1. Find a seat, say hi!
2. Grab a USB key, and plug it in.
3. Run the “Install Neo4j” application (Mac or Windows).
4. Grab a coffee and get ready to begin!

Advanced Cypher

Graph Connect
San Francisco 2015

22nd October 2015

Why are you here?

What is your Cypher/Neo4j background?

Overview of tutorial

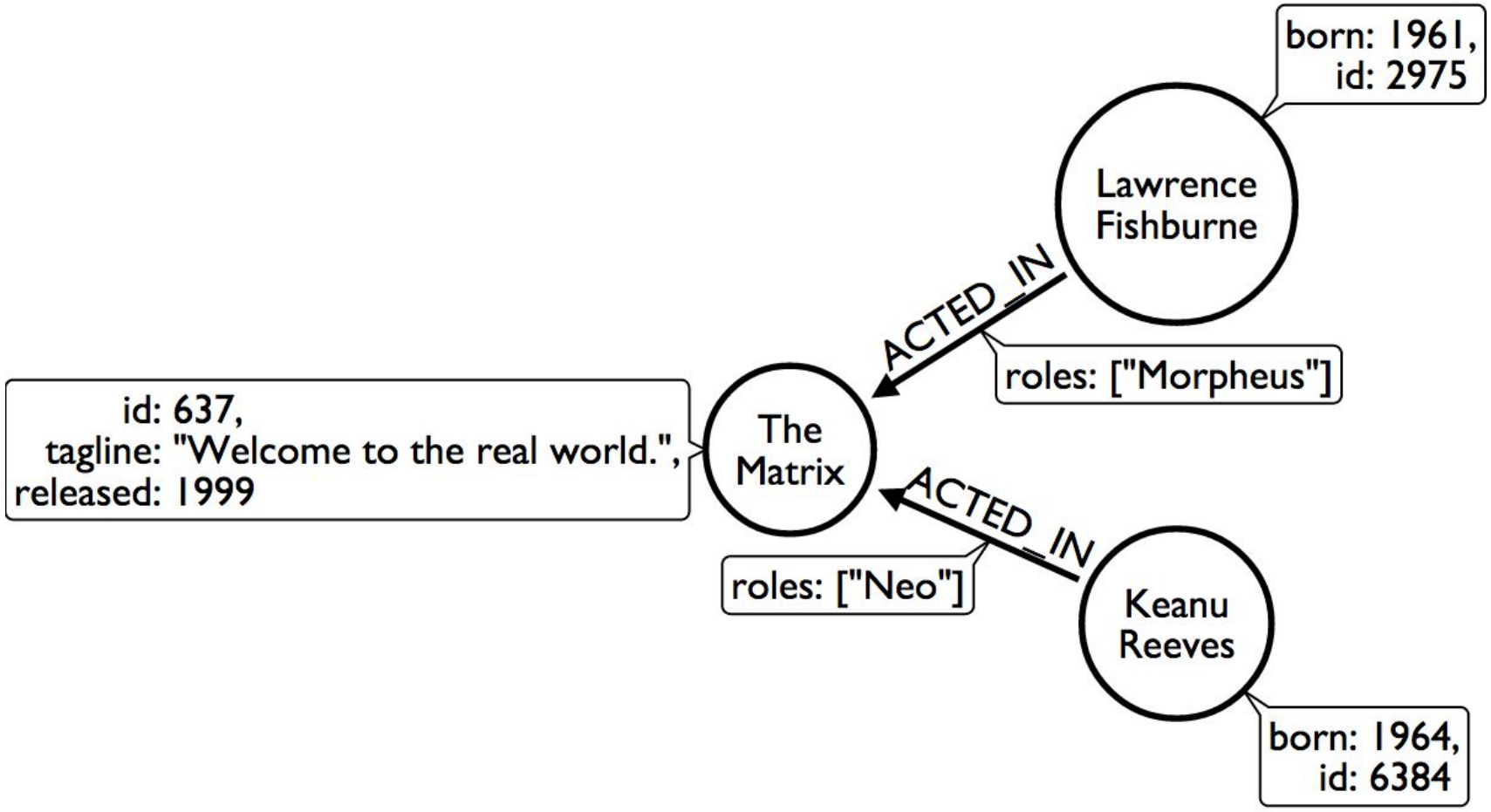
- Quick Cypher refresher
- Data import with Cypher
- Intro to Cypher internals
- Advanced Cypher concept discussions and exercises
- Optimization tips and techniques
- Bring us your hard queries

Cypher resources for the class

- Cypher refcard

  http://neo4j.com/docs/2.3.0/cypher-refcard/

Cineasts data model



Cypher Refresher

Case sensitivity

- Cypher keywords are mostly case insensitive
- But, several things are case sensitive:
  - Labels
  - Relationship types
  - Property names (keys)
  - Identifiers you create in cypher

MATCH

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

MATCH examples

```
... // a simple pattern, with RELTYPE
MATCH (n)-[:LINK]-(m)

... // match a complex pattern
MATCH (n)-->(m)<--(o), (p)-->(m)

... // match a variable-length path
MATCH p=(n)-[:LINKED*]-()

... // use a specialized matcher
MATCH p=shortestPath((n)-[*]-(o))
```

WHERE

- use `MATCH` to find patterns, and `WHERE` to filter them
- use patterns as predicates in `WHERE`, eg. `WHERE NOT (n)-->()`
- can't create new identifiers in `WHERE`, only predicates on existing identifiers defined in `START` or `MATCH`

WHERE examples

```
... // filter on a property value
WHERE n.name = "Andrés"

... // filter with predicate patterns
WHERE NOT (n)<--(m)

... // filter on path/collection length
WHERE length(p) > 3

... // filter on multiple predicates
WHERE n.born < 1980 AND n.name =~ "A.*"
```

RETURN

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

RETURN examples

```
… // p aliased to person
RETURN p AS person

… // implicit group by n, count(*)
RETURN n, count(*) AS count

… // collect things into a collection
RETURN n, collect(r)
```

ORDER BY/SKIP/LIMIT

- Cypher doesn't guarantee ordering unless you `ORDER BY`
- `SORT, LIMIT, SKIP`
- things you order by must be in the `RETURN/WITH` clause
- all optional clauses (you can do `LIMIT` without `ORDER BY`, etc.)

ORDER BY/SKIP/LIMIT examples

```
… // descending sort, limit 5
RETURN n, count(*) AS count
ORDER BY count DESC
LIMIT 5

… // get the next 5
RETURN n, count(*) as count
ORDER BY count DESC
SKIP 5
LIMIT 5
```

Labels

- labels are like type tags for nodes
- label-based indexes and constraints
- 2.x Cypher syntax:

```
CREATE (p:Person)// create labeled node
SET p:Person     // set label on node
REMOVE p:Person  // remove label
MATCH (p:Person) // match nodes with label
WHERE p:Person   // label predicate
RETURN labels(p) // get label collection
```

Indexes overview

- new in 2.x, based on labels
- can be hinted
- used for exact lookup and range queries
- automatic

Index example

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

Index example

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

Indexes (Hints)

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

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

vs

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

Constraints

- Neo4j 2.x allows for constraints on label, property combinations
- `UNIQUE` constraints available
- `NOT NULL` constraints in enterprise version
- soon general expression-based constraints
- creates accompanying index automatically

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

CASE/WHEN

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

CASE/WHEN example

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

FOREACH

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

FOREACH example

```
// we'll create some nodes
// from properties in a collection
WITH [1,2,3] as nums
FOREACH(x in nums|
  CREATE (n:Num {num:x})
)
```

UNWIND

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

UNWIND example

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

UNION

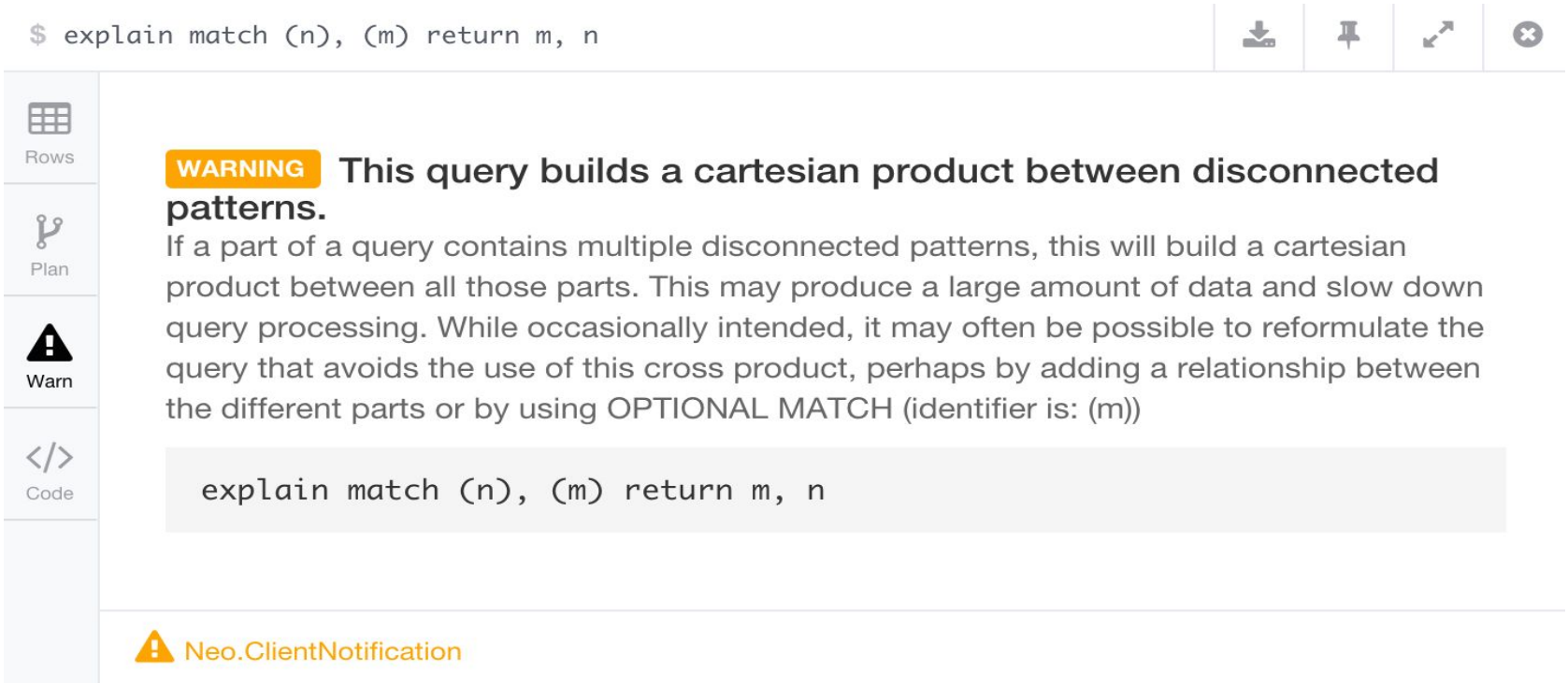
- `UNION` two or more full Cypher queries together
- aliases in `RETURN` must be exactly the same
- `UNION ALL` if you don't want to remove duplicates

UNION example

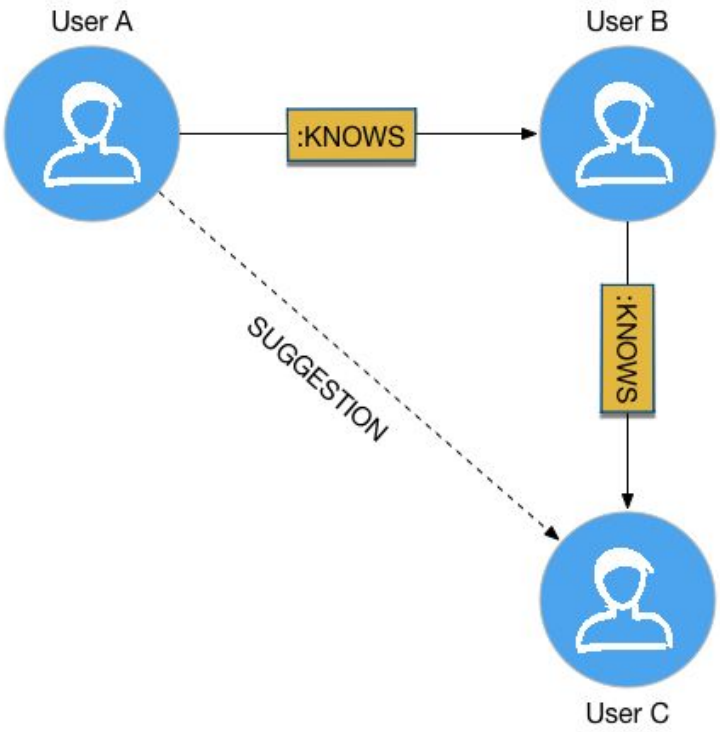
```
// note that aliases are the same
MATCH (a:Actor)
RETURN a.name as name
UNION
MATCH (d:Director)
RETURN d.name as name
```

New in Neo4j 2.3

Warnings


$ explain match (n), (m) return m, n
Rows
Plan
Warn
Code
WARNING This query builds a cartesian product between disconnected patterns.
If a part of a query contains multiple disconnected patterns, this will build a cartesian product between all those parts. This may produce a large amount of data and slow down query processing. While occasionally intended, it may often be possible to reformulate the query that avoids the use of this cross product, perhaps by adding a relationship between the different parts or by using OPTIONAL MATCH (identifier is: (m))
explain match (n), (m) return m, n
Neo.ClientNotification

Triadic Selection



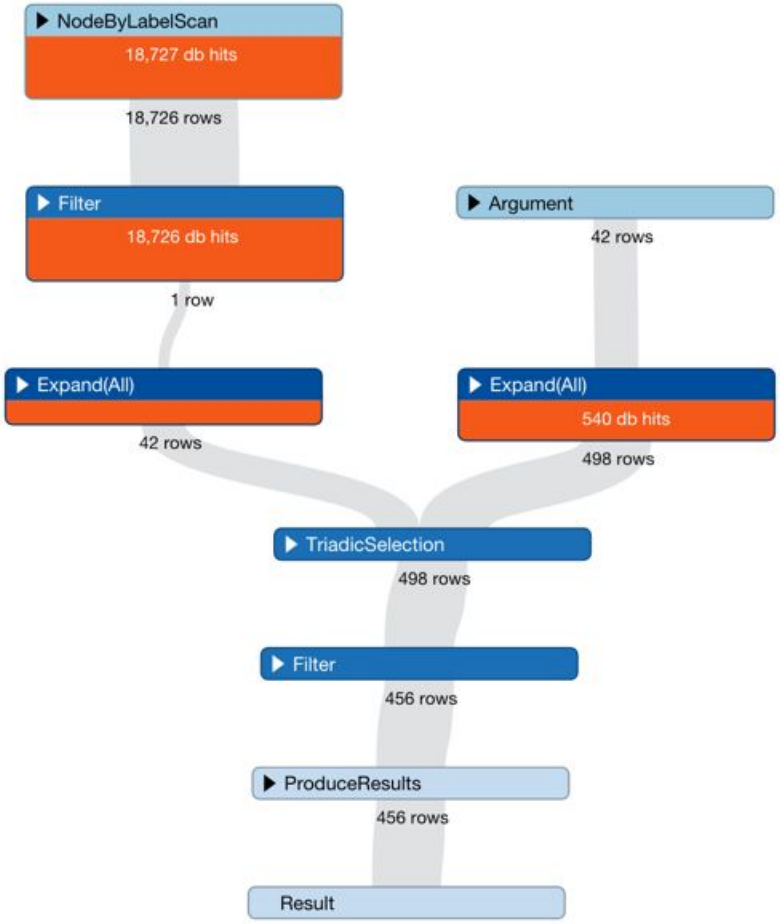
- http://graphaware.com/neo4j/2015/10/20/faster-recommendation-graphaware-reco-neo4j-triadic.html

Triadic Selection

```
MATCH (a:User {name:"A"})
      -[:KNOWS]->(b)-[:KNOWS]->(c)
WHERE a <> c
AND NOT (a)-[:KNOWS]->(c)
RETURN c
```

- http://graphaware.com/neo4j/2015/10/20/faster-recommendation-graphaware-reco-neo4j-triadic.html

Triadic Selection



- http://graphaware.com/neo4j/2015/10/20/faster-recommendation-graphaware-reco-neo4j-triadic.html

Range queries

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

Range queries

- ```
  MATCH (p:Person)
  WHERE p.born > 1980 RETURN p;
  ```
- ```
  MATCH (m:Movie)
  WHERE 2000 <= m.released < 2010
  RETURN m;
  ```
- ```
  MATCH (p:Person)
  WHERE p.name >= "John"
  RETURN p;
  ```
  ```

Text search

- New keywords
- STARTS WITH is index supported

Text search

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

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

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

Detach Delete

- Delete node + relationships attached to it

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

Detach Delete

- Delete node + relationships attached to it

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

Dense node merging

- Now correctly picks the side of smallest cardinality when `MERGE`ing relationships
- Particularly noticeable when you have a dense node follower pattern e.g. `(everyone)-[:FOLLOWS]->(ladyGaga)`

Dynamic property lookup

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

Dynamic property lookup

- ```
  MATCH (question:Question)
  RETURN  [prop IN keys(question)
                  WHERE prop
                  ENDS WITH "_count" |
                question[prop]]
  ```
  ```

Now… you're refreshed. Questions?

Coffee?

Data Import with Cypher

First things first...

Let's create our constraints...

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

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

MERGE

- MERGE is the way to do MATCH/CREATE in one step, or a sort of upsert
- ON MATCH / ON CREATE syntax for conditional declaration after the MERGE

```
MERGE (p:Person {id:123})
  ON CREATE
    SET p.name = "Andrés",
        p.createTime = timestamp()
  ON MATCH
    SET p.updateTime = timestamp()
  ...
```

Data import with Cypher

- Cypher can represent large structures in a single statement, but for efficient import, it's best to break it down into small numbers of nodes and relationships (or combined node + rel if that is easier)
- best is to have short Cypher statements that can be cached

Data import with Cypher

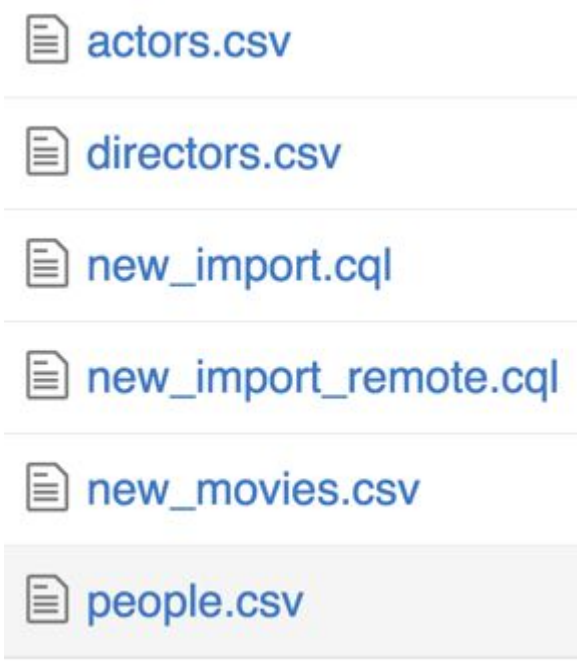
- MERGE for creating unique nodes
  - use indexed fields in the merge node, followed by SET for the rest of the node's properties in the ON CREATE portion
- MERGE for creating unique relationships or attached nodes
- If you're bulk loading, be sure to use larger transaction sizes (test to see what ends up being the fastest)

Data import with LOAD CSV

- `LOAD CSV` gives you a collection of values (or a map, with headers) for the CSV record that you can cast in Cypher, and use to create a graph
- The old movies.cyp import script (several Cypher statements) would take a few minutes to import. With `LOAD CSV`, it takes 10-20 seconds for the same data.
- Extremely effective at translating SQL tables to a graph.

Exercise

- Importing the extended movie graph
- Go to the following page: https://github.com/mneedham/graphconnect-training



Create people

```
LOAD CSV WITH HEADERS FROM 'people.csv' as row

WITH toInt(row.personId) as personId,
     row.name as personName,
     toInt(row.birthYear) as birthYear,
     toInt(row.deathYear) as deathYear

MERGE(person:Person {id: personId})
  ON CREATE SET
    person.name = personName,
    person.born = birthYear,
    person.died = deathYear;
```

Create movies

```
LOAD CSV WITH HEADERS FROM 'new_movies.csv' as record

WITH toInt(record.movieId) as movieId,
     record.title as title,
     toFloat(record.avgVote) as avgVote,
     toInt(record.releaseYear) as releaseYear,
     record.tagline as tagline,
     split(record.genres, ":") as genres

MERGE (movie:Movie {id:movieId})
ON CREATE SET
  movie.title = title,
  movie.avgVote = avgVote,
  movie.releaseYear = releaseYear,
  movie.tagline = tagline,
  movie.genres = genres;
```

Connect directors

```
LOAD CSV WITH HEADERS FROM 'directors.csv' as record
WITH toInt(record.movieId) as movieId, toInt(record.personId) as
personId
MATCH (movie:Movie {id:movieId})
MATCH (person:Person {id: personId})
MERGE (person)-[:DIRECTED]->(movie);
```

Connect actors

```
LOAD CSV WITH HEADERS FROM 'actors.csv' as record
WITH toInt(record.movieId) as movieId, toInt(record.personId) as
personId, split(record.characters, ":") as roles
MATCH (movie:Movie {id:movieId})
MATCH (person:Person {id: personId})
MERGE (person)-[:ACTED_IN {roles:roles}]->(movie);
```

Exercise

- What do you notice in the results of this query?

```
MATCH (a:Person)-[r:ACTED_IN]->(m:
Movie)
WHERE a.name = "Billy West"
AND m.title = "Futurama: Into the
Wild Green Yonder"
RETURN m.title, a.name, r.roles;
```

Let's clean up the import by merging roles.

Exercise

```
MATCH (a)-[r:ACTED_IN]->(m)
WITH a, collect(r) as roles, m
WHERE length(roles) > 1
WITH tail(roles) as extras, head(roles) as keep
UNWIND extras as extra
SET keep.roles = keep.roles + extra.roles
DELETE extra
RETURN keep;
```

Exercise

- Add `:Actor` and `:Director` labels to the dataset by inferring which they are through the relationship type

Exercise

- Add `:Actor` and `:Director` labels to the dataset by inferring which they are through the relationship type

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

MATCH (d:Person)-[:DIRECTED]->(:
Movie)
SET d:Director;
```

Data import without Cypher

- Cypher or Cypher's LOAD CSV are the best options to load while the graph server is running
- consider an unmanaged extension to implement lower level calls on a server endpoint; depending on input, may be faster than batch Cypher
- for optimized bulk load while server is down, check out the neo4j-import tool: http://neo4j.com/docs/stable/import-tool.html

Cypher query continuation: WITH

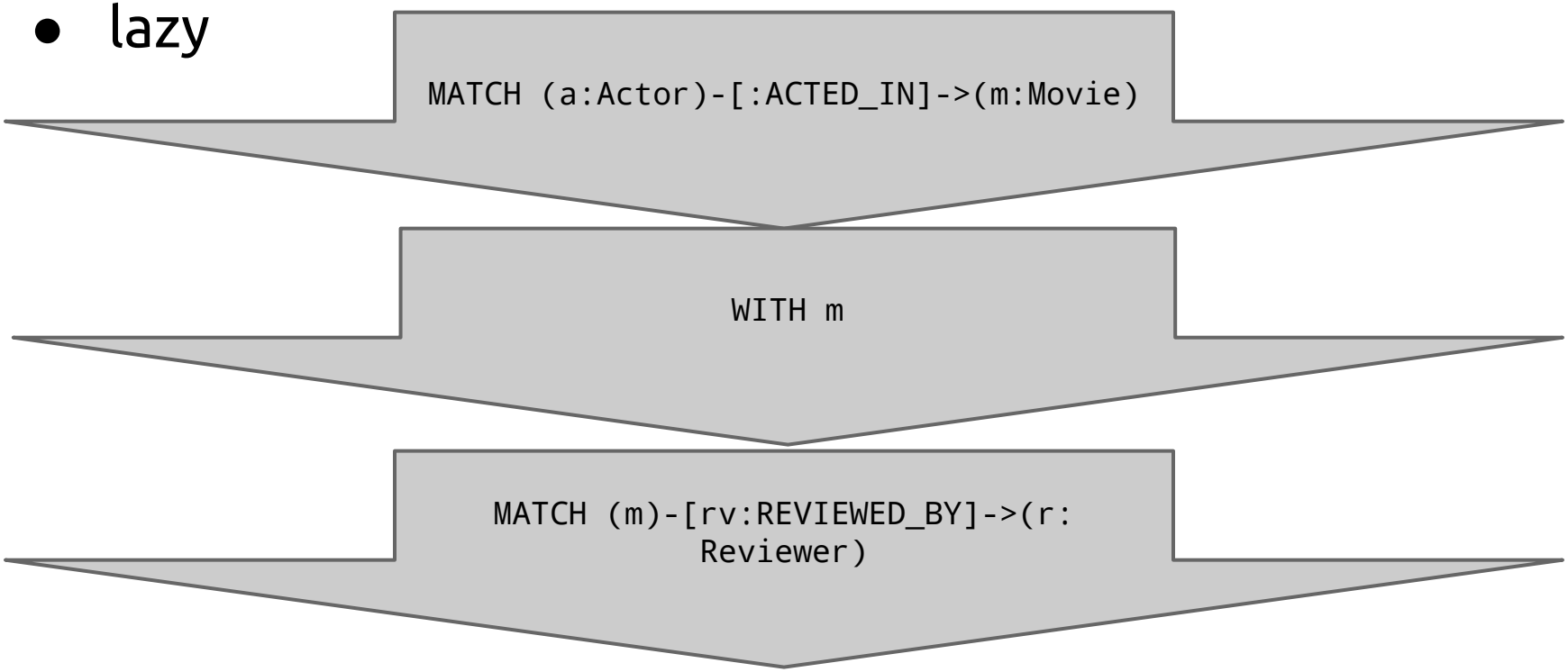
WITH

- get intermediate results, use them for a new query
- `ORDER BY/LIMIT/SKIP` can be used after `WITH`, in order to narrow down your first result set before continuing
- cartesian product with new `MATCH`ed identifiers

```
MATCH (a:Actor)-[:ACTS_IN]->(m:Movie)
WITH DISTINCT m
MATCH (m)<-[rv:REVIEWED_BY]-(r:Reviewer)
RETURN m.title, rv.rating, r.name
```

WITH

- only `m`, not `a`, is carried on from the first `MATCH` to the next
- lazy



Exercise: Get comfortable with WITH, and some aggregations...

1. Find the most prolific actors in our dataset.

2. Find only the actors who have acted in at least 20 movies.

3. Find the movies they acted in as well (and return them).

Bonus: Find actor/directors who have acted in 10 movies AND directed at least 2. Count movies they acted in and list movies they directed.

1. Find the most prolific actors

```
MATCH (a:Actor)-[:ACTED_IN]->(m:Movie)
RETURN a, count(*)
ORDER BY count(*) DESC
LIMIT 10;
```

2. Find the actors who have been in at least 20 movies

```
MATCH (a:Actor)-[:ACTED_IN]->(m:Movie)
WITH a, count(m) AS movie_count
WHERE movie_count >= 20
RETURN a, movie_count
ORDER BY movie_count DESC;
```

3. Find the actors with 20+ movies, and the movies they acted in

```
MATCH (a:Actor)-[:ACTED_IN]->(m:Movie)
WITH a, collect(m.title) AS movies
WHERE length(movies) >= 20
RETURN a, movies
ORDER BY length(movies) DESC;
```

Bonus: Find prolific actors (10+) who have directed at least two films, count films acted in and list films directed.

```
MATCH (a:Actor)-[:ACTED_IN]->(m:Movie)
WITH a, count(m) AS acted
WHERE acted >= 10
WITH a, acted
MATCH (a:Director)-[:DIRECTED]->(m:Movie)
WITH a, acted, collect(m.title) AS directed
WHERE length(directed) >= 2
RETURN a.name, acted, directed
ORDER BY length(directed) DESC, acted DESC;
```

~200ms with warm cache

Bonus optimization: Rewritten to filter both :Actor and :Director labels up front.

```
MATCH (a:Actor:Director)-[:ACTED_IN]->(m:Movie)
WITH a, count(1) AS acted
WHERE acted >= 10
WITH a, acted
MATCH (a:Actor:Director)-[:DIRECTED]->(m:Movie)
WITH a, acted, collect(m.title) AS directed
WHERE length(directed) >= 2
RETURN a.name, acted, directed
ORDER BY length(directed) DESC, acted DESC;
```

~70ms with warm cache

Bonus optimization #2: Using the lowest cardinality label, :Director.

```
MATCH (a:Director)-[:ACTED_IN]->(m)
WITH a, count(m) AS acted
WHERE acted >= 10
WITH a, acted
MATCH (a)-[:DIRECTED]->(m)
WITH a, acted, collect(m.title) AS directed
WHERE length(directed) >= 2
RETURN a.name, acted, directed
ORDER BY length(directed) DESC, acted DESC;
```

~50ms with warm cache

Lunch?

Cypher Query Planning

Cypher query planners

- Currently two query planners
  - (old) rule planner
  - (new) cost planner

The rule planner

- This is the original planner
- It consists of rules that use the indexes to produce the query execution plan
- All **write** queries only use the Rule planner

The rule planner

- This is the original planner
- It consists of rules that use the indexes to produce the query execution plan
- All **write** queries only use the Rule planner

The cost planner

- This is our new, cost-based planner
- Introduced in 2.2.0
- It uses the statistics service in Neo4j to assign costs to various query execution plans, picking the cheapest one
- All read-only queries use this by default

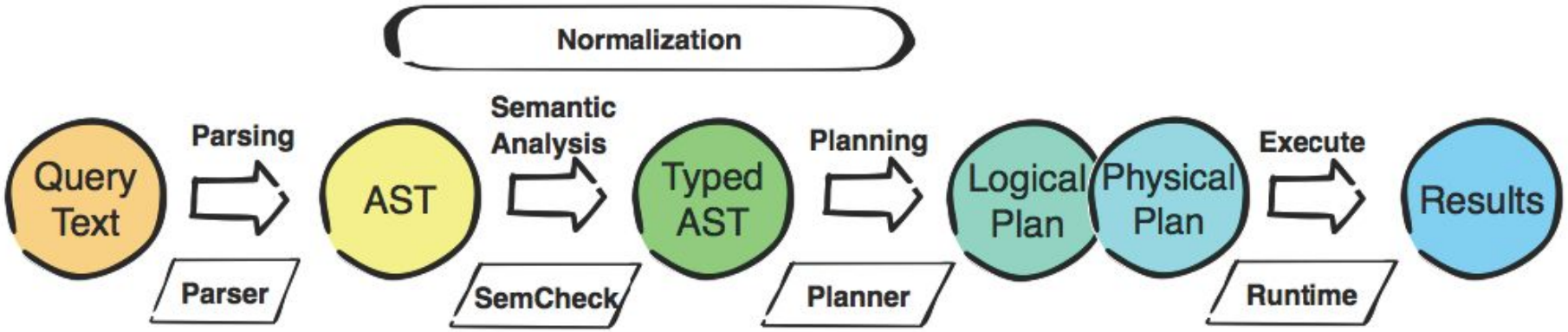
Configuring a planner

- Read-only queries can still be run with the rule planner
  - Prepend query with

    ```
    CYPHER planner=rule
    ```

  - Set `dbms.cypher.planner` to `RULE`

Cypher Query Execution



- http://neo4j.com/docs/snapshot/execution-plans.html
- http://neo4j.com/blog/introducing-new-cypher-query-optimizer

How do I view a query plan?

- EXPLAIN
  - shows the execution plan without actually executing it or returning any results.
- PROFILE
  - executes the statement and returns the results along with profiling information.

Compiling/query cache/parameters

- the Cypher compiler has been getting faster with each release
- compiling still takes time: for best performance, make sure your queries can be cached
- to be able to be cached, the query string must match exactly
- parameters can be used in most cases, for dynamic data

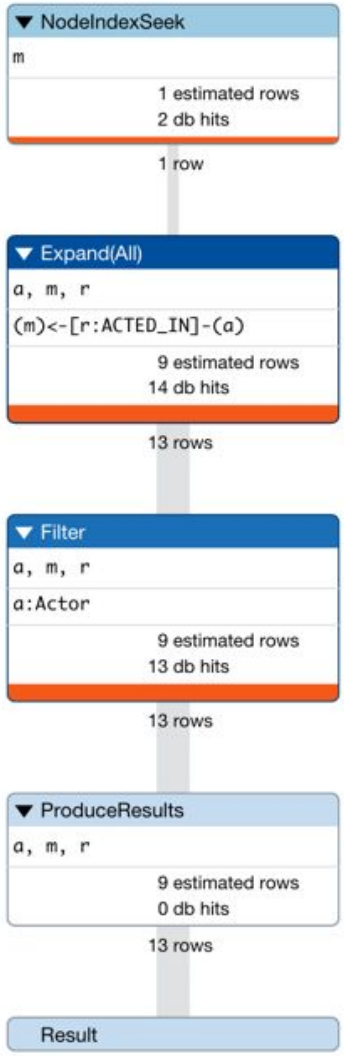
PROFILE

- Use `PROFILE` to ensure that your queries are using indexes as you expect
- `PROFILE` is useful to see the execution plan, and can also be used for performance optimization
- plans are in reverse, think of it as a pipe going step by step from inner to outer

```
PROFILE
MATCH (a:Actor)-[r:ACTED_IN]-(m:Movie)
WHERE m.title="The Matrix"
RETURN *;
```

PROFILE

1. NodeIndexSeek to find the Movie by title
2. Expand the pattern to find the connected Actors
3. Filter by :Actor label



Lazy

- lazy as much as possible
- iterators avoid keeping too much in memory
- things that aren't lazy (or are eager):
  - aggregations
  - `ORDER BY`
  - mixing writing and reading

Operators

- Operators to look out for
  - All nodes scan — expensive
  - Label scan — cheaper
  - Node index seek — cheapest
  - Node index scan — used for range queries

Lab

- let's run a PROFILE on this query:

```
PROFILE
MATCH (a:Actor)-[r:ACTED_IN]-(m:Movie)
WHERE a.name="Keanu Reeves"
RETURN *;
```

Lab

- Do you see anything odd about the profile output?

Lab: Create indexes

- Create indexes for person names and movie titles

Lab: Create indexes

- Create indexes for person names and movie titles:

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

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

Lab: Cypher internals

- try this one, `(a:Person)` instead of `(a:Actor)`:

```
PROFILE
MATCH (a:Person)-[r:ACTED_IN]-(m:Movie)
WHERE a.name="Keanu Reeves"
RETURN *;
```

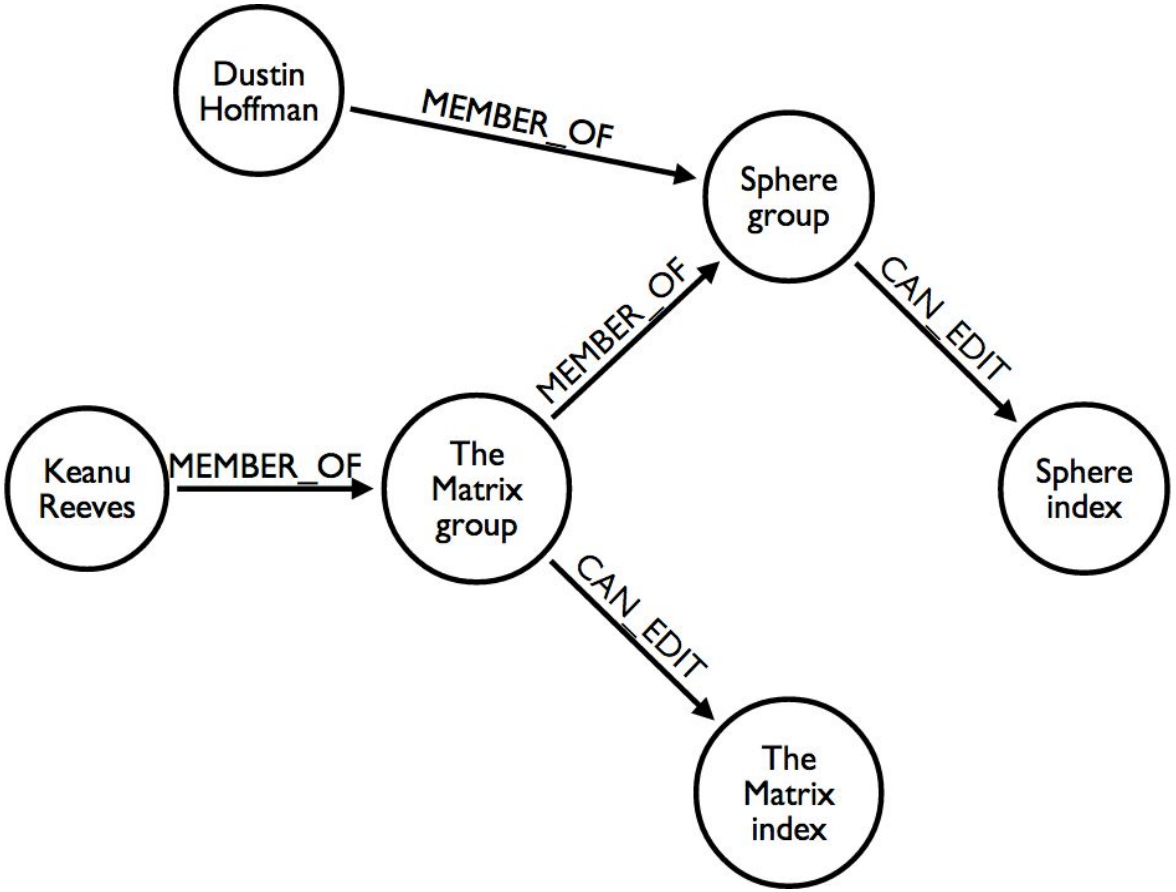
Cypher: Trees and Hierarchies

Cypher to deal with trees/hierarchies

- basic use case: access control
- queries you might need to do:
  - find whether a node is a child of a parent - can person X edit thing Y?
  - Most Recent Common Ancestor (MRCA) - what's the smallest group X people are part of?
  - find leaf nodes beneath a parent - what items are editable by this group?
- Cypher paths are your friends

First, let's build a quick hierarchy

- let's make groups out of our movie casts:



First, let's build a quick hierarchy

- let's make groups out of our movie casts:

```
CREATE INDEX ON :Page(name);
CREATE INDEX ON :Group(name);

MATCH (m:Movie)
MERGE (g:Group {name:m.title + " group", id:m.
id})
MERGE (p:Page {name:m.title + " index"})
MERGE (p)<-[:CAN_EDIT]-(g)
WITH p,m,g
MATCH (m)<-[:ACTED_IN]-(a)
MERGE (g)<-[:MEMBER_OF]-(a);
```

First, let's build a quick hierarchy

- now, let's link some groups to other groups (make them members), to create a hierarchy.

```
MATCH (g1:Group)
WHERE g1.id < 1000
WITH g1
MATCH (g2:Group)
WHERE g1.id + 9550 = g2.id
  AND NOT (g2)-[:MEMBER_OF*]->(g1)  // prevent cycles
MERGE (g1)-[:MEMBER_OF]->(g2);
```

Lab: access control queries

1. Can Keanu Reeves edit the Sphere index? Pages look like:
`(:Page {name:"Sphere index"})`

2. What pages are editable by "The Matrix group" `:Group`?

3. What is the smallest group Dustin Hoffman and Keanu Reeves are part of? Groups look like:
`(:Group {name: "The Matrix group"})`

Can Keanu Reeves access the Sphere index?

```
MATCH (a:Person)-[:MEMBER_OF*]->(group),
      (group)-[:CAN_EDIT]->(p:Page)
WHERE a.name="Keanu Reeves"
  AND p.name="Sphere index"
RETURN CASE
         WHEN count(*) = 0 THEN FALSE
         ELSE TRUE
       END as has_access;
```

Optimization: Can Keanu Reeves access the Sphere index?

```
MATCH (a:Person)-[:MEMBER_OF*]->(team),
      (team)-[:CAN_EDIT]->(p:Page)
WHERE a.name="Keanu Reeves"
  AND p.name="Sphere index"
WITH p
LIMIT 1 // short circuit after 1 is found
RETURN CASE
         WHEN count(*) = 0 THEN FALSE
         ELSE TRUE
       END as has_access;
```

Optimization #2: Can Keanu Reeves access the Sphere index?

- use shortestPath to efficiently find the shortest path between two nodes (this is even faster than finding the first path; sometimes significantly faster)

```
MATCH
  shortestPath((a:Person)-[:MEMBER_OF|CAN_EDIT*]->(t:Page))
WHERE a.name="Keanu Reeves"
  AND t.name="Sphere index"
RETURN CASE
         WHEN count(*) = 0 THEN FALSE
         ELSE TRUE
       END as has_access;
```

What pages are editable by the "The Matrix group" :Group?

```
MATCH (g:Group)-[:MEMBER_OF|CAN_EDIT*]->(p:Page)
WHERE g.name="The Matrix group"
RETURN p;
```

Smallest group Dustin Hoffman and Keanu Reeves are part of?

```
MATCH
 p=(k:Person)-[:MEMBER_OF*]->(g)<-[:MEMBER_OF*]-(w:Person)
WHERE k.name="Keanu Reeves"
  AND w.name="Dustin Hoffman"
RETURN g
ORDER BY length(p) ASC
LIMIT 1;
```

Access control/tree/hierarchy processing

- definitely a strong use case for Neo4j; try writing some of these queries in SQL

Coffee?

Cypher to find the missing links

Cypher to find the missing links

- common use cases
  - recommendations
  - NLP (Natural Language Processing)
  - fraud detection

Cypher to find the missing links

- THE recommendation query:

```
MATCH (p:Person)-[:BOUGHT]->(i:Item),
      (i)<-[:BOUGHT]-(otherPeople),
      (otherPeople)-[:BOUGHT]->(otherItem)
WHERE p.id = {personId}
  AND NOT (p)-[:BOUGHT]->(otherItem)
RETURN otherItem, count(*)
ORDER BY count(*) DESC
LIMIT 3;
```

customers who bought this item also bought:

…

How do we apply this to our movie graph?

Lab: find the missing links

1. I really like The Matrix. Recommend 5 other movies to me based on that fact.

2. Recommend 3 directors that Keanu Reeves should work with (and hasn't yet).

3. Not a query, just a question to consider: Is there any way to use our movie genre properties to help with these queries?

Loving the Matrix recommendation

```
MATCH (m:Movie)<-[:ACTED_IN]-(a:Actor),
    (a)-[:ACTED_IN]->(otherMovie)
WHERE m.title="The Matrix"
RETURN otherMovie, count(1) as c
ORDER BY c DESC
LIMIT 5;
```

3 directors for Keanu

```
MATCH (k:Person)-[:ACTED_IN]->(m:Movie),
(m)<-[:ACTED_IN]-(actor),
    (actor)-[:ACTED_IN]->(otherMovie),    (d:Director)-[:
DIRECTED]->(otherMovie)
WHERE k.name="Keanu Reeves"
AND NOT (k)-[:ACTED_IN]->()<-[:DIRECTED]-(d)
AND m.avgVote > 7
AND NOT has(d.died)
RETURN d.name, count(*) as c
ORDER BY c DESC
LIMIT 3;
```

Refactoring...

- We made collections for genres… maybe they should have been nodes (or, additionally should be nodes)

```
MATCH (m:Movie)
UNWIND m.genres as genre
WITH genre, m
MERGE (g:Genre {genre:genre})
MERGE (m)-[:IS_GENRE]->(g);
```

Refactored:
Loving the Matrix recommendation

- match genres; adds new dimension to query

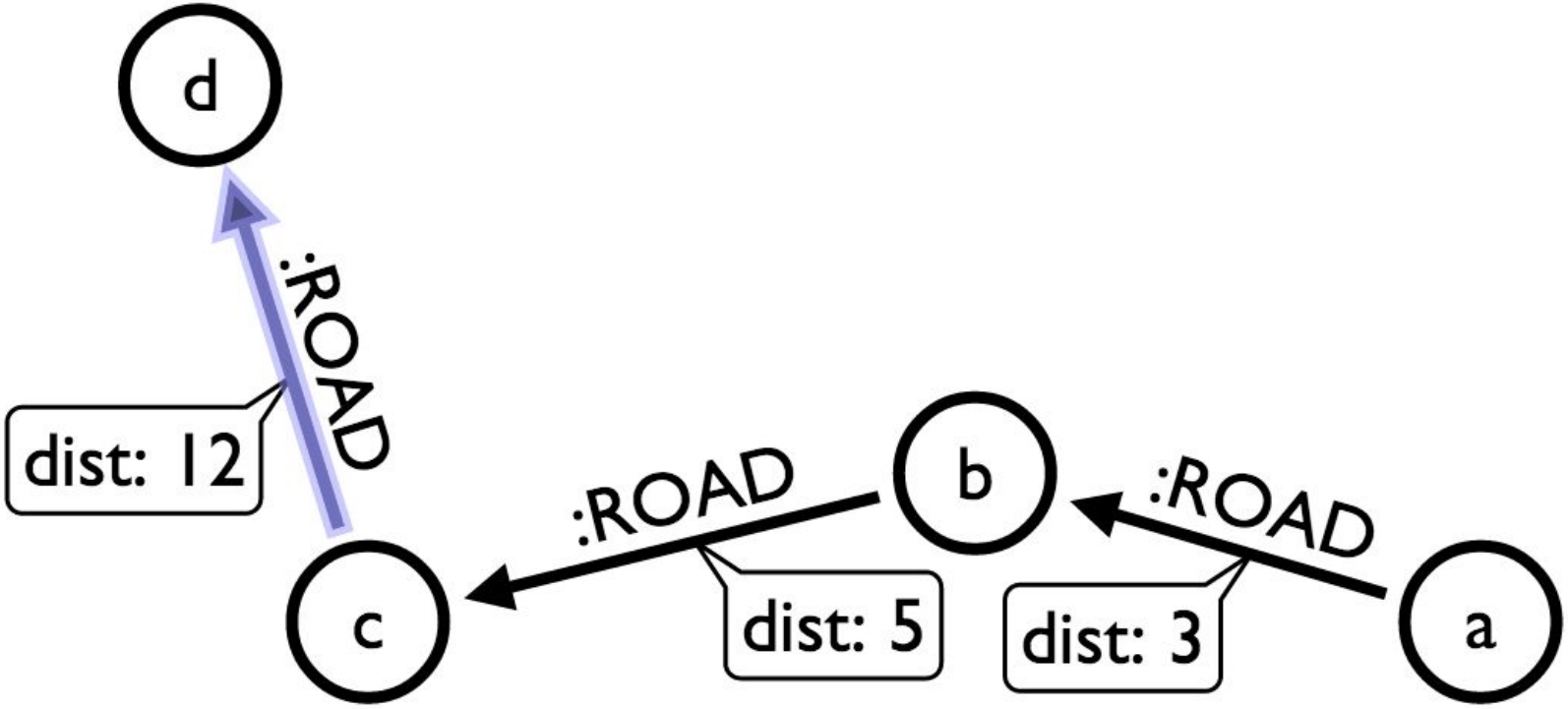
```
MATCH (m:Movie)-[:IS_GENRE]->(g),
    (otherMovie)-[:IS_GENRE]->(g)
WHERE m.title="The Matrix"
WITH otherMovie, count(*) as genres, m
RETURN otherMovie.title,
       genres + m.avgVote * 2 as score
ORDER BY score DESC
LIMIT 5;
```

Quick aside: Cypher shortest paths

- shortest path by rels: shortestPath
  - uses graph-friendly algorithms
  - we've seen a couple of these already
- shortest path by weighted rels/nodes: reduce
  - find every path, calculate weight, order by lowest weight

Weighted shortest paths via reduce

- calculate distance of the path



Weighted shortest paths via reduce

```
CREATE ({n:"a"})-[:ROAD {dist:3}]->({n:"b"})-[:
ROAD {dist:5}]->({n:"c"})-[:ROAD {dist:12}]->
({n:"d"});
```

```
MATCH p=(n)-[r:ROAD*]->(m) // find all paths
RETURN p, reduce(acc=0, x in r |
       acc + x.dist) as totalDistInPath;
```

Cypher collections

Cypher collections

- first class citizens in Cypher's type system
- can hold nested collections in Cypher (node/rel properties can't)
- collection predicates:
  `IN, SOME, ALL, SINGLE`
- slice notation[1..3]

Lab: Getting comfortable with collections

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

Lab: Getting comfortable with collections

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

Lab: Getting comfortable with collections; actors in top 5 movies

```
MATCH (m:Movie)
WITH m
ORDER BY m.avgVote DESC
LIMIT 5
MATCH (a:Actor)-[:ACTED_IN]->(m)
RETURN m.title, collect(a.name)
```

Actors in top 5 movies and their movies

```
MATCH (m:Movie)
WITH m
ORDER BY m.avgVote DESC
LIMIT 5
MATCH (a:Actor)-[:ACTED_IN]->(m)
WITH distinct a
MATCH (a)-[:ACTED_IN]->(m:Movie)
RETURN a.name,
       collect(m.title) as movies;
```

Cypher optimization techniques

Measure

- worth mentioning again: use parameters!
- `PROFILE` to see if you're using indexes as expected, or to see if there's a better label to scan
- use representative test data, time queries with cold and warm cache

Strategy

- let cypher be lazy--avoid aggregations and sorts that force it to not be lazy, if possible
- start with the smallest result set you need to use to filter or aggregate
- expand as needed to gather more data

Strategy

- sometimes a graph refactoring can help your performance
  - aggregate duplicate relationships--add a count property instead of leaving them as separate rels
    - number of times a user has watched/listened to something
- labels to nodes, or nodes to labels (depending on what you need to query)

Keep in mind

- match can easily generate cartesian products
- match expands a pattern quickly (rels^steps), e.g. expanding 1000 rels per node makes 1M in 2 steps and 1BN in 3 steps
- and each predicate has to be evaluated for each of those subgraphs found
- `RETURN`/`WITH` only what you need; it will improve query times

If you can't get your Cypher fast enough

- just wait a bit (if you can): future versions will continue to be a good increase for Cypher performance
- last resort: use unmanaged extensions with the Java API

Case study #1

- very BAD:

```
MATCH (a:Actor), (m:Movie)
RETURN count(a), count(m);
```

- cartesian product generated in MATCH
- count doesn't work (counts cartesian product)
- count distinct works, but is very slow (cartesian product)

Case study #1

- rewrite:

```
MATCH (a:Actor)
WITH count(a) as a_count
MATCH (m:Movie)
RETURN a_count, count(m);
```

- no cartesian product, no distinct necessary

Case study #2

- Cypher compiler: ability to use index…

- bad:

```
MATCH (p:Person)
WHERE p.age + 30 = 34
```

- good:

```
MATCH (p:Person)
WHERE p.age = 4
```

Resources for Cypher

http://stackoverflow.com/questions/tagged/cypher

http://groups.google.com/forum/#!forum/neo4j

http://graphgist.neo4j.com/

http://docs.neo4j.org/refcard/2.1

http://docs.neo4j.org/chunked/stable/data-modeling-examples.html

http://www.markhneedham.com/blog/category/neo4j/

http://maxdemarzi.com/tag/cypher/

http://nicolewhite.github.io/

http://wes.skeweredrook.com/cypher/

http://intelliwareness.org/

http://jexp.de/blog/2013/05/on-importing-data-in-neo4j-blog-series/

http://thought-bytes.blogspot.de/2013/01/optimizing-neo4j-cypher-queries.html

What are your hard queries?