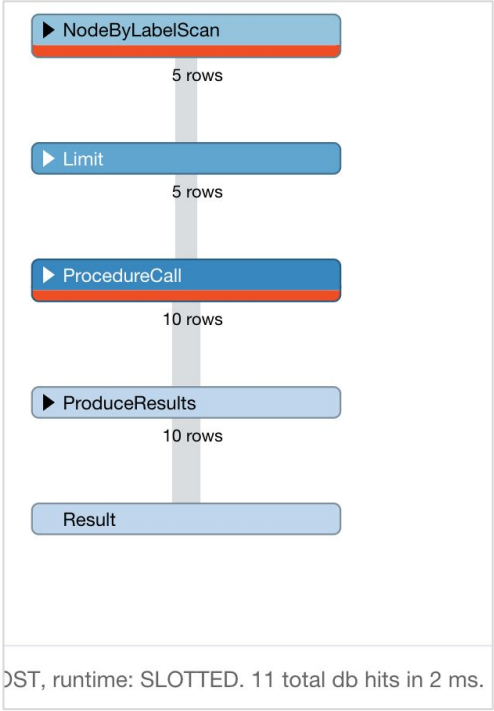


Example:
Returning a custom subgraph with APOC

Here is the Cypher/APOC query for:
Retrieve five movies and for each movie, return the :ACTED_IN path to at most two actors.

```
MATCH (m:Movie) WITH m LIMIT 5
CALL apoc.path.expandConfig(m,
{relationshipFilter:'<ACTED_IN',
labelFilter:'/Person', limit:2, maxLevel: 1}) YIELD path
RETURN path
```



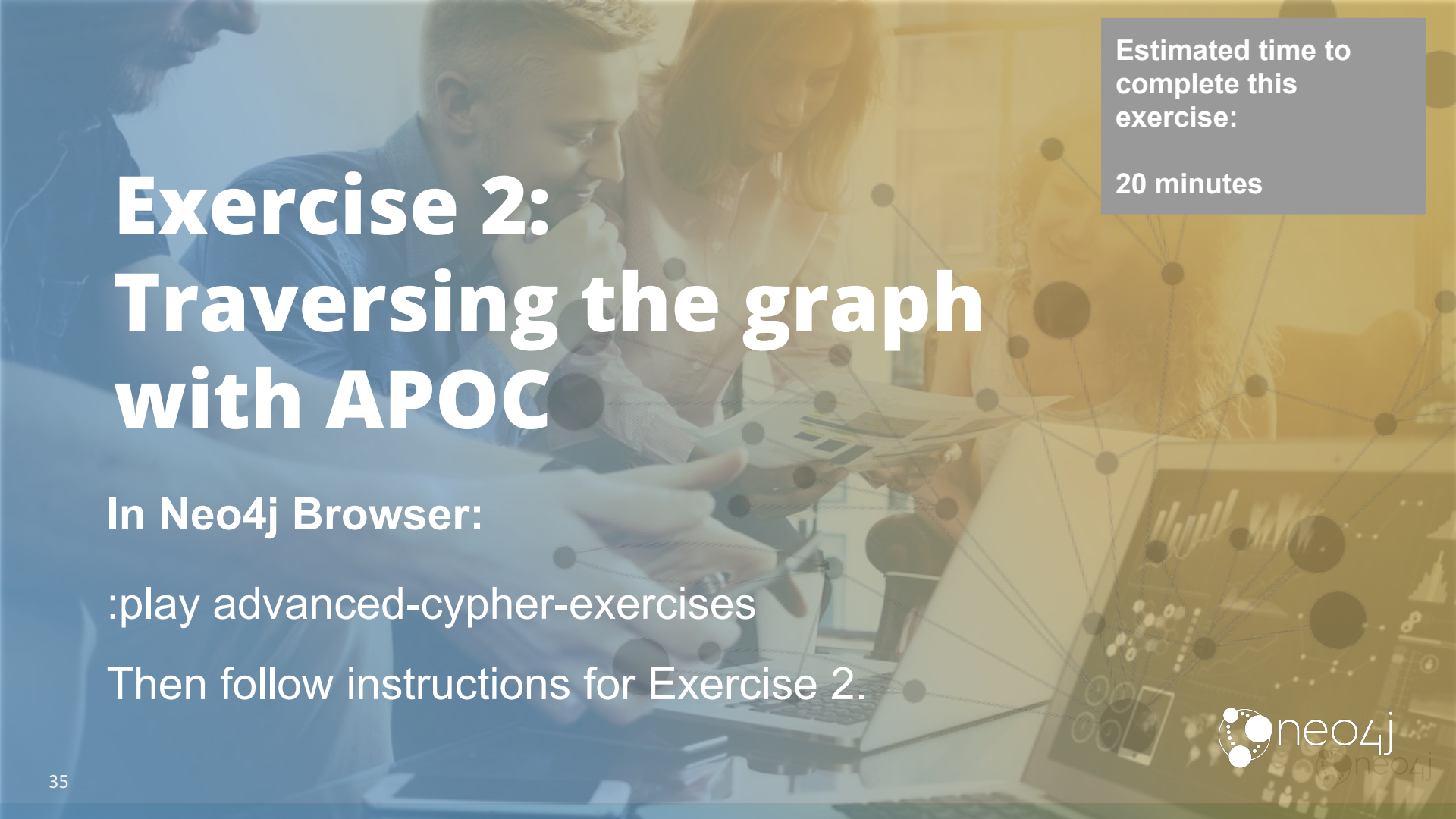


When to use APOC path expander procedures

- Cypher taking too long / hanging
- Need only reachable nodes, not all paths
  - Shortest path or reachability check
  - Different types of expansion (Depth-first, Breadth-first)
  - Different uniqueness constraints for expansion
- Traversal limited to only certain relationships
  - Especially if directions can be different per relationship type
- Traversal limited to only certain labels
  - Need to define sequences of labels and / or relationships
- Need to stop traversals when reaching certain labels or nodes

When not to use APOC path expander procedures

- Some types of expansion are not possible:
  - Path expanders are optimized for filtering on labels during expansion in ways that Cypher can't. With APOC you can define the relationships (types and directions) to follow, and you can define sequences of relationships and/or nodes to follow. But APOC can't currently do additional filtering during expansion, such as anything based on node or relationship properties.
- Expansion requires mix of specific relationships and variable length relationships where ordering matters, for example:
  - MATCH (a)<-[:OWNS]-(:Person)-[:HAS_SHARES*]-(b:Business)



Estimated time to complete this exercise:

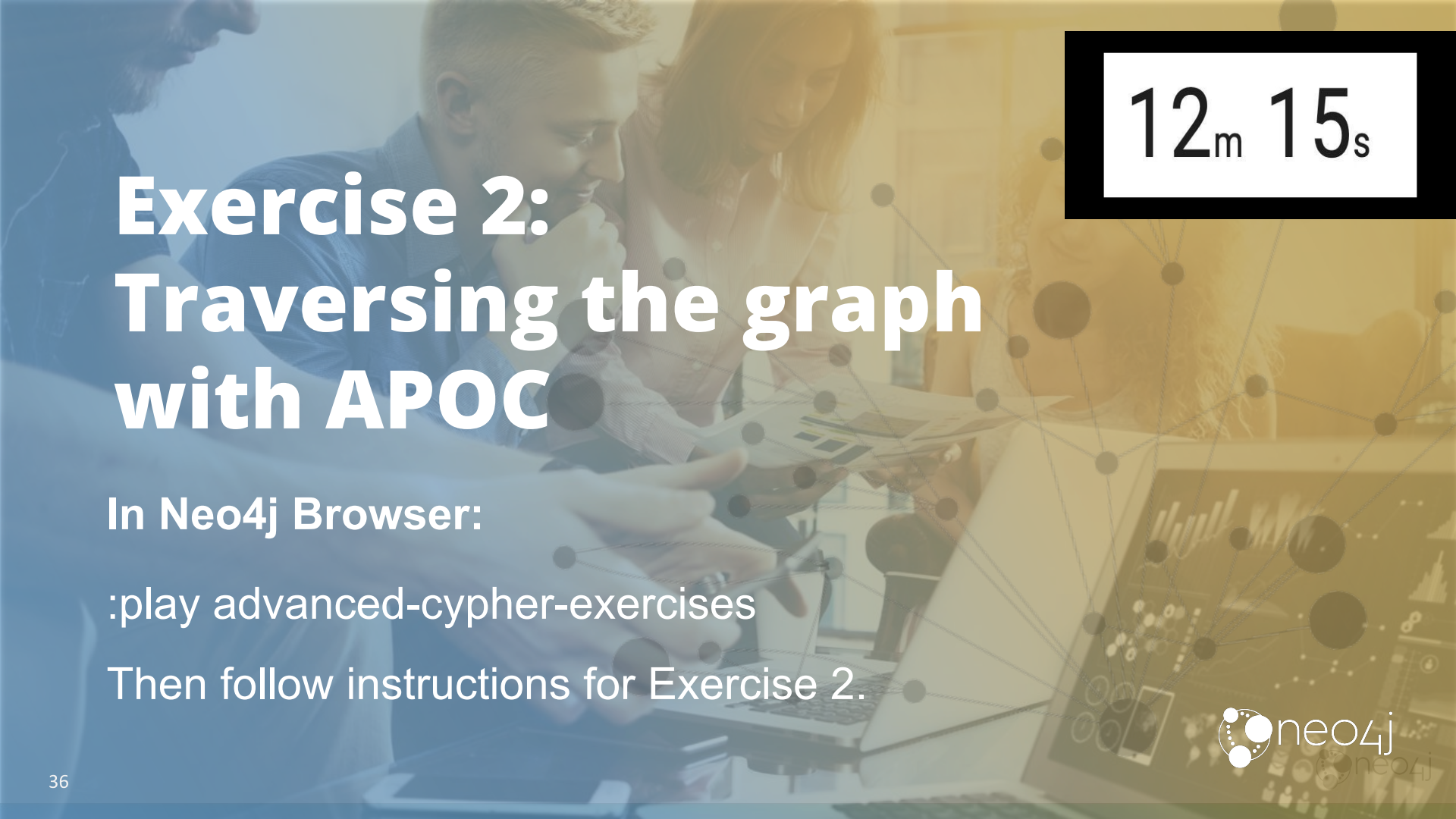
20 minutes

Exercise 2: Traversing the graph with APOC

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 2.

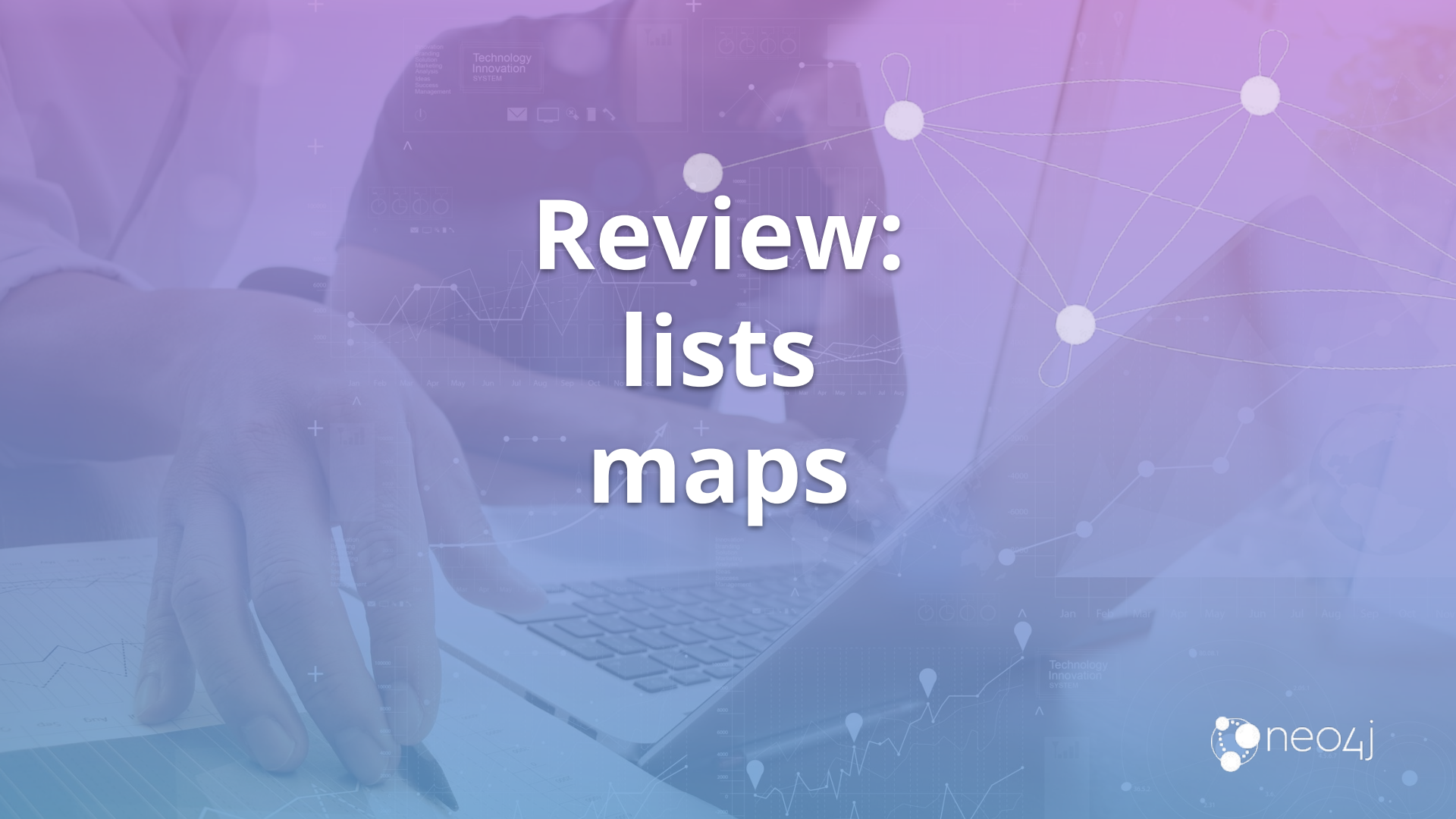


Exercise 2: Traversing the graph with APOC

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 2.


Review:
lists
maps
neo4j

Review: List predicates

List Predicates
`all(x IN coll WHERE exists(x.property))` Returns `true` if the predicate is `true` for all elements in the list.
`any(x IN coll WHERE exists(x.property))` Returns `true` if the predicate is `true` for at least one element in the list.
`none(x IN coll WHERE exists(x.property))` Returns `true` if the predicate is `false` for all elements in the list.
`single(x IN coll WHERE exists(x.property))` Returns `true` if the predicate is `true` for exactly one element in the list.

Review: List expressions

List Expressions
`size($list)` Number of elements in the list.
`reverse($list)` Reverse the order of the elements in the list.
`head($list)`, `last($list)`, `tail($list)` `head()` returns the first, `last()` the last element of the list. `tail()` returns all but the first element. All return `null` for an empty list.
`[x IN list WHERE x.prop <> $value | x.prop]` Combination of filter and extract in a concise notation.
`extract(x IN list | x.prop)` A list of the value of the expression for each element in the original list.
`filter(x IN list WHERE x.prop <> $value)` A filtered list of the elements where the predicate is `true`.
`reduce(s = "", x IN list | s + x.prop)` Evaluate expression for each element in the list, accumulate the results.

Review: Maps

Maps

```
{name: 'Alice', age: 38,
 address: {city: 'London', residential: true}}
```

Literal maps are declared in curly braces much like property maps. Lists are supported.

```
WITH {person: {name: 'Anne', age: 25}} AS p
RETURN p.person.name
```

Access the property of a nested map.

```
MERGE (p:Person {name: $map.name})
  ON CREATE SET p = $map
```

Maps can be passed in as parameters and used either as a map or by accessing keys.

```
MATCH (matchedNode:Person)
RETURN matchedNode
```

Nodes and relationships are returned as maps of their data.

```
map.name, map.age, map.children[0]
```

Map entries can be accessed by their keys. Invalid keys result in an error.

Using UNWIND, pattern and list comprehension

neo4j

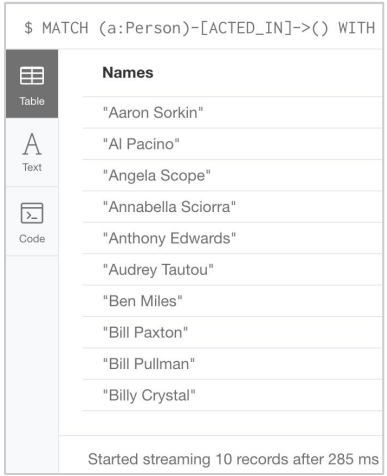
UNWIND

- Transform a collection into **rows**.
- Very useful for working with collections of properties, nodes, paths and sorting, etc.
- Allows collecting a set of nodes to avoid requerying.
- Especially useful after aggregation where you want to partition the data and perform further processing of the selected aggregated data.


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

Another UNWIND example

```
MATCH (a:Person)-[ACTED_IN]->()
WITH collect(a) AS actors
MATCH (d:Person)-[DIRECTED]->()
WITH actors, collect(d) AS directors
UNWIND (actors + directors) AS people
WITH DISTINCT(people)
RETURN  people.name as Names ORDER BY Names LIMIT 10
```



Example: Using UNWIND to create nodes - 1

```
$ :param userData => [ {id:"alice@example.com",properties:{name:"Alice",age:32}},{id:"bob@example.com",properties:{name:"Bob",age:42}}]
```


```
{
  "userData": [
    {
      "properties": {
        "name": "Alice",
        "age": 32
      },
      "id": "alice@example.com"
    },
    {
      "properties": {
        "name": "Bob",
        "age": 42
      },
      "id": "bob@example.com"
    }
  ]
}
```

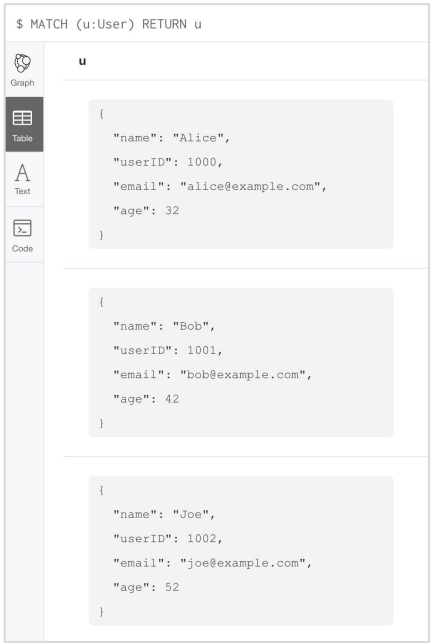

```
UNWIND $userData as row
MERGE (n:User {id: row.id})
ON CREATE SET n += row.properties
```

Example: Using UNWIND to create nodes - 2

```
$ :params
```

```
{
  "seedStart": 1000,
  "userData": [
    {
      "name": "Alice",
      "email": "alice@example.com",
      "age": 32
    },
    {
      "name": "Bob",
      "email": "bob@example.com",
      "age": 42
    },
    {
      "name": "Joe",
      "email": "joe@example.com",
      "age": 52
    }
  ],
  "numUsers": 3
}
```

```
UNWIND range(0, $numUsers - 1) AS index
WITH $userData[index] AS row, index
MERGE (n:User {userID: index + $seedStart})
ON CREATE SET n += row
```

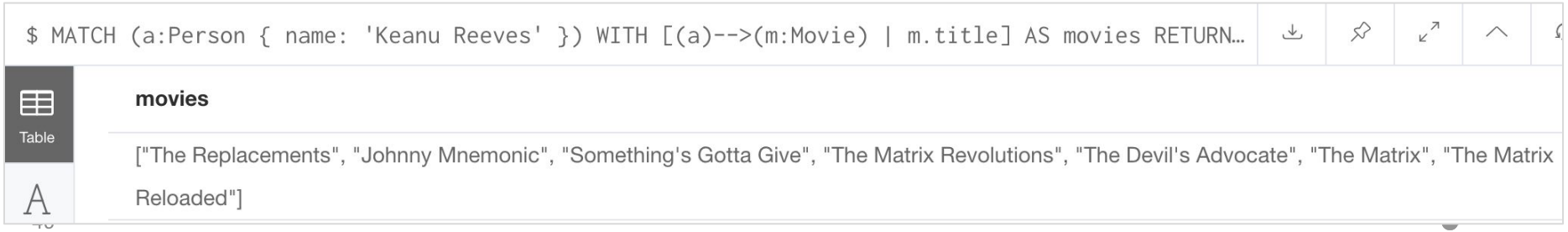


Using pattern comprehension for extraction

- Used to create a list entry (extraction) from a pattern in the graph.
- Eliminates the need for an optional MATCH and an aggregation.

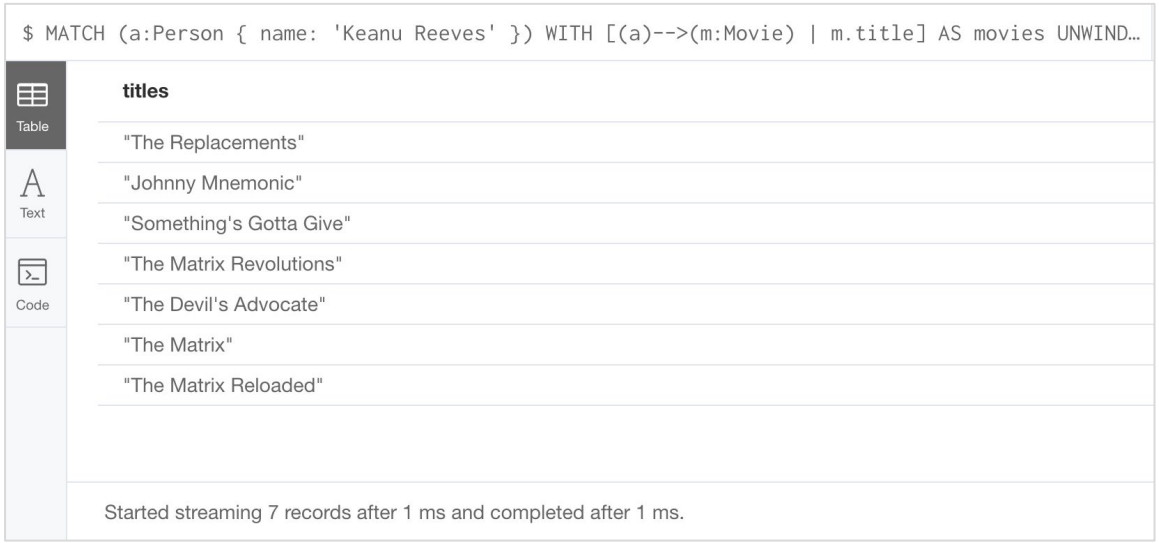
<clause> [<pattern>| <extraction>]

```
MATCH (a:Person { name: 'Keanu Reeves' })
WITH [(a)-->(m:Movie) | m.title] AS movies
RETURN movies
```



Using pattern comprehension and UNWIND

```
MATCH (a:Person { name: 'Keanu Reeves' })
WITH [(a)-->(m:Movie) | m.title] AS movies
UNWIND movies AS titles
RETURN titles
```



Using range() and slices of a list

Expression	Value
range(0,10)	[0,1,2,3,4,5,6,7,8,9,10]
range(1,10,2)	[1,3,5,7,9]
range(0,10) [3]	[3]
range(0,10) [0..3]	[0,1,2,3]
range(0,10) [-4]	[7]
range(0,10) [..4]	[0,1,2,3]
range(0,10) [..-5]	[6,7,8,9,10]

See the documentation for all of the list functions:

keys(), labels(), nodes(), reduce(), relationships(), reverse(), tail()

Using list comprehension for extraction

Used to create a list entry (extraction) from predicate.

<clause> **[**<expression-to-evaluate>**|** <extraction>**]**

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

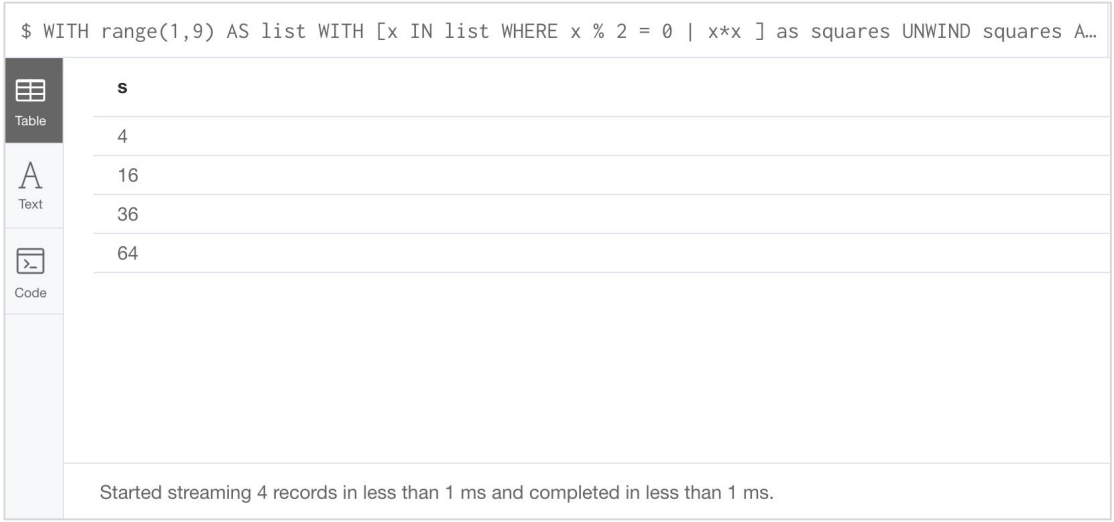
```
$ RETURN [x IN range(0,10) WHERE x % 2 = 0 | x^2] AS result
```

Table

result
[0.0, 4.0, 16.0, 36.0, 64.0, 100.0]

Using list comprehension and UNWIND

```
WITH range(1,9) AS list
WITH [x IN list WHERE x % 2 = 0 | x*x ] as squares
UNWIND squares AS s
RETURN s
```



Exercise 3: Working with UNWIND, pattern and list comprehension

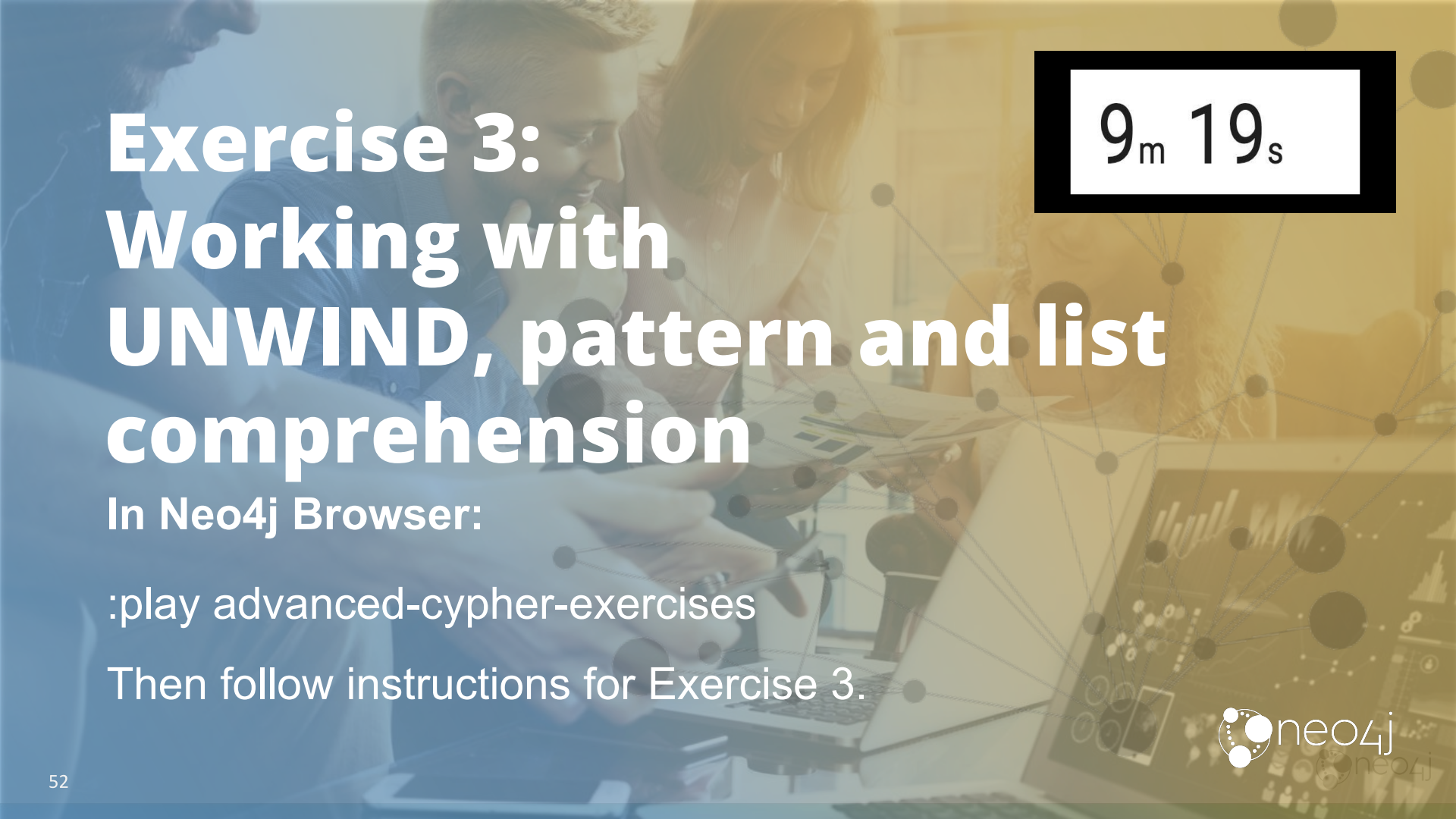
Estimated time to complete this exercise:

10 minutes

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 3.



Exercise 3: Working with UNWIND, pattern and list comprehension

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 3.

Combining query results

neo4j

UNION

- Combines the results of two or more queries.
- Column labels in RETURN clauses must match.
- By default, eliminates duplicate nodes and relationships (implicit DISTINCT).
- Specify UNION ALL to not eliminate duplicates.
- Use when MATCH paths are disjoint and more performant than multiple MATCH clauses.

```
MATCH <match-clause>
RETURN <var1> AS <aliasName>, <var2>
UNION
MATCH <match-clause>
RETURN <var3> AS <aliasName>, <var2>
```

Example: Using UNION

```
MATCH (a:Person)-[:ACTED_IN]->()
WHERE a.born  < 1940
WITH DISTINCT a
RETURN a.name AS name, a.born AS `Year Born`
UNION
MATCH (d:Person)-[:DIRECTED]->()
WHERE d.born < 1940
WITH DISTINCT d
RETURN d.name AS name, d.born AS `Year Born`
```

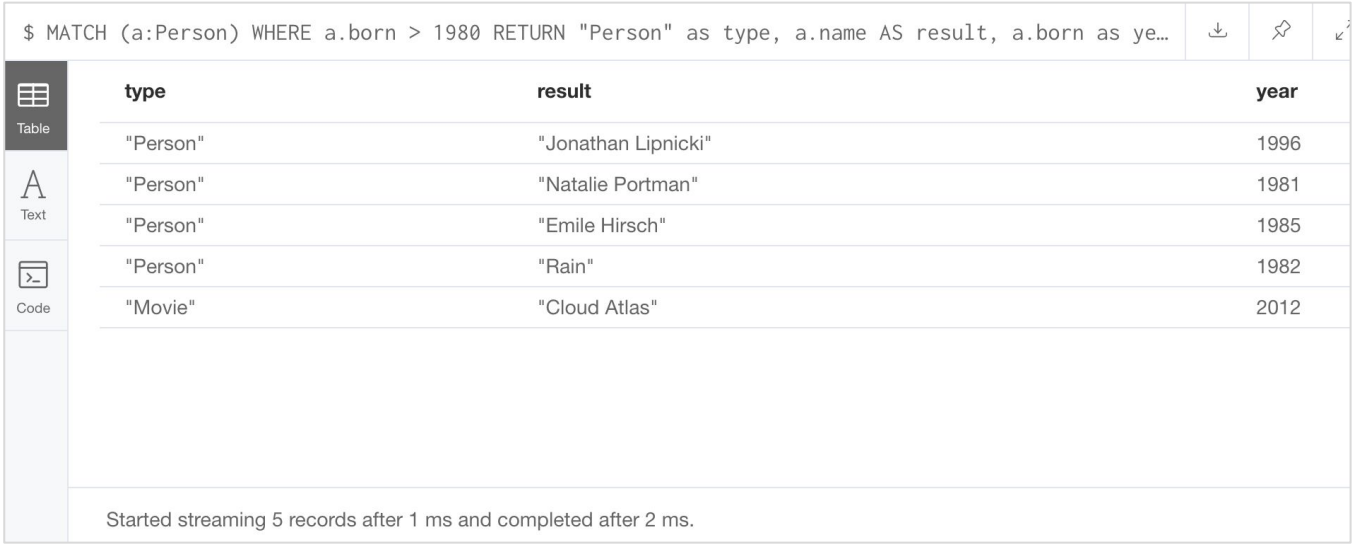
Returns 10 records because UNION automatically eliminates duplicates

```
MATCH (a:Person)-[:ACTED_IN]->()
WHERE a.born  < 1940
WITH DISTINCT a
RETURN a.name AS name, a.born AS `Year Born`
UNION ALL
MATCH (d:Person)-[:DIRECTED]->()
WHERE d.born < 1940
WITH DISTINCT d
RETURN d.name AS name, d.born AS `Year Born`
```

Returns 11 records because one actor is also a director and duplicates are returned

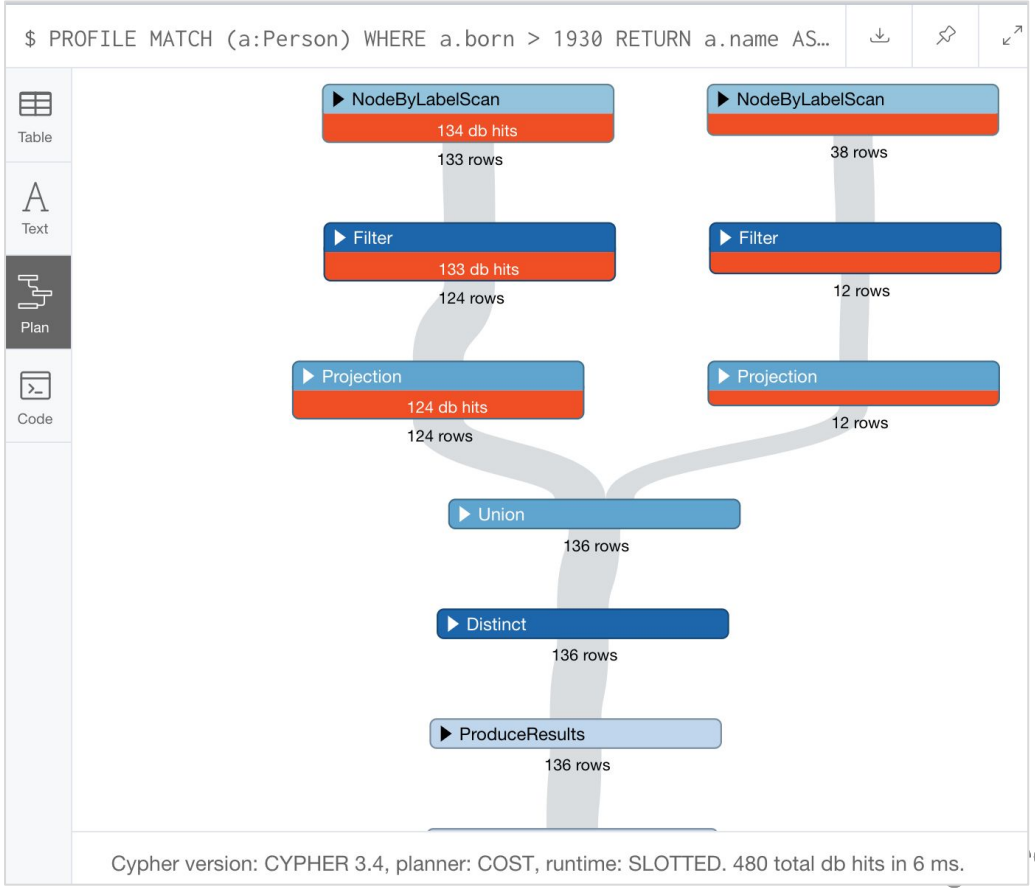
Example: Using UNION for disjoint queries

```
MATCH (a:Person)
WHERE a.born  > 1980
RETURN "Person" AS type, a.name AS result, a.born AS year
UNION
MATCH (m:Movie)
WHERE m.released > 2010
RETURN "Movie" AS type, m.title AS result, m.released AS year
```



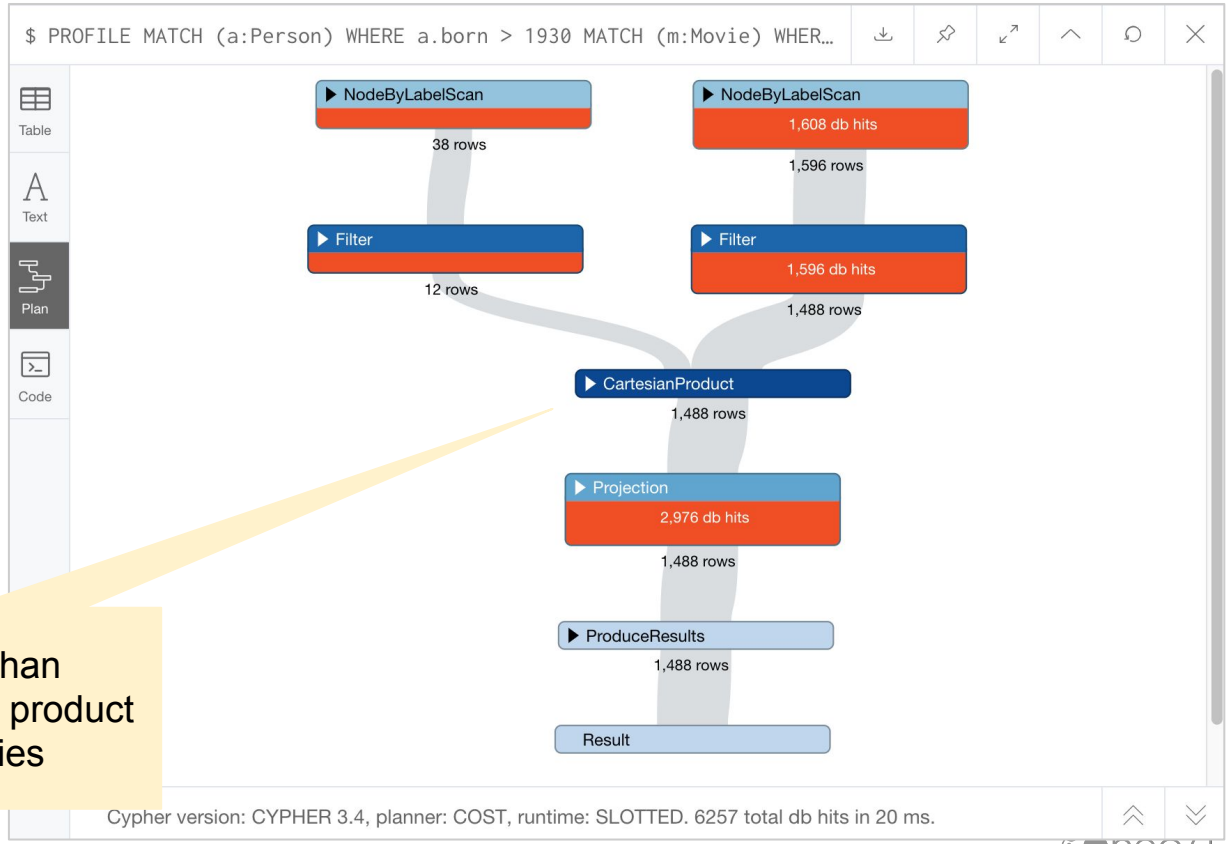
When UNION is better - 1

```
PROFILE MATCH (a:Person)
WHERE a.born  > 1930
RETURN a.name AS result
UNION
MATCH (m:Movie)
WHERE m.released > 2000
RETURN m.title AS result
```



When UNION is better - 2

```
PROFILE MATCH (a:Person)
WHERE a.born  > 1930
MATCH (m:Movie)
WHERE m.released > 2000
RETURN a.name, m.title
```



UNION is better than doing a cartesian product of unrelated queries

Optional connections instead of UNION

Suppose we have this schema:

```
(:Person)-[:FRIENDS_WITH]->(:Person)
(:Person)-[:RECOMMENDS]->(:Movie)
(:Person)-[:RECOMMENDS]->(:Book)
(:Person)-[:RECOMMENDS]->(:Game)
(:Movie)-[:BASED_ON]->(:Book)
(:Movie)-[:BASED_ON]->(:Game)
(:Game)-[:BASED_ON]->(:Movie)
(:Game)-[:BASED_ON]->(:Book)
(:Book)-[:BASED_ON]->(:Game)
(:Book)-[:BASED_ON]->(:Movie)
```

We want to return movies associated with non-movie recommendations (such as movies based on recommended books or other media). This could be done with UNION:

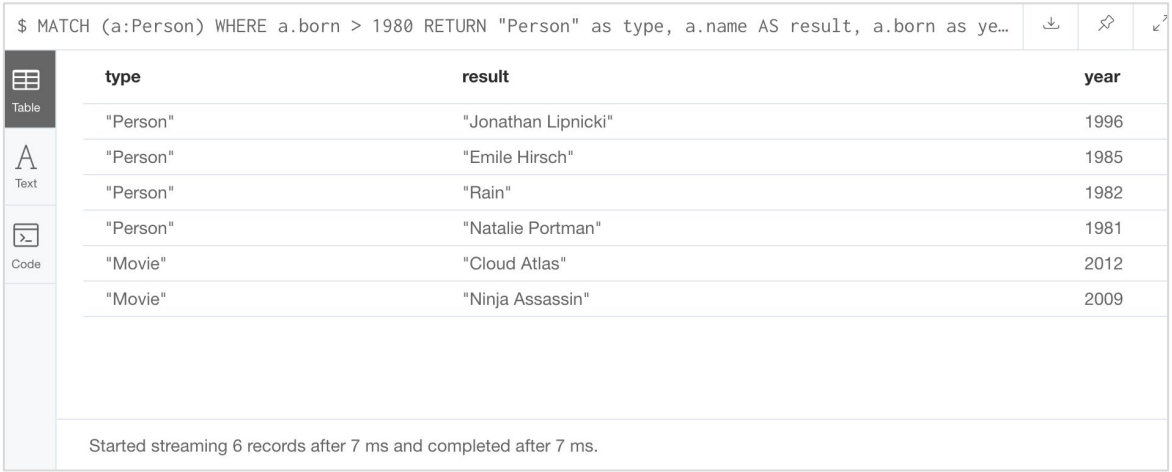
```
MATCH (me:Person
{name:$actorData)-[:FRIENDS_WITH]-(friend)-[:RECOMMENDS]->(movie:
Movie)
RETURN friend, movie
UNION
MATCH (me:Person {name:$actorData})-[:FRIENDS_WITH]-(friend)-
[:RECOMMENDS]->()-[:BASED_ON]->(movie:Movie)
RETURN friend, movie
```

A better approach is to forgo the UNION and use an optional connection. If the recommended item is a movie, it will be included, and if it is something that a movie was based on, that movie will also be included.

```
MATCH (me:Person {name:$actorData)-[:FRIENDS_WITH]-
(friend)-[:RECOMMENDS]->()-[:BASED_ON*0..1]->(movie:Movie)
RETURN friend, movie
```

With UNION you cannot post-process the final results (sort all results)

```
MATCH (a:Person)
WHERE a.born  > 1980
RETURN "Person" as type, a.name AS result, a.born AS year ORDER BY a.born DESC
UNION
MATCH (m:Movie)
WHERE m.released > 2008
RETURN "Movie" as type, m.title AS result, m.released AS year ORDER BY m.released DESC
```

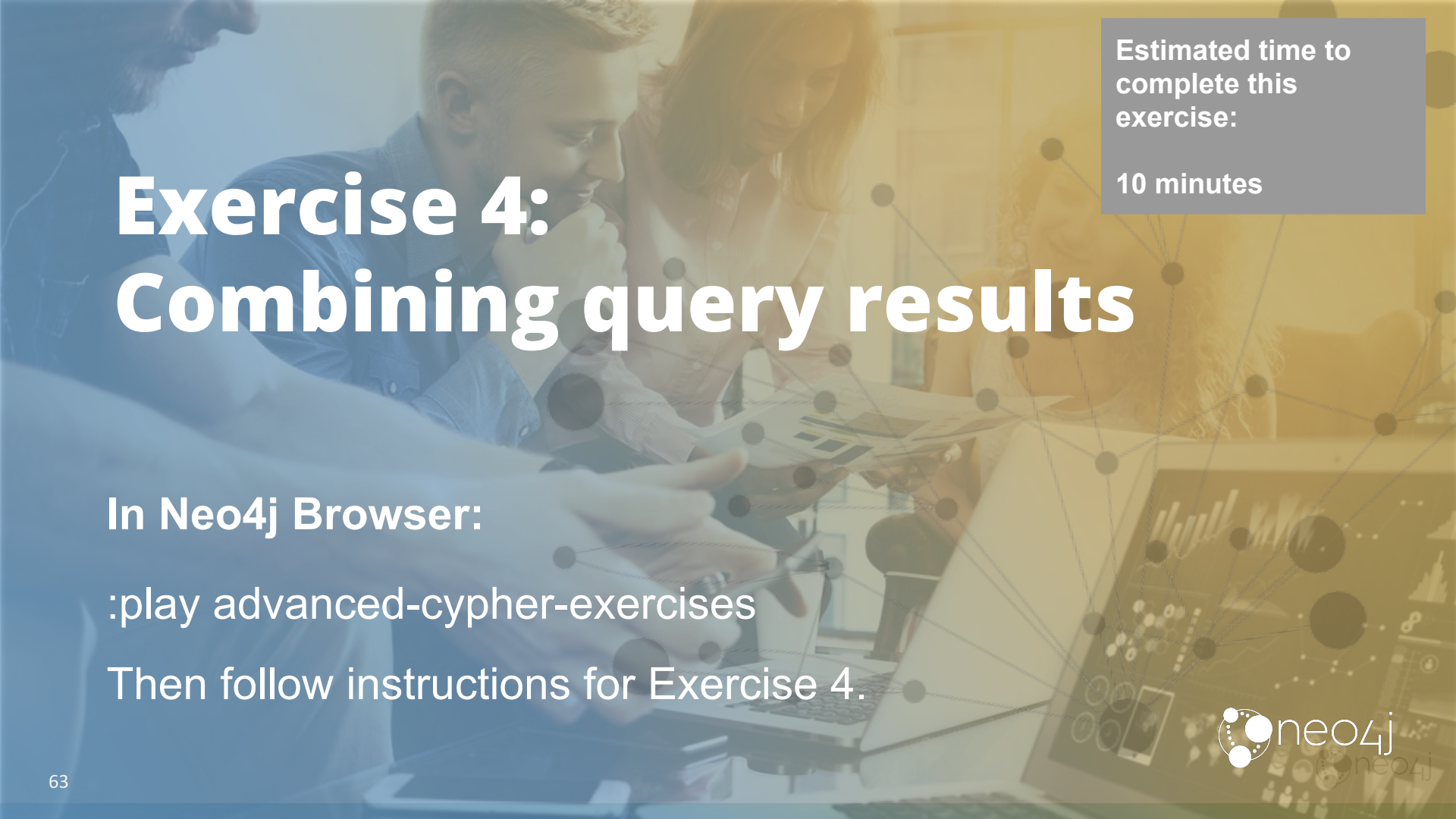


Using APOC to allow UNION post-processing

```
CALL apoc.cypher.run('MATCH (a:Person)
WHERE a.born  > 1980
RETURN "Person" AS type, a.name AS result, a.born AS year
UNION
MATCH (m:Movie)
WHERE m.released > 2008
RETURN "Movie" as type, m.title AS result, m.released AS year
',{}) YIELD value
WITH value
RETURN value.type AS type, value.result AS result, value.year AS year ORDER BY year DESC
```

Alternative to UNION for sorting all results

```
MATCH (a:Person)
WHERE a.born  > 1980
WITH collect({type: "Person", result: a.name, year: a.born}) AS result1
MATCH (m:Movie)
WHERE m.released > 2008
WITH result1 + collect({type: "Movie", result:m.title, year: m.released}) AS allResults
UNWIND allResults AS finalResult
WITH finalResult.type AS type, finalResult.result AS result, finalResult.year AS year
return type, result, year ORDER BY year DESC
```



Estimated time to complete this exercise:

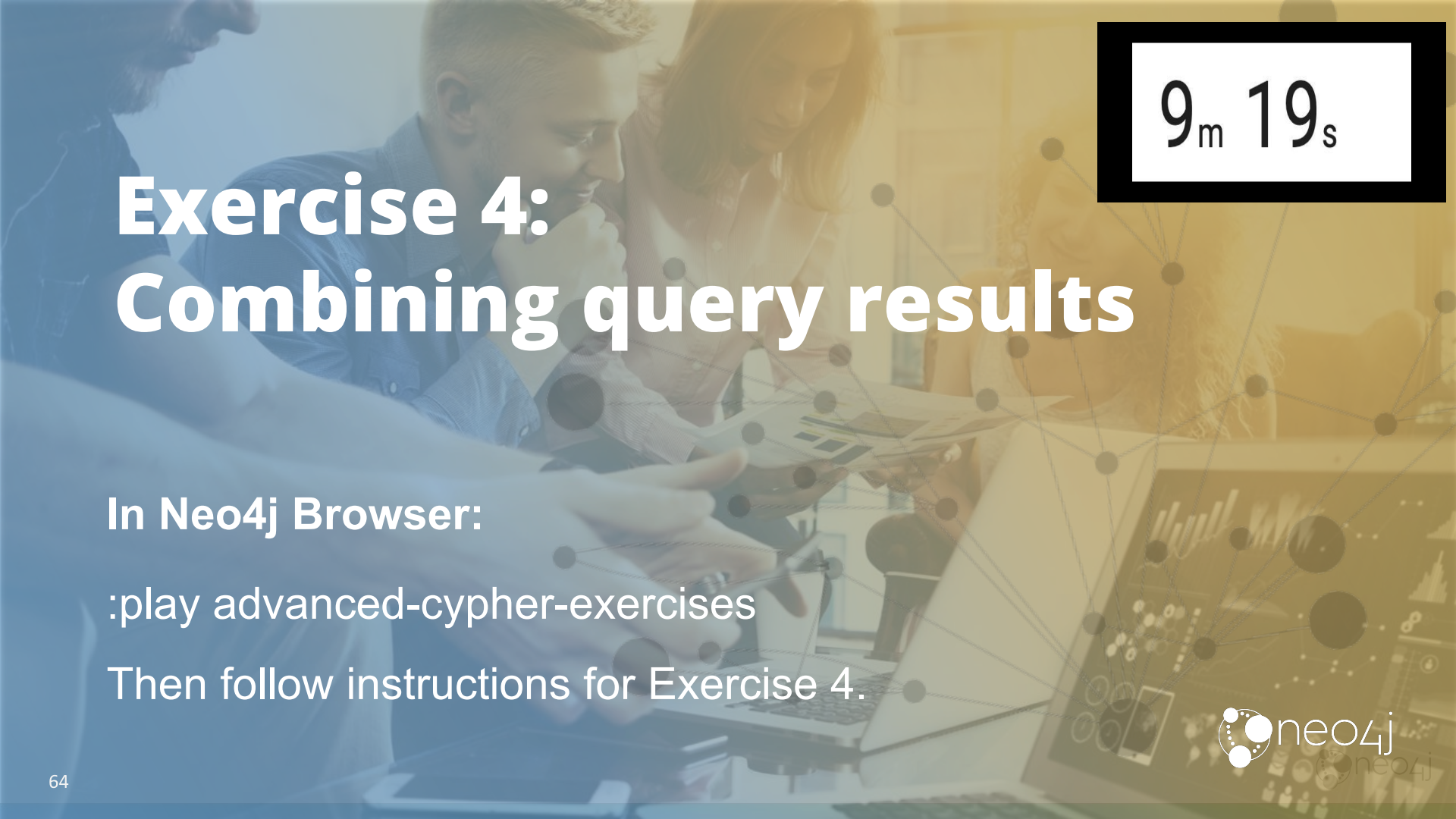
10 minutes

Exercise 4: Combining query results

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 4.



Exercise 4: Combining query results

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 4.

Importing data

neo4j

Importing data into the graph

Import is used by 90% of Neo4j developers to load data into their graphs. It is important that you chose the best way to load data, especially when you have large datasets.

Some things we will cover:

- Review of loading normalized data (multiple CSV files)
- Loading denormalized data (single CSV file)
- Using PERIODIC COMMIT
- Using APOC to help you load data

Steps for importing CSV data into the graph

1. Determine whether headers will be included in the CSV file.
2. Determine whether you will use normalized or denormalized data.
3. Ensure IDs to be used in the data are unique.
4. Ensure data in CSV files is properly formatted and "clean".
5. Execute Cypher code to inspect the data.
6. Ensure constraints are created in the graph.
7. Execute Cypher code to load the data.
8. Add indexes to the graph.

Using headers for CSV import

With headers in the data: **people.csv**:

```
personId,name,birthYear,deathYear
23945,Gérard Pirès,1942,
553509,Helen Reddy,1941,
113934,Susan Flannery,1939,
```

Without headers in the data:
people2_headers.csv:

```
personId,name,birthYear,deathYear
```

people2.csv:

```
23945,Gérard Pirès,1942,
553509,Helen Reddy,1941,
113934,Susan Flannery,1939,
```

Normalized vs. denormalized data

Normalized data:

people.csv:

```
personId,name,birthYear,deathYear
23945,Gérard Pirès,1942,
553509,Helen Reddy,1941,
113934,Susan Flannery,1939,
```

roles.csv:

```
personId,movieId,characters
2295,189,Marv
56731,189,Nancy
16851,189,Dwight
```

movies1.csv:

```
movieId,title,avgVote,releaseYear,genres
10586,The Ghost and the Darkness,7.300000,1996,Action:Adventure:Drama:History:Thriller
2300,Space Jam,6.300000,1996,Animation:Comedy:Drama:Fantasy:Family:Sports Film
11969,Tombstone,7.000000,1993,Action:Adventure:Drama:History:Western
```

Denormalized data:

movies-n.csv:

```
movieId,title,avgVote,releaseYear,tagline,genres,personType,personId,name,birthYear,death
Year,characters
241254,The Prince,5.600000,2014,"",Action:Thriller,ACTOR,62,Bruce Willis,1955,0,Omar
241254,The Prince,5.600000,2014,"",Action:Thriller,ACTOR,12261,Jason Patric,1966,0,Paul
241254,The Prince,5.600000,2014,"",Action:Thriller,DIRECTOR,81613,Brian A. Miller,0,0,""
226486,Tammy,5.200000,2014,"",Comedy,ACTOR,55536,Melissa McCarthy,1970,0,Tammy
226486,Tammy,5.200000,2014,"",Comedy,ACTOR,4038,Susan Sarandon,1946,0,Pearl
226486,Tammy,5.200000,2014,"",Comedy,DIRECTOR,170820,Ben Falcone,0,0,""
```

IDs used in CSV must be unique

Normalized data:

people.csv:

```
personId,name,birthYear,deathYear
23945,Gérard Pirès,1942,
553509,Helen Reddy,1941,
113934,Susan Flannery,1939,
```

roles.csv:

```
personId,movieId,characters
2295,189,Marv
56731,189,Nancy
16851,189,Dwight
```

Denormalized data:

movies2.csv:

```
movieId,title,avgVote,releaseYear,tagline,genres,personType,personId,name,birthYear,death
Year,characters
241254,The Prince,5.600000,2014,"",Action:Thriller,ACTOR,62,Bruce Willis,1955,0,Omar
241254,The Prince,5.600000,2014,"",Action:Thriller,ACTOR,12261,Jason Patric,1966,0,Paul
241254,The Prince,5.600000,2014,"",Action:Thriller,DIRECTOR,81613,Brian A. Miller,0,0,""
226486,Tammy,5.200000,2014,"",Comedy,ACTOR,55536,Melissa McCarthy,1970,0,Tammy
226486,Tammy,5.200000,2014,"",Comedy,ACTOR,4038,Susan Sarandon,1946,0,Pearl
226486,Tammy,5.200000,2014,"",Comedy,DIRECTOR,170820,Ben Falcone,0,0,""
```

Make sure data is properly formatted and clean

movies2.csv:

```
movieId,title,avgVote,releaseYear,tagline,genres,personType,personId,name,birthYear,deathYear,characters
241254,The Prince,5.600000,2014,"Who's the best?",Action:Thriller,ACTOR,62,Bruce Willis,1955,0,Omar
241254,The Prince,5.600000,2014,"Who's the best?",Action:Thriller,ACTOR,12261,Jason Patric,1966,0,Paul
241254,The Prince,5.600000,2014,"Who's the best?",Action:Thriller,DIRECTOR,81613,Brian A. Miller,0,0,""
226486,Tammy,5.200000,2014,"",Comedy,ACTOR,55536,Melissa McCarthy,1970,0,Tammy
226486,Tammy,5.200000,2014,"",Comedy,ACTOR,4038,Susan Sarandon,1946,0,Pearl
226486,Tammy,5.200000,2014,"",Comedy,ACTOR,8534,Kathy Bates,1948,0,Lenore:Grandma
226486,Tammy,5.200000,2014,"",Comedy,DIRECTOR,170820,Ben Falcone,0,0,""
```

- When are quotes used?
  - If string contains single quotes, element must be double-quoted.
  - If string contains double quotes, they must be escaped.
  - If an element has no value will an empty string be used or nothing?
- Make sure lists are properly formed (default is to use colon(:) as the separator.
- Will comma(,) be the delimiter?
- Make sure no special characters in the data that may not be readable.

Define constraints before loading data

people.csv:

```
personId,name,birthYear,deathYear
23945,Gérard Pirès,1942,
553509,Helen Reddy,1941,
113934,Susan Flannery,1939,
```

roles.csv:

```
personId,movieId,characters
2295,189,Marv
56731,189,Nancy
16851,189,Dwight
```

movies-n.csv:

```
movieId,title,avgVote,releaseYear,genres
10586,The Ghost and the Darkness,7.300000,1996,Action:Adventure:Drama:History:Thriller
2300,Space Jam,6.300000,1996,Animation:Comedy:Drama:Fantasy:Family:Sports Film
11969,Tombstone,7.000000,1993,Action:Adventure:Drama:History:Western
```

Add uniqueness constraints for IDs that should be unique to ensure "clean data" in the graph:

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

Accessing data stored locally or remotely

CSV file on the Internet:

```
LOAD CSV WITH HEADERS
FROM 'https://data.neo4j.com/advanced-cypher/people.csv'
AS line
RETURN line LIMIT 10
```

CSV file in the **import** folder for the database:

```
LOAD CSV WITH HEADERS
FROM 'file:///people.csv'
AS line
RETURN line LIMIT 10
```

See the documentation for LOAD CSV for additional keywords and clauses you can use.

Inspect the data before loading

- How large is the file?
  - Larger files require special handling.
- Check for possible problems with the data:
  - Headers that do not match the data
  - Quotes used incorrectly
  - UTF-8 prefixes (for example \uc)
  - Trailing spaces
  - Binary zeros
  - Fields that contain quotes (for example 24" monitor)
  - Typos
- Default type for data is a string. Does data need to be converted?
  - toInteger()
  - toFloat()
- Does data need to be interpreted as a list?
  - split()

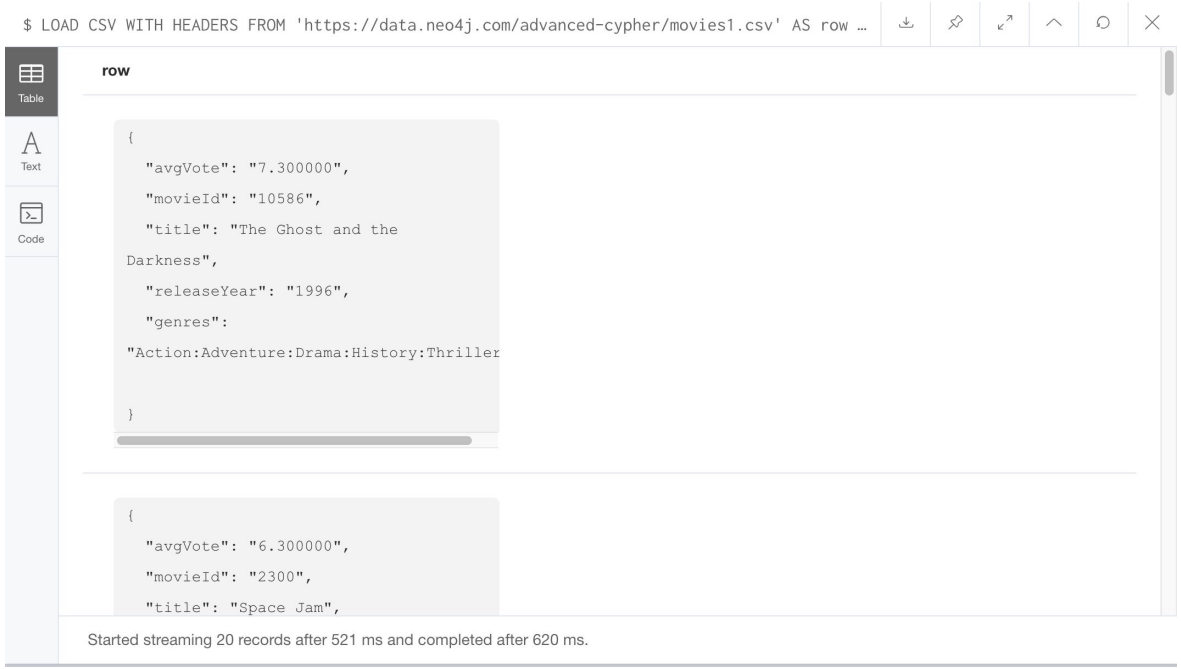
How large is the file?

```
LOAD CSV WITH HEADERS FROM
    'https://data.neo4j.com/advanced-cypher/movies1.csv' AS row
RETURN count(*)
```

Note: If > ~100K lines (with default heap for Neo4j Desktop), need to prefix your LOAD CSV with USING PERIODIC COMMIT to avoid out of memory failure.

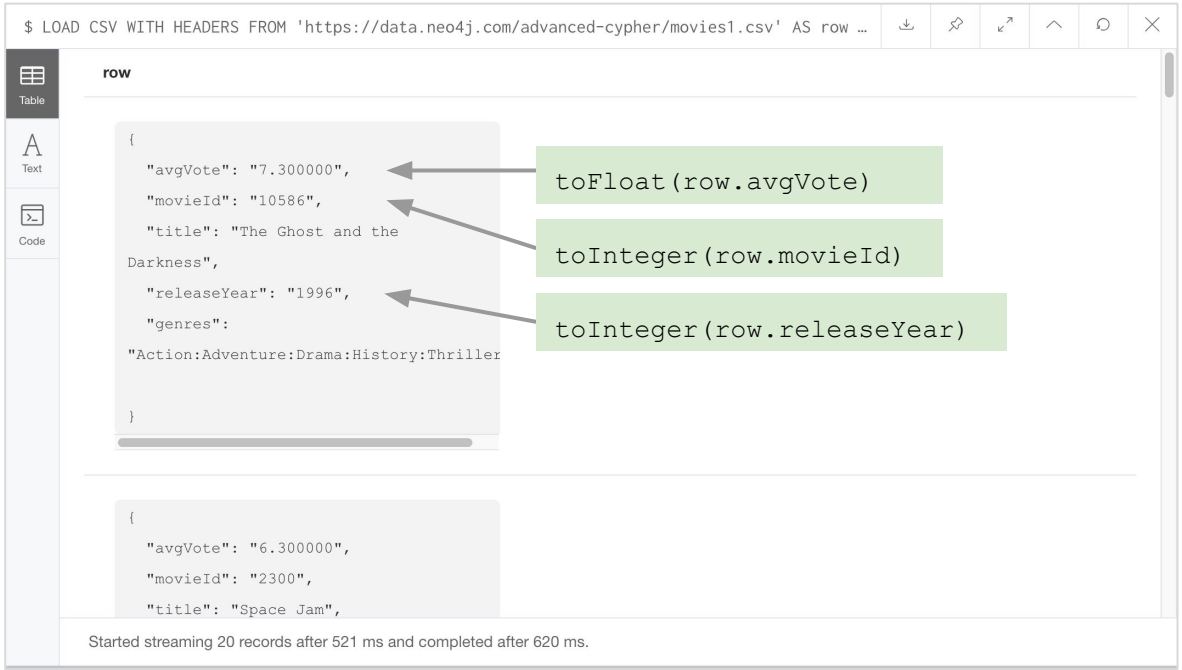
Check for possible problems with the data

```
LOAD CSV WITH HEADERS FROM
    'https://data.neo4j.com/advanced-cypher/movies1.csv' AS row
RETURN row LIMIT 20
```



Does the data need to be converted?

```
LOAD CSV WITH HEADERS FROM
    'https://data.neo4j.com/advanced-cypher/movies1.csv' AS row
RETURN row LIMIT 20
```



Does the data need to be interpreted as a list?

```
LOAD CSV WITH HEADERS FROM
    'https://data.neo4j.com/advanced-cypher/movies1.csv' AS row
RETURN row LIMIT 20
```

What if there is no data for a field?

roles.csv:

```
personId,movieId,characters
381,920,Lizzie (voice)
7907,920,Mack:Hamm Truck:Abominable Snow Plow (voice)
2232,920,Chick Hicks (voice)
6856,209247,Crunch Calhoun
449,209247,Francie
2876,209247,Nicky Calhoun
26723,209247,Lola
28641,209247,Samuel Winter
90498,209247,Ponch
1217648,209247,Guy de Cornet
185805,209247,Agent Bick
230,209247,Dirty Ernie
287,228150,Wardaddy
10959,228150,Boyd 'Bible' Swan
33235,228150,Norman Ellison
19498,228150,Grady Travis
928572,228150,
11355,228150,
109438,228150,Lt. Parker
```

split(row.characters,":")

Creates this list:
["MACK","Hamm Truck","Abominable Snow Plow (voice)"]

split(coalesce(row.characters,""), ":")

Use coalesce() to return null when there is no data for a field. Creates this list:
[""]

Create constraints before load

```
CREATE CONSTRAINT ON (m:Movie) ASSERT m.id IS UNIQUE;

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

// load data

// create indexes
```

MERGE will be very slow if there are no constraints present.

Indexes, however, will slow down the creation of data due to added writes, but are necessary if you want transactionally consistent data and indexes in the database.

If possible, create constraints before loading, but create indexes after loading.

Loading the data to create nodes

```
LOAD CSV WITH HEADERS FROM
    'https://data.neo4j.com/advanced-cypher/movies1.csv' AS row
MERGE (m:Movie {id:toInteger(row.movieId)})
   ON CREATE SET
        m.title = row.title,
        m.avgVote = toFloat(row.avgVote),
        m.releaseYear = toInteger(row.releaseYear),
        m.genres = split(row.genres,":")
```

```
$ LOAD CSV WITH HEADERS FROM 'https://data.neo4j.com/advanced-cypher/movies1.csv' AS
```

Table

Added 6231 labels, created 6231 nodes, set 31155 properties, completed after 619 ms.

Loading the data to create relationships

```
LOAD CSV WITH HEADERS FROM
'https://data.neo4j.com/advanced-cypher/directors.csv' AS row

MATCH (movie:Movie {id:toInteger(row.movieId)})
MATCH (person:Person {id: toInteger(row.personId)})
MERGE (person)-[:DIRECTED]->(movie)
ON CREATE SET person:Director
```

```
$ LOAD CSV WITH HEADERS FROM 'https://data.neo4j.com/advanced-cypher/directors.csv'
```

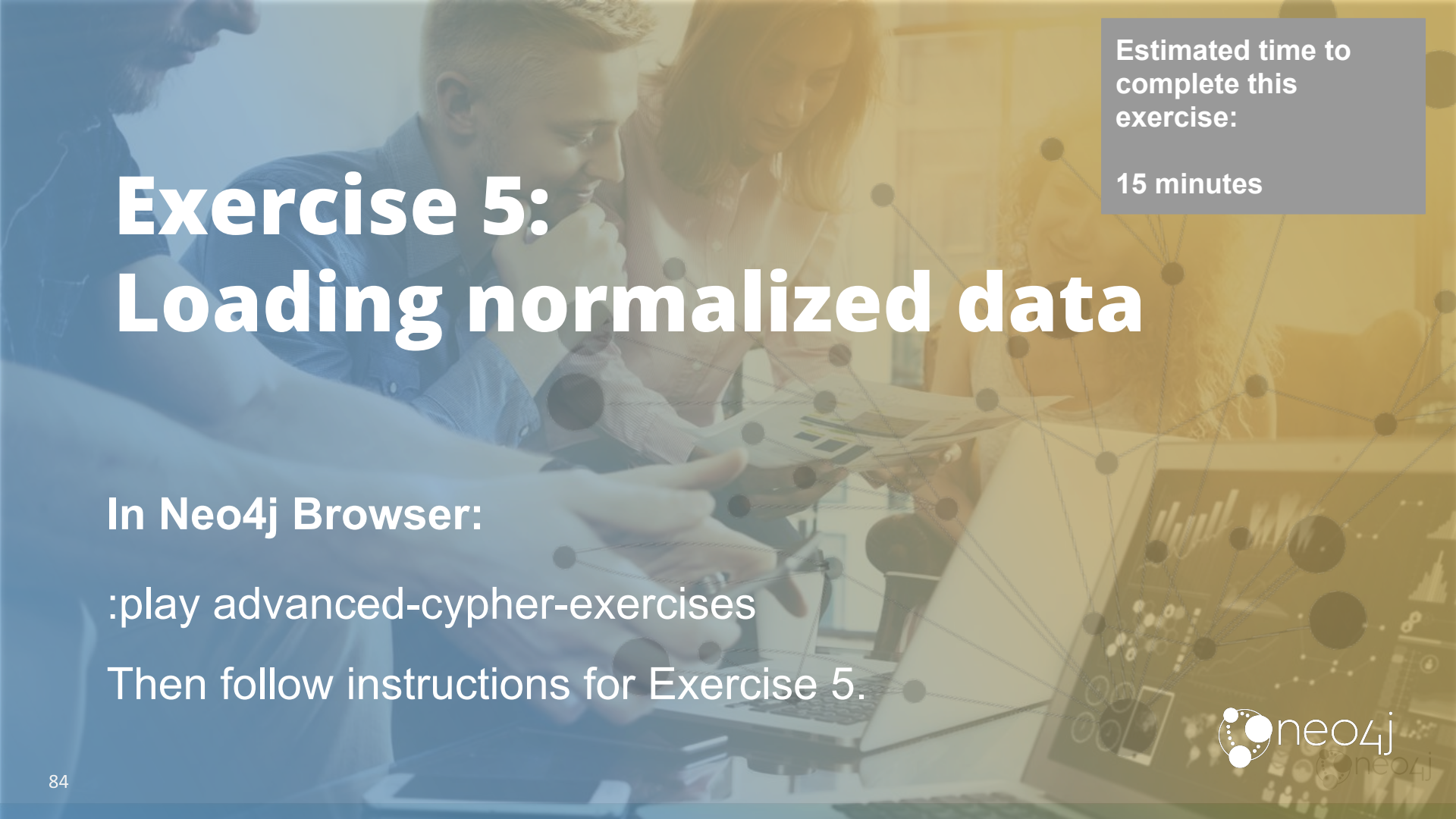
Table

Added 3099 labels, created 6876 relationships, completed after 1252 ms.

Define indexes after loading data

Create indexes for frequently looked up nodes:

```
CREATE INDEX ON :Movie(title);
CREATE INDEX ON :Person(name)
```



Estimated time to complete this exercise:

15 minutes

Exercise 5: Loading normalized data

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 5.

Exercise 5: Loading normalized data

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 5.

Summary: Loading normalized data

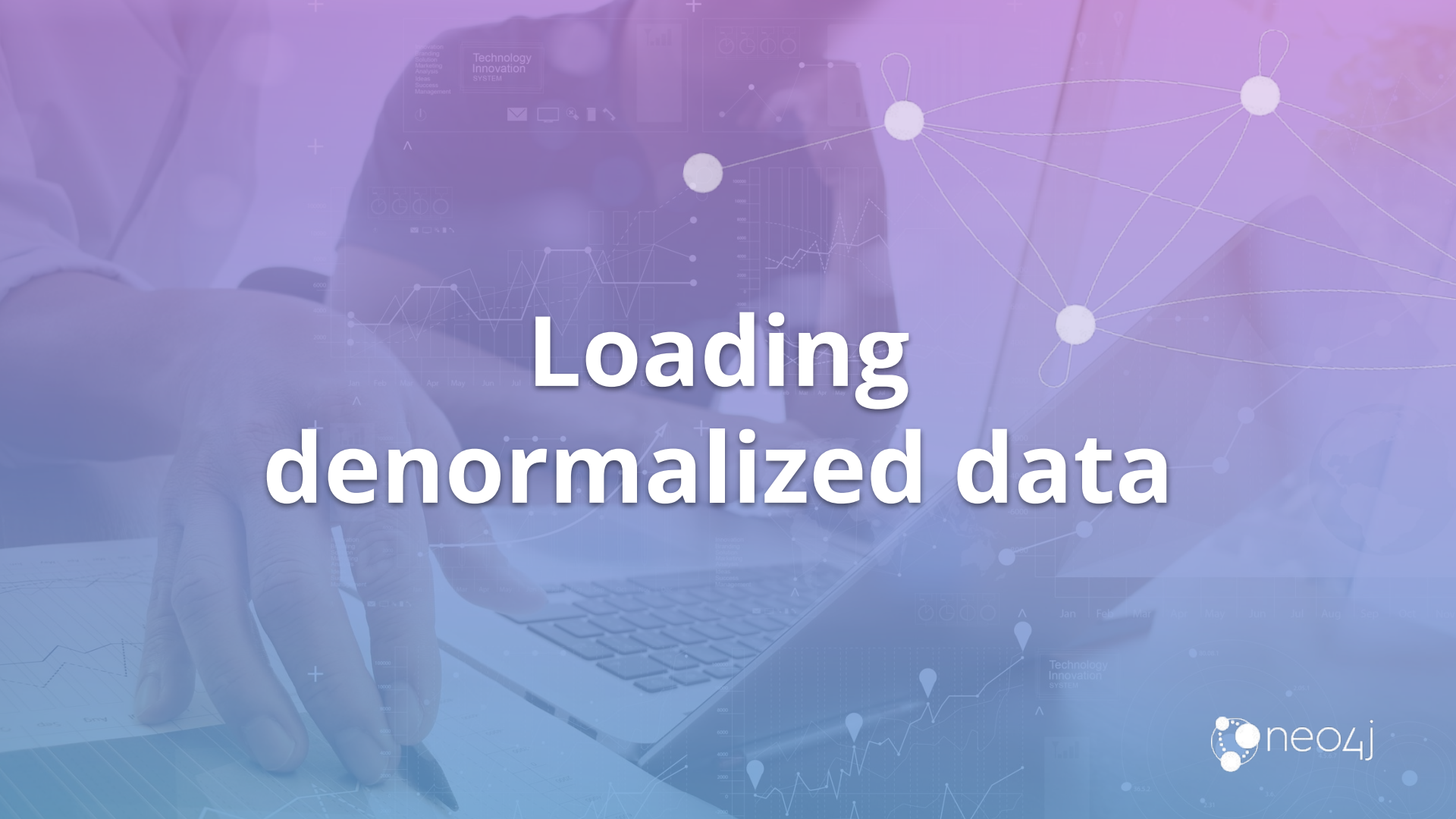
Advantages:

1. Simple MERGE statements to create nodes and relationships.
2. Single pass through each CSV file.

Disadvantages:

1. Large amount of constraint lookups, but required for the load.
2. Deadlocks could occur if loads are done in parallel (relationship creation is the problem).

Note: Later in this module you will learn about measuring Cypher execution times.


Technology
Innovation
SYSTEM
Loading
denormalized data
neo4j

Loading denormalized data

movies2.csv:

```
movieId,title,avgVote,releaseYear,tagline,genres,personType,personId,name,birthYear,deathYear,characters
241254,The Prince,5.600000,2014,"",Action:Thriller,ACTOR,62,Bruce Willis,1955,0,Omar
241254,The Prince,5.600000,2014,"",Action:Thriller,ACTOR,12261,Jason Patric,1966,0,Paul
241254,The Prince,5.600000,2014,"",Action:Thriller,DIRECTOR,81613,Brian A. Miller,0,0,""
226486,Tammy,5.200000,2014,"",Comedy,ACTOR,55536,Melissa McCarthy,1970,0,Tammy
226486,Tammy,5.200000,2014,"",Comedy,ACTOR,4038,Susan Sarandon,1946,0,Pearl
226486,Tammy,5.200000,2014,"",Comedy,DIRECTOR,170820,Ben Falcone,0,0,""
```

Two approaches for loading:

1. MERGE Movie data
2. MERGE Person data
3. Create relationships using conditional processing

1. MERGE Movie data and collect people (if memory permits)
2. UNWIND and MERGE Person data from people
3. Create relationships using conditional processing

Conditional processing with APOC

Syntax:

```
CALL apoc.do.when(<condition>,
     "<if-true-execute-cypher-statement>",
     "<else-execute-cypher-statement>",
       {<variable map>}) YIELD value
```

WITH required before this APOC call

Example:

```
WITH row, p, m
CALL apoc.do.when(row.personType = 'ACTOR',
    "MERGE (p)-
     [:ACTED_IN {roles: split(coalesce(row.characters,''), ':')}]
     ->(m)
        ON CREATE SET p:Actor",
     "MERGE(p)-[:DIRECTED]->(m)
        ON CREATE SET p:Director",
    {row:row, m:m, p:p}) YIELD value
RETURN count(*) // must have additional "dummy" clause after this APOC call
```

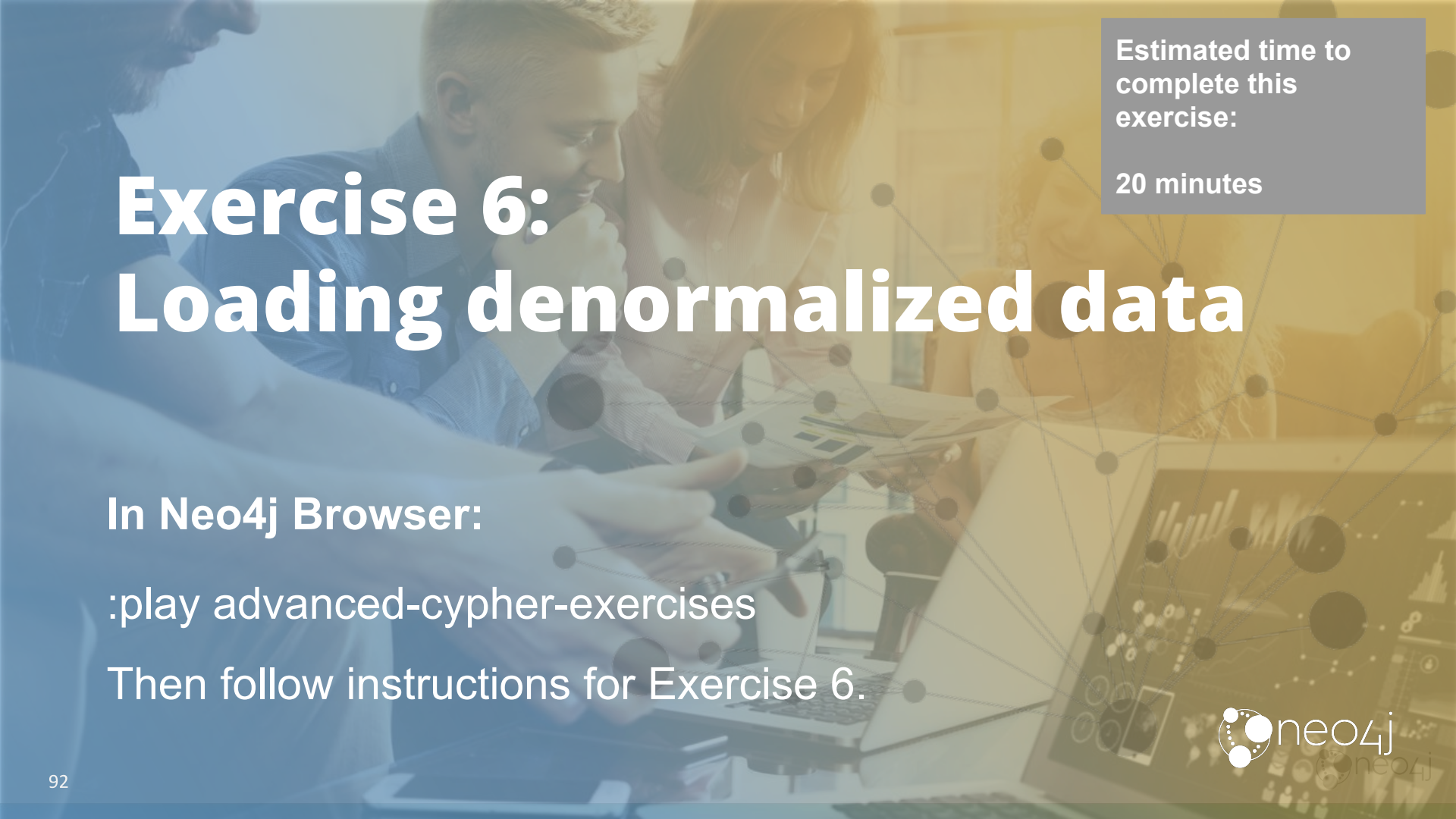
Cannot end a Cypher statement with this APOC call

Example: Loading data with APOC

```
LOAD CSV WITH HEADERS FROM 'https://data.neo4j.com/advanced-cypher/movies2.csv' AS row
MERGE (m:Movie {id:toInteger(row.movieId)})
   ON CREATE SET m.title=row.title, m.avgVote=toFloat(row.avgVote),
      m.releaseYear=toInteger(row.releaseYear), m.genres=split(row.genres,":")
MERGE (p:Person {id: toInteger(row.personId)})
   ON CREATE SET p.name = row.name, p.born = toInteger(row.birthYear),
      p.died = toInteger(row.deathYear)
WITH row, m, p
CALL apoc.do.when(row.personType = 'ACTOR',
     "MERGE (p)-[:ACTED_IN {roles: split(coalesce(row.characters,''), ':')}]->(m)
         ON CREATE SET p:Actor",
     "MERGE (p)-[:DIRECTED]->(m)
         ON CREATE SET p:Director",
      {row:row, m:m, p:p}) YIELD value AS value
RETURN count(*) // cannot end query with APOC call
```

Example: loading data with APOC/UNWIND

```
LOAD CSV WITH HEADERS FROM
    'https://data.neo4j.com/advanced-cypher/movies2.csv' AS row
WITH row.movieId as movieId, row.title AS title, row.genres AS genres, toInteger(row.releaseYear) AS releaseYear,
toFloat(row.avgVote) AS avgVote,
collect({id: row.personId, name:row.name, born: toInteger(row.birthYear), died: toInteger(row.deathYear),personType:
row.personType, roles: split(coalesce(row.characters,""),':')}) AS people
MERGE (m:Movie {id:movieId})
  ON CREATE SET m.title=title, m.avgVote=avgVote,
    m.releaseYear=releaseYear, m.genres=split(genres,":")
WITH *
UNWIND people AS person
MERGE (p:Person {id: person.id})
  ON CREATE SET p.name = person.name, p.born = person.born, p.died = person.died
WITH  m, person, p
CALL apoc.do.when(person.personType = 'ACTOR',
    "MERGE (p)-[:ACTED_IN {roles: person.roles}]->(m)
            ON CREATE SET p:Actor",
    "MERGE (p)-[:DIRECTED]->(m)
       ON CREATE SET p:Director",
    {m:m, p:p, person:person}) YIELD value AS value
RETURN count(*)  // cannot end query with APOC call
```



Estimated time to complete this exercise:

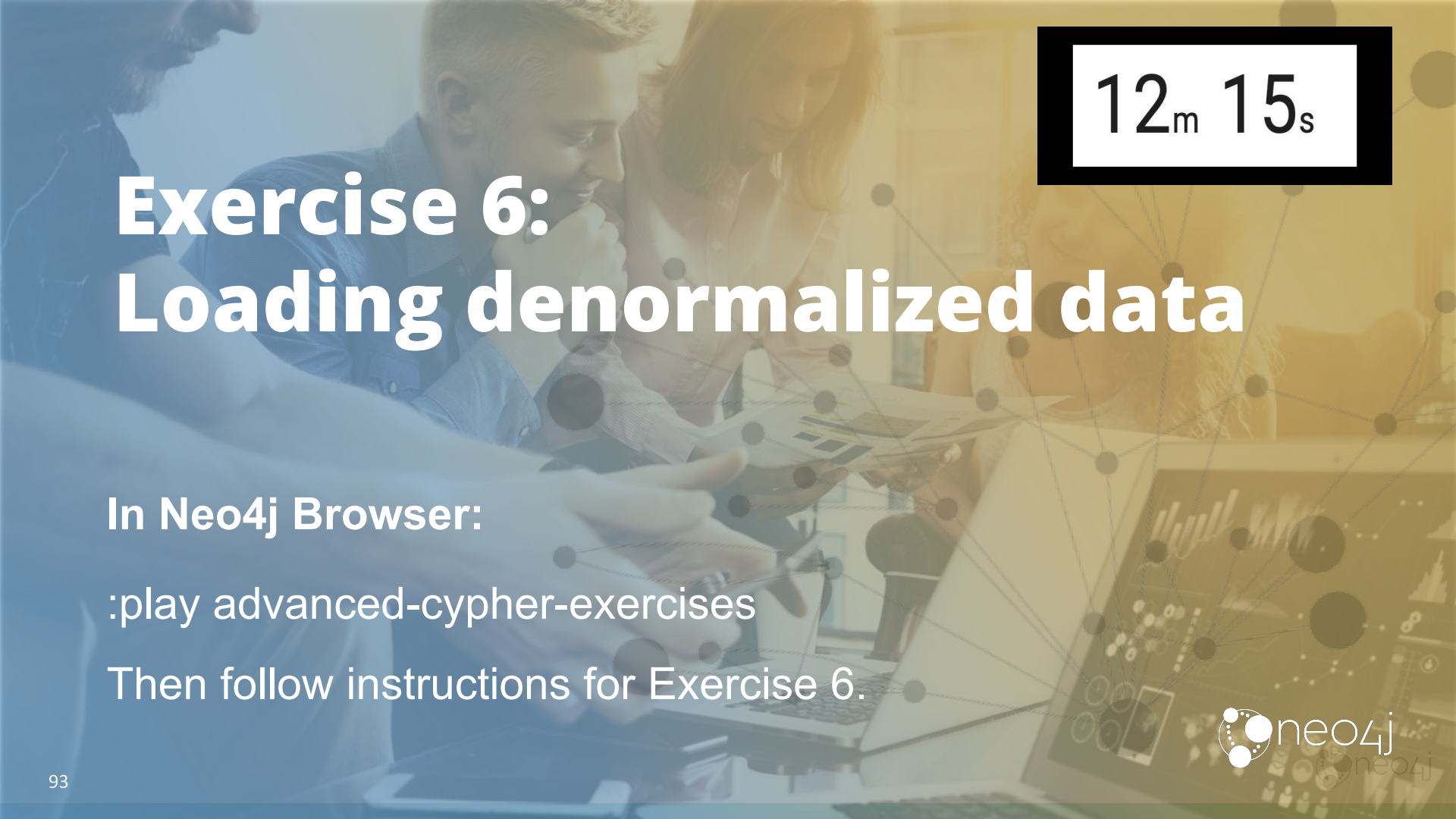
20 minutes

Exercise 6: Loading denormalized data

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 6.



Exercise 6: Loading denormalized data

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 6.

Summary: Loading denormalized data

Advantages:

1. Multiple passes through a single CSV file.
2. No deadlocks since a single process creates relationships

Disadvantages:

1. Still a large amount of constraint checking (MERGE), but required for the load.

Loading large datasets

neo4j

Loading large datasets

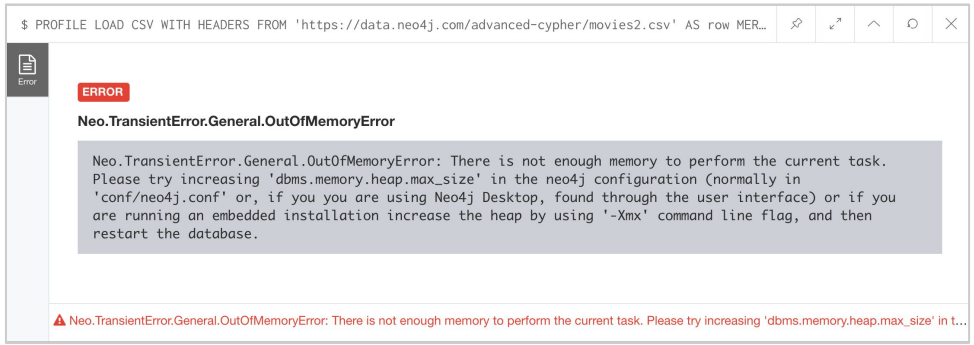
You can run out of memory if a large dataset is loaded or if many operations need to be performed per row read.

Solutions:

- USING PERIODIC COMMIT LOAD CSV…
  - Provided no eager operations performed during the load
  - Any eager operations in the query plan disables periodic commit.
- CALL apoc.periodic.iterate()

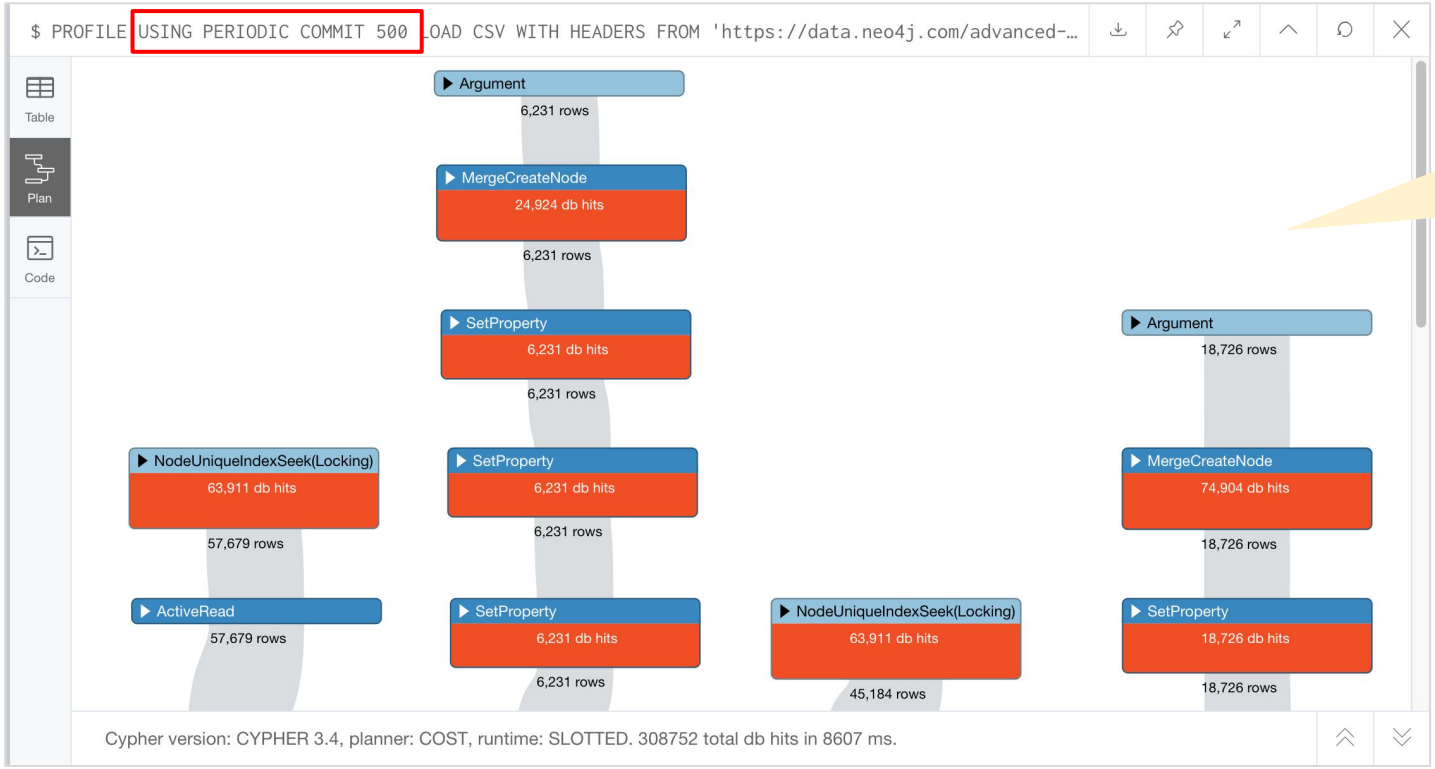
Out of memory for LOAD

```
LOAD CSV WITH HEADERS FROM 'https://data.neo4j.com/advanced-cypher/movies2.csv' AS row
MERGE (m:Movie {id:toInteger(row.movieId)})
  ON CREATE SET m.title=row.title, m.avgVote=toFloat(row.avgVote),
    m.releaseYear=toInteger(row.releaseYear), m.genres=split(row.genres,":")
MERGE (p:Person {id: toInteger(row.personId)})
  ON CREATE SET p.name = row.name, p.born = toInteger(row.birthYear),
    p.died = toInteger(row.deathYear)
```




```
dbms.memory.heap.initial_size=128m
dbms.memory.heap.max_size=128m
```

USING PERIODIC COMMIT



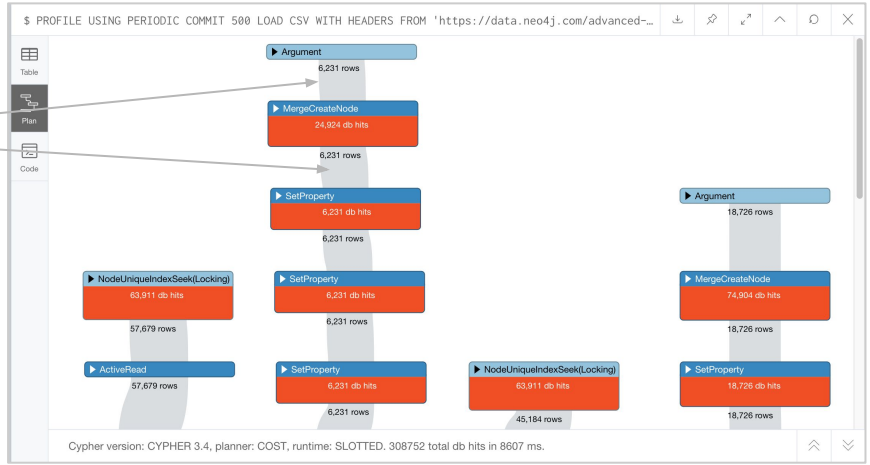
USING PERIODIC COMMIT works here because we have no eager operators

neo4j

Cypher query plans

Plan evaluation are **Eager** or **Lazy**

- Most query evaluation is lazy:
  - Operators pipe their output rows to their parent operators as soon as they are produced.
  - Child operator may not be finished before the parent receives and processes rows.
- An Eager operation can take 2 forms:
  - An **EagerAggregation** step caused by any of the aggregation functions (e.g. count, sum). This is normal and of lesser concern.
  - An **Eager** step caused by a reference later in the query to an object modified earlier in the query.

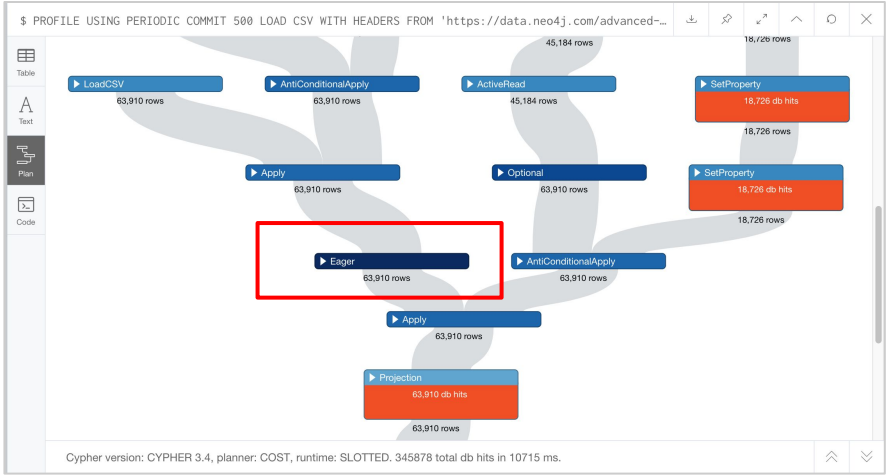


https://neo4j.com/docs/cypher-manual/current/execution-plans/

Eager operators disable PERIODIC COMMIT

- Sorting
- Aggregation
- Distinct
- Updating
- Eager

```
PROFILE USING PERIODIC COMMIT 500 LOAD CSV WITH HEADERS FROM
'https://data.neo4j.com/advanced-cypher/movies2.csv' AS row
MERGE (m:Movie {id:toInteger(row.movieId)})
  ON CREATE SET m.title=row.title, m.avgVote=toFloat(row.avgVote),
    m.releaseYear=toInteger(row.releaseYear), m.genres=split(row.genres,":")
WITH m, row
MERGE (p:Person {id: toInteger(row.personId)})
  ON CREATE SET p.name = row.name, p.born = toInteger(row.birthYear),
    p.died = toInteger(row.deathYear)
RETURN m.title ORDER BY m.title
```



Example: Using APOC for the CSV loading

```
CALL apoc.periodic.iterate(
"CALL apoc.load.csv('https://data.neo4j.com/advanced-cypher/movies2.csv' )
YIELD map AS row RETURN row",
"MERGE (m:Movie {id:toInteger(row.movieId)})
   ON CREATE SET m.title=row.title, m.avgVote=toFloat(row.avgVote),
      m.releaseYear=toInteger(row.releaseYear), m.genres=split(row.genres,':')
WITH m, row
MERGE (p:Person {id: toInteger(row.personId)})
   ON CREATE SET p.name = row.name, p.born = toInteger(row.birthYear),
      p.died = toInteger(row.deathYear)",
{batchSize: 500}
)
```

Example: apoc.periodic.iterate() results

```
$ CALL apoc.periodic.iterate( "CALL apoc.load.csv('https://data.neo4j.com/advanced-cypher/movies2.csv' ) YIELD map AS row RETURN row", "MERGE (m:Movie {id:to…
```

Table | Text | Code

batches	total	timeTaken	committedOperations	failedOperations	failedBatches	retries	errorMessages	batch	operations	wasTerminated
128	63910	11	63910	0	0	0	{ }	{ "total": 128, "committed": 128, "failed": 0, "errors": { } }	{ "total": 63910, "committed": 63910, "failed": 0, "errors": { } }	false

Exercise 7: Loading large datasets without running out of memory

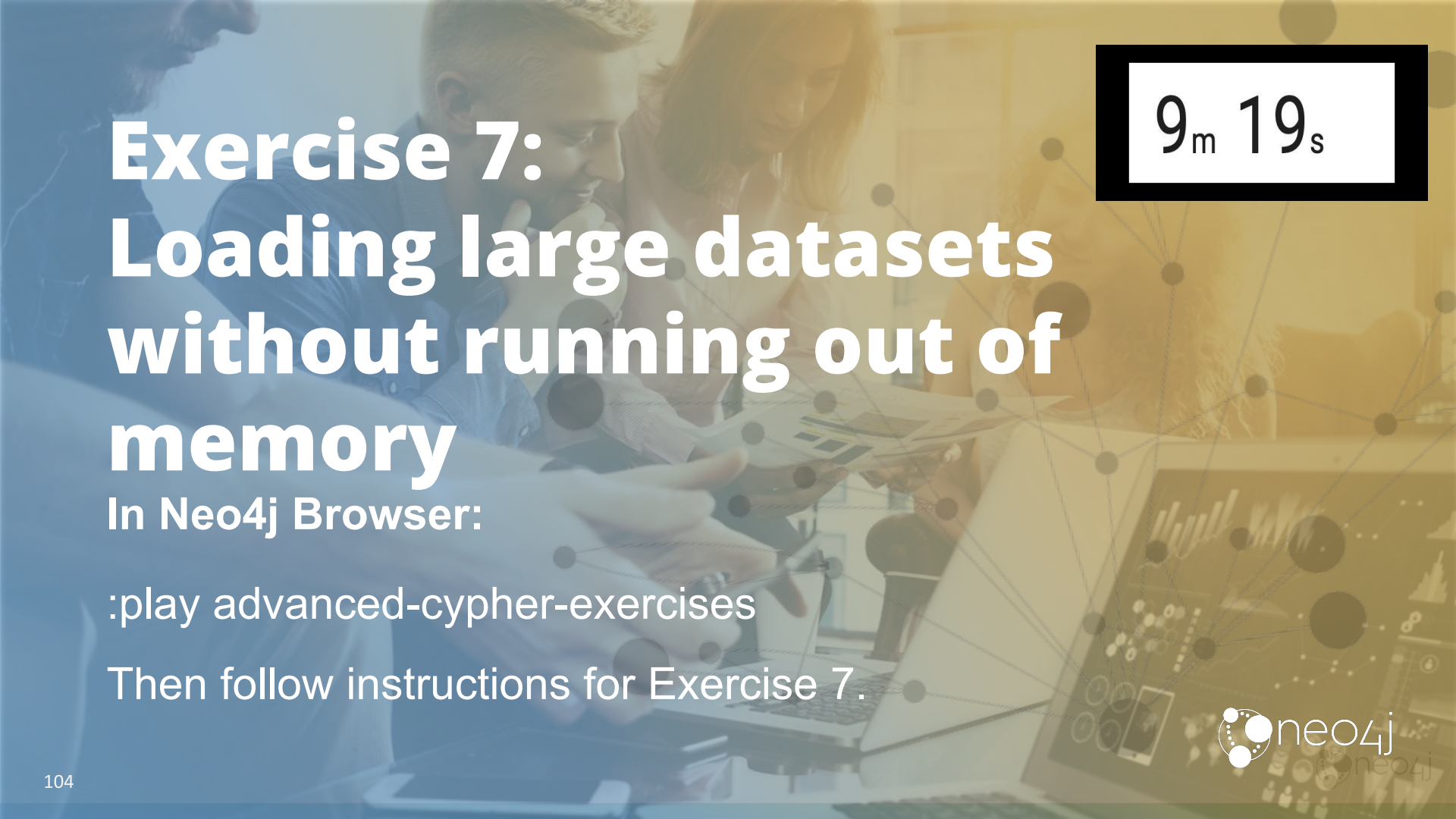
Estimated time to complete this exercise:

10 minutes

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 7.



Exercise 7: Loading large datasets without running out of memory

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 7.

Aggregating data

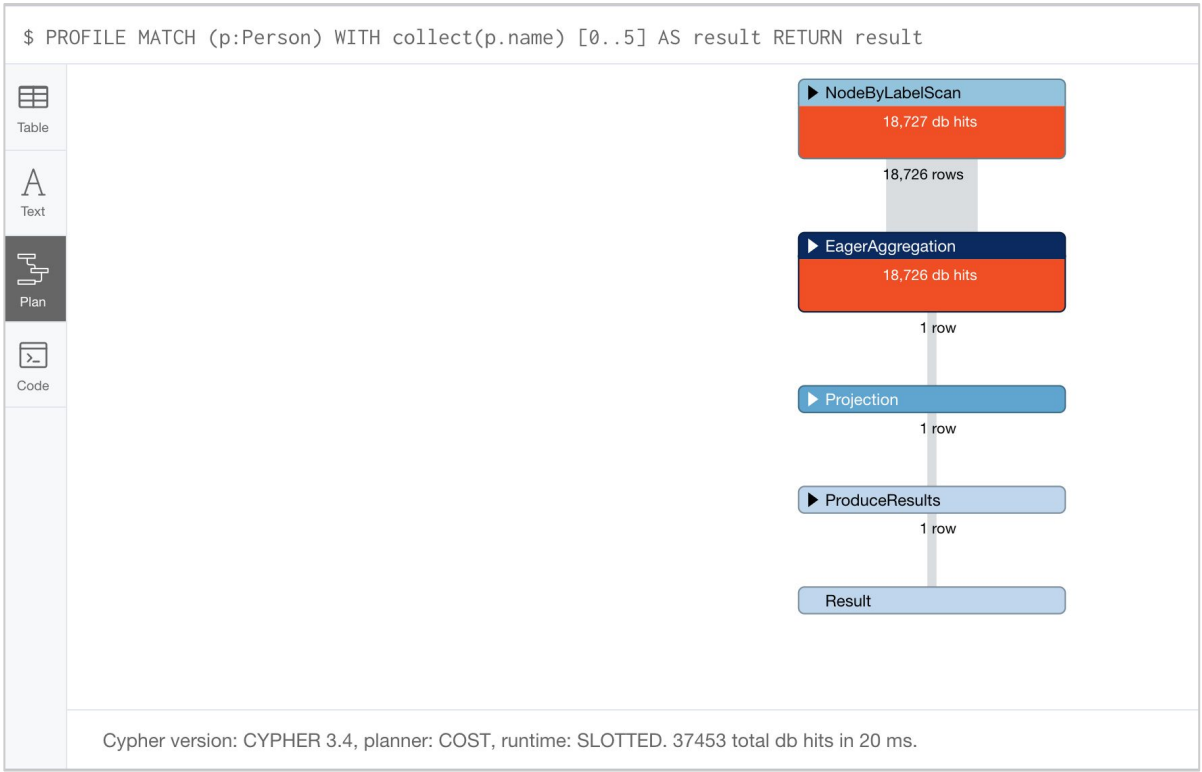
neo4j

What is aggregation?

- Traversing paths to gather data about a group of nodes.
- These keywords/functions trigger eager aggregation:
  - count()
  - collect()
  - ORDER BY (if no index)
- Properties can also be represented by lists which is what collect() returns or can be represented by maps (key/value pairs) which is what you can manipulate using the apoc.map functions.

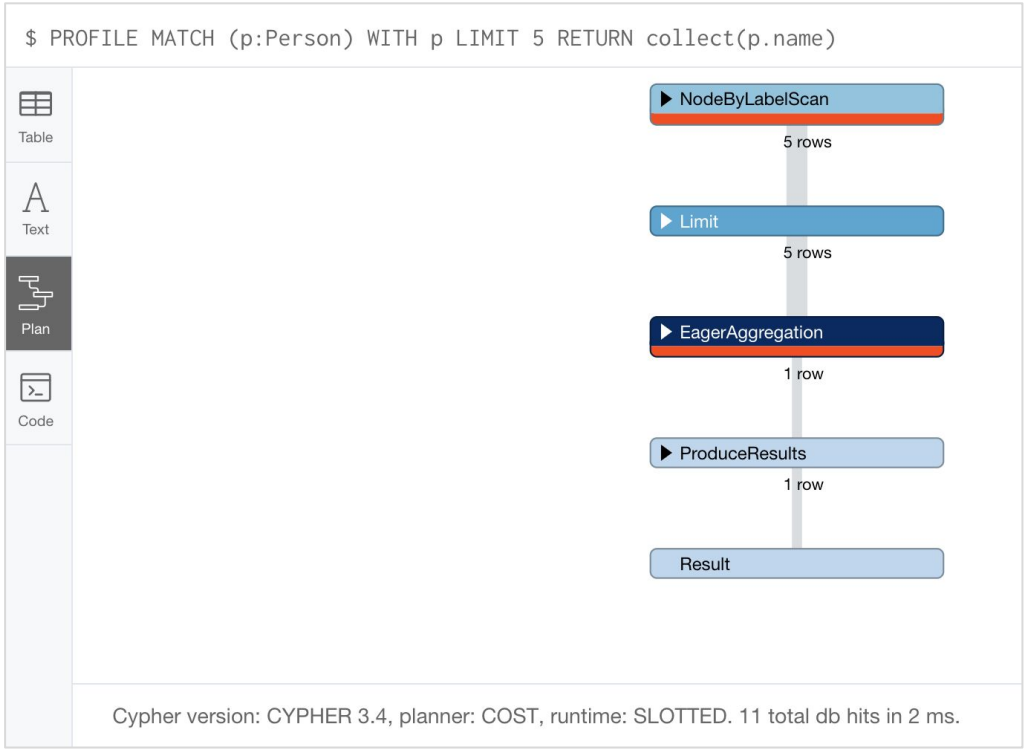
Limiting collection size - 1

You should <u>never</u> collect the data and then take a slice. (High number of db hits)



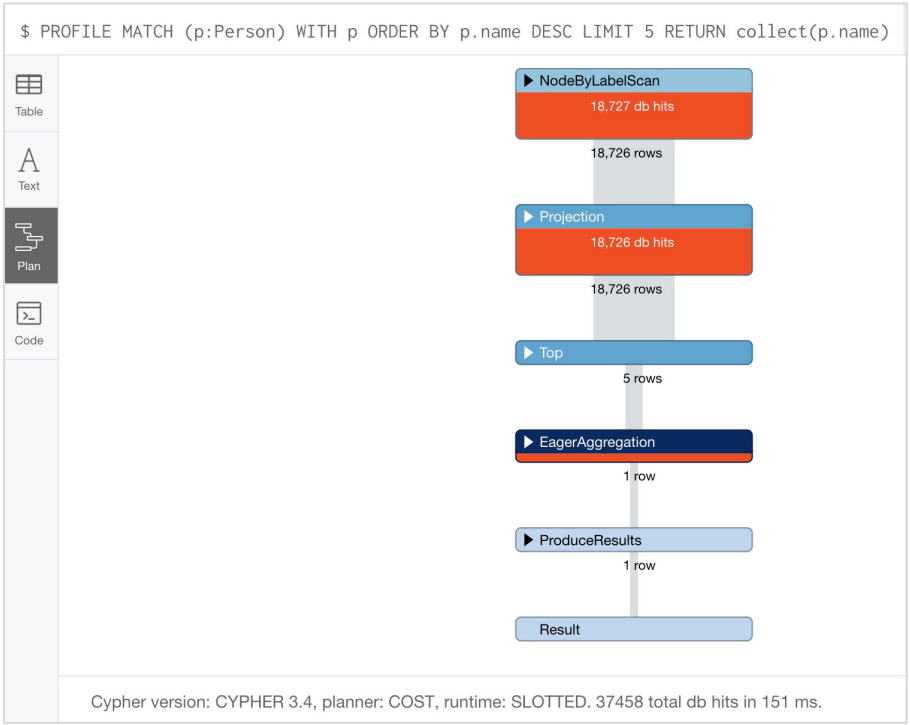
Limiting collection size - 2

Best practice: Collect the results after LIMIT is specified.



Limiting collection size - 3

But, if we need to sort the data before adding to a collection, even if LIMIT is specified, it is expensive.



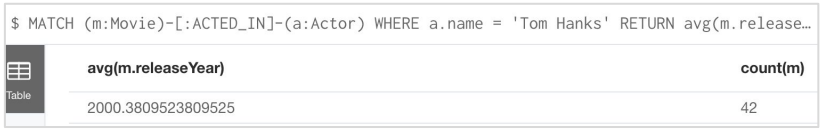
APOC collection functions

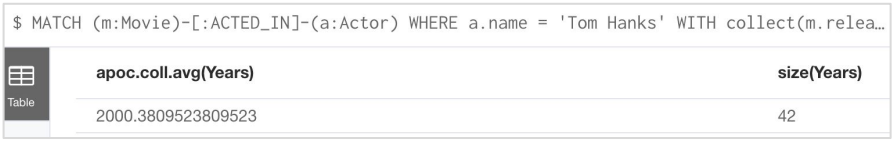
- sum, avg, min,max,stdev,
- zip, partition, pairs
- sort, toSet, contains, split
- indexOf, different
- occurrences, frequencies, flatten
- disjunct, subtract, union, …
- set, insert, remove
- intersection

Difference between Cypher and APOC collection/aggregation functions

```
MATCH (m:Movie)-[:ACTED_IN]-(a:Actor)
WHERE a.name = 'Tom Hanks'
RETURN avg(m.releaseYear), count(m)
```

```
MATCH (m:Movie)-[:ACTED_IN]-(a:Actor)
WHERE a.name = 'Tom Hanks'
WITH  collect(m.releaseYear) AS Years
// other Cypher code
RETURN apoc.coll.avg(Years), size(Years)
```





Cypher operates on aggregation of rows.

APOC operates on a an existing list, whether it is from a row or aggregated in Cypher. apoc.coll.avg is a scalar function that sometimes makes your coding easier.

Example: Manipulating a list with APOC

```
MATCH (p:Actor {name:'Keanu Reeves'})
-[act:ACTED_IN]->(m:Movie)
WITH collect(act.roles) AS lists_of_roles
RETURN lists_of_roles, size(lists_of_roles)
```

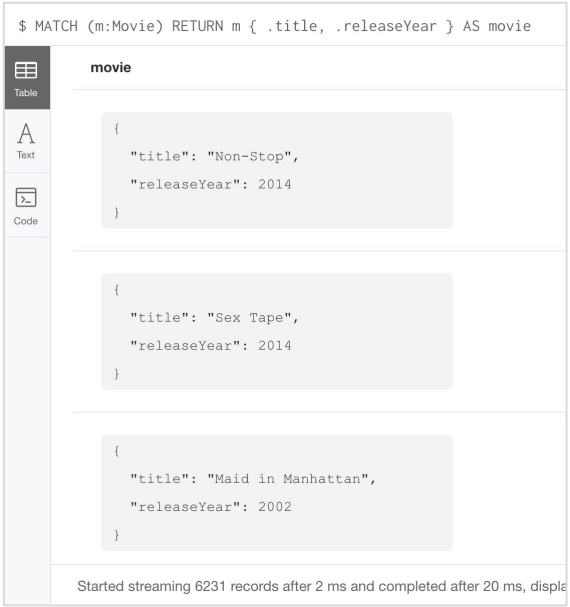
```
MATCH (p:Actor {name:'Keanu Reeves'})
-[act:ACTED_IN]->(m:Movie)
WITH collect(act.roles) AS lists_of_roles
// remove embedded lists
WITH apoc.coll.flatten(lists_of_roles) AS all_roles
// remove duplicates
WITH apoc.coll.toSet(all_roles) AS roles
RETURN roles, size(roles)
```

lists_of_roles	size(lists_of_roles)
[["Shane Falco"], ["Alex Wyler"], ["David Allen Griffin"], ["Tod Higgins"], ["Le Chevalier Raphael Danceny"], ["Nelson"], ["Donnie Barksdale"], ["Julian Mercer"], ["Scott Favor"], ["Johnny Mnemonic"], ["Kai"], ["Detective Tom Ludlow"], ["Neo (Voice)"], ["Ted Logan"], ["Fred", "Bob Arctor"], ["Klaatu"], ["Eddie Kasalivich"], ["FBI Special Agent John 'Johnny' Utah"], ["Henry"], ["Kevin Lomax"], ["Donaka Mark"], ["Jonathan Harker"], ["Jack Traven"], ["Don Juan"], ["Neo"], ["Neo"], ["Himself", "Neo"], ["Conor O'Neill"], ["John Constantine"], ["Neo"], ["Himself"], ["Ted Logan"], ["Paul Sutton"], ["Chris Nadeau"]]	34

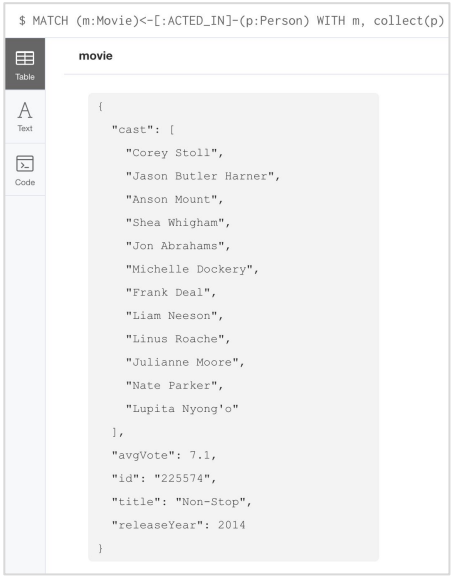
Map projections

Use the properties of a node to create a map.

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



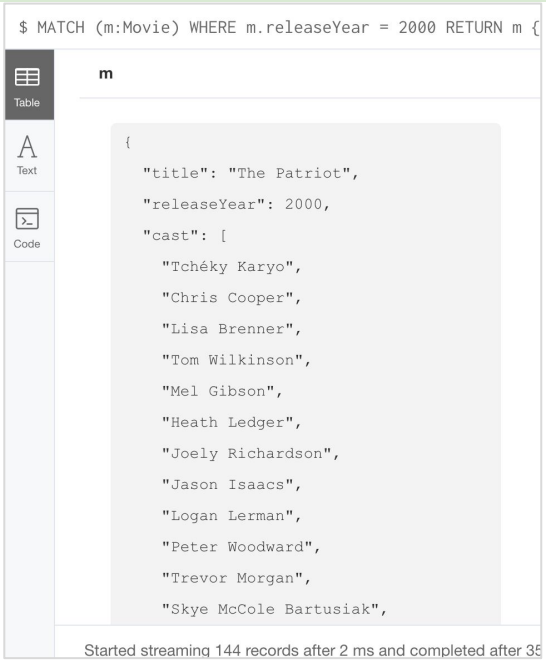

```
MATCH (m:Movie)<-[:ACTED_IN]-(p:Person)
WITH m, collect(p) AS people
RETURN m { .*, cast: [x in people | x.name] }
AS movie
```



Example: Using map projection and pattern comprehension to return custom data

```
MATCH (m:Movie)
WHERE m.releaseYear = 2000
RETURN m {.title, .releaseYear}
```


```
MATCH (m:Movie)
WHERE m.releaseYear = 2000
RETURN m {.title, .releaseYear, cast: [(m)<--(a:Actor) | a.name]}
```



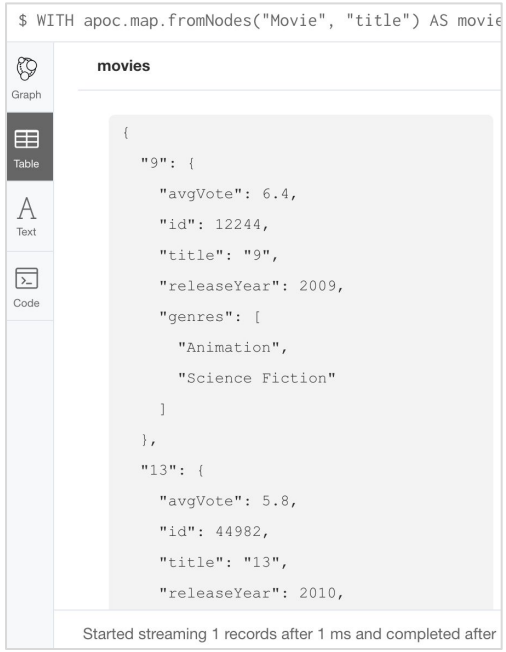
neo4j

Map functions - apoc.map.*

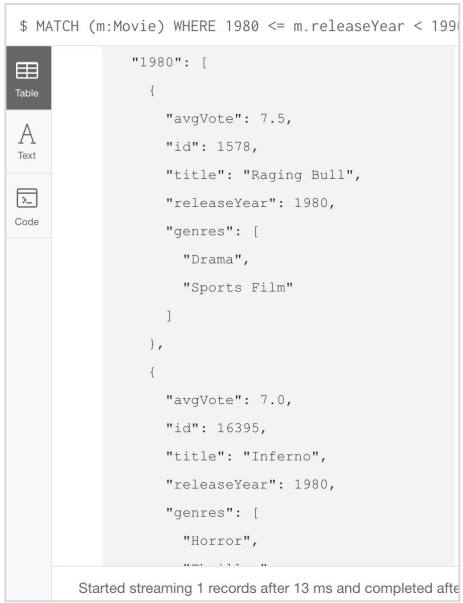
- fromNodes, fromPairs, fromLists, fromValues
- merge
- setKey, removeKey
- clean(map,[keys],[values])
- groupBy[Multi]

Examples: Using APOC map functions

```
WITH apoc.map.fromNodes("Movie", "title")
AS movies
RETURN movies
```



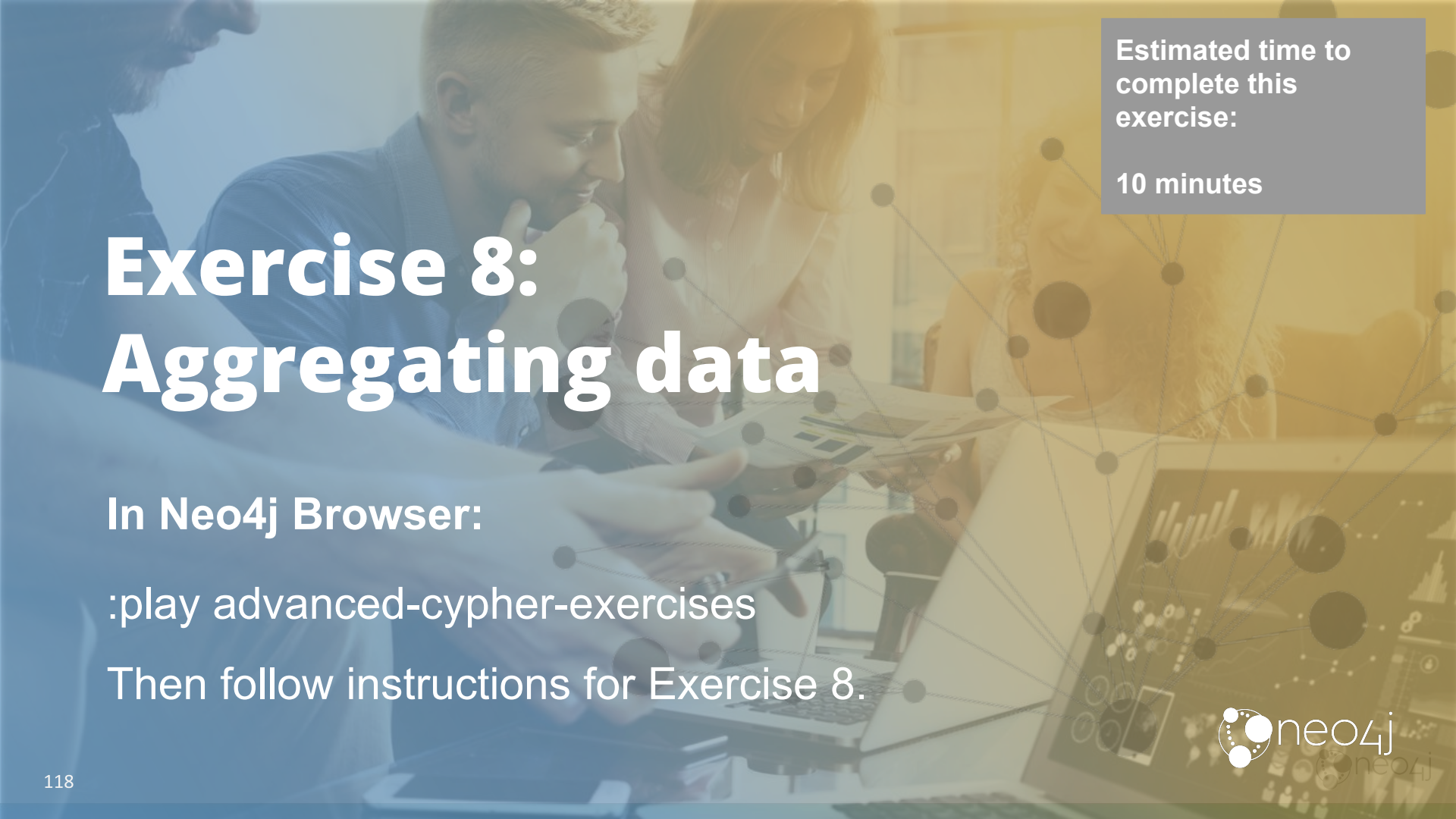

```
MATCH (m:Movie)
WHERE 1980 <= m.releaseYear < 1990
WITH collect(properties(m)) AS movies
RETURN apoc.map.groupByMulti(movies,"releaseYear")
```



Best practice: Create a map for faster lookup

```
MATCH (g:Genre)
WITH apoc.map.fromPairs(collect([g.name, g])) AS genres

MATCH (m) WHERE not exists (m)-[:IS_GENRE]->()
WITH genres, m LIMIT 10000
UNWIND m.genres AS genre
WITH genres[genre] AS g, m
CREATE (m)-[:IS_GENRE]->(g)
```



Estimated time to complete this exercise:

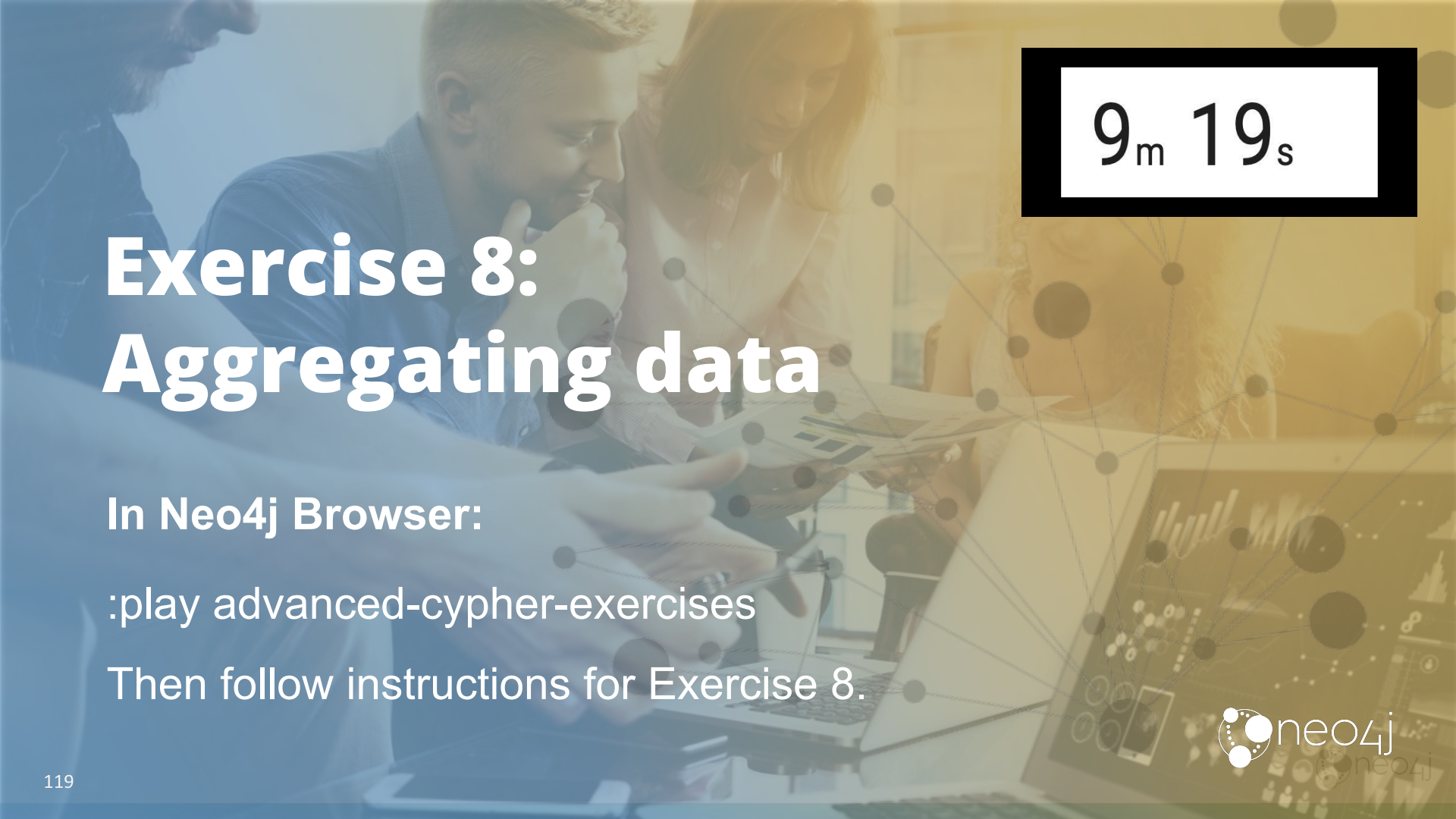
10 minutes

Exercise 8: Aggregating data

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 8.

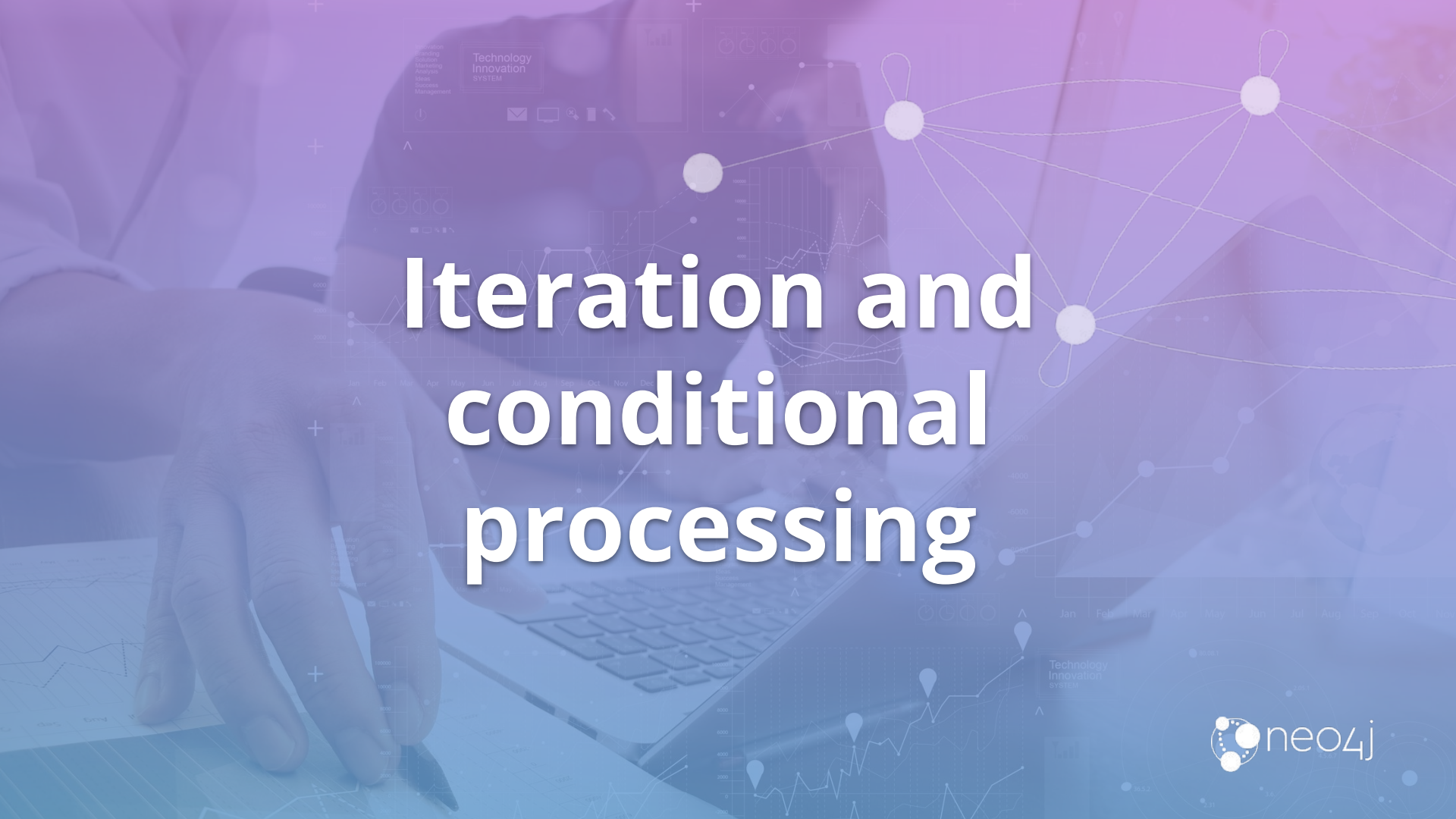


Exercise 8: Aggregating data

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 8.



Iteration and conditional processing

neo4j

Iteration and conditional processing

- Iterate through a list of nodes and perform write or update operations.
  - FOREACH
- Perform conditional processing
  - FOREACH + CASE
  - CASE
  - apoc.do.when

FOREACH

FOREACH (<elements to iterate> | <code to execute>)

- A block of code that iterates through a list where the list is created before the FOREACH block. The list can contain nodes, relationships, or paths.
- Variables set within the FOREACH block are not available when the FOREACH block ends.
- Code to execute is code to create or update data in the graph.
- MATCH clauses are not permitted within a FOREACH block.
- FOREACH blocks can be nested.
- A FOREACH step in a query plan is eager.

Examples: FOREACH

```
MATCH (m:Movie)-[:IS_GENRE]->(g:Genre)
WHERE g.name = 'Family'
WITH collect(m) AS FamilyMovies
FOREACH (x IN FamilyMovies | SET x.rating = 'G')
```

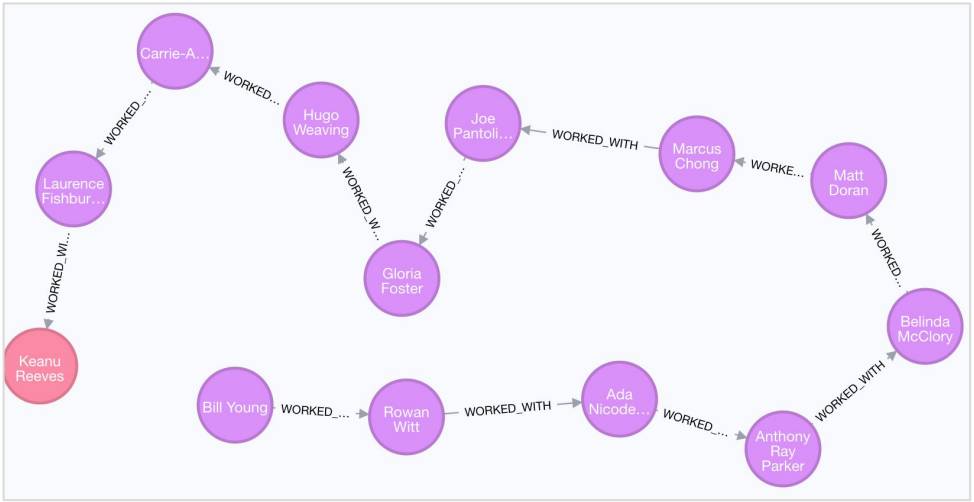
=

```
MATCH (m:Movie)-[:IS_GENRE]->(g:Genre)
WHERE g.name = 'Family'
SET m.rating = 'G'
```

```
MATCH p = (a:Actor)-[:ACTED_IN]-(m:Movie)
WHERE 1942 <= m.releaseYear <= 1945
FOREACH (x IN nodes(p) | SET x.USWarTime = true)
```

Example: Nested FOREACH

```
MATCH (a:Actor)-[:ACTED_IN]->(m:Movie)
WHERE m.title = 'The Matrix'
WITH collect(a) AS cast
FOREACH (i IN range(0, size(cast) - 2) |
 FOREACH (node1 IN [cast[i]] |
  FOREACH (node2 IN [cast[i+1]] |
   MERGE (node1)-[:WORKED_WITH]->(node2))))
```



Using CASE for conditional assignment - 1

```
CASE <expression-to-test>
   WHEN <value1>  THEN <result1>
   WHEN <value2>  THEN <result2>
   . . .
   ELSE <default-result>
END
```

```
MATCH (m:Movie)
WHERE m.title contains('Matrix')
WITH m,
CASE m.releaseYear
   WHEN 1999 THEN 'Oldest in Series'
   WHEN 2010 THEN 'Newest in Series'
   ELSE 'One in Series'
END
   AS series
RETURN m.title, m.releaseYear, series
```

m.title	m.releaseYear	series
"The American Matrix - Age Of Deception"	2010	"Newest in Series"
"The Matrix"	1999	"Oldest in Series"
"The Matrix Reloaded"	2003	"One in Series"
"The Matrix Revisited"	2001	"One in Series"
"The Matrix Revolutions"	2003	"One in Series"

Note: You cannot update the graph in a CASE clause.

Using CASE for conditional assignment - 2

```
CASE
   WHEN <expression1>  THEN <result1>
   WHEN <expression2>  THEN <result2>
   . . .
   ELSE <default-result>
END
```

```
MATCH (m:Movie)
WHERE m.title contains('Matrix')
WITH m,
CASE
   WHEN m.releaseYear < 2000 THEN 'Older in Series'
   WHEN m.releaseYear >= 2010 THEN 'Newer in Series'
   ELSE 'One in Series'
END
   AS series
RETURN m.title, m.releaseYear, series
```

m.title	m.releaseYear	series
"The American Matrix - Age Of Deception"	2010	"Newer in Series"
"The Matrix"	1999	"Older in Series"
"The Matrix Reloaded"	2003	"One in Series"
"The Matrix Revisited"	2001	"One in Series"
"The Matrix Revolutions"	2003	"One in Series"

Conditionally updating the graph

```
MATCH (m:Movie)
WHERE m.title contains("Matrix")
FOREACH (ignoreMe IN CASE WHEN m.releaseYear < 2000 THEN [1] ELSE [] END |
  SET m.series = "Older in Series")
FOREACH (ignoreMe IN CASE WHEN m.releaseYear >= 2010 THEN [1] ELSE [] END |
  SET m.series = "Newer in Series")
FOREACH (ignoreMe IN CASE WHEN 1999 < m.releaseYear < 2010 THEN [1] ELSE [] END |
  SET m.series = "One in Series")
RETURN m.title, m.releaseYear, m.series
```

m.title	m.releaseYear	m.series
"The American Matrix - Age Of Deception"	2010	"Newer in Series"
"The Matrix"	1999	"Older in Series"
"The Matrix Reloaded"	2003	"One in Series"
"The Matrix Revisited"	2001	"One in Series"
"The Matrix Revolutions"	2003	"One in Series"

FOREACH vs. UNWIND

- Both can iterate through a set of entities/rows to update the graph.
- You cannot use MATCH within a FOREACH clause.
- Variables set in a FOREACH cannot be used later on in the processing.

```
WITH
  [{name: "Event 1", timetree: {day: 1, month: 1, year: 2014}},
   {name: "Event 2", timetree: {day: 2, month: 1, year: 2014}}]
  AS events
FOREACH (event IN events |
  CREATE (e:Event {name: event.name})
  MERGE (year:Year {year: event.timetree.year })
  MERGE (year)-[:HAS_MONTH]->
      (month {month: event.timetree.month })
  MERGE (month)-[:HAS_DAY]->
      (day {day: event.timetree.day })
  CREATE (e)-[:HAPPENED_ON]->(day))
```

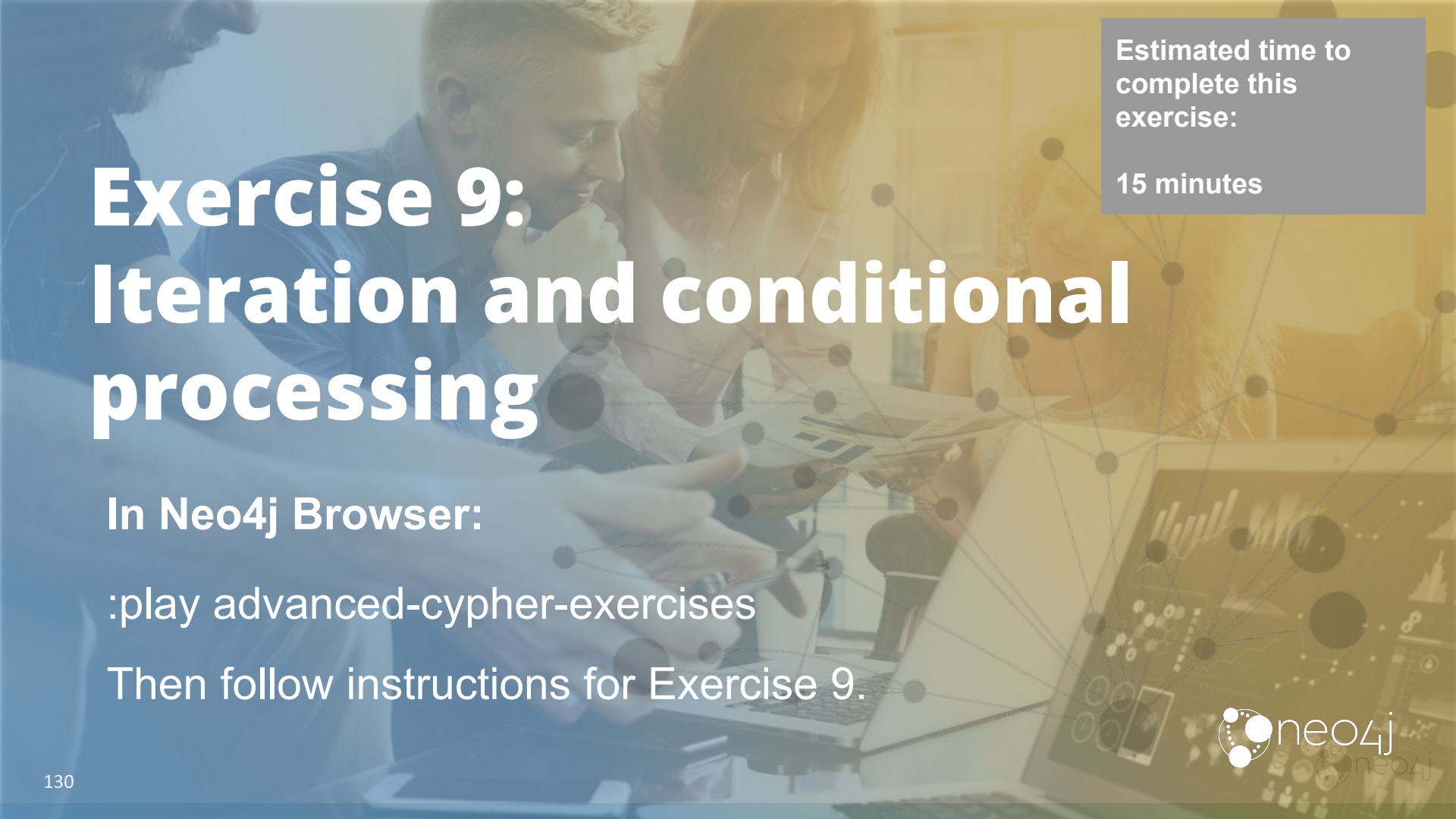
```
WITH
  [{name: "Event 1", timetree: {day: 1, month: 1, year: 2014}},
   {name: "Event 2", timetree: {day: 2, month: 1, year: 2014}}]
  AS events
UNWIND events AS event
CREATE (e:Event {name: event.name})
WITH e, event.timetree AS timetree
MATCH (year:Year {year: timetree.year }),
      (year)-[:HAS_MONTH]->
            (month {month: timetree.month }),
      (month)-[:HAS_DAY]->(day {day: timetree.day })
CREATE (e)-[:HAPPENED_ON]->(day)
```

More efficient and can use variables further down in the processing.

More resources

cypher-tips-and-tricks.adoc (by Michael Hunger)

https://gist.github.com/jexp/caeb53acfe8a649fecade4417fb8876a



Estimated time to complete this exercise:

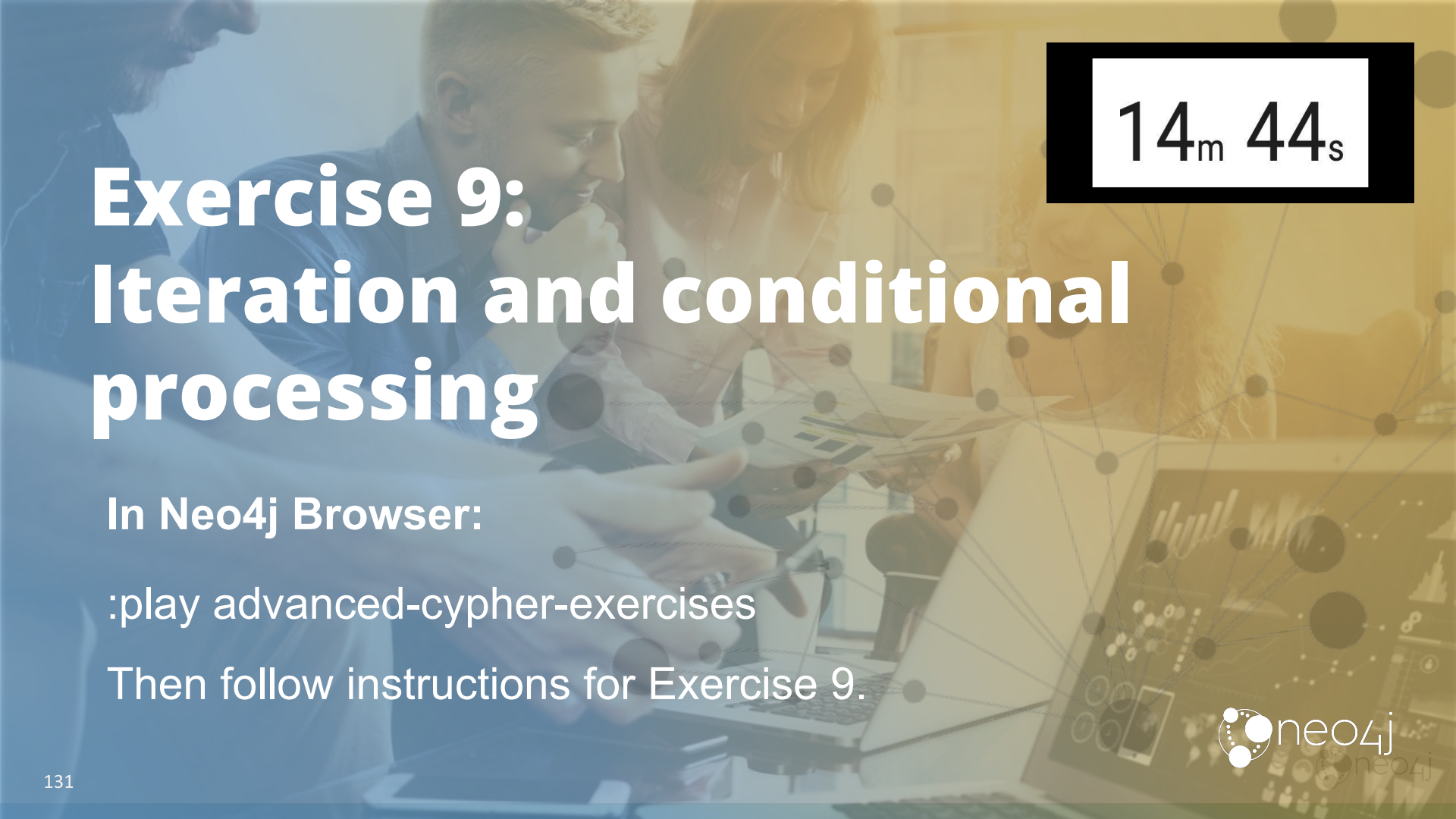
15 minutes

Exercise 9: Iteration and conditional processing

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 9.



Exercise 9: Iteration and conditional processing

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 9.


Check your understanding

Question 1

What APOC procedure can be used to retrieve distinct nodes where you are interested in nodes that have relationships to a nodes of type "X" with a hard limit on the depth of the relationship from a starting node?

Select the correct answer.

- ❏ apoc.query.getDistinct()
- ❏ apoc.query.subgraphNodes()
- ❏ apoc.path.subgraphNodes()
- ❏ apoc.path.getDistinct()

Answer 1

What APOC procedure can be used to retrieve distinct nodes where you are interested in nodes that have relationships to a nodes of type "X" with a hard limit on the depth of the relationship from a starting node?

Select the correct answer.

- ❏ apoc.query.getDistinct()
- ❏ apoc.query.subgraphNodes()
- ✔ apoc.path.subgraphNodes()
- ❏ apoc.path.getDistinct()

Question 2

Suppose you have this code to combine the results of two queries using UNION:

```
MATCH (x:Movie) WHERE x.title contains('Matrix') RETURN x.title AS title

UNION

MATCH (y:Movie) WHERE y.title STARTS WITH 'The'  RETURN y.title AS XX
```

What must XX be in the second MATCH clause?

Select the correct answer.

- ❏ Something other than Title, for example Title2
- ❏ title
- ❏ +title
- ❏ There should be no alias used for the second MATCH, it should return y.title.

Answer 2

Suppose you have this code to combine the results of two queries using UNION:

```
MATCH (x:Movie) WHERE x.title contains('Matrix') RETURN x.title AS title

UNION

MATCH (y:Movie) WHERE y.title STARTS WITH 'The'  RETURN y.title AS XX
```

What must XX be in the second MATCH clause?

Select the correct answer.

- ❏ Something other than Title, for example Title2
- ✔ title
- ❏ +title
- ❏ There should be no alias used for the second MATCH, it should return y.title.

Question 3

What Cypher clauses do you use to conditionally update a graph?

Select the correct answers.

- ❑ IF/THEN/ELSE
- ❑ CASE
- ❑ FOREACH
- ❑ REPEAT

Answer 3

What Cypher clauses do you use to conditionally update a graph?

Select the correct answers.

❑ IF/THEN/ELSE

✔ CASE

✔ FOREACH

❑ REPEAT

Answer 3

In general, what is the maximum number of nodes or relationships that you can easily create using LOAD CSV?

Select the correct answer.

- ❑ 1K
- ❑ 10K
- ❑ 1M
- ✔ 10M

Summary

You should now be able to:

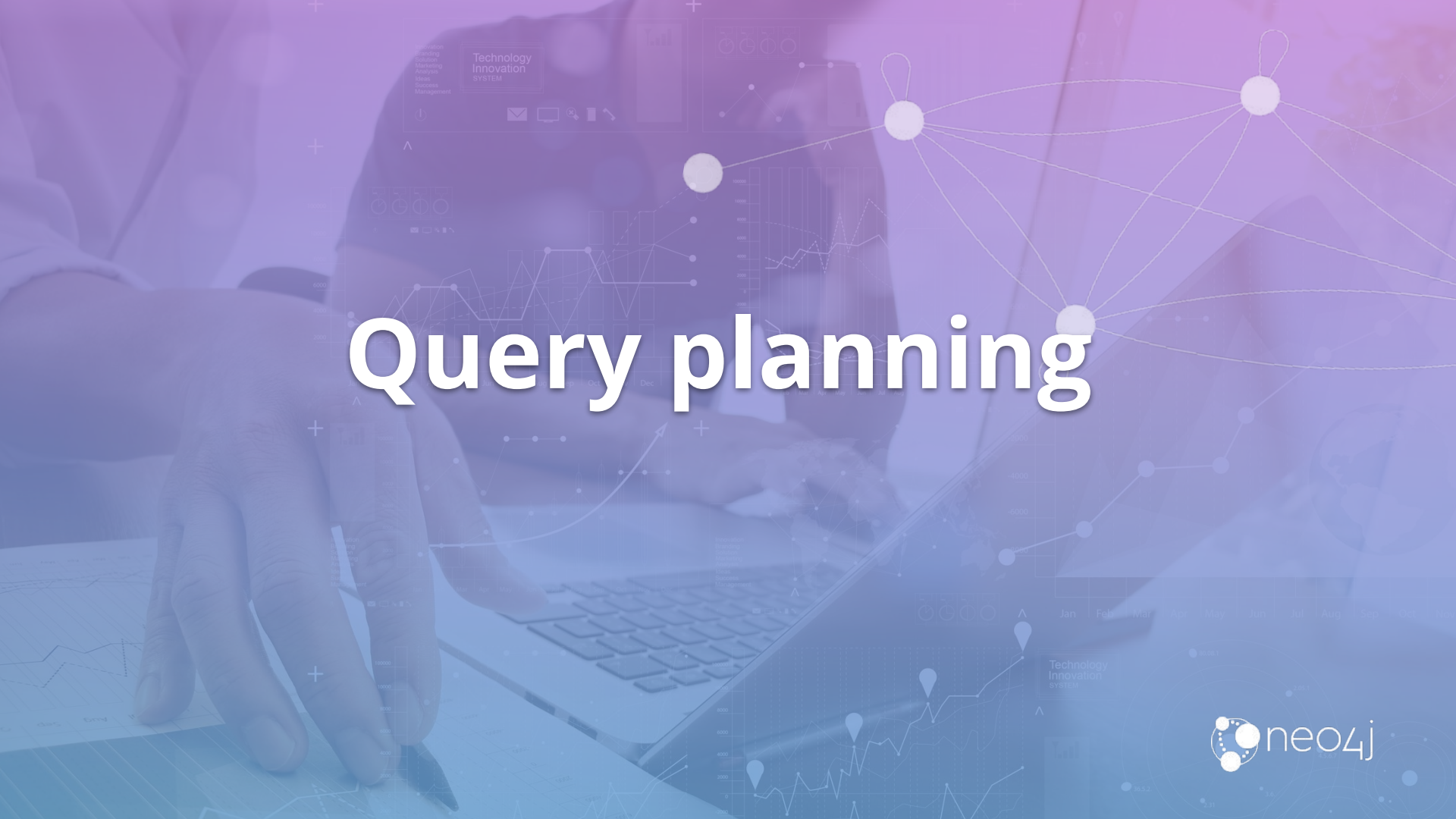
- Minimize graph traversals for queries.
- Use APOC for graph traversal.
- Use UNWIND as well as pattern and list comprehension.
- Combine query results.
- Import normalized and denormalized data into the graph.
- Aggregate data and work with lists and maps.
- Iterate and perform conditional processing.


Query Tuning
v 1.0
neo4j

Overview

At the end of this module, you should be able to:

- Describe query planning.
- Use DB stats and the count store.
- Use some best practices for writing optimal queries.
- Create and use indexes
- Install and use the Query Analyzer Tool.



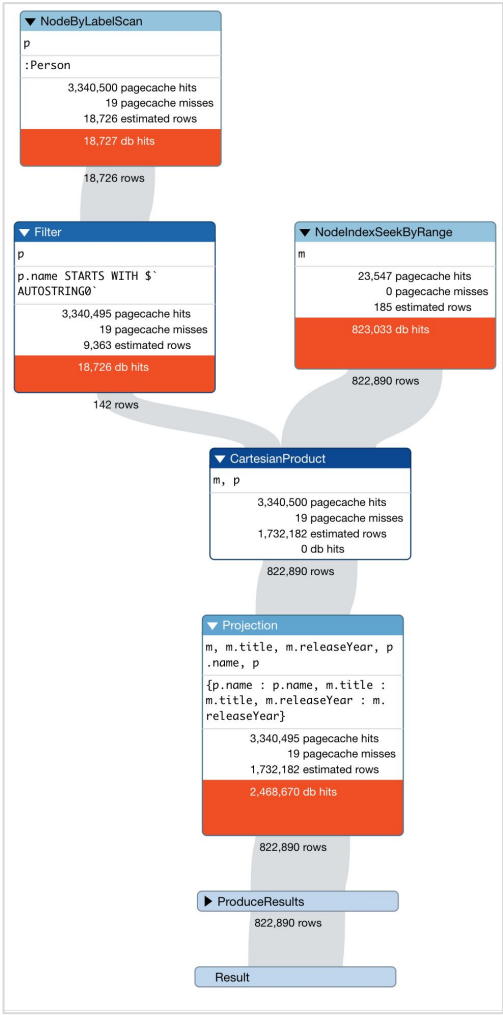
Query planning

neo4j

Cypher query planning - 1

- Query execution is decomposed into operators, that define a unit of work.
- Each operator becomes a node in the execution plan “tree”.
- Operators take zero or more rows as input and output
- Operators can converge inputs or split outputs to next operator in the execution plan tree

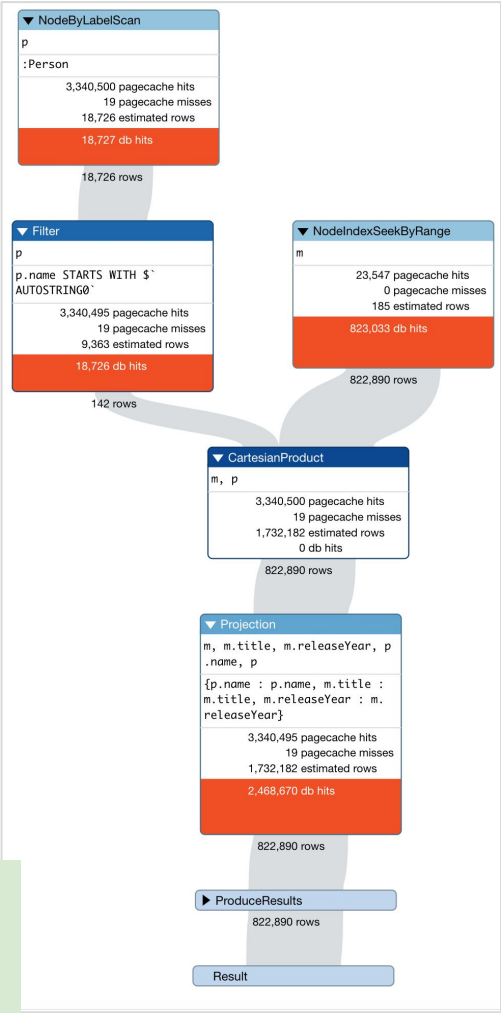
```
PROFILE MATCH (p:Person) WHERE p.name STARTS WITH 'Tom'
MATCH (m:Movie) WHERE m.releaseYear > 1975
RETURN p.name, m.title, m.releaseYear
```



Note: This query creates a cartesian product. This is just for illustrative purposes to show an example query plan. Sometimes cartesian products are useful, especially when creating nodes in the graph and you want to relate them.

Cypher query planning - 2

- Evaluation of the query plan begins at the leaf nodes of the tree.
  - Leaf nodes have no input rows and generally are scan and index seek operators.
  - These operators obtain the data directly from the storage engine or index, thus incurring database hits (db hits).
- Each operator will send a request to the storage engine to do work such as retrieving or updating data. A database hit is an abstract unit of this storage engine work.




```
PROFILE MATCH (p:Person) WHERE p.name STARTS WITH 'Tom'
MATCH (m:Movie) WHERE m.releaseYear > 1975
RETURN p.name, m.title, m.releaseYear
```

Actions that incur DB hits - 1

Create actions:
Create a node
Create a relationship
Create a new node label
Create a new relationship type
Create a new ID for property keys with the same name

Delete actions:
Delete a node
Delete a relationship

Update actions:
Set one or more labels on a node
Remove one or more labels from a node
Set, update, or remove properties

Node-specific actions:
Get a node by its ID
Get the degree of a node
Determine whether a node is dense
Determine whether a label is set on a node
Get the labels of a node
Get a property of a node
Get an existing node label

Relationship-specific actions:
Get a relationship by its ID
Get a property of a relationship
Get an existing relationship type

Actions that incur DB hits - 2

General actions:
Lookup a node or relationship
Find a path in a variable-length expand
Find a shortest path
Ask the count store for a value

Schema actions:
Add an index
Drop an index
Get the reference of an index
Create a constraint
Drop a constraint

Call a procedure (possibly)

Call a user-defined function (possibly)

Important: Reported as only 1 DB hit in a PROFILE.

Lazy or eager evaluation

Plan evaluation is **Eager** or **Lazy**

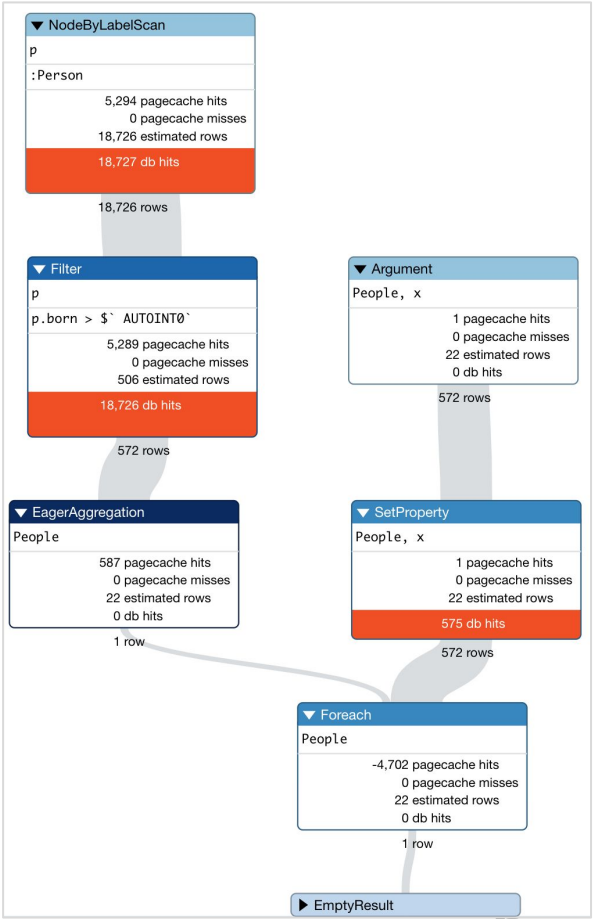
Most query plan evaluation is **lazy**:

- Operators pipe their output rows to their parent operators as soon as they are produced. Child operator may not be finished before the parent receives and processes rows.

Eager operations must complete before next step can start:

- Aggregation and sorting need to aggregate and process all their rows before they can produce output. This is not the case if an index is used.

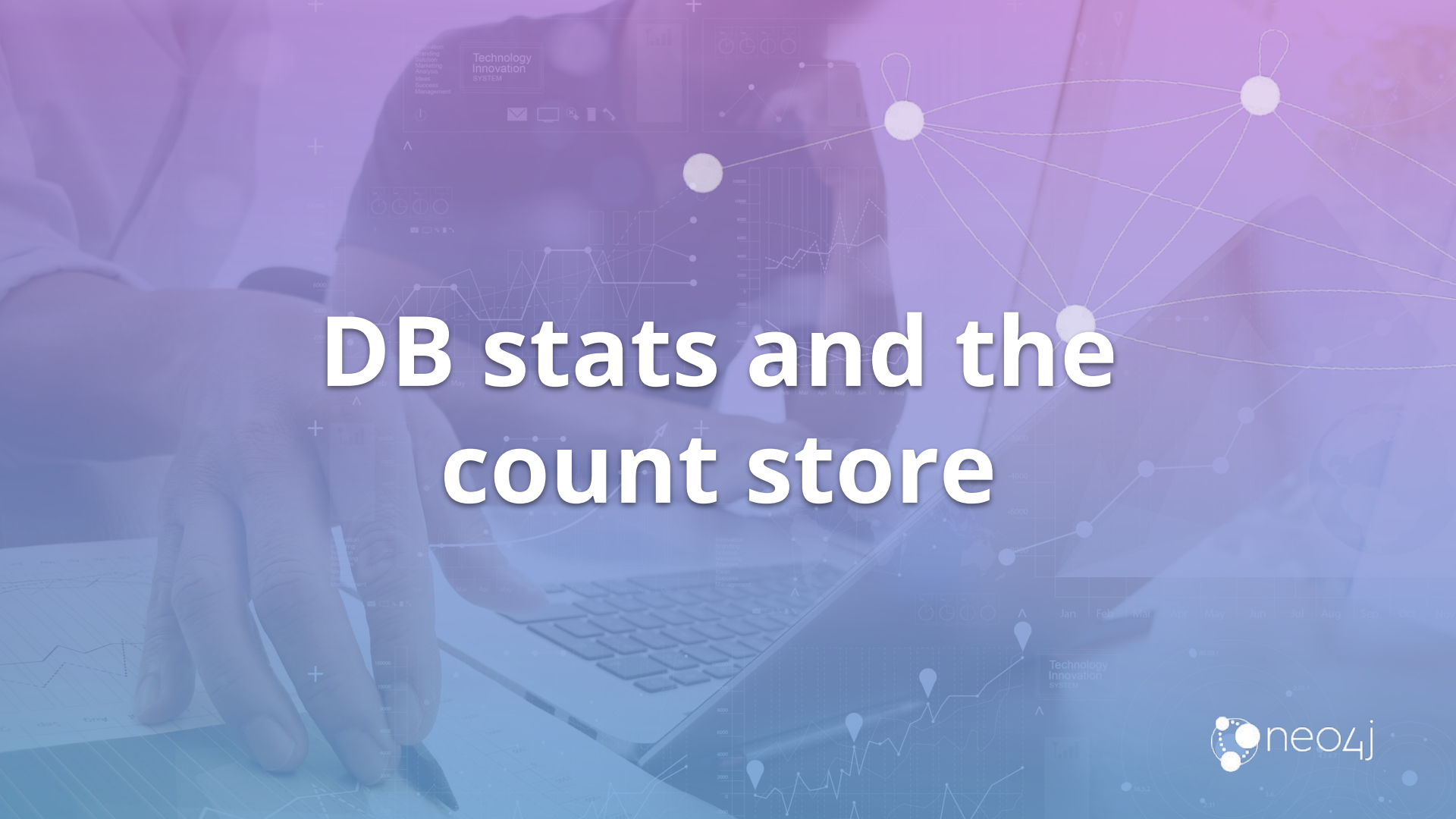
```
PROFILE MATCH (p:Person) WHERE p.born > 1990
WITH collect(p) AS people
FOREACH (x IN people | SET x.millenials = true)
```



Things to investigate in a query plan

- **AllNodesScan** - very expensive
- Plans that start with **NodeByLabelScan** with a lot of DB hits
- Large number of rows used in cartesian products and joins
- Many intermediate rows that are duplicates
- **Eager operators** (especially early in the plan; but, eager aggregation is good)
- Duplicated operations
- Property reads early in the query plan (ideally defer to later if possible)
- Over-expansion due to lack of rel-type specificity (e.g. everything is :CONNECTED_TO)

Query Plan Operations are described here:
https://neo4j.com/docs/cypher-manual/current/execution-plans/operators/#query-plan-expand-all


DB stats and the count store
neo4j

DB stats are used for query optimization

The Cypher query planner optimizes using indexes and constraints as well as DB stats collected by the graph engine during a transaction and used by the query planner:

- Count stores
- Selectivity per index or constraint value

How DB stats for indexes and constraints are collected is controlled by these configuration settings where you can enable the collection of DB stats if indexes have been updated by a certain threshold percentage:

dbms.index_sampling.background_enabled (default: true)
dbms.index_sampling.update_percentage (default: 5)

You can also manually force the collection of DB stats by executing:

```
//update DB stats for a specific index
CALL db.resampleIndex(':Person(name)')

//update DB stats for all indexes
CALL db.resampleOutdatedIndexes()
```

Neo4j count stores

- Neo4j keeps transactional database statistics that contain cardinalities of various graph entities.
- Count stores are used to determine execution plans for some queries.
- Count stores are used to select whether to use an index or not (more on indexes later).

What does the count store contain?

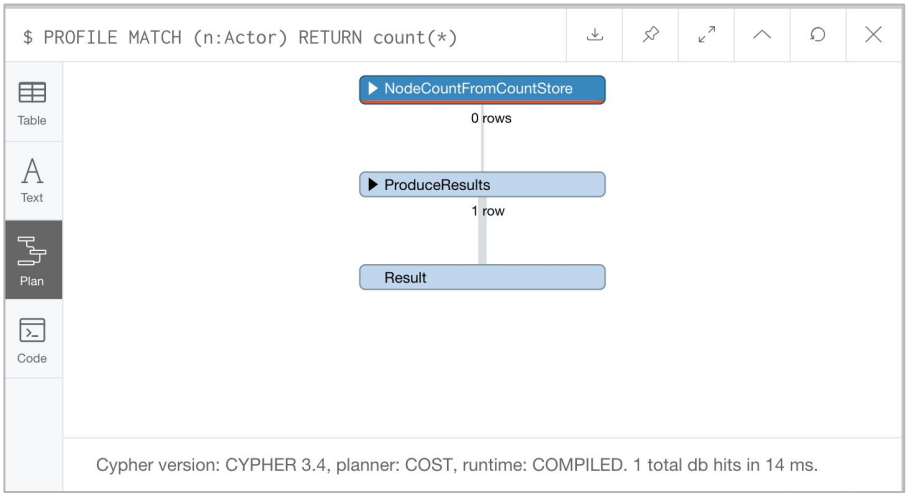
Number of nodes	(n)
Number of nodes with a specific label	(n:Label) (single label only)
Number of directed relationships	()-[]->()
Number of directed relationships of a specific type	()-[r:REL_TYPE]->()
Number of outgoing relationships of a specific type from a node with the label	(n:Label)-[r:REL_TYPE]->()
Number of incoming relationships of a specific type to a node with the label	(n:Label)<-[r:REL_TYPE]-()

Important: Relationship counts with labels on the start and end nodes are not counted.

Using the count store - node labels

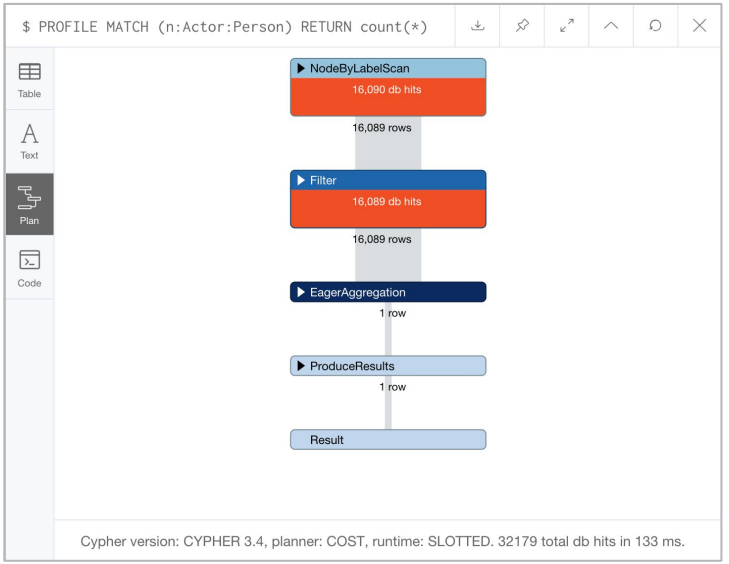
Count store used - single label node:

```
PROFILE MATCH (n:Actor)
RETURN count(*)
```



Count store not used - multiple label node:

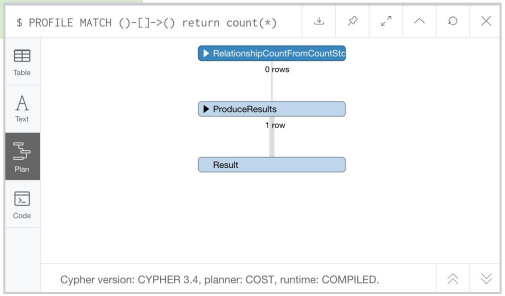
```
PROFILE MATCH (n:Actor:Person)
RETURN count(*)
```



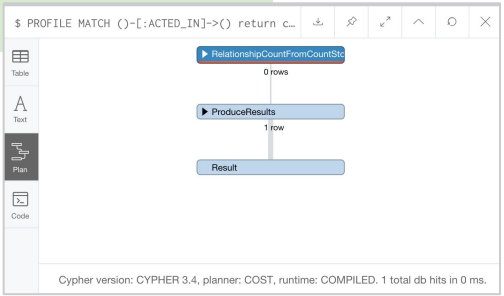
Using the count store - relationship direction

Count store used - directed relationship specified:

```
PROFILE MATCH ()-[]->()
RETURN count(*)
```

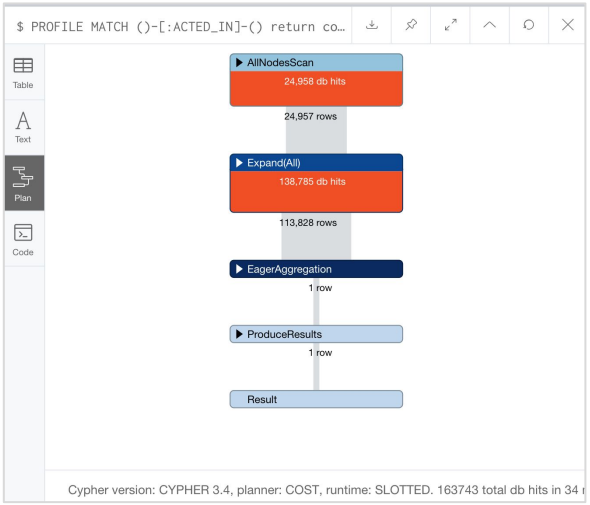



```
PROFILE MATCH ()-[:ACTED_IN]->()
RETURN count(*)
```



Count store <u>not</u> used - undirected relationship specified:

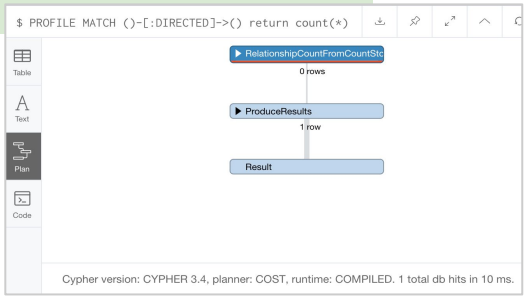
```
PROFILE MATCH ()-[:ACTED_IN]-()
RETURN count(*)
```



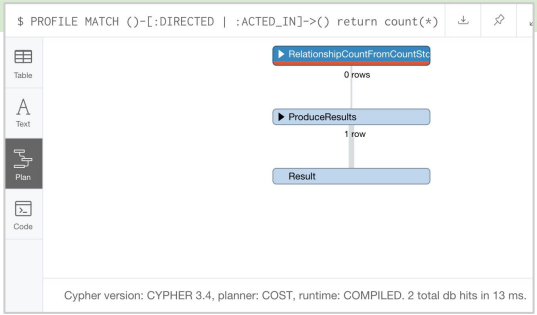
Using the count store - relationship types

Count store used - directed relationship specified with type(s) specified:

```
PROFILE MATCH ()-[:DIRECTED]->()
RETURN count(*)
```

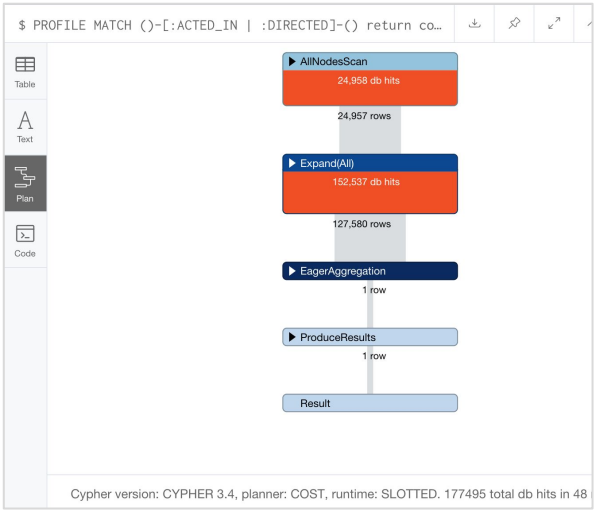



```
PROFILE MATCH ()-[:DIRECTED | :ACTED_IN]->()
RETURN count(*)
```



Count store <u>not</u> used - undirected relationship specified with type(s) specified:

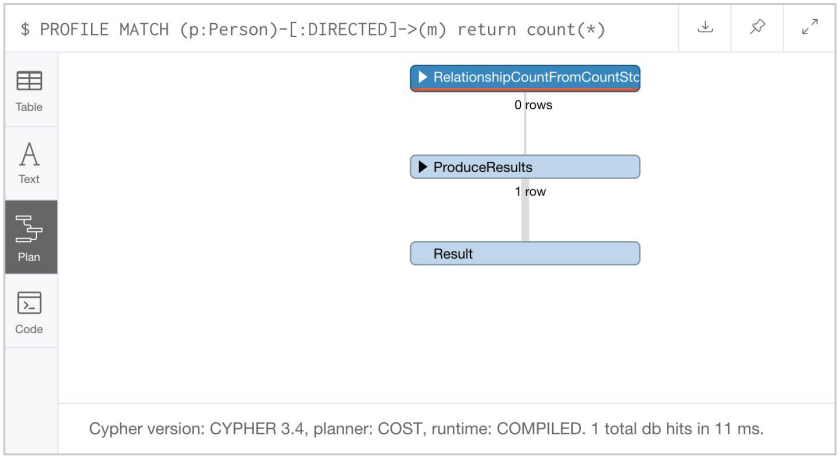
```
PROFILE MATCH ()-[:DIRECTED | :ACTED_IN]-()
RETURN count(*)
```



Using the count store - outgoing relationship types with one node label

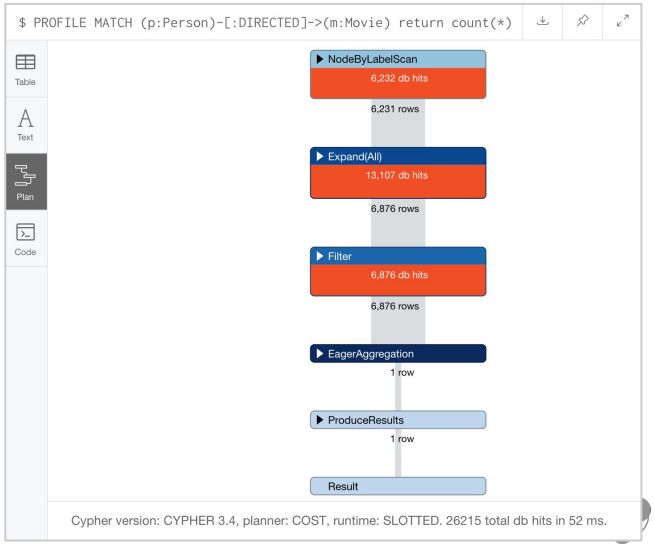
Count store used - directed relationship specified with type(s) specified and a singly-labeled node on one end of the relationship:

```
PROFILE MATCH (p:Person)-[:DIRECTED]->(m)
RETURN count(*)
```



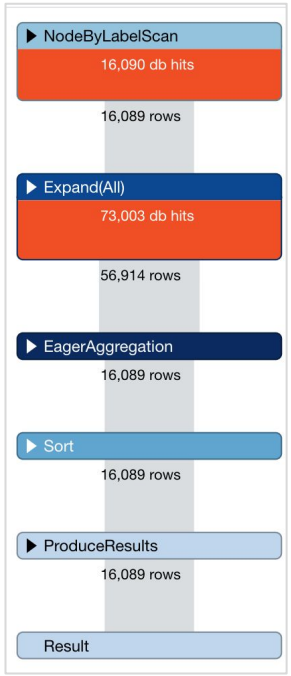
Count store not used - directed relationship specified with type(s) specified and a singly-labeled node on both ends of the relationship:

```
PROFILE MATCH (p:Person)-[:DIRECTED]->(m:Movie)
RETURN count(*)
```



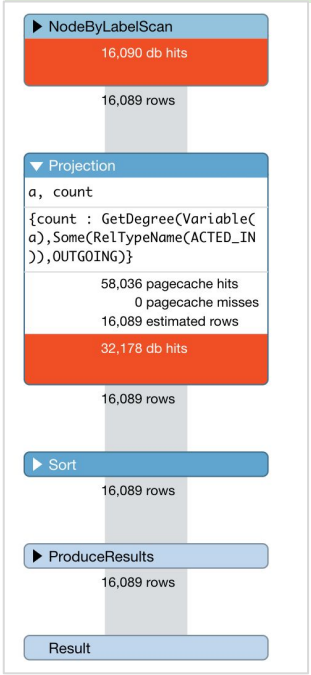
size() is alternative to count store

```
PROFILE MATCH (a:Actor)-[:ACTED_IN]->()
RETURN a, count(*) AS count ORDER BY count DESC
```



89,093 DB hits

```
PROFILE MATCH (a:Actor)
RETURN a, size((a)-[:ACTED_IN]->()) AS count
  ORDER BY count DESC
```



GetDegree is used by size() if a node has more than 50 relationships.

48,268 DB hits

neo4j

Retrieving DB stats - 1

```
CALL db.stats.retrieve('GRAPH COUNTS' | 'TOKENS' | 'QUERIES' | 'META')
```

- GRAPH COUNTS are the count stores for the graph and the uniqueness constraints defined for the graph. This is very useful for query planning.
- TOKENS are the names used for node labels, relationship types, property names.
- QUERIES are the details of all queries run against the graph.
- META are the configuration details for the database.

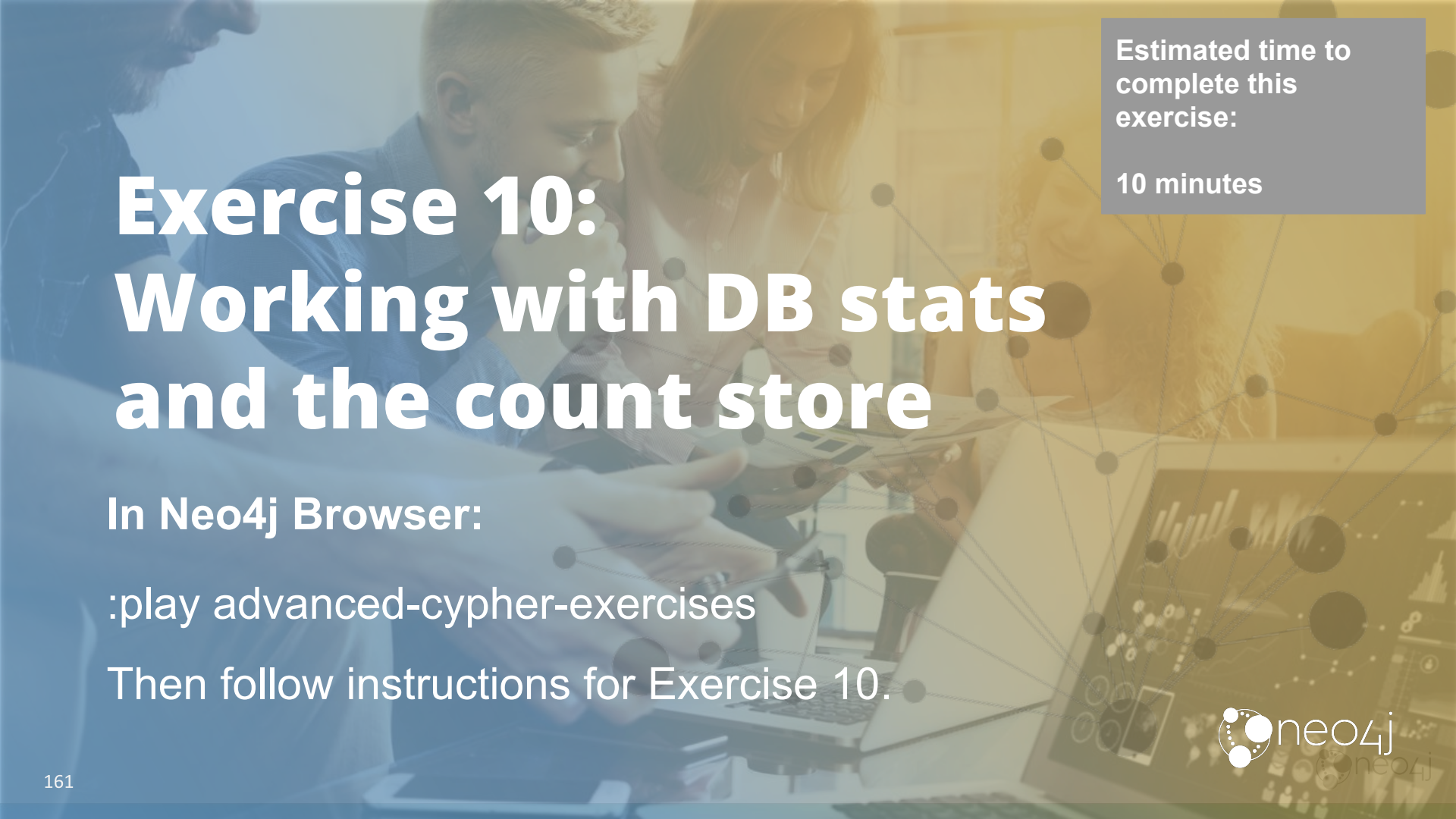
Retrieving DB stats - 2

```
$ CALL apoc.meta.stats() YIELD labelCount, relTypeCount, propertyKeyCount, nodeCount, relCount, labels, relTypes, stats return labelCount, relTy…
```

Table | Text | Code

labelCount	relTypeCount	propertyKeyCount	nodeCount	relCount	labels	relTypes	stats
12	6	16	24994	81094	{ "Director": 3099, "Movie": 6231, "Genre": 35, "Person": 18726, "Actor": 16089 }	{ "()-[:DIRECTED]->(:Movie)": 6876, "()-[:DIRECTED]->()": 6876, "(:Person)-[:DIRECTED]->()": 6876, "()-[:IS_GENRE]->()": 17304, "()-[:IS_GENRE]->(:Genre)": 17304, "(:Actor)-[:ACTED_IN]->()": 56914, "(:Actor)-[:DIRECTED]->()": 1659, "()-[:ACTED_IN]->	{ "relTypeCount": 6, "propertyKeyCount": 16, "labelCount": 12, "nodeCount": 24994, "relCount": 81094, "labels": { "Director": 3099, "Movie": 6231, "Genre": 35, "Person": 18726, "Actor": 16089 }, "relTypes": { "()-[:DIRECTED]->(:Movie)": 6876,

Started streaming 1 records after 1 ms and completed after 6 ms.



Estimated time to complete this exercise:

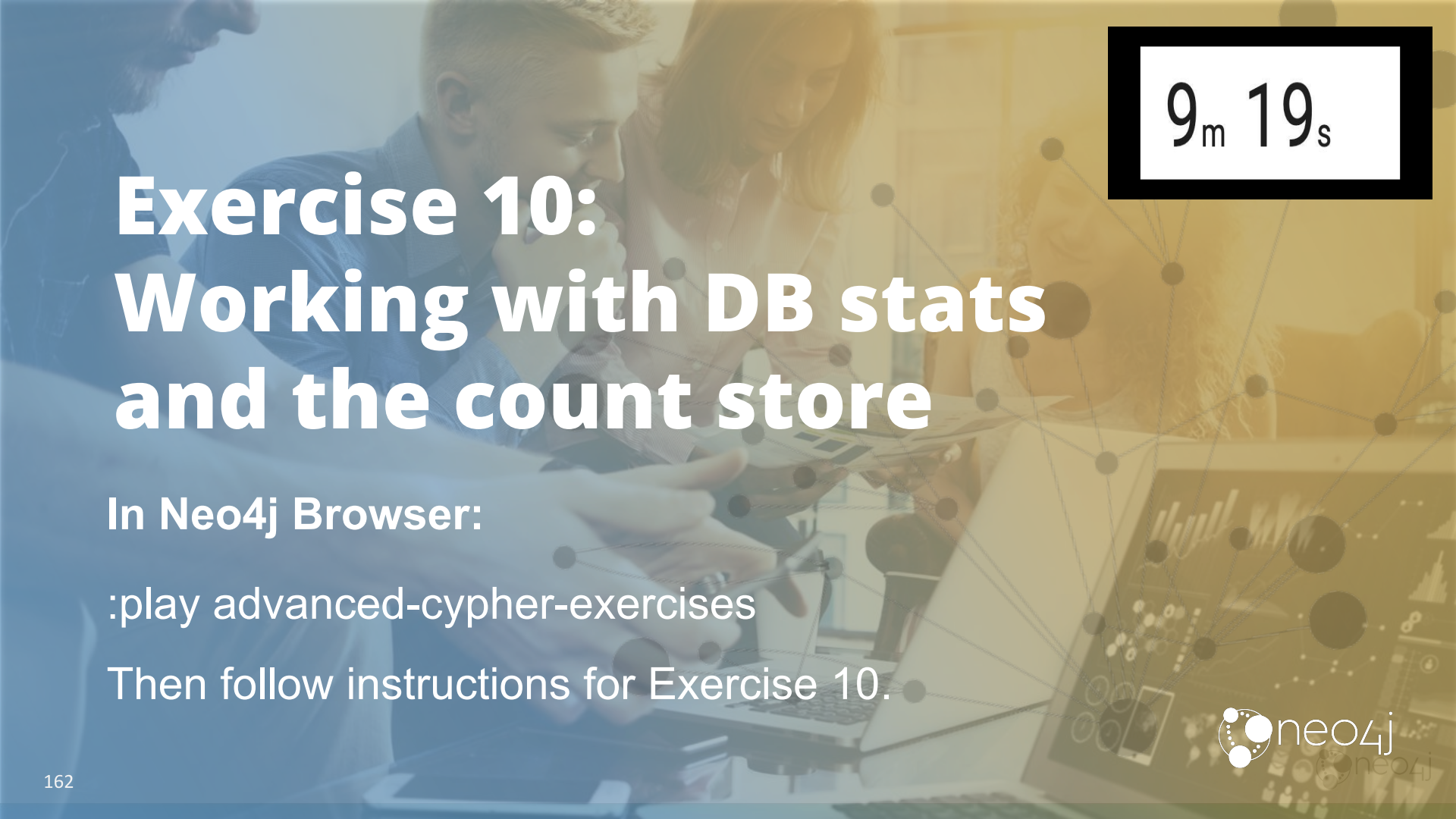
10 minutes

Exercise 10: Working with DB stats and the count store

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 10.

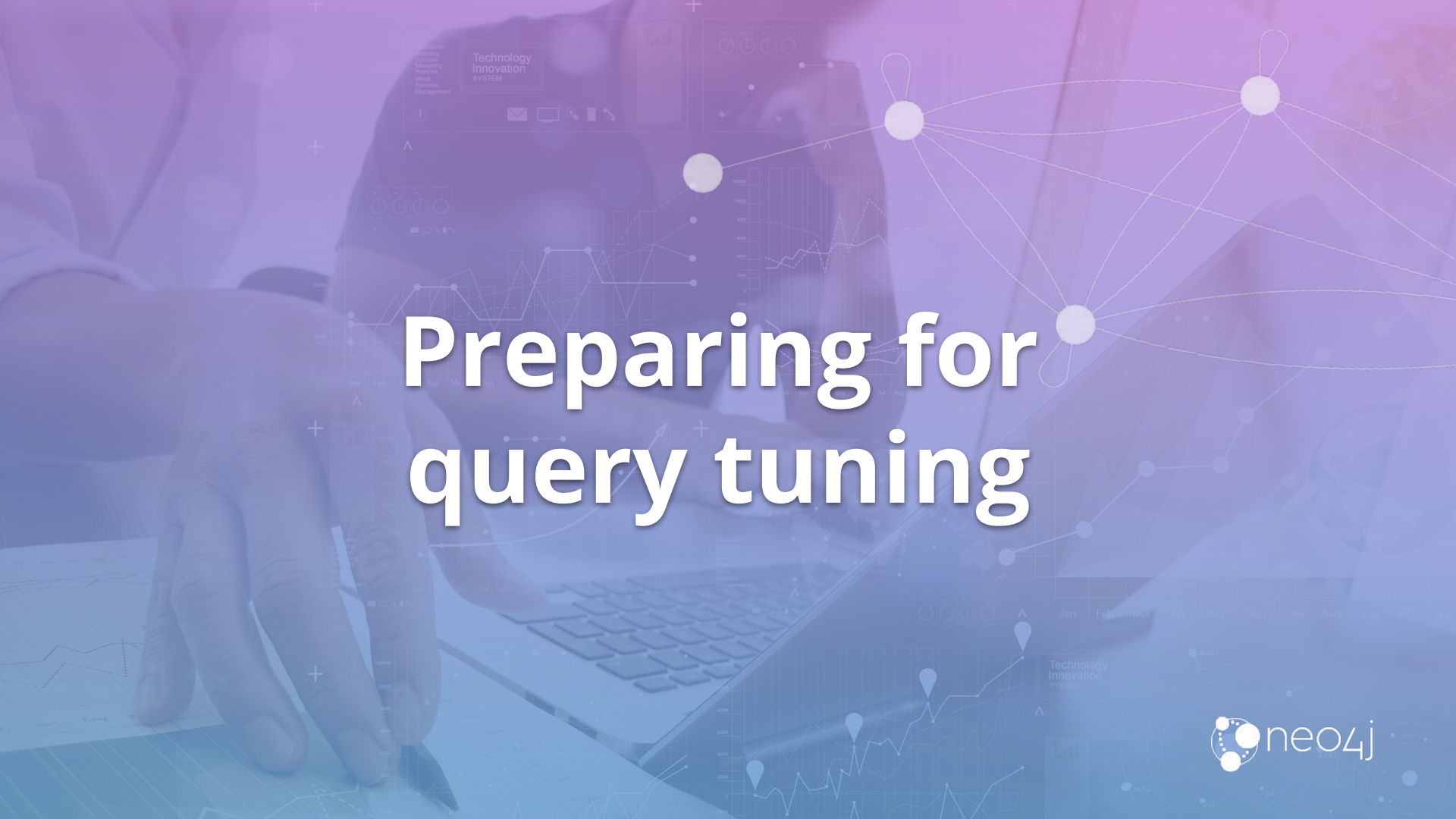


Exercise 10: Working with DB stats and the count store

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 10.



Preparing for query tuning

neo4j

Steps for query tuning

1. Ensure hardware and configuration ready for serious query tuning.
2. Ensure queries are correct by examining the query plan. (EXPLAIN)
3. Run queries to ensure the Page Cache is warmed up.
4. **Test**: PROFILE query on representative dataset.
5. **Measure**: Examine Query Plan & note hotspots.
    a. Number of intermediate rows
    b. DB hits
    c. Elapsed time
6. **Change**:
    a. Modify the model or change the query.
    b. Tune the query.
        i. *Focus on hotspots*
        ii. *Build query up*
        iii. *Apply best practices*

Preparations for query tuning - 1

- Hardware
  - SSD (or enough IOPS)
  - Plenty of RAM
  - For concurrent operations, multiple CPU cores
  - On Linux, configure disk scheduler to **noop** or **deadline**, mount the database volume with **noatime**.
- Be prepared to monitor system during query testing for bottlenecks::
  - IO waits
  - TOP for CPU and memory usage
    - *htop*
    - *iotop*

Preparations for query tuning - 2

- Configuration
  - Use the latest GA of Neo4j.
  - Set dbms.memory.pagecache.size=4G or to the size of the graph:
    - *ls -lt <graph-name>/neostore.*.db*
    - total size from *:sysinfo*
  - Set the heap from 8G to 16G, depending on the RAM of the machine:
    - *dbms.memory.heap.initial_size=8G*
    - *dbms.memory.heap.max_size=16G*

Page Cache

- The Neo4j database maps the data files from disk to the Page Cache.
- Data not in the Page Cache must be read from disk.

Best practice: For best performance, size the Page Cache to match the size of the database.

Query Cache

- Query plans are stored in a Query Cache for reuse.
  - Hash value based on query (case-sensitive)
  - Queries with different literal values are counted as separate queries.
    - ***Best practice****: Use parameters instead of literals.*
- Query plans are stale if graph has changed too much, or Query Cache is full.
- First query execution will always be slower than second for the same query due to cache misses, query planning, etc.

Warming up the page cache

Application in production or during a query tuning exercise should be ready for queries by ensuring the page cache is warmed up (in memory):
Here are some ways to warm up the page cache:

- MATCH (n) RETURN max(id(n))
- MATCH ()-[rel]->() RETURN max(id(rel))
  or
- CALL apoc.warmup.run() //nodes and relationships
- CALL apoc.warmup.run(true) // include properties
- CALL apoc.warmup.run(true,true) // include large strings and arrays
- CALL apoc.warmup.run(true,true,true) // include indexes

Page cache info

```
$ :sysinfo
```

Store Sizes	
Count Store	2.59 KiB
Label Store	16.02 KiB
Index Store	2.69 MiB
Schema Store	8.01 KiB
Array Store	7.72 MiB
Logical Log	97.23 MiB
Node Store	375.93 KiB
Property Store	6.65 MiB
Relationship Store	5.12 MiB
String Store	40.01 KiB
Total Store Size	119.84 MiB

ID Allocation	
Node ID	25004
Property ID	96044
Relationship ID	81114
Relationship Type ID	9

Page Cache	
Faults	2524
Evictions	0
File Mappings	55
Bytes Read	15786195
Flushes	60
Eviction Exceptions	0
File Unmappings	33
Bytes Written	16372540
Hit Ratio	99.93%
Usage Ratio	3.87%

Transactions	
Last Tx Id	94
Current	1
Peak	4
Opened	12446
Committed	9647

Warming up query cache

- Run all queries you will be measuring.
- Best practice is to use parameters in any query that would have used a string constant for different values at runtime.

Best practice: Use query parameters

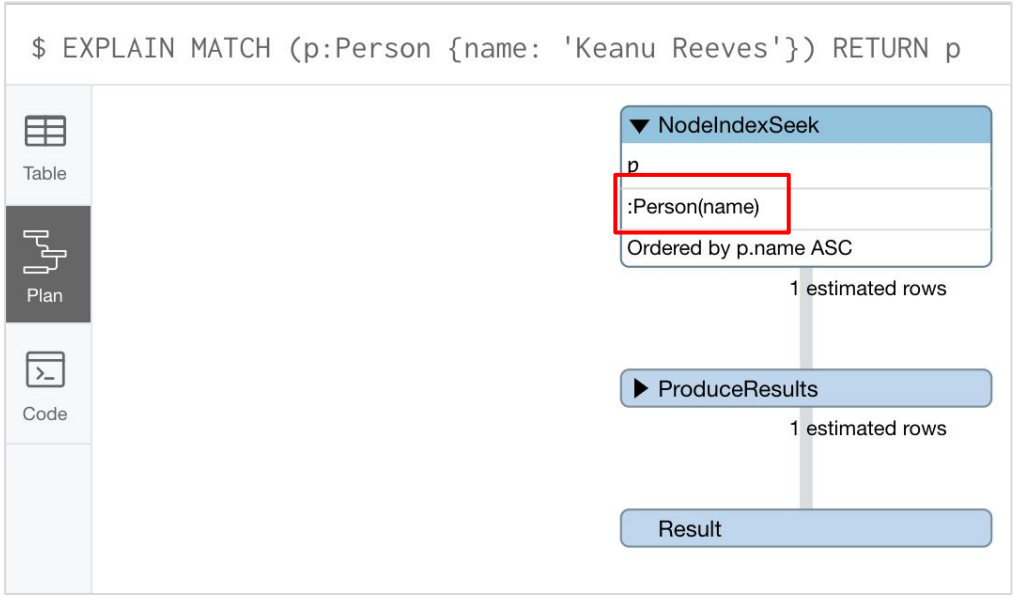
New query plan required	Query plan re-used
MATCH …. WHERE n.name = 'John' RETURN …	Parameters: { name : "John" } MATCH …. WHERE n.name = $name RETURN …
MATCH …. WHERE n.name = 'Ellis' RETURN …	Parameters: { name : "Ellis" } MATCH …. WHERE n.name = $name RETURN …

Query performance components

1. Query planning
    - Can be reduced if the Query Cache holds query plan.
2. Query execution
    - DB hits can correlate to performance, but the best measure is execution time (with a warm cache).
    - Not all DB hits are created equal and can depend on what they access in the graph.
    - Streaming results back to client
    - Result of query; carefully consider whether large result sets are necessary, especially if streamed over a network.

Prepare for tuning: Ensure indexes are used

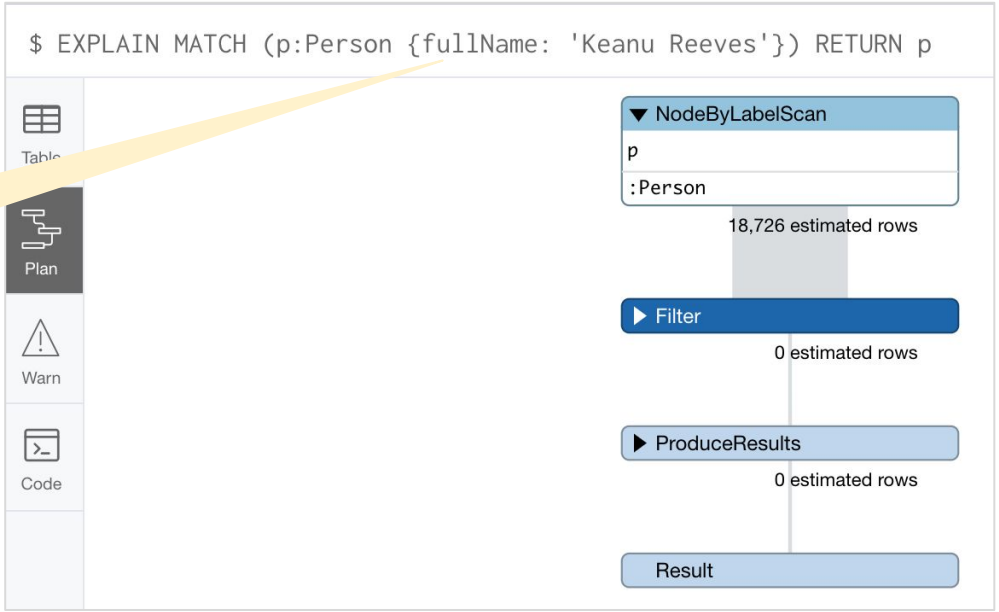
Use EXPLAIN on all queries that you expect to use an index and confirm that the query planner will use the correct index and that the name (case-sensitive) of the property matches what is defined for the index.



Prepare for tuning: Ensure properties and node labels are correct

Use EXPLAIN on all queries first to ensure that any (case-sensitive) names you reference in the query actually exist in the graph, otherwise a full label scan and post-filter will occur to find them.

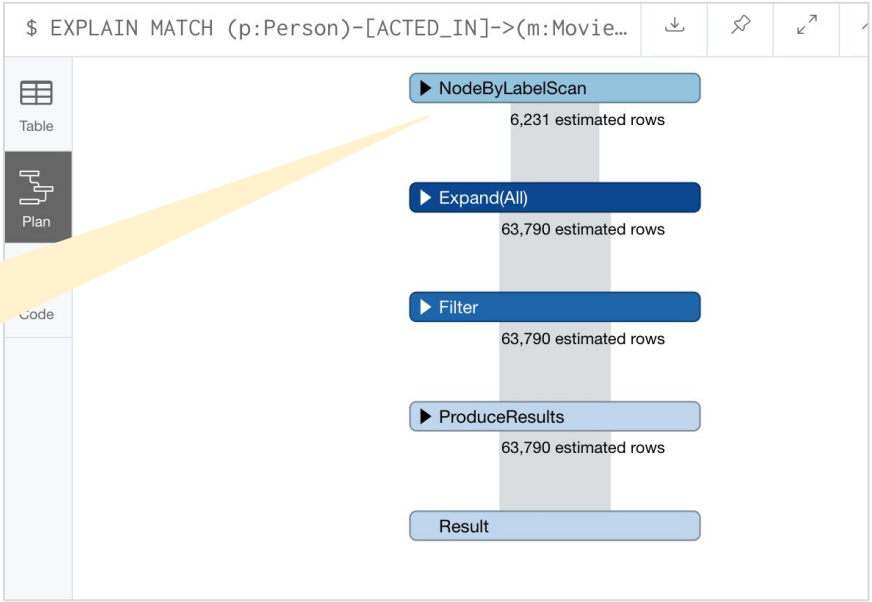
Property name is wrong.

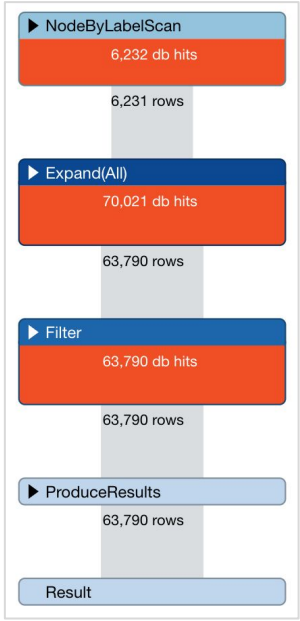


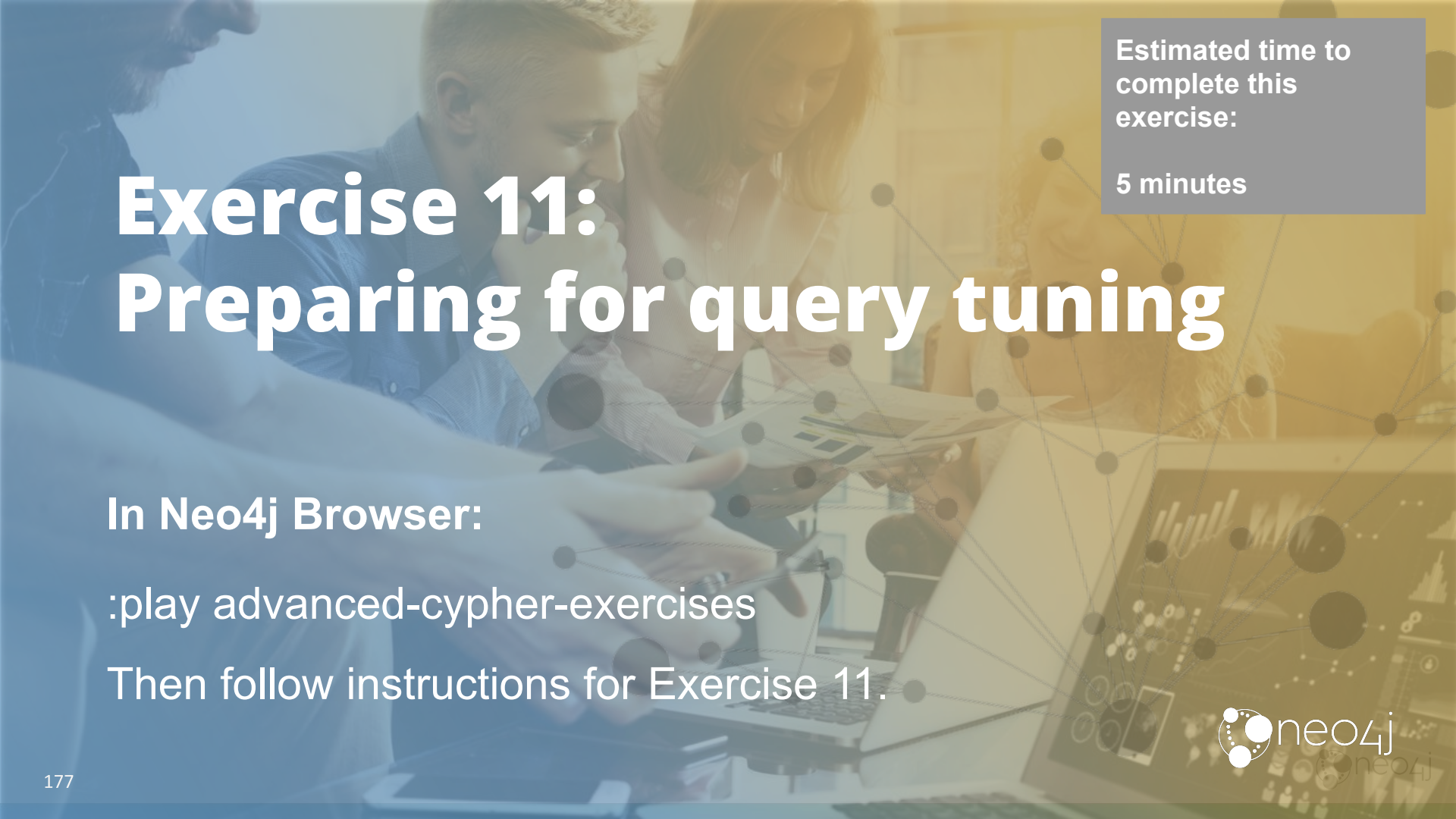
Prepare for tuning: Ensure relationship types are properly specified

Use EXPLAIN on all queries first to ensure that any relationship types are specified with the ":" in them, otherwise, the planner interprets it as a variable and traverses all relationships.

ACTED_IN should be :ACTED_IN







Estimated time to complete this exercise:

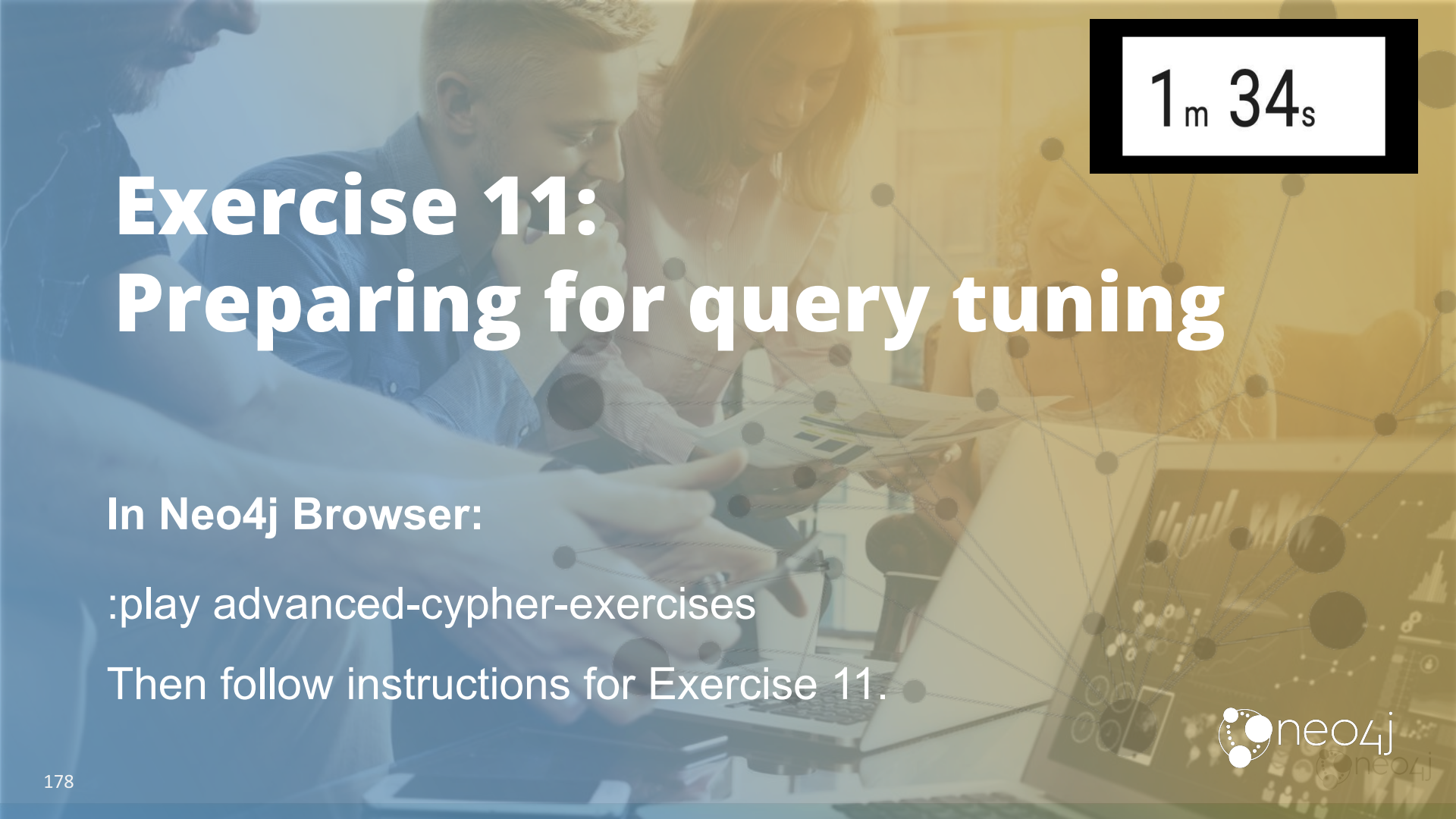
5 minutes

Exercise 11: Preparing for query tuning

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 11.



Exercise 11: Preparing for query tuning

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 11.

Optimizing queries

neo4j

Query tuning goal: reduce number of rows

Input rows into an operator:

- Number of times the operator is executed.
- Increases the number of DB hits and the output rows generated.

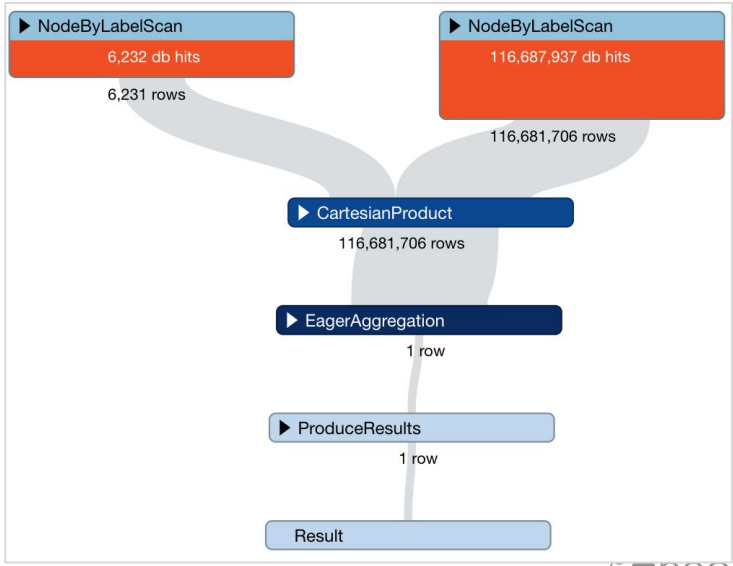
Reduce output rows to reduce input rows for next operator:

- Get distinct results, not number of paths.
- Use label or index with the higher selectivity.

Important: You should try to minimize the use of cartesian products, except for creating relationships.

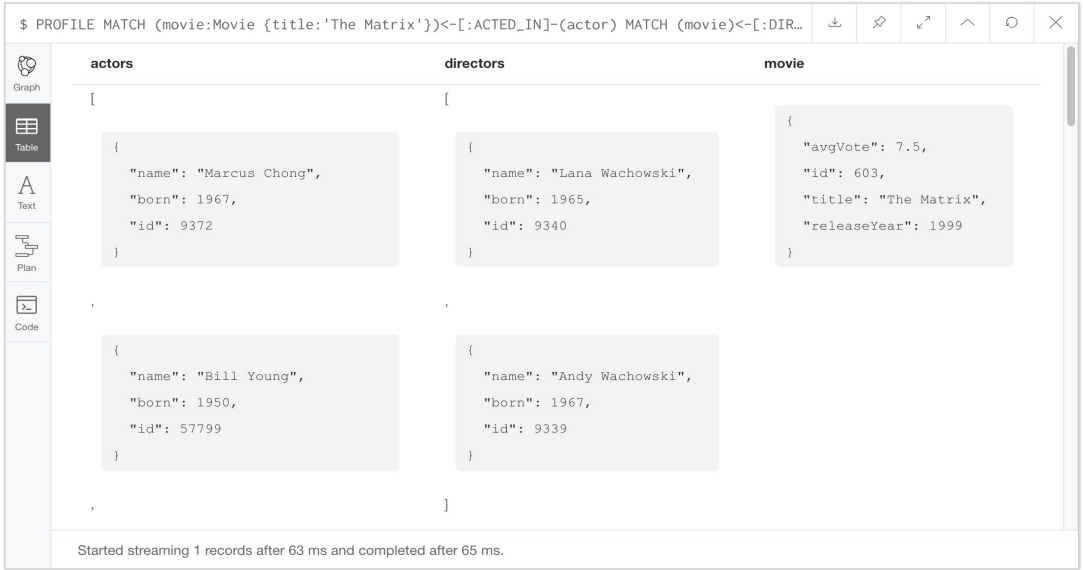
```
// do NOT do this
PROFILE MATCH (m:Movie), (p:Person)
RETURN count(DISTINCT m), count(DISTINCT p)
```

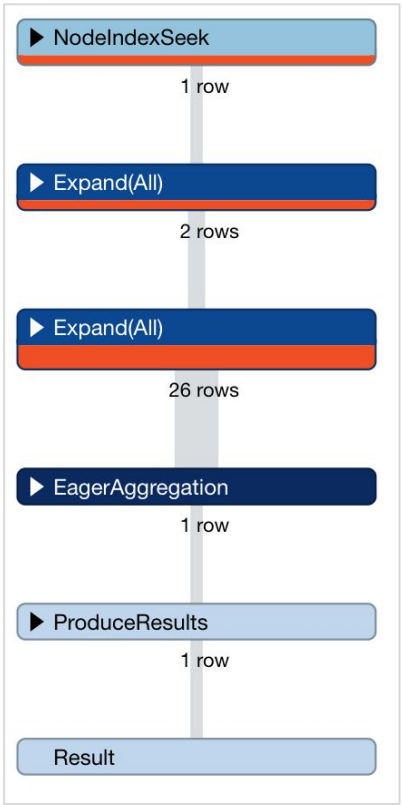
(116,694,169 DB hits)



Reducing rows by aggregating data - 1

```
PROFILE MATCH (movie:Movie {title:'The Matrix'})<-[:ACTED_IN]-(actor)
MATCH (movie)<-[:DIRECTED]-(director)
RETURN collect(DISTINCT actor) AS actors, collect(DISTINCT director)
AS directors, movie
```





Reducing rows by aggregating data - 2

```
PROFILE MATCH (movie:Movie {title:'The Matrix'})<-[:ACTED_IN]-(actor)
WITH movie, collect(actor) AS actors
MATCH (movie)<-[:DIRECTED]-(director)
WITH movie, actors, collect(director) AS directors
RETURN actors, movie, directors
```

actors

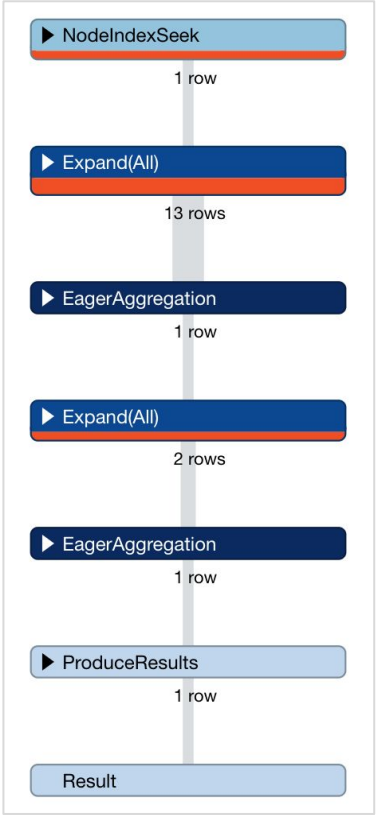
```
[
  {
    "name": "Keanu Reeves",
    "born": 1964,
    "id": 6384
  }
,
  {
    "name": "Marcus Chong",
    "born": 1967,
    "id": 9372
  }
,
```

movie

```
{
  "avgVote": 7.5,
  "id": 603,
  "title": "The Matrix",
  "releaseYear": 1999,
  "genres": [
    "Action",
    "Adventure",
    "Science Fiction",
    "Thriller"
  ]
}
```

directors

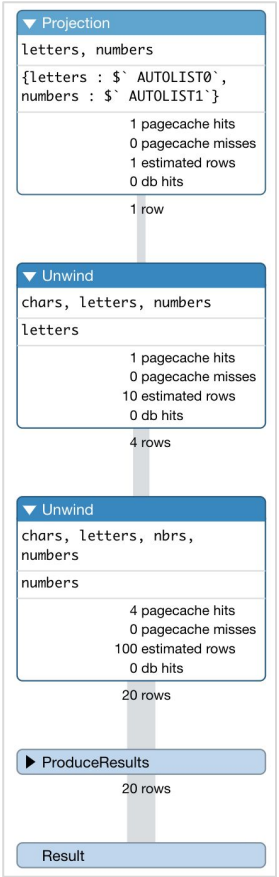
```
[
  {
    "name": "Andy Wachowski",
    "born": 1967,
    "id": 9339
  }
,
  {
    "name": "Lana Wachowski",
    "born": 1965,
    "id": 9340
  }
]
```



Query tip: Minimize multiple UNWINDS

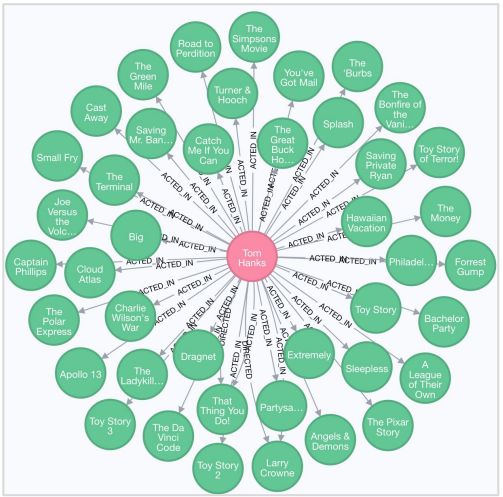
```
PROFILE WITH ['a','b','c','d'] AS letters, [1,2,3,4,5] AS numbers
UNWIND letters AS chars
UNWIND numbers AS nbrs
RETURN chars,nbrs
```

chars	nbrs
"a"	1
"a"	2
"a"	3
"a"	4
"a"	5
"b"	1
"b"	2
"b"	3
"b"	4
"b"	5
"c"	1
"c"	2
"c"	3
"c"	4
"c"	5
"d"	1

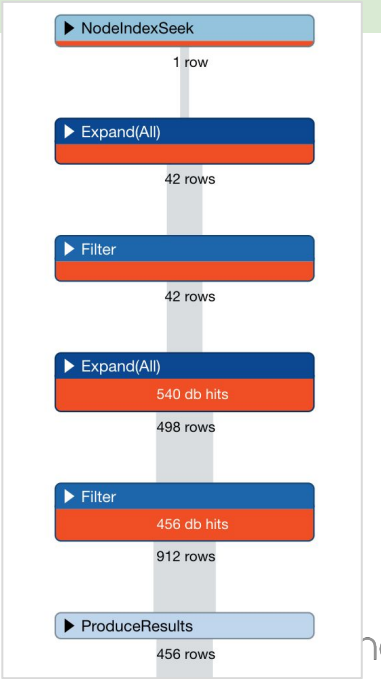


Fanout and cardinality

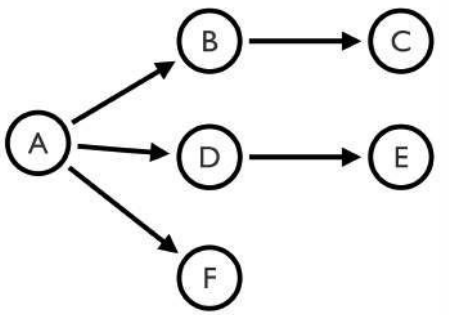
```
// returns 44 nodes and 43 relationships
MATCH  (p:Person)-[:ACTED_IN]->(m:Movie)
          WHERE p.name = "Tom Hanks"
RETURN p,m
```




```
// returns 587 nodes and 353 relationships
PROFILE MATCH  (p:Person)-[:ACTED_IN]->(m:Movie)
          <-[:ACTED_IN]- (p2:Person)
WHERE p.name = "Tom Hanks"
RETURN p, m, p2
```



With large fanouts, limit multiple MATCHes



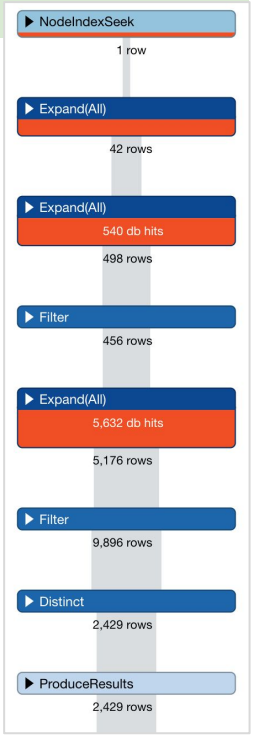

```
MATCH (a)
OPTIONAL MATCH (a)-->(b)-->(c)
OPTIONAL MATCH (a)-->(d)-->(e)
OPTIONAL MATCH (a)-->(f)
...
```

- Multiple MATCHes can cause a cartesian product.
- Each MATCH results in a stream of paths.
- For example, with a single (a) node, the (a)->(b)->(c) path produces 10 entries
  - Then the (a)->(d)->(e) path produces 10*10 entries
  - Then the (a)->(f) path produces 100 * 10 entries
- Results in 10 * 10 * 10 = 1000 intermediate rows per (a) node.

Control cardinalities with DISTINCT - 1

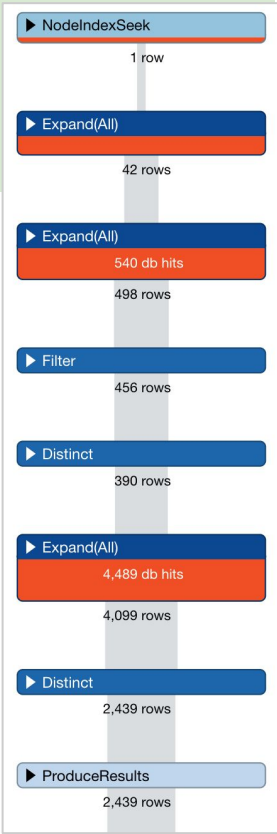
```
PROFILE MATCH  (:Person {name: 'Tom
Hanks'})-[:ACTED_IN]->()
            <-[:ACTED_IN]-()-[:ACTED_IN]->(m)
RETURN  DISTINCT m
```

- 20,297 rows processed
- 6,217 DB hits



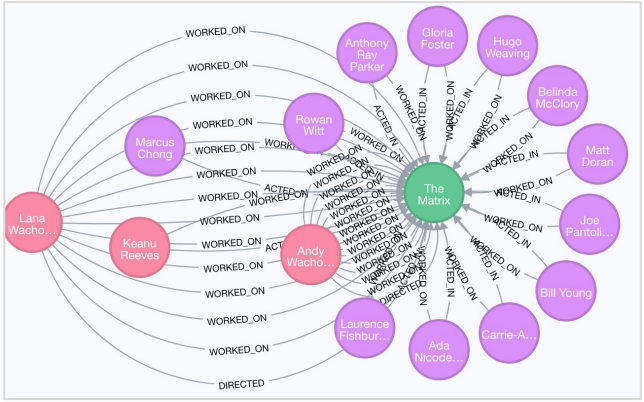

```
PROFILE MATCH  (:Person {name: 'Tom
Hanks'})-[:ACTED_IN]->()
            <-[:ACTED_IN]-(p)
WITH DISTINCT p
MATCH (p)-[:ACTED_IN]->(m)
RETURN  DISTINCT m
```

- 10,363 rows processed
- 5,074 DB hits



Control cardinalities with DISTINCT - 2

```
// Do not do this,  movie will have 5 rows (one per actor)
PROFILE MATCH (movie:Movie {title:'The
Matrix'})<-[:ACTED_IN]-(actor)
CREATE (actor)-[:WORKED_ON {role:'actor'}]->(movie)
WITH movie
MATCH (movie)<-[:DIRECTED]-(director)
CREATE (director)-[:WORKED_ON {role:'director'}]->(movie)
```

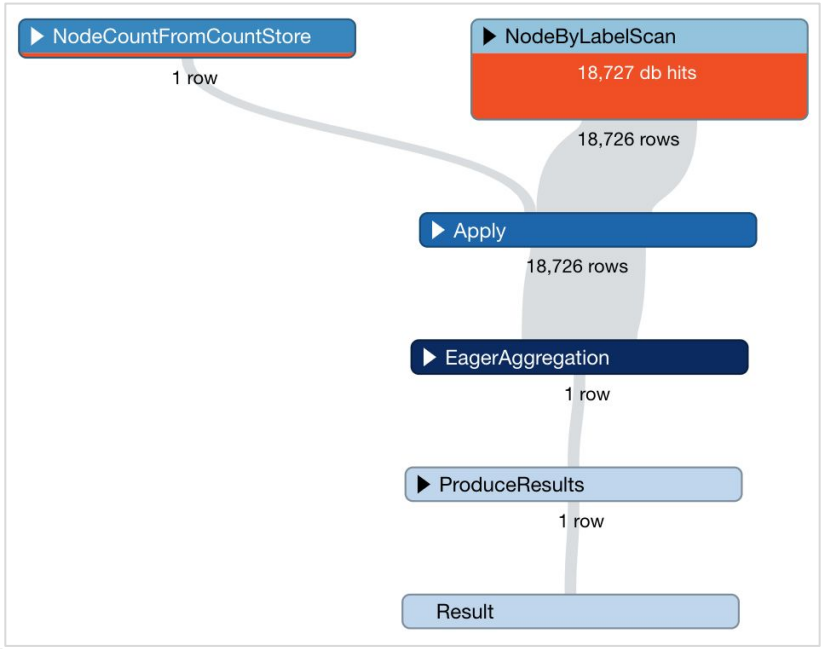



```
// Do this
PROFILE MATCH (movie:Movie)<-[:ACTED_IN]-(actor)
CREATE (actor)-[:WORKED_ON {role:'actor'}]->(movie)
WITH DISTINCT movie
MATCH (movie)<-[:DIRECTED]-(director)
CREATE (director)-[:WORKED_ON {role:'director'}]->(movie)
```

Query tip: Use the count store

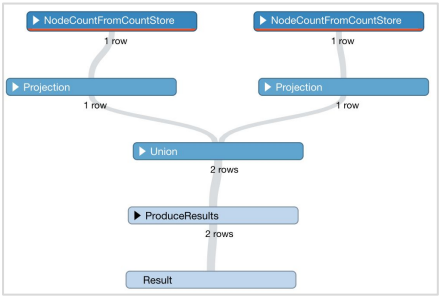
```
// Count store cannot be used twice
PROFILE MATCH (m:Movie) WITH count(*) AS movies
MATCH (p:Person) RETURN movies, count(*) AS people
```

(18,728 DB hits)




```
// Force the use of the count store for each query
PROFILE
MATCH (m:Movie) RETURN {movies: count(*)} AS c
UNION ALL
MATCH (p:Person) RETURN {people: count(*)} AS c
```

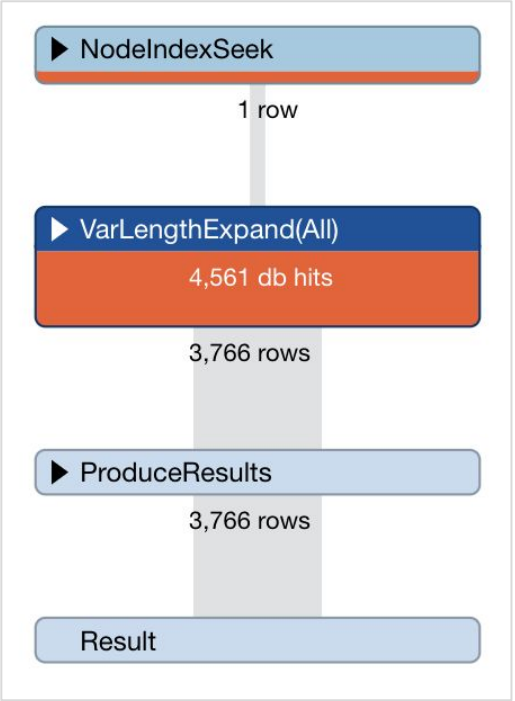
(2 DB hits)



Query tip: Use DISTINCT nodes in variable-length queries - 1

```
PROFILE
MATCH (:Person{name:'Keanu Reeves'})-[:ACTED_IN*..3]-(movie)
RETURN movie
```

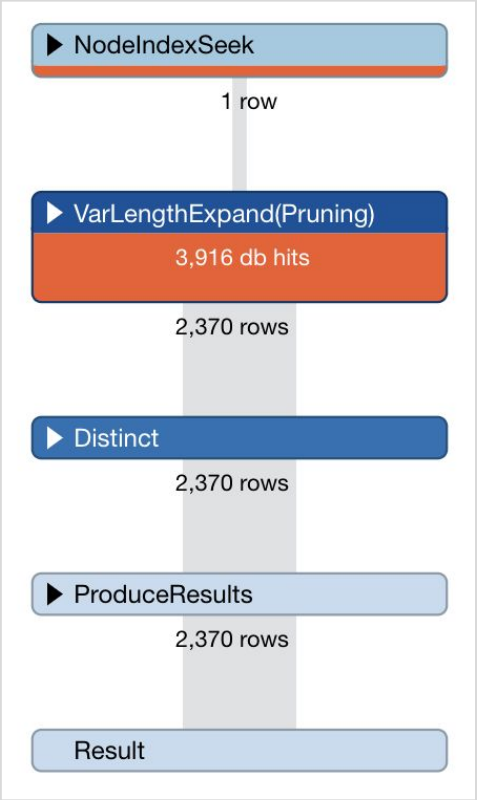
3766 rows returned, some of which are duplicates



Query tip: Use DISTINCT nodes in variable-length queries - 2

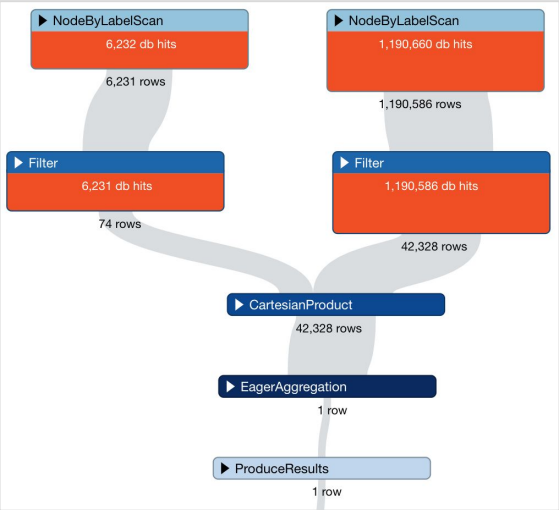
```
PROFILE
MATCH (:Person{name:'Keanu Reeves'})-[:ACTED_IN*..3]-(movie)
WITH DISTINCT movie
RETURN movie
```

2370 rows returned, with no duplicates

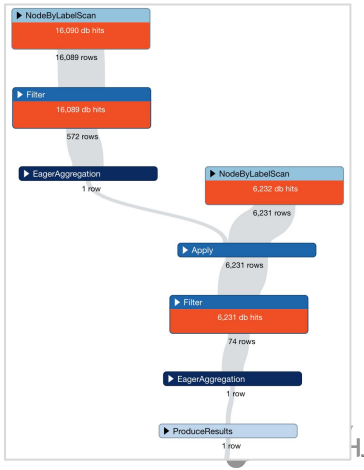


Query tip: Eliminate cartesian products

```
// Do  not do this
PROFILE MATCH (a:Actor), (m:Movie)
WHERE m.releaseYear = 1990 AND a.born
> 1990
RETURN collect(DISTINCT a) AS actors,
collect(DISTINCT m) AS movies
```




```
// Do this
PROFILE MATCH (a:Actor)
WHERE a.born > 1990
WITH collect(a) AS actors
MATCH (m:Movie)
WHERE m.releaseYear = 1990
WITH collect(m) AS movies, actors
RETURN actors, movies
```



Pattern comprehension - 1

Pattern comprehension is a Cypher expression to create a list based on matchings of a pattern:

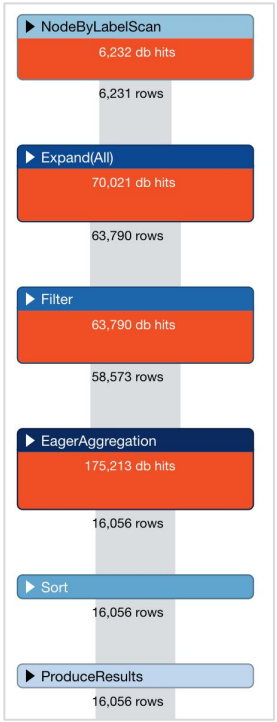
A pattern comprehension:

- Matches the specified pattern just like an optional MATCH clause.
  - Can introduce new variables for nodes, relationships, paths
- Has predicates just like a normal WHERE clause.
- Yields the defined projection.

```
MATCH (a:Person)
RETURN DISTINCT a.name AS actor,
[ (a:Person)-->(b) WHERE b:Movie | b.title] AS titles ORDER BY actor
```

Pattern comprehension - 2

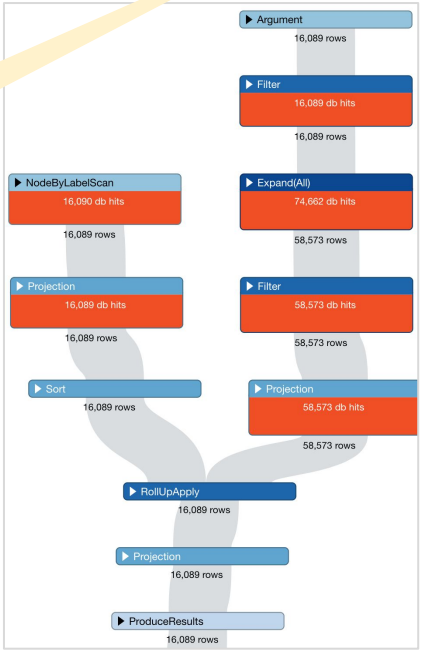
```
PROFILE MATCH (a:Actor)-->(b:Movie)
RETURN a.name AS actor,
collect(DISTINCT b.title) AS titles ORDER BY actor
```



315, 256 DB hits

```
PROFILE MATCH (a:Actor)
RETURN a.name AS actor,
[ (a)-->(b:Movie) | b.title] AS titles
ORDER BY actor
```

DISTINCT no longer required!



240,076 DB hits

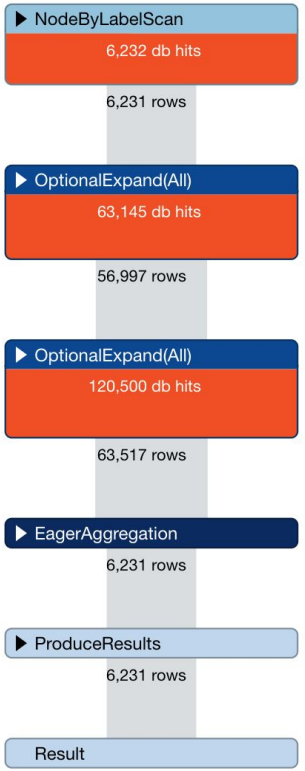
neo4j

Query tip: Aggregate early - 1

Here is an example where we aggregate in the RETURN clause:

```
PROFILE MATCH (movie:Movie)
OPTIONAL MATCH (movie)<-[:ACTED_IN]-(actor)
OPTIONAL MATCH (movie)<-[:DIRECTED]-(director)
RETURN movie, collect(DISTINCT actor) AS actors,
       collect(DISTINCT director) AS directors
```

189, 877 DB hits

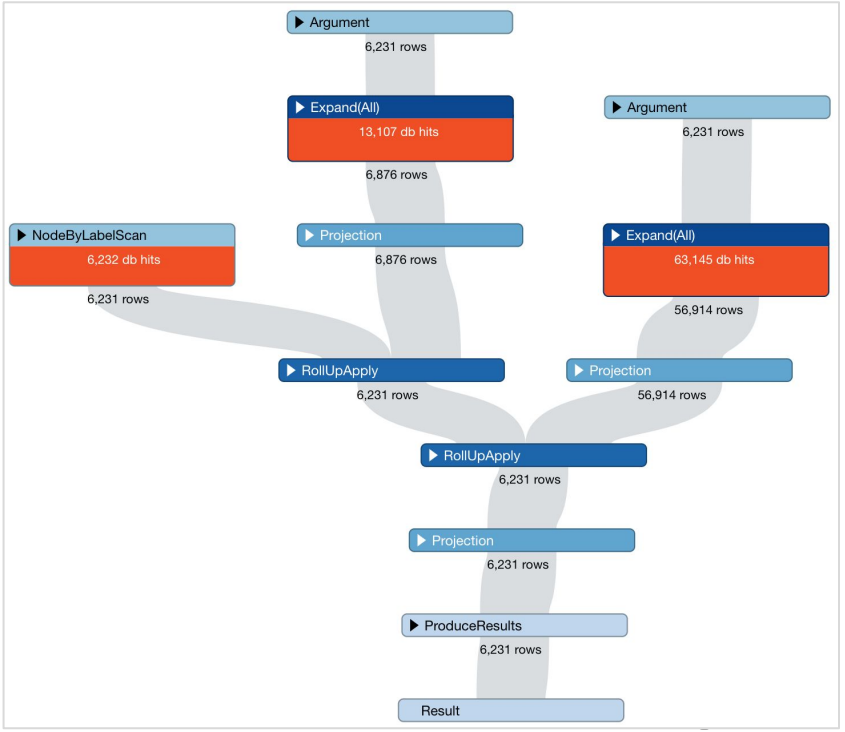


Query tip: Aggregate early - 2

Here we use pattern comprehension to aggregate earlier:

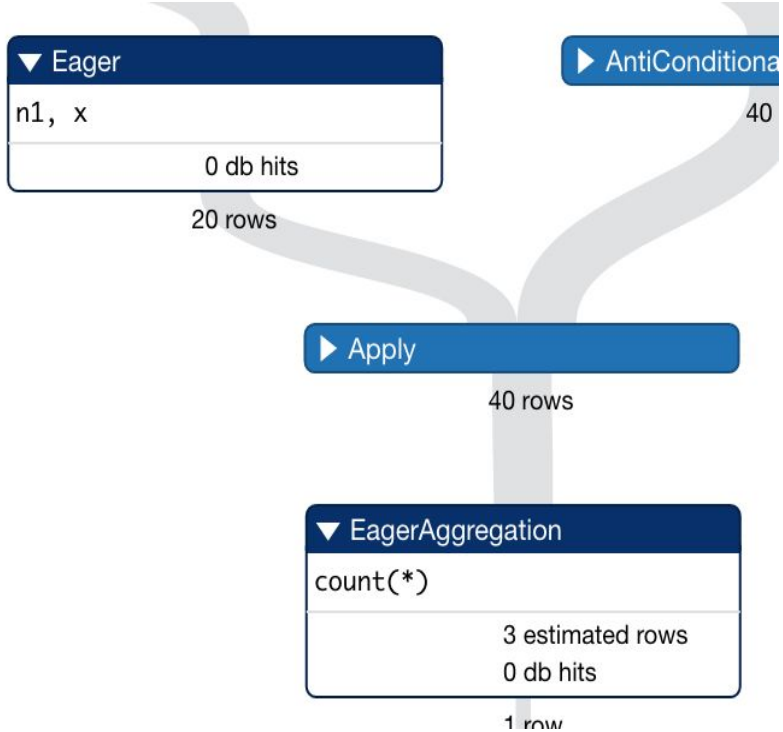
```
PROFILE MATCH (movie:Movie)
WITH movie, [(movie)<-[:DIRECTED]-(director) | director] AS directors, [(movie)<-[:ACTED_IN]-(actor) | actor] AS actors
RETURN movie, actors, directors
```

82,484 DB hits



Query tip: Reduce eagerness

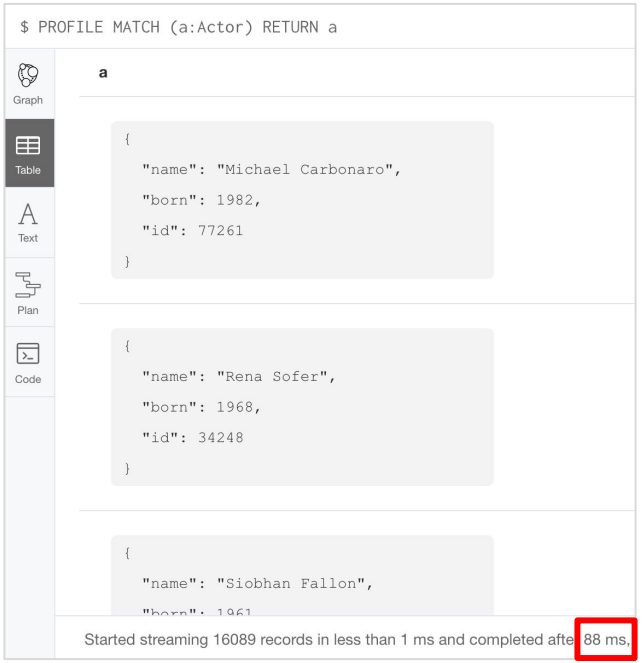
```
PROFILE
UNWIND range(1, 10) AS x
MERGE (n1:Node {id: x})
MERGE (n2:Node {id: x^2})
RETURN count(*)
```



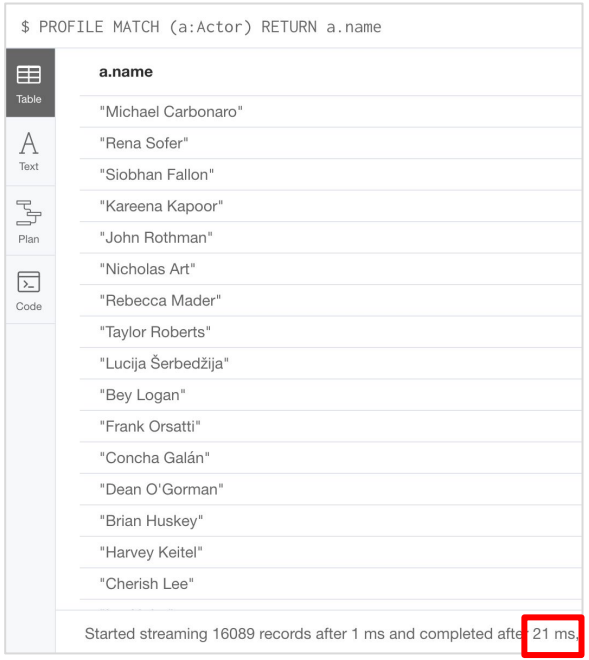
- The second MERGE operation will depend on the result of the first MERGE operation as it matches on the same label and property so newly-created nodes need to be visible to it also. This also causes Eager.
- To resolve, separate the creation into two statements and execute independently.

Best practice: Only return what you need

```
MATCH (a:Actor)
RETURN a
```




```
MATCH (a:Actor)
RETURN a.name
```



Query tip: Be selective

- Start with the smallest starting set (most selective)
  - Can I move a distinct to an earlier point in the query?
  - Can I move a limit to an earlier point in the query?
  - Can I use collect on places in the query to reduce the amount of rows to be processed during execution of the query?
  - Do I use order by on the right place in the query?
- Use WITH and COLLECT and DISTINCT to reduce the intermediate results.
- Use pattern comprehension as much as possible.
- Use label with fewest members
  - :Actor instead of :Person
- Use relationships with smallest degree
- Use index with highest selectivity
- Make sure you have an index for large retrievals (100k+) that will need to be ordered

Query tip: Use LIMIT early

```
PROFILE MATCH (movie:Movie)
OPTIONAL MATCH (movie)<-[:ACTED_IN]-(actor)
WITH movie, collect(actor) AS actors
OPTIONAL MATCH
(movie)<-[:DIRECTED]-(director)
WITH movie, actors, collect(director) AS directors
RETURN movie, actors, directors
LIMIT 1
```

82,484 DB hits

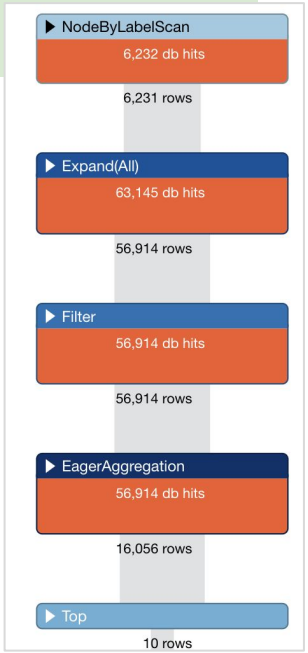
```
PROFILE MATCH (movie:Movie)
WITH movie
LIMIT 1
OPTIONAL MATCH (movie)<-[:ACTED_IN]-(actor)
WITH movie, collect(actor) AS actors
OPTIONAL MATCH (movie)<-[:DIRECTED]-(director)
WITH movie, actors, collect(director) AS directors
RETURN movie, actors, directors
```

13 DB hits

Query tip: Defer property access

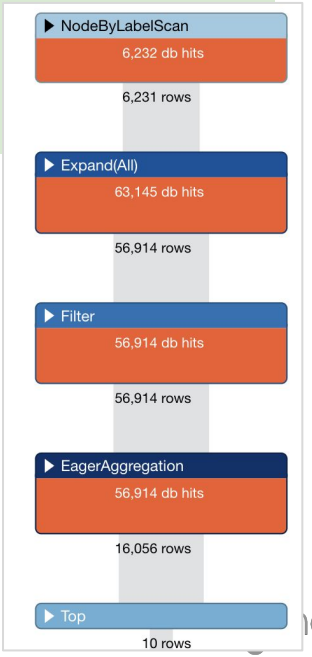
```
PROFILE
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
RETURN p.name, count(*) AS c
ORDER BY c DESC LIMIT 10
```

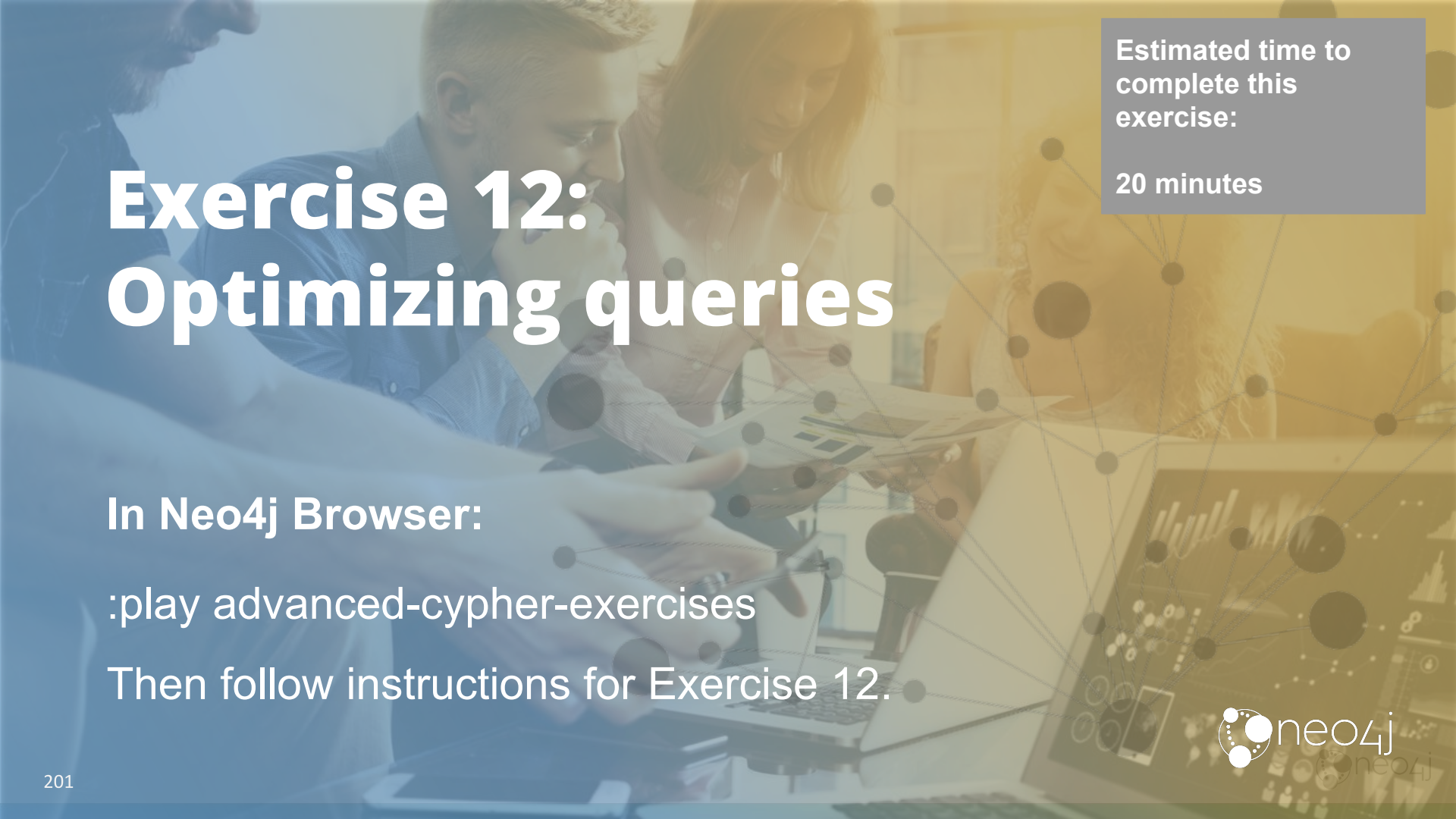
183,205 DB hits




```
PROFILE
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WITH p, count(*) AS c
ORDER BY c DESC LIMIT 10
RETURN p.name, c
```

126,301 DB hits





Exercise 12: Optimizing queries

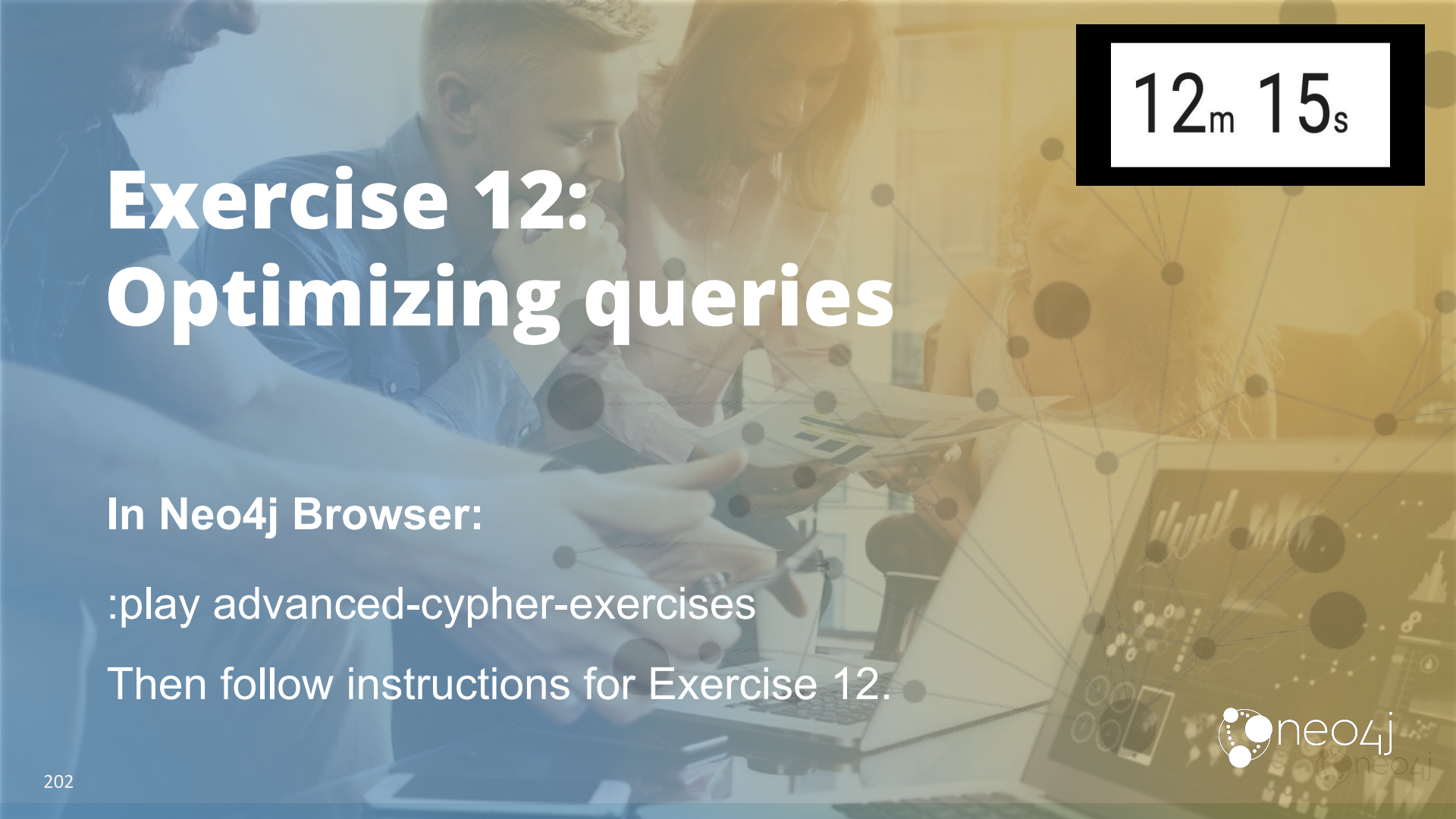
Estimated time to complete this exercise:

20 minutes

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 12.



Exercise 12: Optimizing queries

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 12.

Using indexes

neo4j

Indexes in Neo4j

Two types of indexes supported:

- Index on properties of nodes and constraints on properties
- Fulltext schema index on properties of nodes or relationships

Best practice: Always create indexes for frequently queried properties.

Indexes and constraints on properties

- Index on single node property
- Composite index on multiple node properties
- Uniqueness constraint index on a single node property
- Uniqueness constraint on multiple node properties (node key, which also enforces an existence constraint on all elements in the key)
- Existence constraint on a single node property or a single relationship property

Fulltext schema indexes

Use **fulltext schema indexes** for node or relationship properties that are queried frequently:

- Used via procedures so you need to start the query to provide the anchors.
- Fulltext index on a single property of a node label or relationship type
- Fulltext index on multiple properties of a node label or multiple properties of a relationship type
- Fulltext index on multiple properties of multiple node labels or multiple properties of multiple relationship types (multi-token)

When single property indexes are used

Used to determine the starting point for graph traversal using:

- Equality checks =
- Range comparisons >,>=,<, <=
- List membership IN
- String comparisons STARTS WITH, ENDS WITH, CONTAINS
- Existence checks exists()
- Spatial distance searches distance()
- Spatial bounding searches point()

When composite property indexes are used

Used to determine the starting point for graph traversal using:

- Equality checks =
- List membership IN

Note: The index is only used if all properties for the index are specified in the query.

Index-backed property lookup

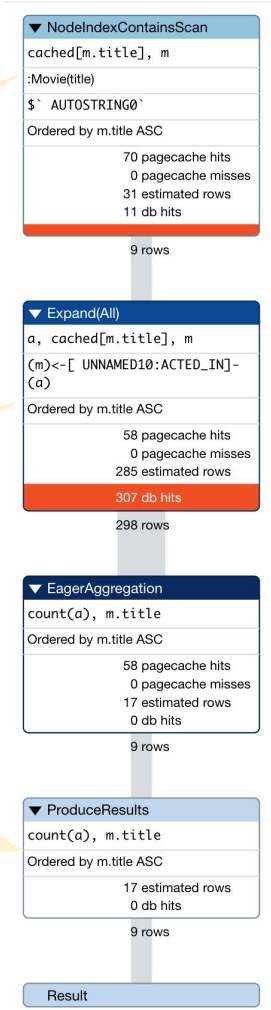
Index on a property means that the property for the node need not be accessed later in the query requiring fewer DB hits.

```
PROFILE
MATCH (a)-[:ACTED_IN]->(m:Movie)
WHERE m.title CONTAINS "Potter"
RETURN m.title, count(a)
```

Index used to select the Movie titles; 11 DB hits

DB hits to retrieve the count of :ACTED_IN relationships

No DB hits to return m.title since m.title was cached in the index.



Index-backed sorting

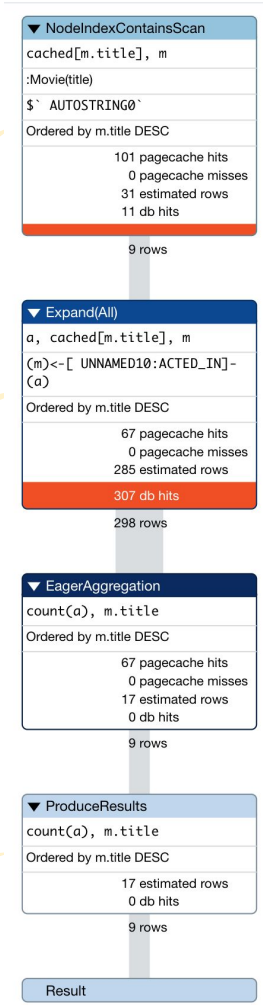
Index on a property means that the property need not be sorted later in the query since it is already sorted in the index.

```
PROFILE
MATCH (a)-[:ACTED_IN]->(m:Movie)
WHERE m.title CONTAINS "Potter"
RETURN m.title, count(a)
ORDER BY m.title DESC
```

Index used to select the Movie titles and sort in descending order; 11 DB hits

DB hits to retrieve the count of :ACTED_IN relationships

No DB hits to sort and return m.title since m.title was cached in the index.



Query tip: Use query hints

USING SCAN

- Force the planner to start on a specific **label**.
- Use when the label would be more selective than any indexes. (which is rare), or when a lookup by a node by one label may be more efficient in the overall plan than starting by looking up some other node by label.
- Good when there are a low number of nodes for a particular label.

```
MATCH (p:Person:Olympian)
USING SCAN p:Olympian
WHERE p.age = 18
RETURN p.name
```

USING INDEX

- Force the planner to start on a specific **index**.
- Use when the planner is using a less efficient index for lookup, or is starting with an index lookup by some other node, when it would be more efficient to start with an index lookup by a different node.
- If you specify multiple indexes the results will be joined.

```
MATCH (p:Person:Parent)
USING INDEX p:Person(age)
WHERE p.age < 16
RETURN p.name
```

USING JOIN

- Force how logical branches should **join** together. Use this when the cost of traversing TO a node is far less expensive than expanding FROM it or THROUGH it (and when you have efficient start node lookups from both sides of the pattern.
- Good for super nodes that have a lot of relationships.

```
MATCH (me:Person {name:'Me'})-
   [:FRIENDS]-(friend)-[:LIKES]->
   (a:Artist {name: "Taylor Swift"})
   <-[:LIKES]-(me)
USING JOIN ON a
RETURN a, count(a) AS likesInCommon
```

Example: Using multiple indexes

```
// One Index used; By default, graph engine uses one index
// as starting point and filters on related nodes
PROFILE
MATCH p = (p1:Person)-[:ACTED_IN*6]-(p5:Person)
WHERE p1.name = 'Tom Cruise'
  AND p5.name = 'Kevin Bacon'
RETURN [n IN nodes(p) | coalesce(n.title, n.name)]
```

12,201,331 DB hits

```
// Forcing the index to be used in two places
PROFILE
MATCH p = (p1:Person)-[:ACTED_IN*6]-(p5:Person)
USING INDEX p1:Person(name)
USING INDEX p5:Person(name)
WHERE p1.name = 'Tom Cruise'
  AND p5.name = 'Kevin Bacon'
RETURN [n IN nodes(p) | coalesce(n.title, n.name)]
```

8,075,477 DB hits

Warning: Use caution with query hints

- The planner will take the selectivity of an index into account when evaluating equality.
- Forcing a plan means that planner cannot adapt when the underlying data changes.
- Your plan may be more efficient specifically while being less efficient generally.
- Hints can inform the planner about the structure of your data in ways the planner cannot infer itself.
- If you do use hints, use them to force the plan around aspects of the data model that will remain consistent.

Fulltext schema indexes

- Create for a node or a relationship.
- Specify 1 or more labels/relationship types with corresponding property names.
- Implemented using Lucene.

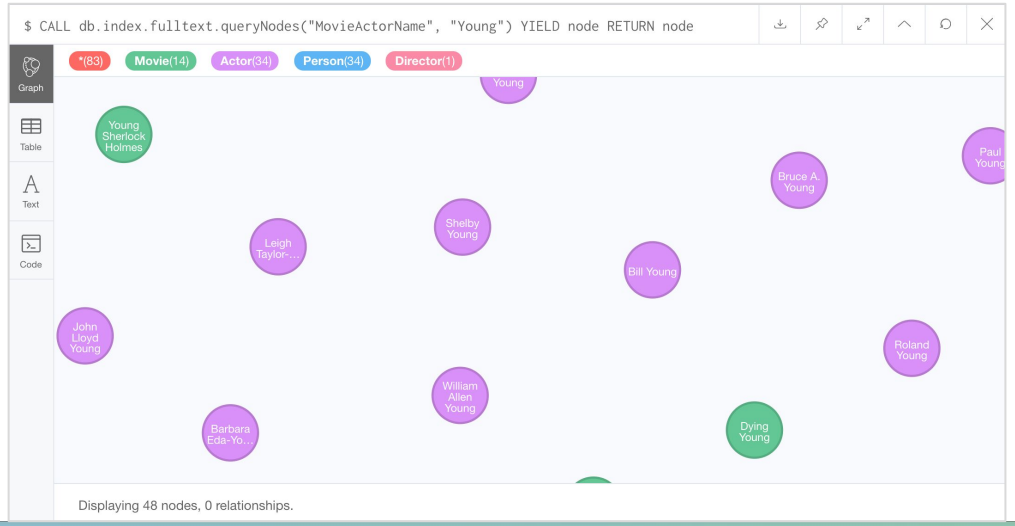
```
CALL db.index.fulltext.createNodeIndex('<indexName>',
['<label1>'[, '<labeln><']], ['<label1-property>'[, '<labeln-property']])
```

```
CALL db.index.fulltext.createRelationshipIndex('<indexName>',
['<label1>'[, '<labeln><']], ['<label1-property>'[, '<labeln-property']])
```

Example: Creating and using a fulltext index

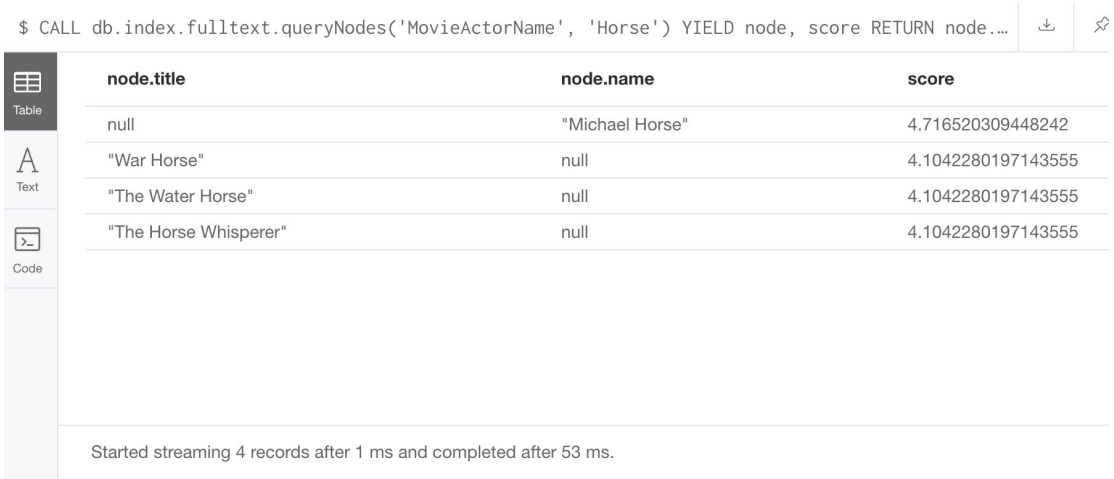
```
CALL db.index.fulltext.createNodeIndex('MovieActorName',['Movie', 'Actor'], ['title', 'name'])
```

```
CALL db.index.fulltext.queryNodes('MovieActorName', 'Young') YIELD node
RETURN node
```



Example: Getting the "hit" score for a fulltext index

```
CALL db.index.fulltext.queryNodes('MovieActorName', 'Horse' ) YIELD node, score
RETURN node.title, node.name, score
```



Note: See the Cypher documentation for examples of using fulltext indexes. See the Operations documentation about configuring fulltext index analyzers and providers.

Example: Getting an exact match

```
CALL db.index.fulltext.queryNodes('MovieActorName',
    'The Water Horse' ) YIELD node, score
RETURN node.title, node.name, score
```

```
CALL db.index.fulltext.queryNodes('MovieActorName',
    ' "The Water Horse" ')  YIELD node, score
RETURN node.title, node.name, score
```

Example: Logical expressions for match

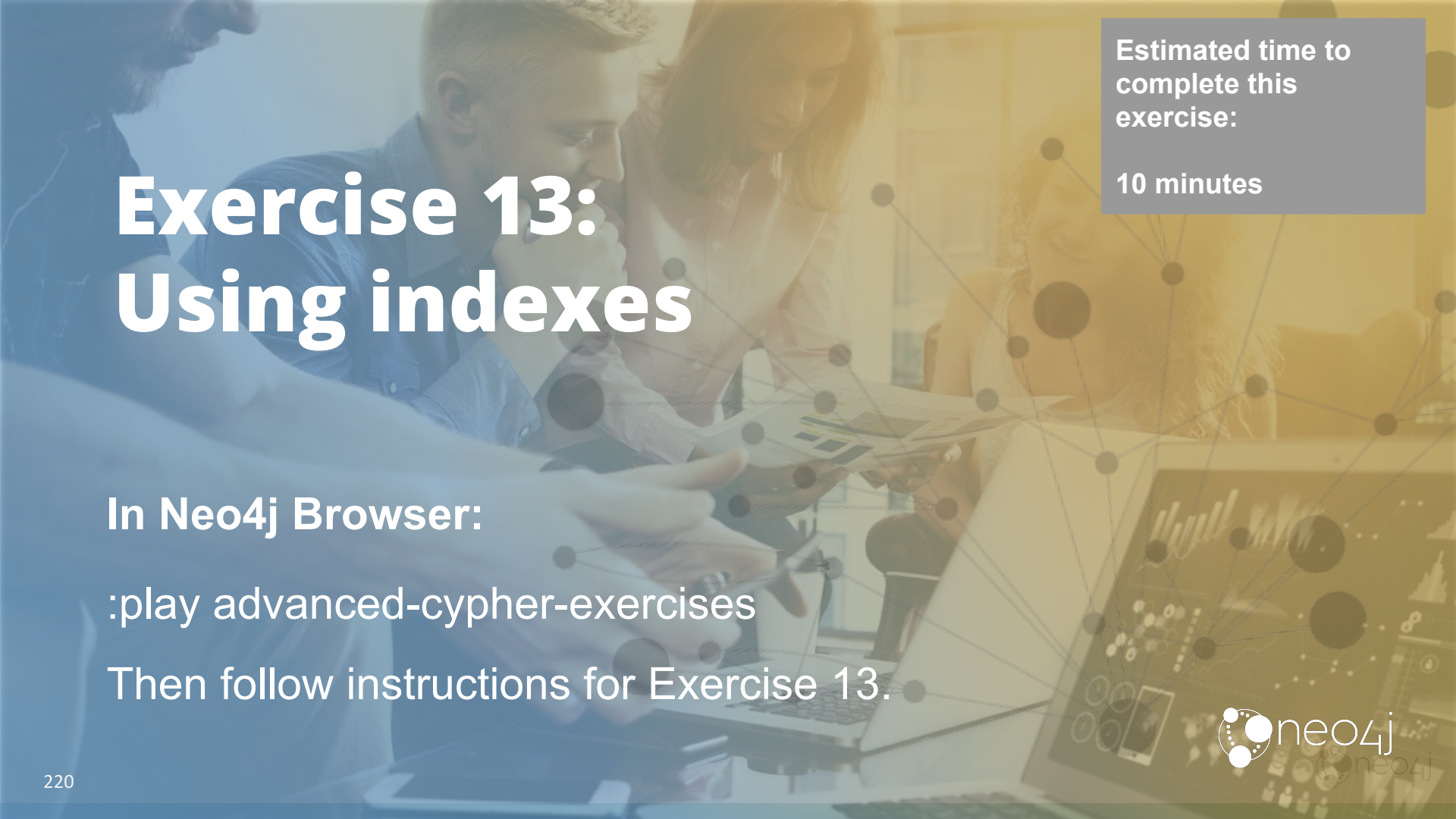
```
CALL db.index.fulltext.queryNodes("MovieActorName", "Horse OR Keanu") YIELD
node, score
RETURN node.title, node.name, score
```

node.title	node.name	score
null	"Keanu Reeves"	1.6106988191604614
null	"Michael Horse"	1.6106988191604614
"War Horse"	null	1.4016001224517822
"The Water Horse"	null	1.4016001224517822
"The Horse Whisperer"	null	1.4016001224517822

Example: Search for specific property value

```
CALL db.index.fulltext.queryNodes("MovieActorName", "title: Horse") YIELD node, score
RETURN node.title, node.name, score
```

node.title	node.name	score
"War Horse"	null	6.016839981079102
"The Water Horse"	null	6.016839981079102
"The Horse Whisperer"	null	6.016839981079102



Exercise 13: Using indexes

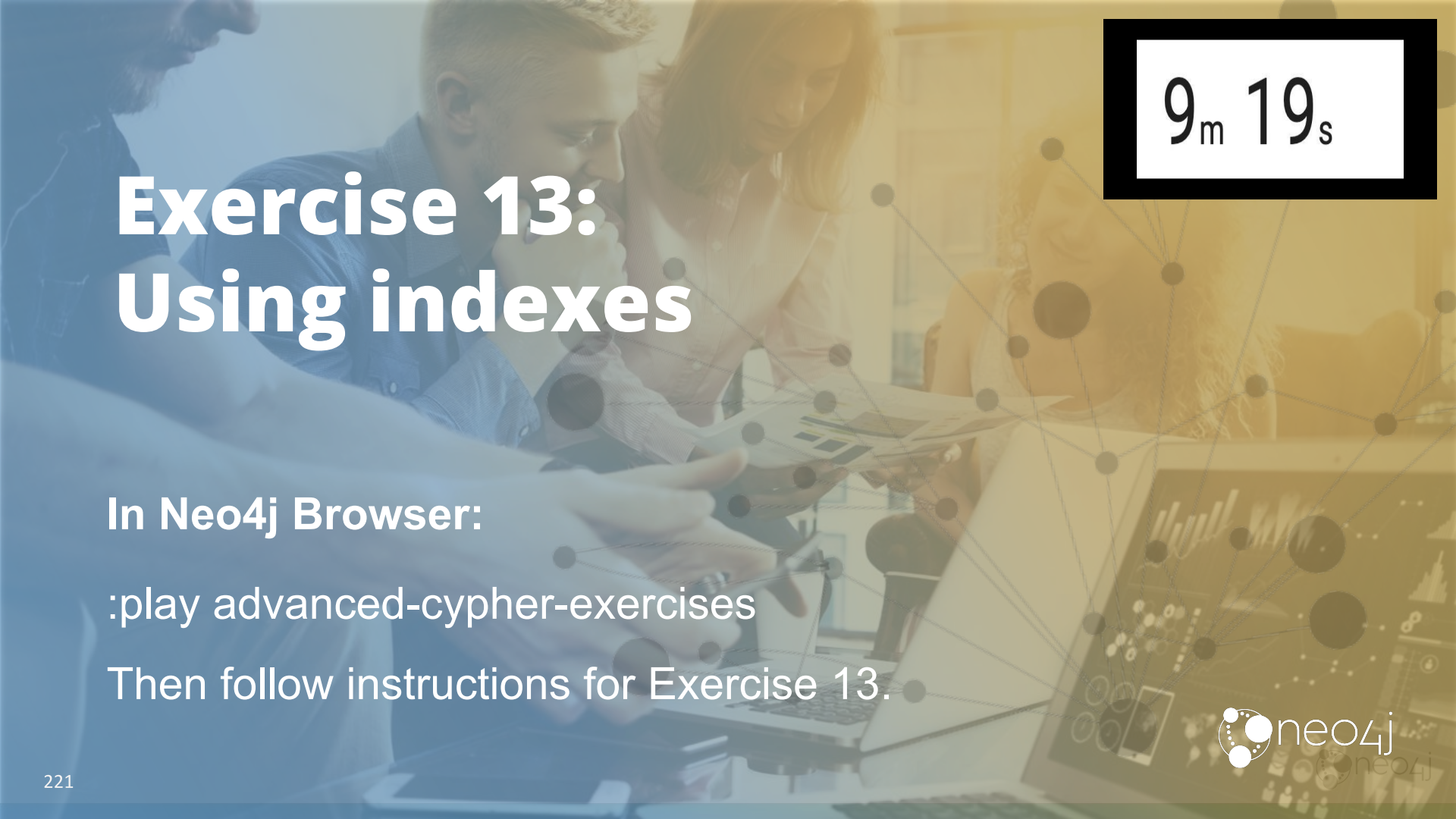
Estimated time to complete this exercise:

10 minutes

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 13.



Exercise 13: Using indexes

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 13.

Monitoring queries

neo4j

Monitoring queries

There are two reasons why a Cypher query may take a long time:

1. The query returns a lot of data. The query completes execution in the graph engine, but it takes a long time to create the result stream to return to the client.

```
MATCH (a)--(b)--(c)--(d)--(e)--(f) RETURN a
```

You should avoid these types of queries!

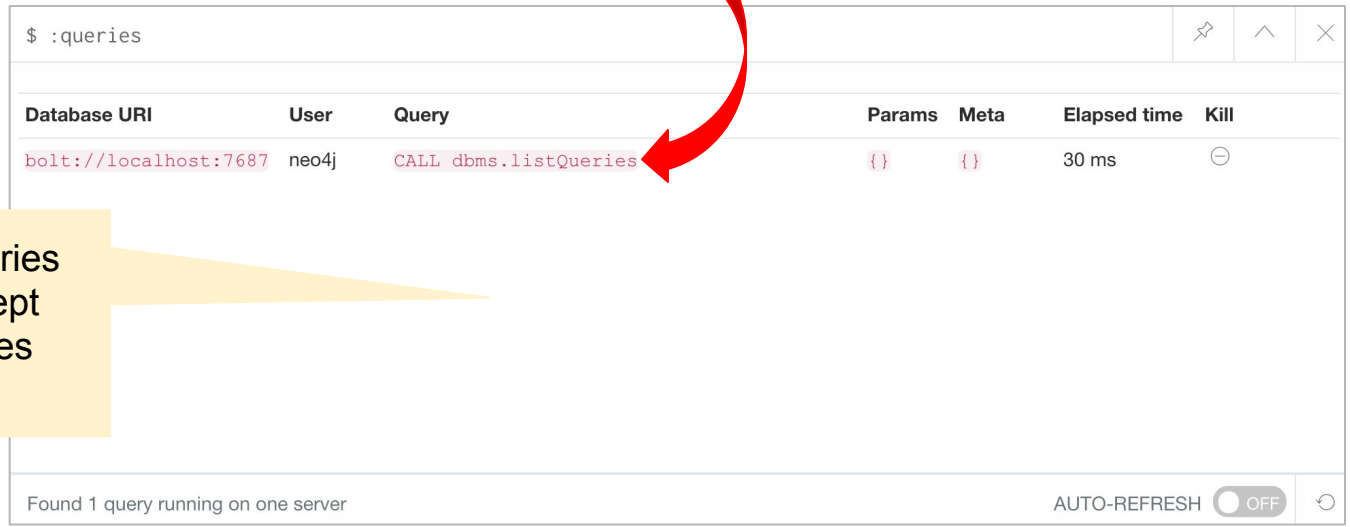
2. The query takes a long time to execute in the graph engine.

```
MATCH (a), (b), (c), (d), (e) RETURN count(id(a))
```

Viewing running queries

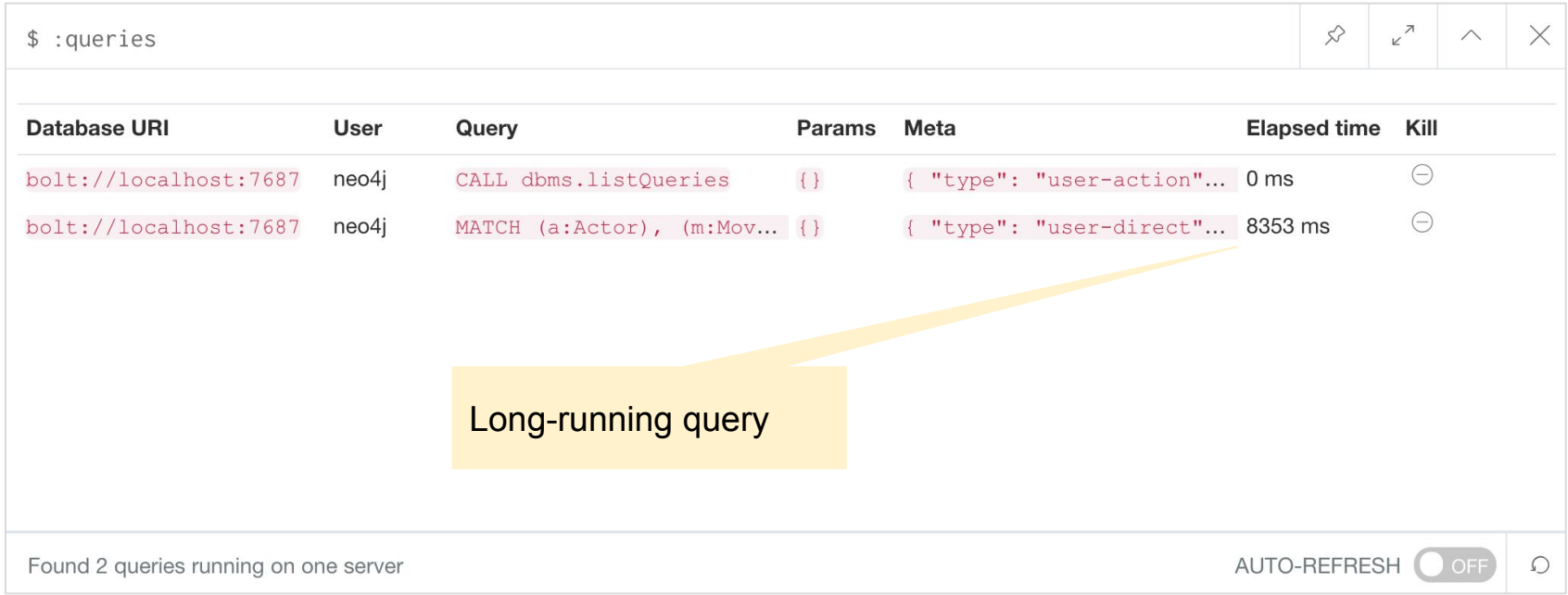
If your query is taking a long time to execute your first have to determine if it is running in the graph engine:

1. Open a new Neo4j Browser session.
2. Execute the **:queries** command.



No other queries running, except for the :queries command.

Viewing long-running queries



Getting information about running queries

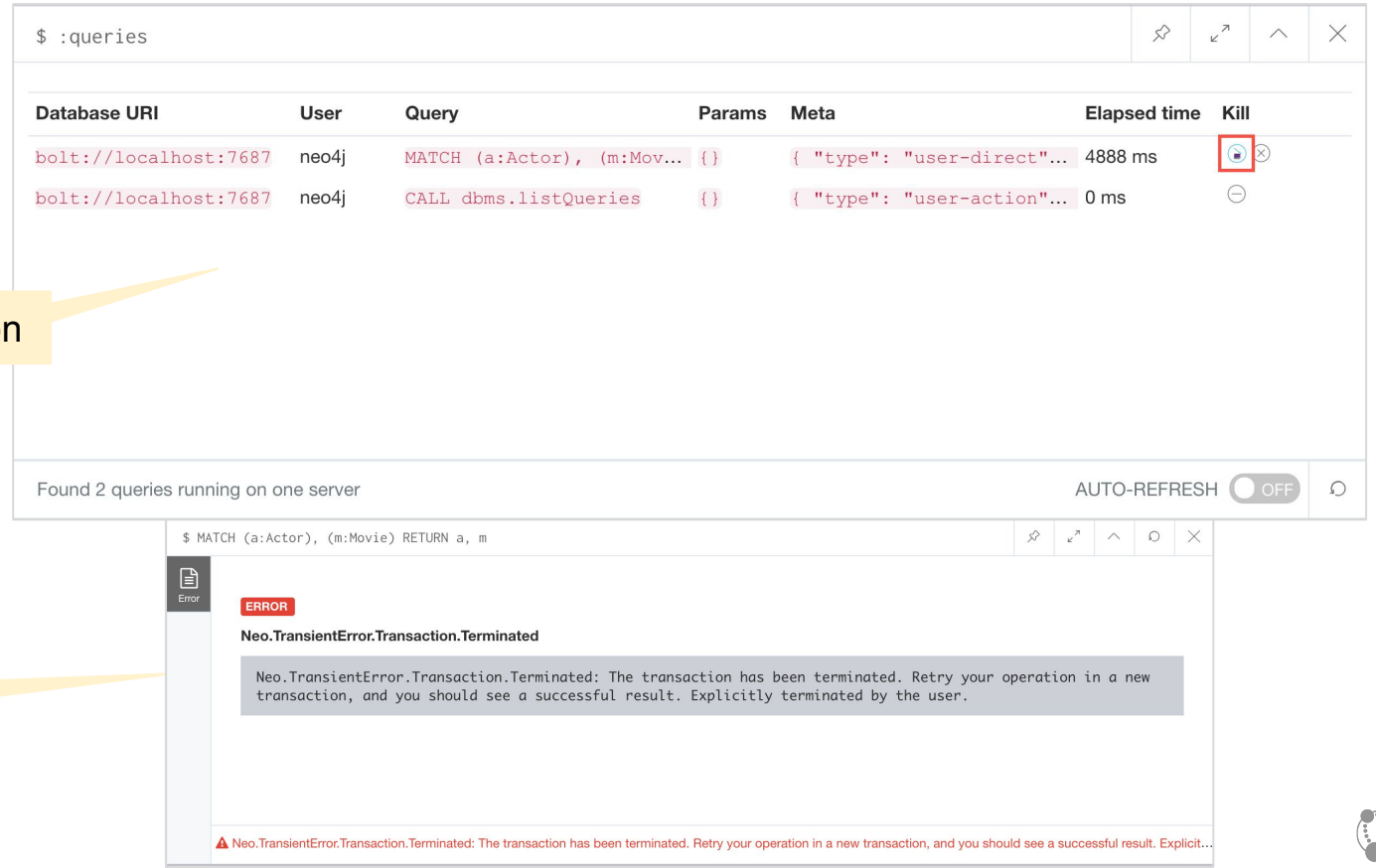
CALL dbms.listQueries() YIELD queryId, username, metaData, query, planner , runtime , indexes , startTime , protocol , clientAddress , requestUri , status , resourceInformation , activeLockCount , elapsedTimeMillis , cpuTimeMillis , waitTimeMillis , idleTimeMillis , allocatedBytes , pageHits , pageFaults

```
$ CALL dbms.listQueries() YIELD query, queryId, elapsedTimeMillis
```

Table

query	queryId	elapsedTimeMillis
"CALL dbms.listQueries() YIELD query, queryId, elapsedTimeMillis"	"query-3134"	0
"MATCH (a:Actor), (m:Movie) RETURN a, m"	"query-3131"	6855

Killing long-running queries - 1



Monitoring session

Query session

Killing long-running queries - 2

```
CALL dbms.listQueries() YIELD query, queryId, elapsedTimeMillis
WHERE query STARTS WITH 'MATCH' AND elapsedTimeMillis > 5000
CALL dbms.killQuery(queryId) YIELD queryId AS qid, message
RETURN "Query: " + " " +  qid +" " + message AS result
```

Monitoring session

```
$ CALL dbms.listQueries() YIELD query, queryId, elapsedTimeMillis
```

Table

result
"Query: query-4111 Query found"

Query session

```
$ MATCH (a:Actor), (m:Movie) RETURN a, m
```

Error

ERROR

Neo.TransientError.Transaction.Terminated

Neo.TransientError.Transaction.Terminated: The transaction has been terminated. Retry your operation in a new transaction, and you should see a successful result. Explicitly terminated by the user.

Neo.TransientError.Transaction.Terminated: The transaction has been terminated. Retry your operation in a new transaction, and you should see a successful result. Explicit...

Query Log Analyzer Tool

- Neo4j Desktop app to help you understand queries that have run against an Enterprise server.
- Use in a test environment where you can run expected queries in the time-frame to be used in the production environment.
- Reads and compiles query statistics from an existing query.log file.
- Focus on expensive queries and exhaustion of machine resources due to concurrent queries.
- Good for comparing all of the application's queries to view:
  - Planning cost
  - CPU
  - Elapsed time
  - Wait time
  - Cache hits
  - Memory
  - Client types
  - Historical data to show "clusters" of queries

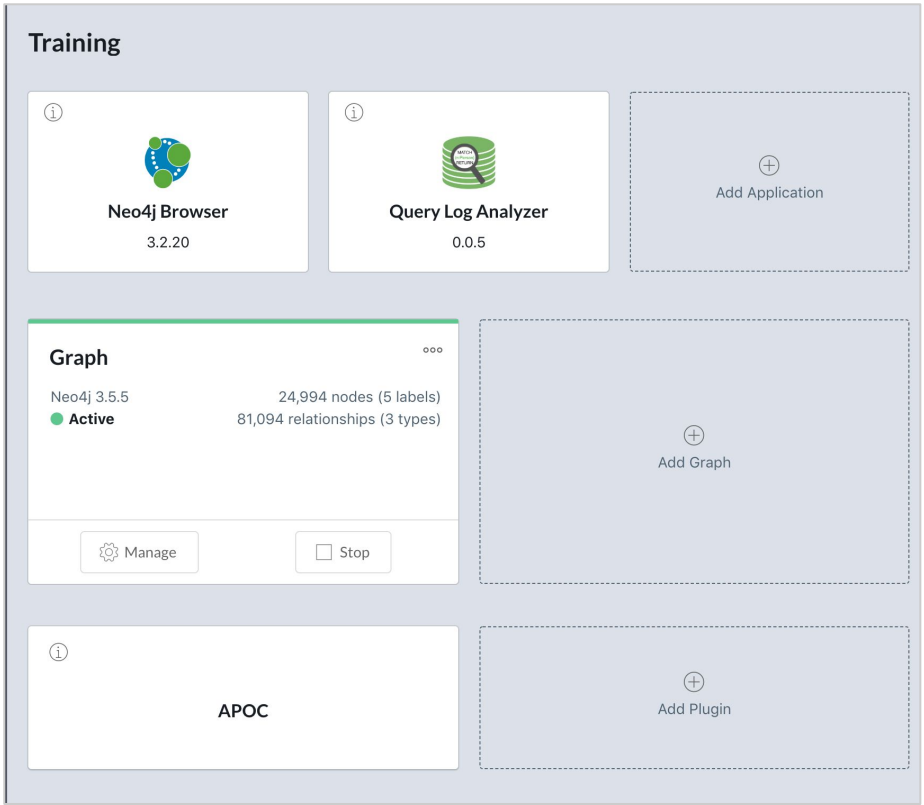
Note: For more information read: *Meet the Query Log Analyzer* (Medium blog)

Preparing to analyze queries

Configure query logging and associated properties in neo4j.conf:

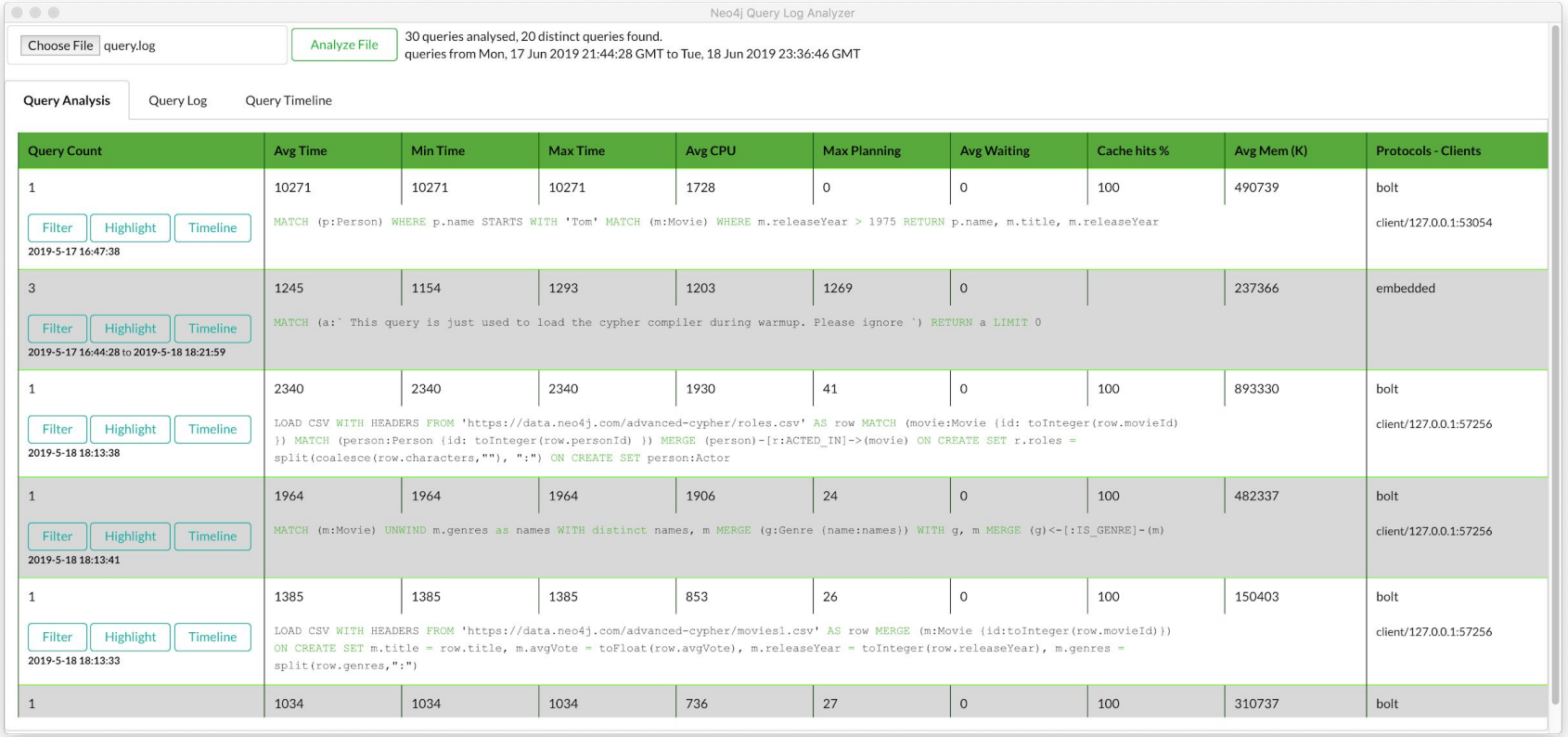
```
dbms.logs.query.enabled=true
# If the execution of query takes more time than this threshold,
# the query is logged. If set to zero then all queries
dbms.logs.query.threshold=100ms
dbms.logs.query.parameter_logging_enabled=true
dbms.logs.query.time_logging_enabled=true
dbms.logs.query.allocation_logging_enabled=true
dbms.logs.query.page_logging_enabled=true
dbms.track_query_cpu_time=true
dbms.track_query_allocation=true
```

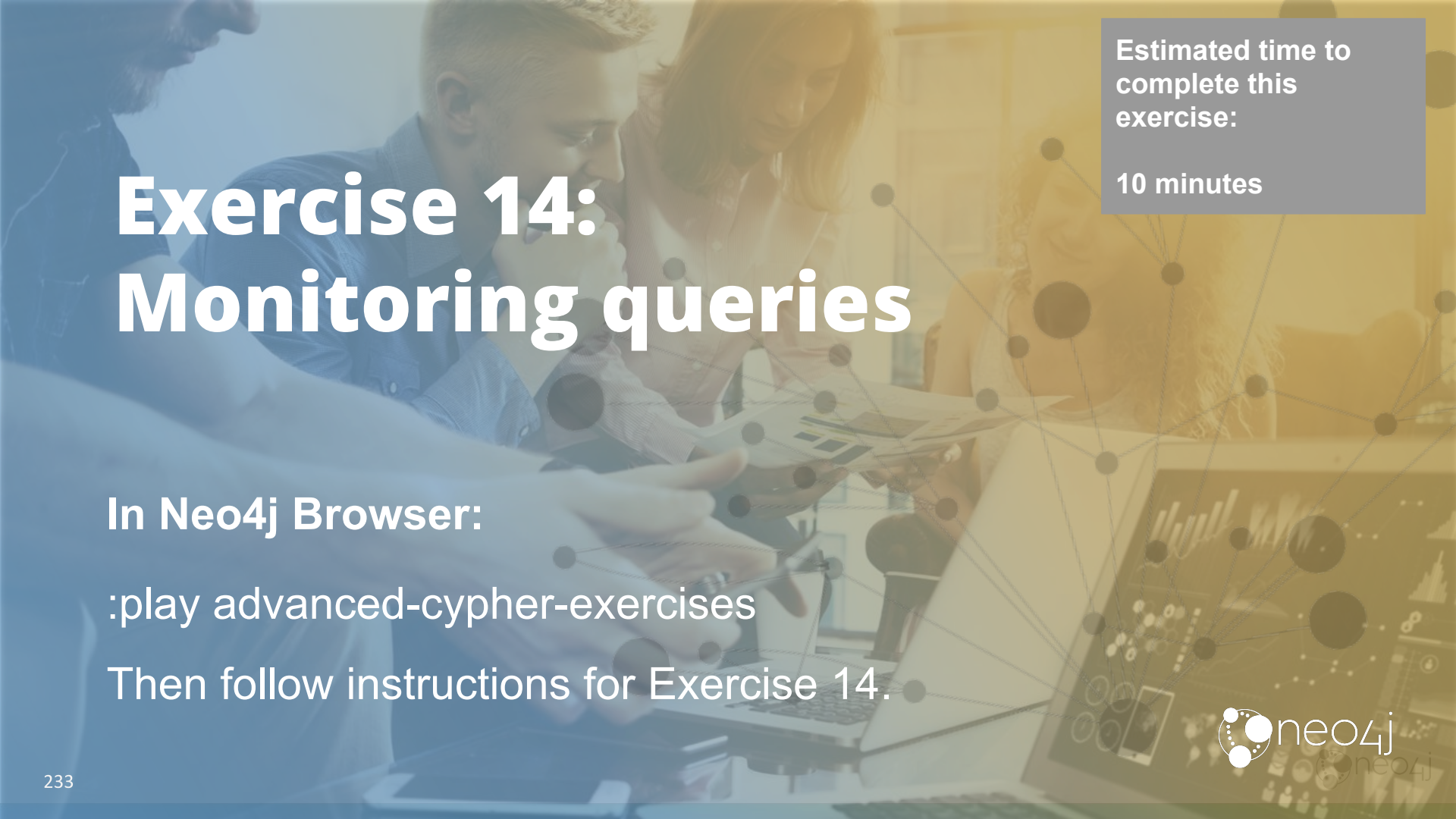
Install the Query Log Analyzer Tool



1. Click **Add Application**.
2. Click **Add** for the Query Log Analyzer.
3. Click **OK**.
4. Restart the database.

Using the Query Log Analyzer





Estimated time to complete this exercise:

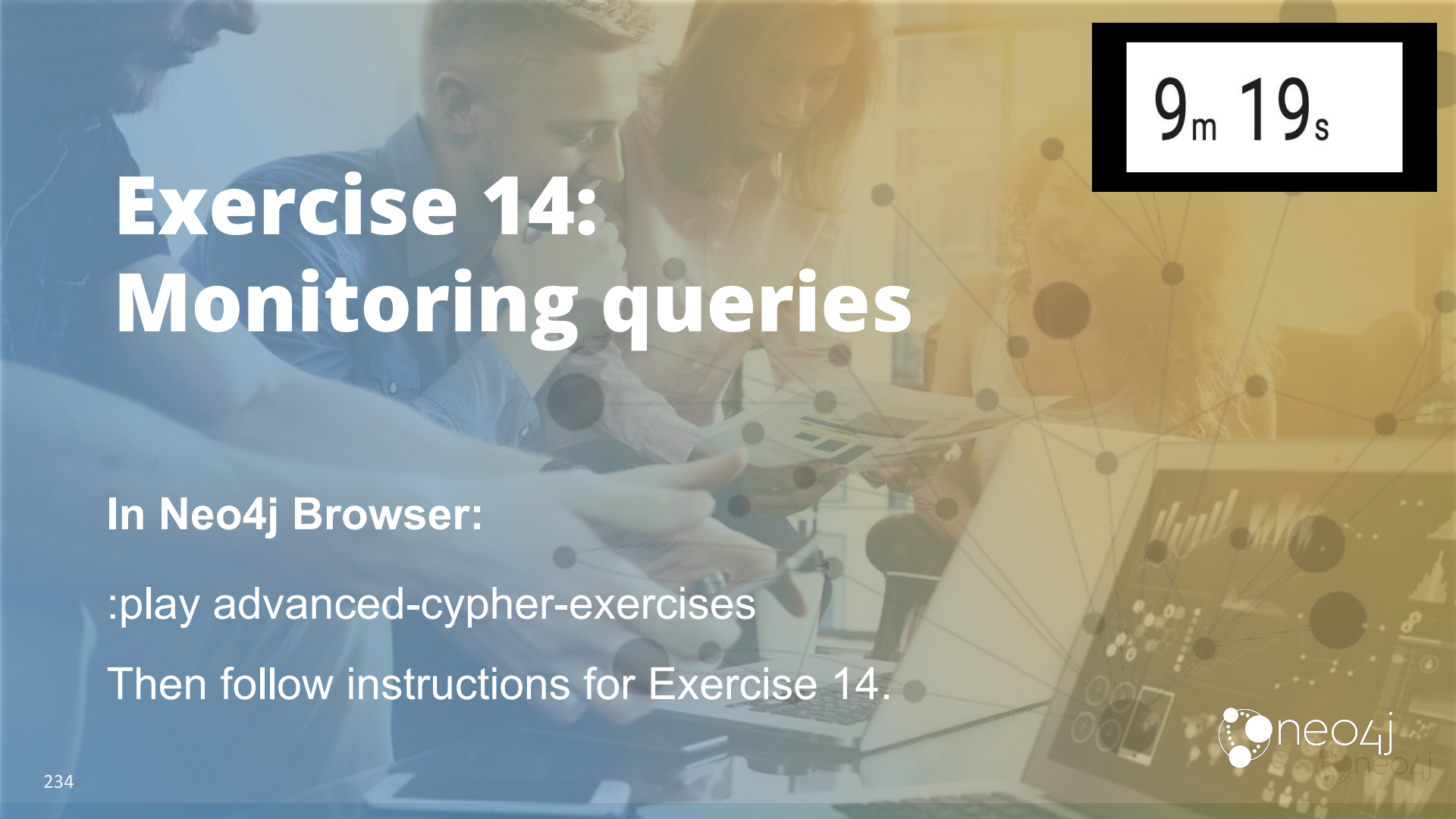
10 minutes

Exercise 14: Monitoring queries

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 14.



Exercise 14: Monitoring queries

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 14.


Check your understanding

Question 1

What information from the query plan is most helpful for determining whether you are optimizing a query?

Select the correct answers.

- ❑ Reduction in the number of steps (operators) in the plan.
- ❑ Reduction in the number of rows passed between steps in the plan.
- ❑ Reduction in the number of total DB hits.
- ❑ Eager aggregation is moved to early in the plan.

Answer 1

What information from the query plan is most helpful for determining whether you are optimizing a query?

Select the correct answers.

❑ Reduction in the number of steps (operators) in the plan.

✔ Reduction in the number of rows passed between steps in the plan.

✔ Reduction in the number of total DB hits.

❑ Eager aggregation is moved to early in the plan.

Question 2

Suppose you want a count (degree) of how many :ACTED_IN relationships each actor has in the graph (a:Actor)-[:ACTED_IN]->()? You can RETURN count(*) to return this information from the count store for each actor. What is a better alternative to using the count store where the MATCH clause is simply MATCH (a:Actor)?

Select the correct answer.

- ❏ RETURN collect(a)
- ❏ RETURN count(collect((a)-[:ACTED_IN]->()))
- ❏ RETURN collect((a)-[:ACTED_IN]->())
- ❏ RETURN size((a)-[:ACTED_IN]->())

Answer 2

Suppose you want a count (degree) of how many :ACTED_IN relationships each actor has in the graph (a:Actor)-[:ACTED_IN]->()? You can RETURN count(*) to return this information from the count store for each actor. What is a better alternative to using the count store where the MATCH clause is simply MATCH (a:Actor)?

Select the correct answer.

- ❏ RETURN collect(a)
- ❏ RETURN count(collect((a)-[:ACTED_IN]->()))
- ❏ RETURN collect((a)-[:ACTED_IN]->())
- ✔ RETURN size((a)-[:ACTED_IN]->())

Question 3

What type of index do you create if you want to optimize queries that involve properties of relationships?

Select the correct answer.

- ❏ Relationship Key
- ❏ Fulltext Schema
- ❏ Relationship Property
- ❏ Lucene

Answer 3

What type of index do you create if you want to optimize queries that involve properties of relationships?

Select the correct answer.

- ❏ Relationship Key
- ✔ Fulltext Schema
- ❏ Relationship Property
- ❏ Lucene

Summary

You should now be able to:

- Describe query planning.
- Use DB stats and the count store.
- Use some best practices for writing optimal queries.
- Create and use indexes
- Install and use the Query Analyzer Tool.


Using Neo4j with
Multiple Clients/Threads
v 1.0
neo4j

Overview

At the end of this module, you should be able to:

- Describe how locking works in Neo4j.
- Monitor locking in Neo4j.
- Use best practices when locking data in the graph.

Locking in Neo4j

neo4j

How does locking work in Neo4j?

- A client can read data in the graph without a lock.
- An EXCLUSIVE lock must be acquired to write data in the graph.
- What can be locked:
  - Node (and its properties)
  - Relationship (and its properties)
  - Index
- Two types of locks:
  - EXCLUSIVE (single writer)
  - SHARED (multiple readers; blocks writers; used for schema changes only)
- A lock is held for the entire lifetime of a transaction.

How are locks acquired?

- A client can always read data if the Cypher statement is read-only:
  - Will never be blocked by other clients updating the same data.
  - Will only read data that has been committed.
- EXCLUSIVE locks are held until the Cypher statement completes execution.
- An EXCLUSIVE lock will be acquired for write operations such as:
  - Setting a property
  - Setting a label
  - Creating a relationship (start/end nodes are also locked, as well as neighboring relationships in the chain)
  - Updating an index
- A client will need to wait for an EXCLUSIVE lock if another client has an EXCLUSIVE lock on the same data.

What is a transaction in Cypher?

- A unit of work that involves reading data and acquiring EXCLUSIVE locks if the graph is updated.
- Scope of a transaction is one or more Cypher statements that may contain read clauses and write clauses.

```
MATCH (a:Actor)
WHERE a.name = 'Doris Day'
RETURN a.died
```

- read-only transaction
- **no locking**
- only read committed data

```
MATCH (a:Actor)
WHERE a.name = 'Doris Day'
WITH a
SET a.died = 2019
RETURN a.died
```

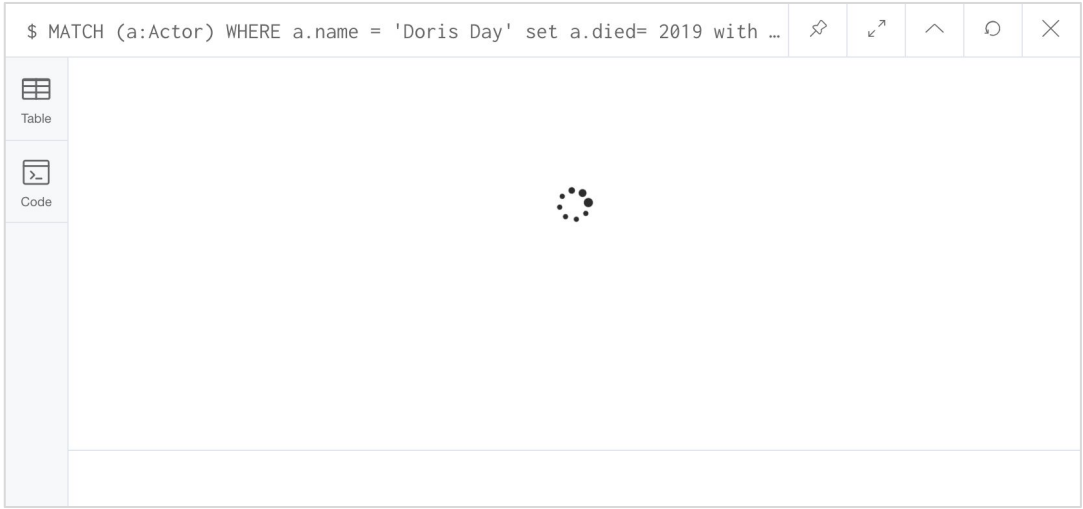
- read-only clause
- no locking
- only read committed data

acquire EXCLUSIVE lock on node

EXCLUSIVE lock released when Cypher execution ends

Performing a (blocking) write transaction

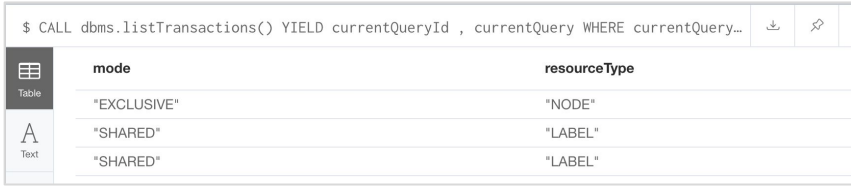
```
MATCH (a:Actor)
WHERE a.name = 'Doris Day'
SET a.died= 2019
WITH a
call apoc.util.sleep(200000)
RETURN a.died
```

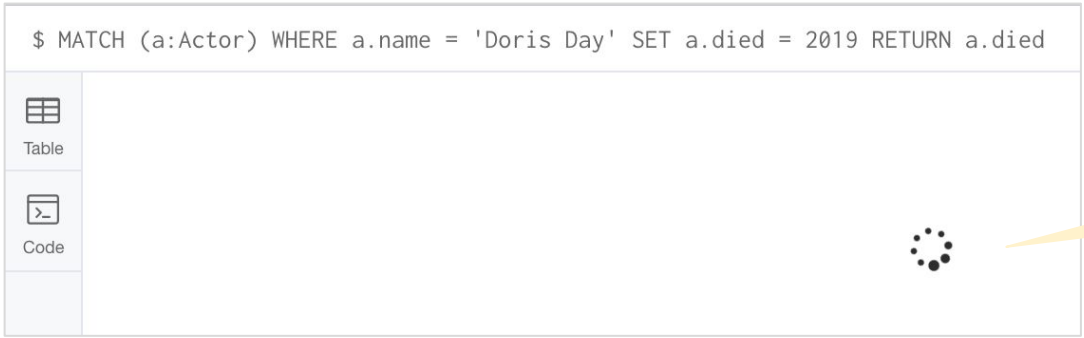


Keep this write transaction open for 200 seconds.

Getting the locking information

```
CALL dbms.listTransactions() YIELD currentQueryId , currentQuery
WHERE currentQuery STARTS WITH "MATCH (a:Actor)"
CALL dbms.listActiveLocks(currentQueryId) YIELD mode, resourceType
RETURN mode, resourceType
```





Another client is blocked

What is deadlock?

Time	Transaction	Cypher Statement
08:00:01	tx1001	Begin
08:00:02	tx1001	MATCH (n:Person {name:'Tom Hanks'}) set n.age=59;
08:00:03	tx1002	Begin
08:00:04	tx1002	MATCH (n:Movie {title:'Cast Away'}) set n.gross=233630478;
08:00:05	tx1001	MATCH (n:Movie {title:'Cast Away'}) set n.budget=90000000;
08:00:06	tx1002	MATCH (n:Person {name:'Tom Hanks'}) set n.residence=California;

Avoiding deadlock - 1

Use **optimistic** locking - lock early

```
MATCH (p:Person {name:'Tom Hanks'}), (m:Movie {title:'Cast Away'})
SET p._lock = true, m._lock = true
WITH p, m
// do more processing (simulate with sleep)
call apoc.util.sleep(200000)
WITH p,m
SET p.age=59
SET m.gross=233630478
REMOVE p._lock, m._lock
RETURN p,m
```

Avoiding deadlock - 1

Use APOC for **optimistic** locking - lock early

```
MATCH (p:Person {name:'Tom Hanks'}), (m:Movie {title:'Cast Away'})
CALL apoc.lock.nodes([p,m])
WITH p, m
// do more processing (simulate with sleep)
call apoc.util.sleep(200000)
WITH p,m
SET p.age=59
SET m.gross=233630478
RETURN p,m
```

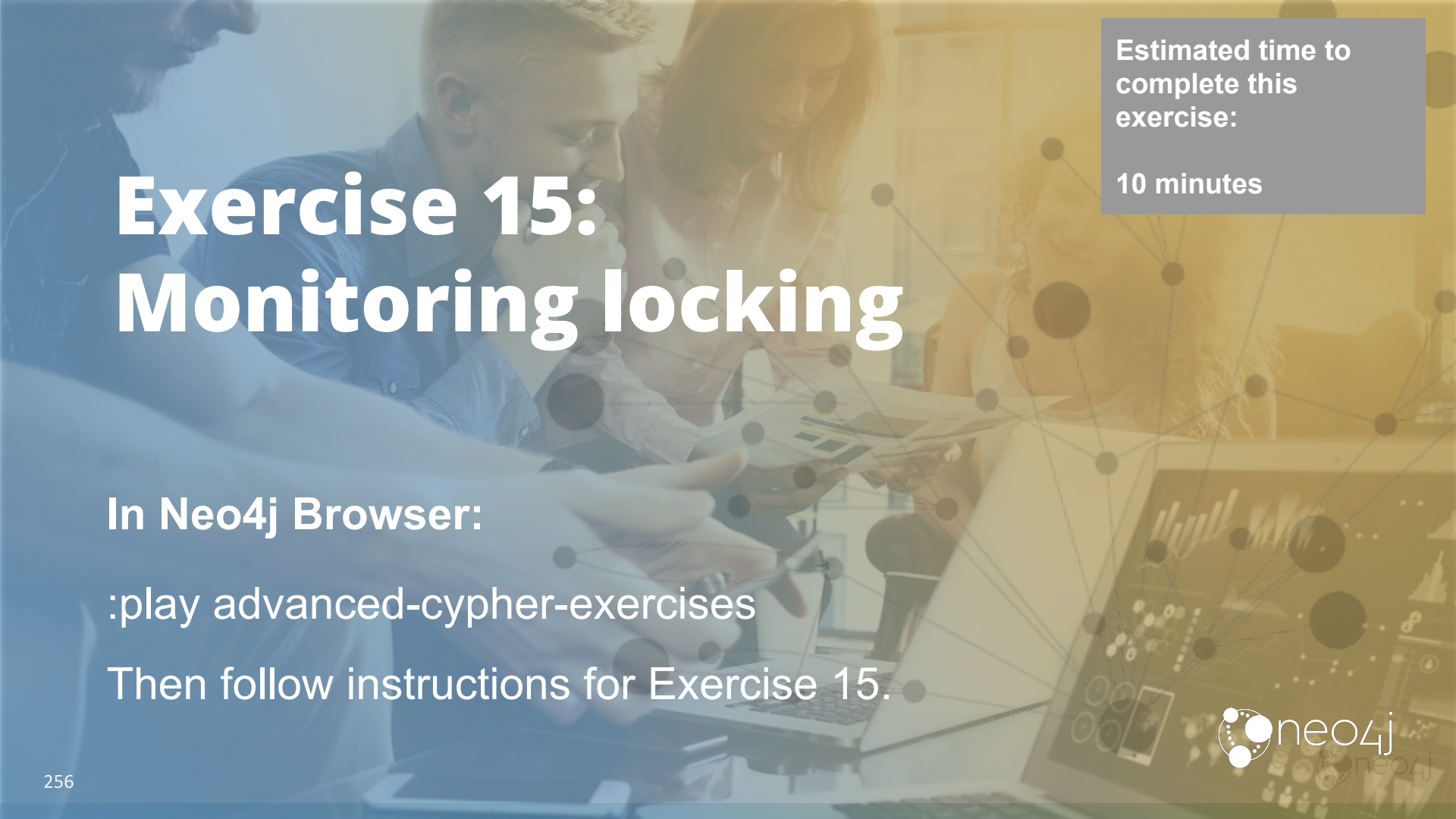
Minimizing lock contention

Use **pessimistic** locking - lock later

```
MATCH (p:Person {name:'Tom Hanks'}), (m:Movie {title:'Cast Away'})
WITH p, m
// do more processing (simulate with sleep)
call apoc.util.sleep(200000)
WITH p,m
// perform all updates at the end of the transaction
SET p.age=59
SET m.gross=233630478
RETURN p,m
```

Lock configuration settings

```
# transaction guard: max duration of any transaction
dbms.transaction.timeout=10s
# max time to acquire write lock
dbms.lock.acquisition.timeout=10ms
```



Estimated time to complete this exercise:

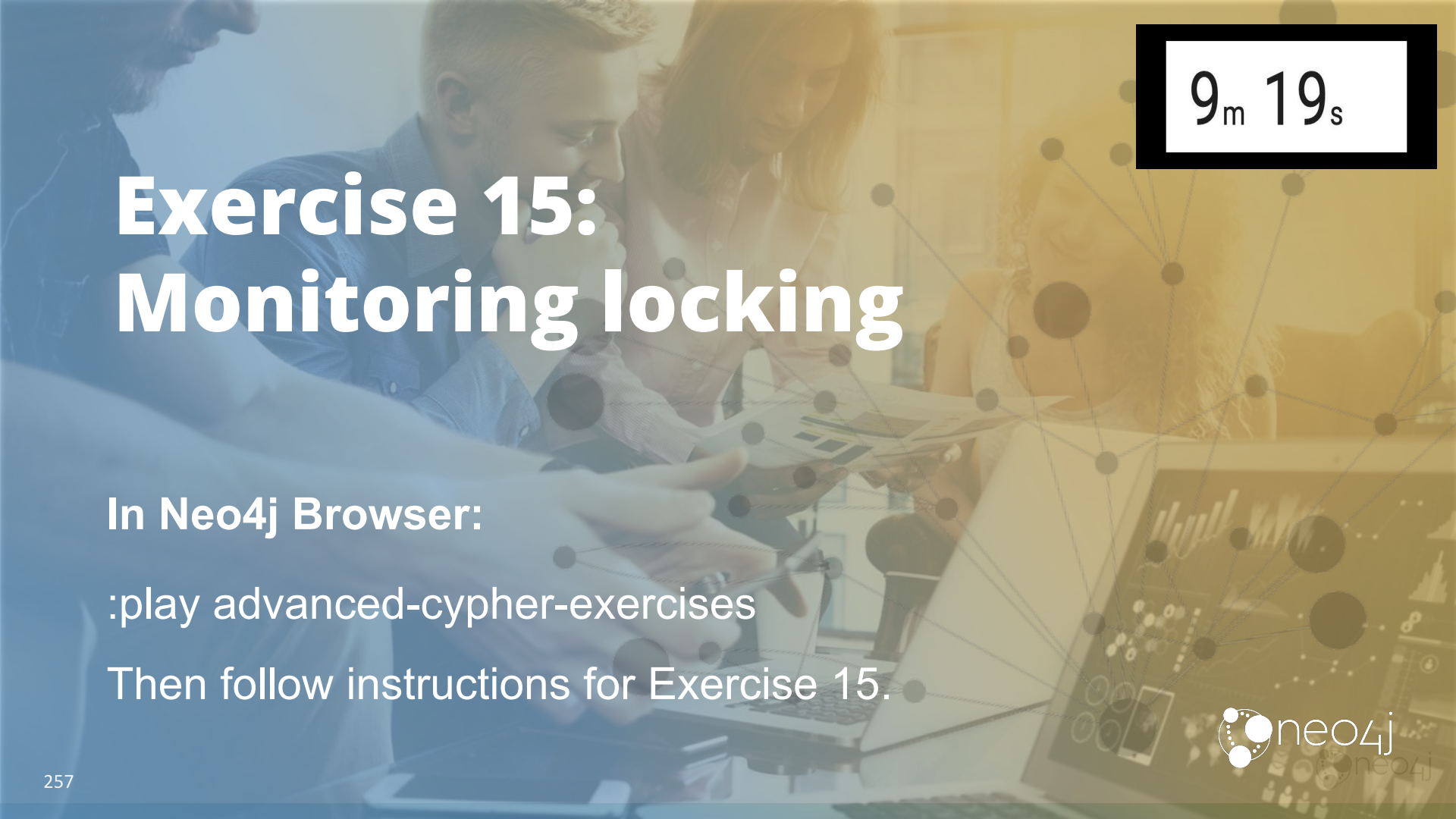
10 minutes

Exercise 15: Monitoring locking

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 15.



Exercise 15: Monitoring locking

In Neo4j Browser:

:play advanced-cypher-exercises

Then follow instructions for Exercise 15.

Check your understanding

Question 1

Suppose you have retrieved node A and node B and you want to write Cypher code to add a :DEPENDS_ON relationship between A and B. What types of locks are acquired?

Select the correct answers.

- ❑ READ-ONLY
- ❑ SHARED
- ❑ EXCLUSIVE
- ❑ NODE-LOCK

Answer 1

Suppose you have retrieved node A and node B and you want to write Cypher code to add a :DEPENDS_ON relationship between A and B. What types of locks are acquired?

Select the correct answers.

- ❏ READ-ONLY
- ✔ SHARED
- ✔ EXCLUSIVE
- ❏ NODE-LOCK

Question 2

When is pessimistic locking useful?

Select the correct answers.

❏ When you want to eliminate deadlocks.

❏ When there are a lot of read-only and then a final write step in a Cypher statement.

❏ When you want to minimize lock acquisition timeouts.

❏ When it is OK for data to change by another user after you have read it and are still working with the older values.

Answer 2

When is pessimistic locking useful?

Select the correct answers.

❏ When you want to eliminate deadlocks.

✔ When there are a lot of read-only and then a final write step in a Cypher statement.

✔ When you want to minimize lock acquisition timeouts.

✔ When it is OK for data to change by another user after you have read it and are still working with the older values.

Question 3

What can you do to minimize deadlocks in your code?

Select the correct answer.

- ❏ Set a transaction guard in **neo4j.conf**.
- ❏ Use pessimistic locking.
- ❏ Use optimistic locking.
- ❏ Use explicit BEGIN/END clauses.

Answer 3

What can you do to minimize deadlocks in your code?

Select the correct answer.

❏ Set a transaction guard in **neo4j.conf**.

❏ Use pessimistic locking.

✔ Use optimistic locking.

❏ Use explicit BEGIN/END clauses.

Summary

You should now be able to:

- Describe how locking works in Neo4j.
- Monitor locking in Neo4j.
- Use best practices when locking data in the graph.