Cypher Refresher

(Just in Case)

Resources for learning Cypher

- Cypher Reference Card
  neo4j.com/docs/cypher-refcard/
- Cypher Railroad Diagrams
  bit.ly/cypher-railroad
- Neo4j Developer Pages
  neo4j.com/developer/cypher
- Neo4j Documentation
  neo4j.com/docs

Cypher Query Structure

```
MATCH pattern
WHERE predicate
RETURN/WITH expression AS alias ...
ORDER BY expression
SKIP ... LIMIT ...
```

Case sensitivity

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

Labels

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

```
CREATE (p:Person)// create labeled node
SET p:Person     // set label on node
REMOVE p:Person  // remove label

MATCH (p:Person) // match nodes with label
WHERE p:Person   // label predicate
RETURN labels(p) // get label collection
```

MATCH

MATCH patterns

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

MATCH examples

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

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

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

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

WHERE

WHERE

- use MATCH to find patterns, and WHERE to filter them
- use patterns as predicates in WHERE

  e.g. `WHERE NOT EXISTS ((n)-->())`
- can't create new identifiers in WHERE - only predicates on existing identifiers defined in MATCH or WITH

  e.g. we can’t do `WHERE NOT EXISTS ((n)-->(newIdentifier))`

WHERE examples

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

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

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

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

RETURN

RETURN

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

RETURN examples

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

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

ORDER BY/LIMIT/SKIP

ORDER BY/SKIP/LIMIT

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

ORDER BY/SKIP/LIMIT examples


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

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

Data flow using WITH

WITH

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

WITH examples

```
// intermediate projection, ordering, pagination
WITH a.name as name, a.born as born
ORDER BY born DESC
LIMIT 5
...

// aggregation + filter
WITH m, count(*) as cast, collect(a.name) as actors
WHERE cast > 5
RETURN m.title, actors
```

Exercise: Translate English to Cypher

:play movies

Open your browser to http://localhost:7474 and execute the following command:

```
:play movies
```

Task: Complex Graph Operation

Find all Actors and Movies they acted in

Whose name contains the letter "a"

Aggregate the frequency and movie titles

Filter by who acted in more than 5 movies

Return their name, birth year and movie titles

Ordered by number of movies

Limited to top 10

SQL version of the query

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

Exercise: Write and execute the query

Find all Actors and Movies they acted in

Whose name starts with "a"

Aggregate the frequency and movie titles

Filter by who acted in more than 5 movies

Return their name, birth year and movie titles

Ordered by number of movies

Limited to top 10

Write the query one step at a time and don't forget to use the Cypher refcard if you get stuck: neo4j.com/docs/cypher-refcard

Solution on next slide

Solution: Complex Graph Query

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

Breakdown

MATCH
describes the pattern

MATCH the pattern

```
MATCH (a:Person)
RETURN a
LIMIT 10
```

WHERE
filters the result set

Filter using WHERE

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

MATCH
describes the pattern

MATCH the pattern

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

RETURN
returns the results

RETURN the results

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

Aggregation
with auto grouping

Aggregation

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

ORDER BY / LIMIT / SKIP
Sort and paginate

ORDER BY LIMIT - Paginate

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

WITH + WHERE computes intermediate results + filter

WITH + WHERE - filter

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

Solution

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

Quick Review of Update Operations

CREATE
creates nodes, relationships, and patterns

CREATE nodes, relationships, structures


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

MERGE
matches or creates

MERGE - get or create

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

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

Dense node merging and matching

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

SET, REMOVE
update attributes and labels

SET

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

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

REMOVE

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

SET + REMOVE

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

DELETE
remove nodes and relationships

DELETE

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

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

DETACH DELETE

- Delete node + relationships attached to it

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

Delete everything in the database

- Delete node + relationships attached to it

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

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

End of Cypher Refresher

Questions?

Data import with Cypher

Clean the database

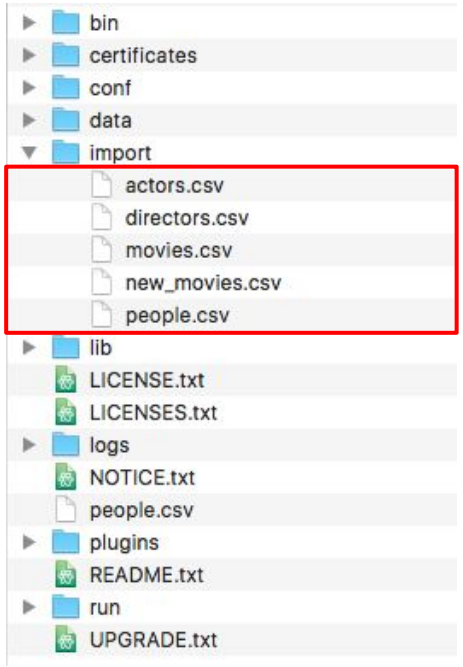
Before the next section we need to clean the database. Run the following query:

```
MATCH (n)
DETACH DELETE n
```

Prepare your Neo4j graph.db directory

Copy the import folder
From the **USB Stick**
to the **`default.graphdb`** folder
(or **`$NEO4J_HOME`**)

After you've done this restart Neo4j



Importing data with Cypher

As a movie enthusiast

I want to import IMDB data into neo4j as quickly as possible

So that I can start writing queries about my favorite movie stars

Load CSV

A clause in the Cypher query language that lets us iterate over CSV files and create graph structures based on the data contained in each row.

Load CSV

```
LOAD CSV      // load csv data
WITH HEADERS // optionally use first header row as keys in "row" map
FROM "url"    // file:// URL relative to $NEO4J_HOME/import or http://
AS row        // return each row of the CSV as list of strings or map
// ... rest of the Cypher statement ...
```

Load CSV

```
[USING PERIODIC COMMIT] // optionally batch transactions
LOAD CSV      // load csv data
WITH HEADERS // optionally use first header row as keys in "row" map
FROM "url"    // file:// URL relative to $NEO4J_HOME/import or http://
AS row        // return each row of the CSV as list of strings or map
[FIELDTERMINATOR ";"] // optionally alt. delimiter
// ... rest of the Cypher statement ...
```

Exploring the data

```
$ ls -lh import/*.csv
1.3M 21 Sep 10:03 actors.csv
 19M 21 Sep 10:03 dict.csv
 77K 21 Sep 10:03 directors.csv
355K 21 Sep 10:03 new_movies.csv
489K 21 Sep 10:03 people.csv
```

Exploring the data

```
$ wc -l import/*.csv
   57137 actors.csv
    6883 directors.csv
    6232 new_movies.csv
   18727 people.csv
```

Exploring the data

new_movies.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
8271	Disturbia	6.400000	2007	Drama:Mystery:Thriller
230896	Iron Man & Hulk: Heroes United	5.600000	2013	Action:Adventure:Animation

Exploring the data

people.csv

personId	name	birthYear	deathYear
23945	Gérard Pirès	1942	
553509	Helen Reddy	1941	
113934	Susan Flannery	1939	
26706	David Rintoul	1948	
237537	Rita Marley	1946	

Exploring the data

actors.csv

personId	movieId	characters
2295	189	Marv
56731	189	Nancy
16851	189	Dwight
24045	189	Johnny
5916	189	Gail

Exploring the data

directors.csv

personId	movieId
2293	189
2294	189
15218	118340
59521	225574
51851	225886

Data import with Cypher

Open your browser to http://localhost:7474 and execute the following command:

```
:play http://guides.neo4j.com/advanced_cypher
```

End of Module
Data import with Cypher

Advanced Cypher Concepts

UNION

UNION

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

UNION example

```
// note that aliases are the same

MATCH (a:Actor)

RETURN a.name as name

UNION

MATCH (d:Director)

RETURN d.name as name
```

CASE/WHEN

CASE/WHEN

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

CASE/WHEN example

```
// group by age-range

RETURN CASE

 WHEN p.age < 20 THEN 'under 20'

 WHEN p.age < 30 THEN 'twenties'

 ...

 END AS age_group
```

Collections

Cypher collections

- first class citizens in Cypher's type system
- nested collections in Cypher (not in properties)
- collection predicates: `IN`, `SOME`, `ALL`, `SINGLE`
- collection operations: `extract`,`filter`,`reduce`,
- `[x IN list WHERE predicate(x) | expression(x)]`
- slice notation `[1..3]`, `map[key]` access
- clauses: `UNWIND`, `FOREACH`

Collections

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

Exercise: Collections Basics

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

Answers: Collections Basics

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

Fun with collections

```
WITH range(1,9) AS list

WHERE all(x IN list WHERE x < 10)

  AND any(x in [1,3,5] WHERE x IN list)

WITH [x IN list WHERE x % 2 = 0 | x*x ] as squares

UNWIND squares AS s

RETURN s
```

Dynamic property lookup

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

Dynamic property lookup

```
WITH "title" AS key

MATCH (m:Movie)

RETURN m[key]
```

Dynamic property lookup

```
MATCH (movie:Movie)

UNWIND keys(movie) as key

WITH movie, key

WHERE key ENDS WITH "_score"

RETURN avg(movie[key])
```

FOREACH

FOREACH

- iterate over a collection and update the graph

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

FOREACH example

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

FOREACH example - conditional logic

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

UNWIND

UNWIND

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

UNWIND Example

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

UNWIND Example: Post UNION processing

```
MATCH (a:Actor)

RETURN a.name AS name

UNION

MATCH (d:Director)

RETURN d.name AS name

// no means for sort / limit
```

UNWIND Example: Post UNION processing

```
MATCH (a:Actor)

WITH collect(a.name) AS actors

MATCH (d:Director)

WITH actors, collect(d.name) AS directors

UNWIND (actors + directors) AS name

RETURN DISTINCT name

ORDER BY name ASC LIMIT 10
```

INDEXes, CONSTRAINTs

represent optional schema

Indexes Overview

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

Index Example

```
// create and drop an index

CREATE INDEX ON :Director(name);

DROP INDEX ON :Director(name);
```

Index Example

```
// use an index for a lookup

MATCH (p:Person)

WHERE p.name="Clint Eastwood"

RETURN p;
```

Range Queries

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

Range Queries

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

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

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

Text Search

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

Index Hints: USING SCAN

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

vs

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

Text Search

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

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

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

Index Hints: USING SCAN

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

```
USING SCAN m:Comedy
```

Composite Indexes

Neo4j doesn’t have composite indexes at the moment but we can create a "dummy" property to simulate one.

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

Constraints

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

Constraints

```
CREATE CONSTRAINT ON (p:Person)

ASSERT p.id IS UNIQUE
```

Constraints

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

CREATE CONSTRAINT ON (p:Person)
ASSERT exists(p.name)

CREATE CONSTRAINT ON (:Person)-[r:ACTED_IN]->(:Movie)
ASSERT exists(r.roles)
```

Map Projections and Pattern Comprehensions

(>= Neo4j 3.1)

Map Projections

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

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

Pattern Comprehensions

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


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

End of Module
Advanced Cypher Concept

Questions?

Cypher Query Tuning

My query is too slow!
I can't change the model
I need to tune!
But how?

What are we going to learn?

- Stepwise query tuning
- How the Cypher query planner works
- Query Profiling
- Operation Costs, Cardinalities and Rows
- Batching Transaction
- Index Lookup performance

Steps for Query Tuning

Steps for Query Tuning

1. **Test:** Run slow query with `PROFILE` on representative dataset
2. **Measure:** Examine Query Plan & note hotspots
   a. intermediate rows
   b. db-hits
   c. total runtime
3. **Change: the model/adapt question** or
4. **Tune: the query**
5. **Step by step:** Focus on hotspots, build query up, apply tuning techniques

Let's look at some of the query tuning tools

Query Planning determines how a Query is executed

The Cypher Query Planner

Cypher Query Planner

Historically RULE planner based on heuristics

Since Neo4j 2.2 (greedy) COST planner based on database statistics

Since Neo4j 2.3 (IDP) COST planner based on database statistics

Since Neo4j 3.0 COST planner for write statements

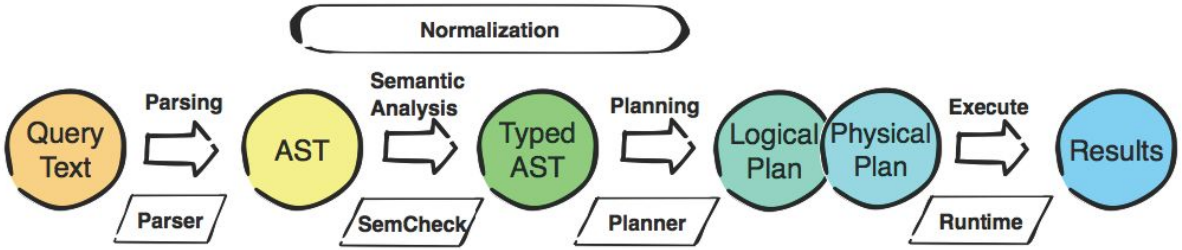
Continuous improvement

- rewriting of the query
- operation improvements
- new operations

Plans are compiled and **cached**

Cypher Query Processing

1. Parse into Abstract Syntax Tree (AST)
2. Semantic Checking
3. Rewrite
4. Logical Planning using transactional DB-Statistics
5. Physical Execution Plan (cached)
6. Execute with interpreted or compiled runtime



neo4j.com/blog/introducing-new-cypher-query-optimizer/

Query Plan Cache

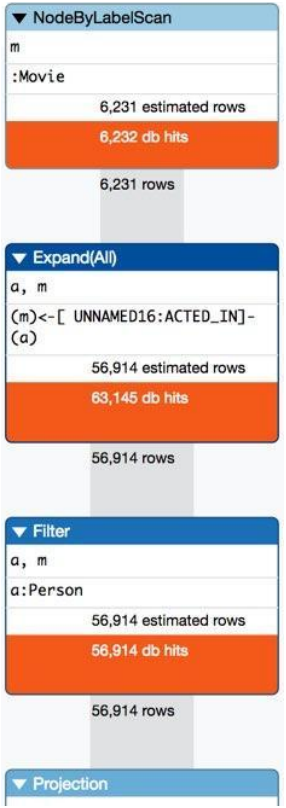
1. Check query string against cache
2. Auto-Parameterize (parse + replace literals with parameters)
3. Check statement string again
4. Parse and Rewrite
5. Check against AST cache
6. Compile
7. Add to cache(s)
8. Execute

To utilize the caches:
Use statements with *parameters* and the *same simple structure*

Execution Plan, AKA Profile

Clauses are decomposed into **multiple** operators which have:

- **input rows**
- **output rows**
- perform database operators, represented by **db hits.**



Profile a query using EXPLAIN

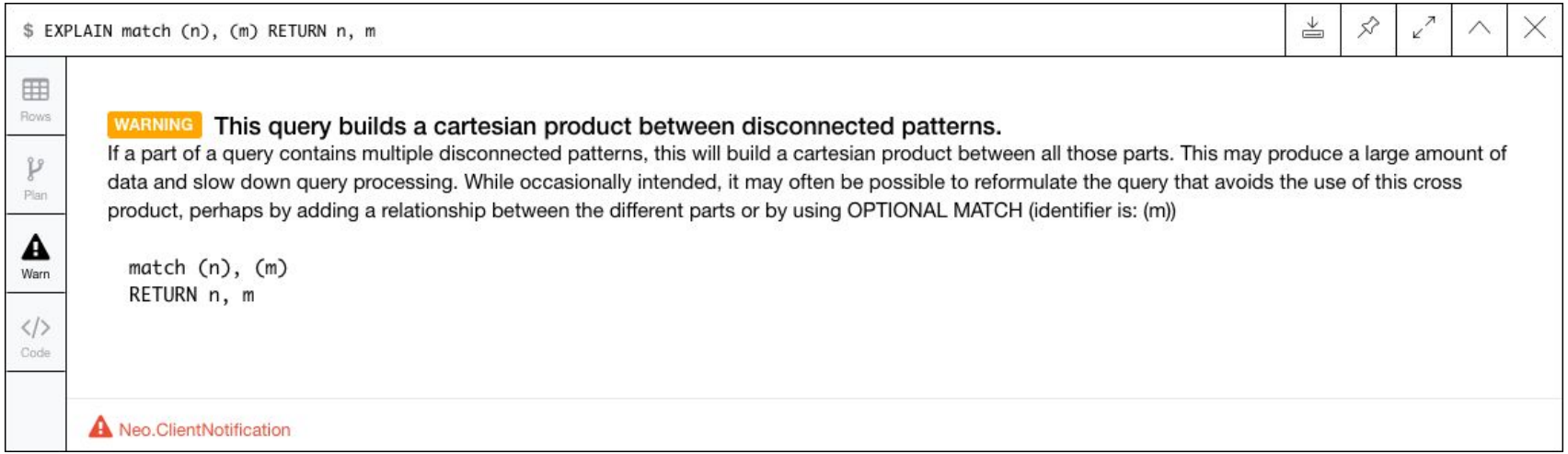
EXPLAIN shows the execution plan **without** actually executing it or returning any results.

It also does syntactic and semantic checking and generates warnings, provides **estimated** rows from **database statistics**.

EXPLAIN provides warnings

on cardinalities, unused symbols, eagerness, and more.

Pay Attention to these



Profiling a query: EXPLAIN vs PROFILE

EXPLAIN

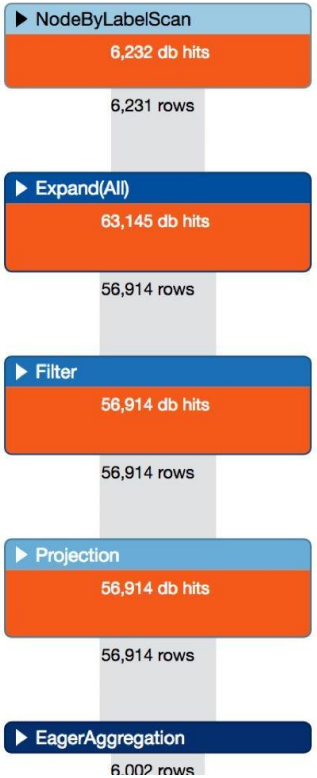
shows the execution plan **without** actually executing it or returning any results. Also does syntactic and semantic checking and generates warnings, provides **estimated** rows from **database statistics**.

PROFILE

executes the statement and returns the results along with profiling information, **actual** rows and **db-hits**

How do I profile a query?

```
PROFILE
MATCH (m:Movie)<-[:ACTED_IN]-(a:Person)
RETURN m.title, count(*) as cast
ORDER BY cast DESC
LIMIT 10
```

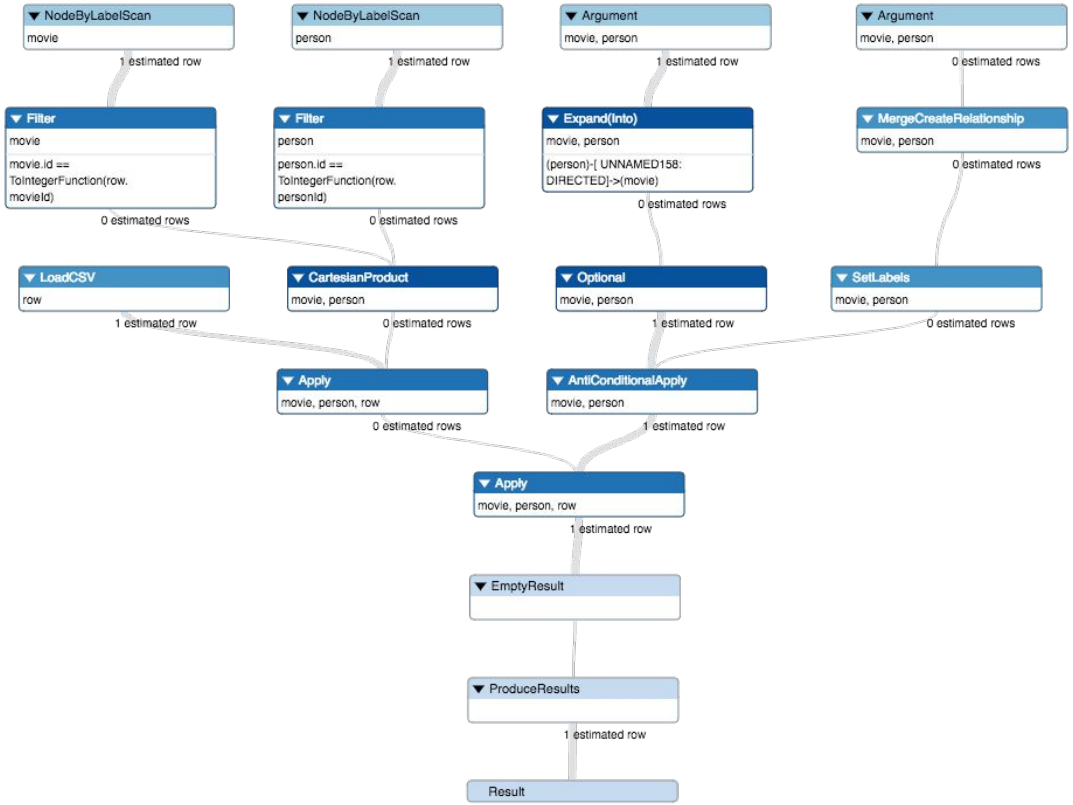



```
Cypher version: CYPHER 3.0, planner: COST, runtime: INTERPRETED.
183205 total db hits in 546 ms.
```

Example profile

```
EXPLAIN
LOAD CSV WITH HEADERS FROM {directors_url} 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
```

Example profile




```
EXPLAIN
LOAD CSV WITH HEADERS FROM {directors_url} 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;
```

The Counts-Store
Transactional Database Statistics

Introducing the Counts Store

Neo4j keeps **transactional database statistics**, which contain cardinalities of various graph entities:

- node counts for labels `(:Label)`
- outgoing / incoming per type and label
- `(:Label)-[:TYPE]->()`
- `()-[:TYPE]->(:Label)`
- Index selectivity 0.0 ... 1.0 `(:Label {property})`

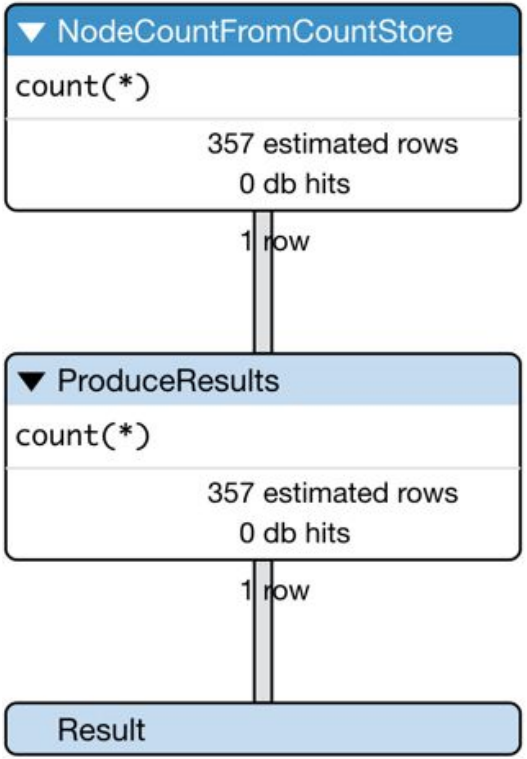
It uses these determine execution plans for queries.

What gets counted?

```
MATCH (n)
MATCH (n:Label)
MATCH ()-[r]->()
MATCH (:Label1)-[r]->()
MATCH ()-[r]->(:Label1)
MATCH ()-[r:X]->()
MATCH (:Label1)-[r:X]->()
MATCH ()<-[r:X]-(:Label1)
```

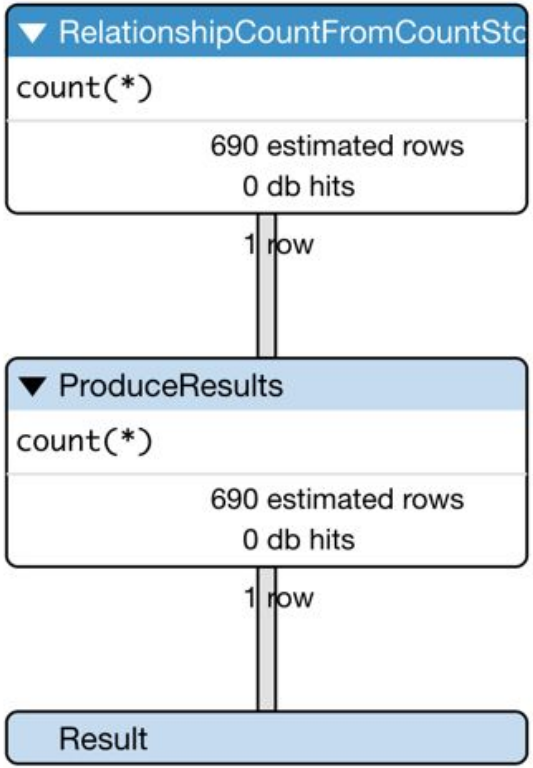
NodeCountFromCountStore

```
MATCH (n)
RETURN count(*)

MATCH (n)
RETURN count(n)

MATCH (n:Label)
RETURN count(*)
```



RelationshipCountFromCountStore

```
MATCH ()-->()
RETURN count(*)

MATCH (:Label)-->()
RETURN count(*)

MATCH ()-->(:Label)
RETURN count(*)
```



Count Store Tricks

Only works with no other aggregation key

Works with `count(*)` and `count(n)`

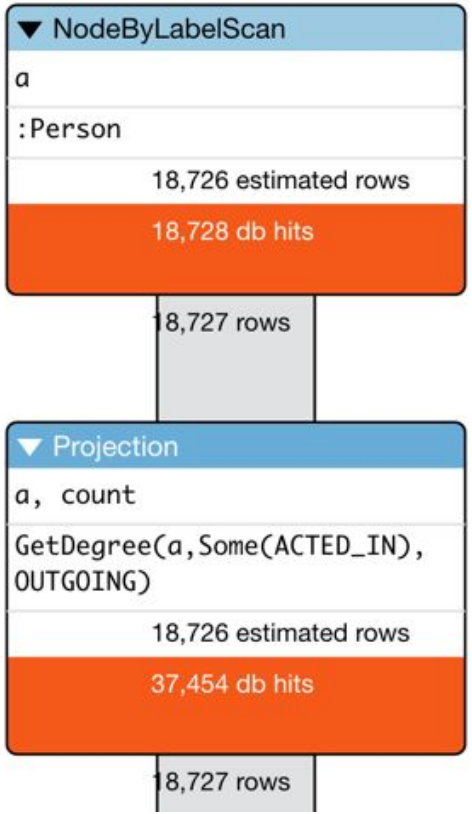
Solution: `UNION ALL` of Maps

```
MATCH (:Person) RETURN {person:count(*)} as count
UNION ALL
MATCH (:Movie)  RETURN {movie:count(*)} as count
```

Node Degree - GetDegree

Neo4j stores the degrees of dense nodes in the rel-group-store.

```
MATCH (a:Person)
RETURN a,
       size( (a)-[:ACTED_IN]->() ) as count
ORDER BY count DESC
LIMIT 10
```



The Counts-Store
Try it out:

```
:play
```

```
http://guides.neo4j.com/advanced_cypher
```

Query Tuning Goal

What's our goal?

At a high level, the goal is simple: get the numbers down

How?

- database hits
- rows
- runtime

Keep it lazy.

What is a database hit?

neo4j

> “an abstract unit of storage engine work.”

Operators cause database hits

Operators cause db-hits

Each operator generates db hits, they are **multiplied by input rows**

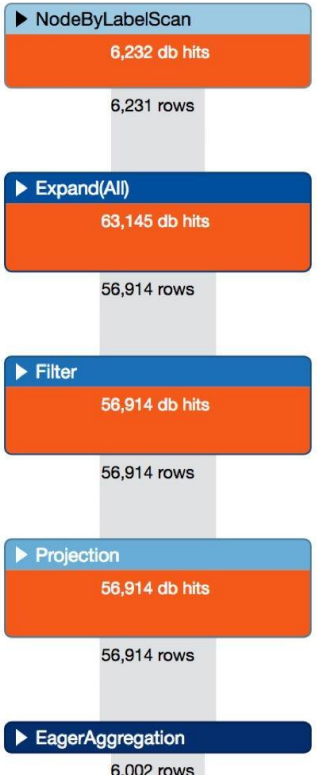
Some operators have a lower cost.

e.g.

- Exchange Expand(All) with **Expand(Into)**
- Exchange Expand(All) + Count with **GetDegree()**
- Exchange Aggregation on unique property with Aggregation on **entity**
- Remove label check if relationship-type is specific enough

How do I profile a query?

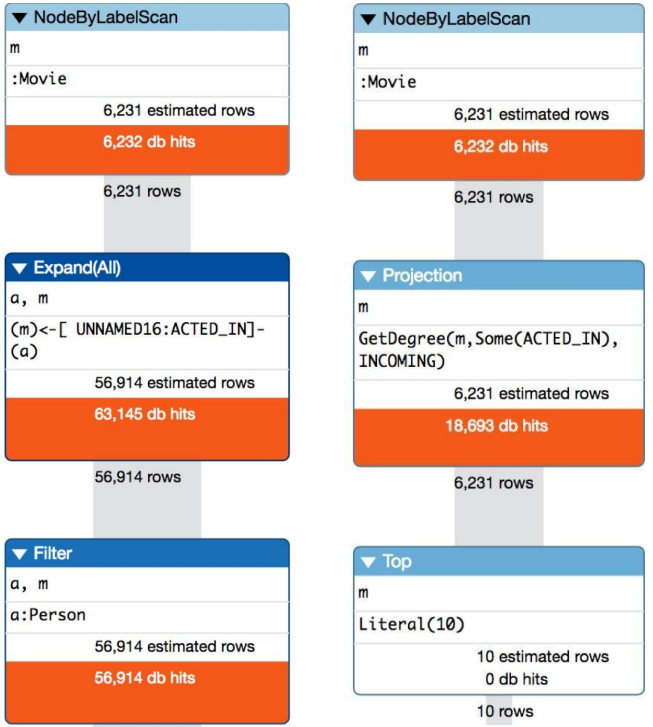
```
PROFILE
MATCH (m:Movie)<-[:ACTED_IN]-(a:Person)
RETURN m.title, count(*) as cast
ORDER BY cast DESC
LIMIT 10
```




```
Cypher version: CYPHER 3.0, planner: COST, runtime: INTERPRETED.
183205 total db hits in 546 ms.
```

Example: Expand -> GetDegree

```
PROFILE
MATCH (m:Movie)
RETURN m.title,
       size( (m)<-[:ACTED_IN]-() )
       AS cast
ORDER BY cast DESC
LIMIT 10
```




```
Cypher version: CYPHER 3.0, planner: COST,
runtime: INTERPRETED. 24925 total db hits in 207 ms.
```

Rows (Cardinalities)

Controlling Cardinalities

Input rows into an operator
= **times** the operator is executed
= multiplies the number of db-hits and individual **output rows**

Getting output rows down, reduces input rows for next operator

.g. you're only interested in distinct results, not number of paths,
get the data/rows down to distinct in-between values.

Or get the number of times a index lookup is done down.
Or use the label or index with the higher selectivity to drive the query.

Example: Cardinalities - Cross Product

Avoid Cross Products like this

```
MATCH (m:Movie), (p:Person)
RETURN count(distinct m), count(distinct p)
```

Example: Cardinalities - Cross Product

Avoid Cross Products like this

```
MATCH (m:Movie), (p:Person)
RETURN count(distinct m), count(distinct p)
```

Instead do one MATCH at a time

```
MATCH (m:Movie) WITH count(*) AS movies
MATCH (p:Person) RETURN movies, count(*) AS people
```

Example: Cardinalities - Cross Product

Avoid Cross Products like this

```
MATCH (m:Movie), (p:Person)
RETURN count(distinct m), count(distinct p)
```

Instead do one MATCH at a time

```
MATCH (m:Movie) WITH count(*) AS movies
MATCH (p:Person) RETURN movies, count(*) AS people
```

Or use the count store

```
MATCH (m:Movie) RETURN {movies: count(*)} AS c UNION ALL
MATCH (p:Person) RETURN {people: count(*)} AS c
```

Controlling Cardinalities Example

Assume 100 actors per Movie.
Even if we're just interested in distinct movies, we touch **1M paths**.

```
(p:Person)-[:ACTED_IN]->(m:Movie)<-[:ACTED_IN]-(p2:Person)-[:ACTED_IN]->(m2:Movie)
    1          x100         100         x100          10000        x100        1000000

(p:Person)-[:ACTED_IN]->(:Movie)<-[:ACTED_IN]-(p2)
   1          x100         100         x100       10000
WITH DISTINCT p2
               100
(p2:Person)-[:ACTED_IN]->(m2:Movie)
    100         x100         10000
```

If we keep distinct data in between, we touch **10k paths**, i.e. 100 times less.

Controlling Cardinalities Example

```
PROFILE
MATCH  (p:Person)-[:ACTED_IN]->(m:Movie)
                <-[:ACTED_IN]- (p2:Person)
                 -[:ACTED_IN]->(m2:Movie)
WHERE p.name = "Tom Hanks"

RETURN count(*), count(distinct m2)
```

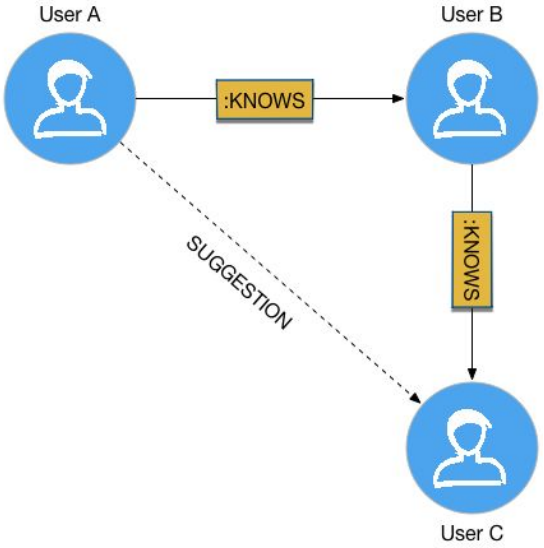
Cypher version: CYPHER 3.0, planner: COST,
runtime: INTERPRETED. **11435 total db hits** in 138 ms.

Controlling Cardinalities Example

```
PROFILE
MATCH  (p:Person)-[:ACTED_IN]->(m:Movie)
                <-[:ACTED_IN]- (p2:Person)

WHERE p.name = "Tom Hanks"

WITH DISTINCT p2

MATCH  (p2)-[:ACTED_IN]->(m2:Movie)
RETURN count(*), count(distinct m2)
```

Cypher version: CYPHER 3.0, planner: COST,
runtime: INTERPRETED. **9671 total db hits** in 60 ms.

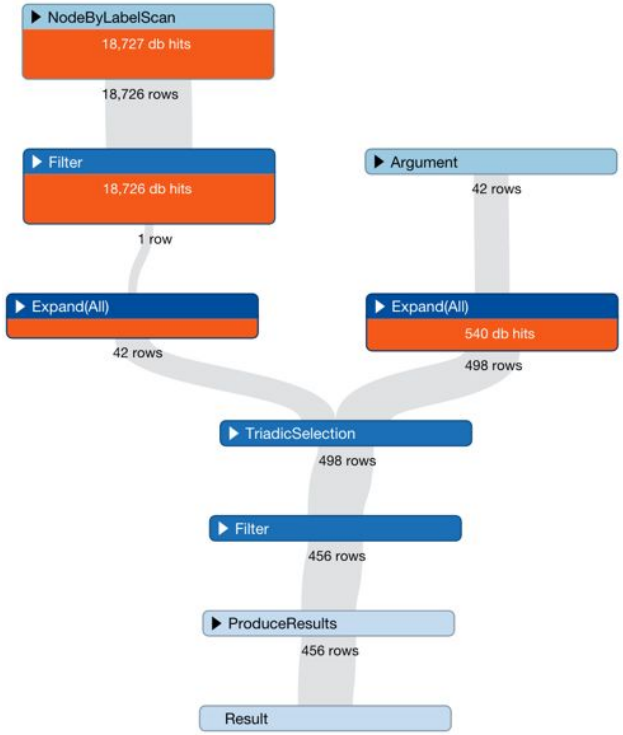
Triadic Selection



- Recommendations form triangles
- Up to 2 steps out
- Check against elements contained in first step

Triadic Selection

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



Caveats:

- need same label
- same relationship-type and direction
- only works on 2-step patterns

Work In Progress (Memory)

Work in Progress

Rows represent the “work in progress“ data, which consumes memory.

Cypher is lazily streaming except when using:

- Sorting
- Aggregation
- Distinct
- Updating (Transaction State)
- Eager

Example

Reduce work in progress by:

- only keep distinct values
- access properties late
- “band filter“ potential inputs (e.g. remove top and bottom 25%)
- use sorting with limit (Top-K window select)
- used index or label with high selectivity to start from

Example

Reduce work in progress by:

- only keep distinct values
- access properties late
- “band filter“ potential inputs (e.g. remove top and bottom 25%)
- use sorting with limit (Top-K window select)
- used index or label with high selectivity to start from

```
MATCH (:Movie) WITH count(*) as movies
MATCH (u:User)
WITH u,
     size( (u)-[:RATED]->() ) as ratings,
     movies * 0.8 as upperBound
WHERE 20 < ratings < upperBound
```

Updating Data (Transaction Size)

Transaction Size

- Neo4j and Cypher are **transactional**
- Isolation requires to keep Transaction-State (delta changes) separate
- Tx-State consumes memory
- *roughly* 1M updates work with 4G heap

- especially critical with:
  - Graph Refactorings
  - Larger Import (Periodic Commit, Eager)
  - Larger Computation / Mass Updates

Transaction Size

- `USING PERIODIC COMMIT` with `LOAD CSV`
- Batch Graph Updates
  - externally
  - with sliding window
  - with pre-condition filter
  - with a batching procedure (`apoc.periodic.iterate`)

Pre-Condition Filter

```
MATCH (m) WHERE not exists (m)-[:GENRE]->()
WITH m LIMIT 10000
UNWIND m.genres as genre
MATCH (g:Genre {name: genre})
CREATE (m)-[:GENRE]->(g)
RETURN count(*);
```

Faster map lookup

We can do even better by populating a map and using that to lookup the genres.

Faster map lookup

```
MATCH (g:Genre)
WITH collect([g.name, g]) AS pairs
CALL apoc.map.fromPairs(pairs) YIELD value AS genres
MATCH (m) WHERE not exists (m)-[:GENRE]->()
WITH genres, m LIMIT 10000
UNWIND m.genres as genre
WITH genres[genre] as g, m
CREATE (m)-[:GENRE]->(g)
RETURN count(*)
```

Mark with Label, Process Batch Window


```
// mark nodes to process
MATCH (m:Movie) SET :Process;

// process marked nodes in batch
MATCH (m:Movie:Process) WITH m LIMIT 10000
REMOVE m:Process // remove mark
UNWIND m.genres as genre
MATCH (g:Genre {name: genre})
CREATE (m)-[:GENRE]->(g)
RETURN count(*);
```

Batching Procedure

```
CALL apoc.periodic.iterate("
  MATCH (m:Movie) RETURN m // driving statement
","
  WITH {m} AS m
  UNWIND m.genres as genre // processing statement
  MATCH (g:Genre {name: genre})
  CREATE (m)-[:GENRE]->(g)
",{batchSize:10000})       // 10k per transaction
```

Selectivity

Selectivity

- Start with the smallest starting set
- Label with fewest members
  - e.g. `:Actor` instead of `:Person`
- Relationship with smallest degree
- Index with highest selectivity

Index Lookups (Performance)

Index Lookups

- For Lookups of individual nodes
- Neo4j uses an index provider (Lucene) for
  - lookups by label
  - lookup by property
  - range scans and text lookups
- Fast for individual lookups
- **Many million lookups** (double for relationship-creation) add up
- Lookups by node and relationship id are very fast, but shouldn‘t be used outside of a narrow scope (e.g. write back computation results)

Tips - Index Lookups

Use `EXPLAIN` and `PROFILE` to see if indexes are used

Provide hints: `USING INDEX n:Label(name)`

Make sure **most selective** index is used

Cypher uses by default **only one index** per pattern.

- Index hints force it to use multiple

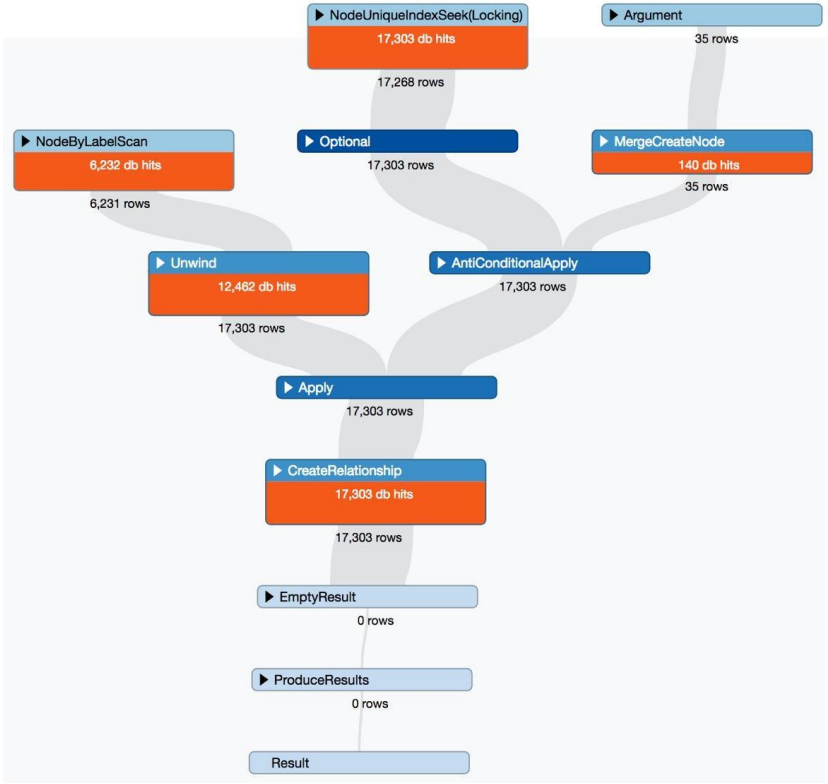
Keep the left side just the **plain indexed property**

- Move all computation to the other side

```
MATCH (g:Genre) WHERE g.name CONTAINS {adjective} + " drama"
```

Index Lookups: Example

```
MATCH (m:Movie)
UNWIND m.genres as genre

// 1 genre lookup per movie-genre
MERGE (g:Genre {name: genre})
CREATE (m)-[:GENRE]->(g);
```



Cypher version: CYPHER 3.0, planner: COST, runtime: INTERPRETED. 53440 total db hits in 1655 ms.

Index Lookups: Example

```
MATCH (m:Movie)
UNWIND m.genres AS genre
WITH genre, collect(m) AS movies
// 1 lookup per genre
MERGE (g:Genre {name: genre})
WITH g, movies
UNWIND movies AS m
CREATE (m)-[:GENRE]->(g);
```

```
Cypher version: CYPHER 3.0,
planner: COST, runtime: INTERPRETED.
36173 total db hits in 2388 ms.
```

Index Lookups: Multi faceted Search

Do **multiple** separate index lookups

Declare the nodes to be the **same**

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

MATCH (m:Movie) WHERE m.title CONTAINS "The"
MATCH (n:Movie) WHERE 1990 < n.releaseYear < 2000 AND n = m
RETURN m;
```

Caches & Warmup

Caches & Warmup

- Configuration: Heap + Page-Cache
  - Page-Cache possibly datastore-size
  - Heap: 8, 16, 32G depending on operations
- Caches keep active dataset available
  - Query Plan Cache
    - Run query once, e.g. with EXPLAIN
    - **Use Parameters**
  - Page-Cache - memory mapping datastore from disk
  - warmup with
    - all nodes / all relationships operation / properties if needed
    - MATCH (n) RETURN max(id(n))
    - apoc.warmup.run() - node & relationship-pages

End of Module
Cypher Query Tuning

User Defined Procedures and Functions

Extending Cypher

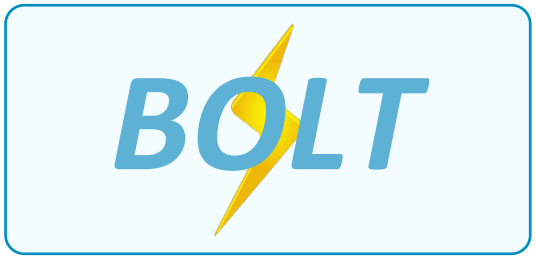
What will we learn?

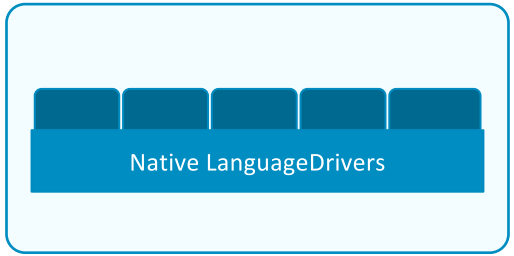
- Why do we need user defined procedures and functions?
- Syntax & built in procedures
- The APOC procedure library
  - Helper functions
  - Data integration
  - Graph algorithms
  - Graph refactoring / Transaction management
- How to write and test your own procedure / function
- Turn Cypher queries into procedures

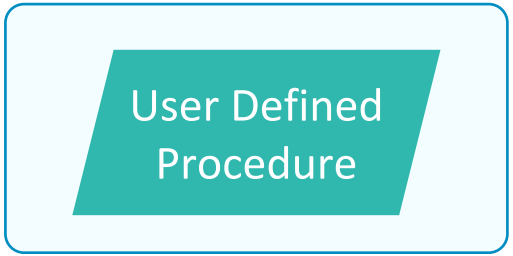
What are User Defined Procedures and Functions?

Neo4j Developer Surface

2000-2010	0.x - Embedded Java API
2010-2014	1.x - REST
2014-2015	2.x - Cypher over HTTP
2016 -	3.0.x - Bolt, Official Language Drivers, and User Defined Procedures 3.1.x - User Defined Functions



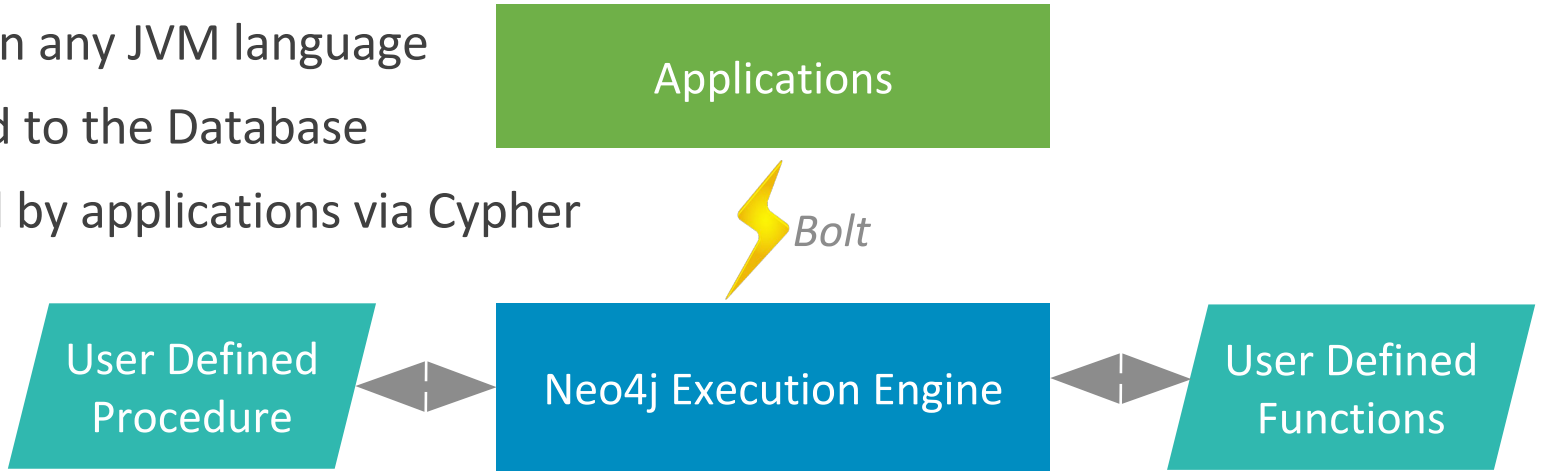




User Defined Procedures & Functions

User Defined Procedures & Functions let you write custom code that is:

- Written in any JVM language
- Deployed to the Database
- Accessed by applications via Cypher



Procedures vs Functions

Procedures	Functions
More complex operations	Simple computations / conversions
Generate streams of results	Return a single value
Used within the **CALL** clause as a stand-alone element and **YIELD** their result columns	Used in any expression or predicate

User Defined Procedures

- written in a JVM Language, e.g. Java, Scala
- deployed into the database,
- and called from Cypher with the CALL clause
- take named parameters and return a stream of columns

http://neo4j.com/docs/developer-manual/current/procedures

Built-in procedures

```
CALL dbms.procedures()
```

$ CALL dbms.procedures()

Rows | Text | Code

name	signature	description	roles
db.awaitIndex	db.awaitIndex(index :: STRING?, timeOutSeconds = 300 :: INTEGER?) :: VOID	Wait for an index to come online (for example: CALL db.awaitIndex(":Person(name)")).	[reader, publisher, architect, admin]
db.constraints	db.constraints() :: (description :: STRING?)	List all constraints in the database.	[reader, publisher, architect, admin]
db.indexes	db.indexes() :: (description :: STRING?, state :: STRING?, type :: STRING?)	List all indexes in the database.	[reader, publisher, architect, admin]
db.labels	db.labels() :: (label :: STRING?)	List all labels in the database.	[reader, publisher, architect, admin]

Started streaming 34 records after 1 ms and completed after 1 ms.

Listing all relationship types

```
CALL db.relationshipTypes()
```

Using User Defined Procedures

Using - User Defined Procedures

```
// shortcut for stand alone call
CALL db.procedures()


// process result columns
CALL db.procedures()

YIELD name, signature, description

RETURN count(name)
```

http://neo4j.com/docs/developer-manual/current/procedures

Exercise: List available procedures

1. List all procedures in the "dbms" namespace
2. List all procedures whose signature contains "NODE"
3. Group procedures by first part of the names (e.g. "db") and return a count of the number of procedures and a collection containing just the last part of the name. (e.g. "awaitIndex")
4. Play around with at least 3 procedures in the list. Which ones did you try?

Listing procedures

```
CALL dbms.procedures()
YIELD name, signature, description
WITH * WHERE name STARTS WITH "dbms."
RETURN *
```

Listing procedures

```
CALL dbms.procedures()
YIELD name, signature, description
WITH * WHERE signature CONTAINS "NODE"
RETURN *
```

Listing procedures

```
CALL dbms.procedures()
YIELD name, signature, description
WITH split(name,".") AS parts
RETURN parts[0] AS package,
       count(*) AS count,
       collect(parts[-1]) AS names
```

Cons of User-Defined Procedures

- Quite involved **`CALL`** clause for simple, read-only computation, conversion functions or predicates
- Need to **`YIELD`** and select result columns
- Can’t be part of expressions

User Defined Functions

User-Defined Functions

- Allows users to create their own functions and use them with Cypher
- Useful for expressing common computations, rules, conversions, predicates
- Functions can be used in any expression, predicates
- Extend the Neo4j 3.0 Stored Procedure mechanism

Creating a UUID with a Procedure vs. with a Function


```
CALL apoc.create.uuid() YIELD uuid
CALL apoc.data.formatDefault(timestamp(), "ms")
YIELD value AS date
CREATE (:Document {id: uuid, created:date})
```

vs

```
CREATE (:Document {
  id: apoc.create.uuid(),
  date: apoc.data.formatDefault(timestamp(), "ms")})
```

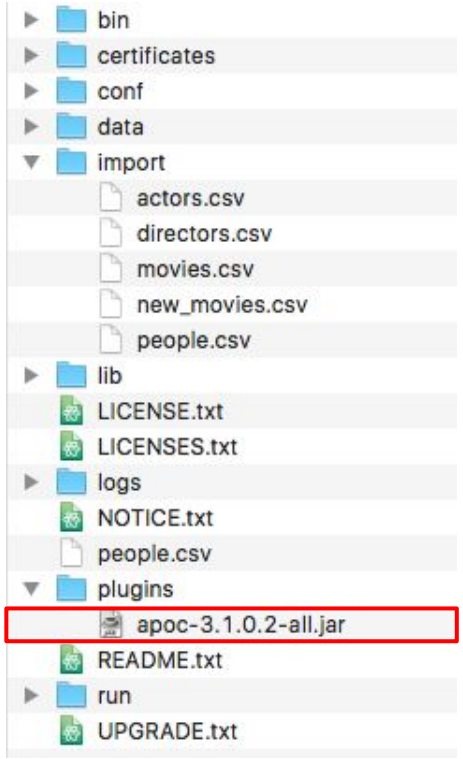
APOC
Awesome Procedures on Cypher

Why and How?

Install APOC

Copy the plugins folder
From the **USB Stick**
to the **`default.graphdb`** folder
(or **`$NEO4J_HOME`**)

After you've done this restart Neo4j



APOC Procedures

- Cypher is expressive and great for graph operations but misses some utilities so people built their own one-off solutions
- APOC is a "Standard Library" of helpful procedures and functions
- Started as an experiment but has evolved into an active community project with 200+ procedures and 75+ functions
- github.com/neo4j-contrib/neo4j-apoc-procedures

What does APOC cover?

- Functions for date, time, data conversion, collection handling
- Procedures for data integration, graph algorithms, graph refactoring, metadata, Cypher batching
- TimeToLive (TTL), triggers, parallelization
- And much more!

Documentation

- documentation site github.com/neo4j-contrib/neo4j-apoc-procedures
- browser guide `:play http://guides.neo4j.com/apoc`
- many articles & blog posts neo4j.com/tag/apoc

User-Defined Functions in APOC - Packages

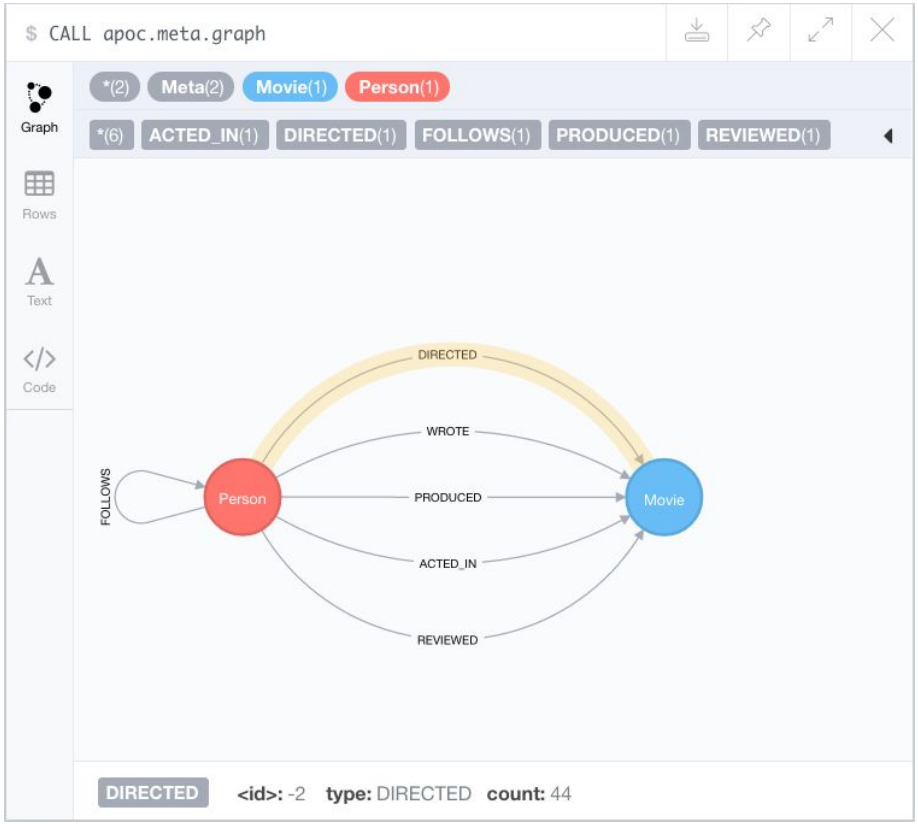
package	# of functions	example function
date & time conversion	3	`apoc.date.parse("time",["unit"],["format"])`
number conversion	3	`apoc.number.parse("number",["format"])`
general type conversion	8	`apoc.convert.toMap(value)`
type information and checking	4	`apoc.meta.type(value)`
collection and map functions	25	`apoc.map.fromList(["k1",v1,"k2",v2,"k3",v3])`
JSON conversion	4	`apoc.convert.toJson(value)`
string functions	7	`apoc.text.join(["s1","s2","s3"],"delim")`
hash functions	2	`apoc.util.md5(value)`

(Meta)-Utilities, Converters

(Meta)-Utilities, Converters

- `apoc.date.format(timestamp(),"ms","YYYY-MM-dd")`
- `apoc.number.parse("12.000,00")`

The Meta Graph



Exercise: Explore APOC functions

Use `dbms.functions()` to:

1. List all functions in the "apoc" namespace
2. Play around with at least 3 other functions. Which ones did you try?

APOC functions

```
RETURN apoc.coll.toSet([1,2,3,4,4,4,4,5,6,7])
```

APOC functions

```
RETURN apoc.text.clean("      Neo4j    ")
```

Data Import and Export

Data Integration / Import

- Databases
  - JDBC
  - MongoDB
  - Cassandra
  - Elastic
  - CouchDB
- File formats
  - JSON
  - XML

LOAD JDBC

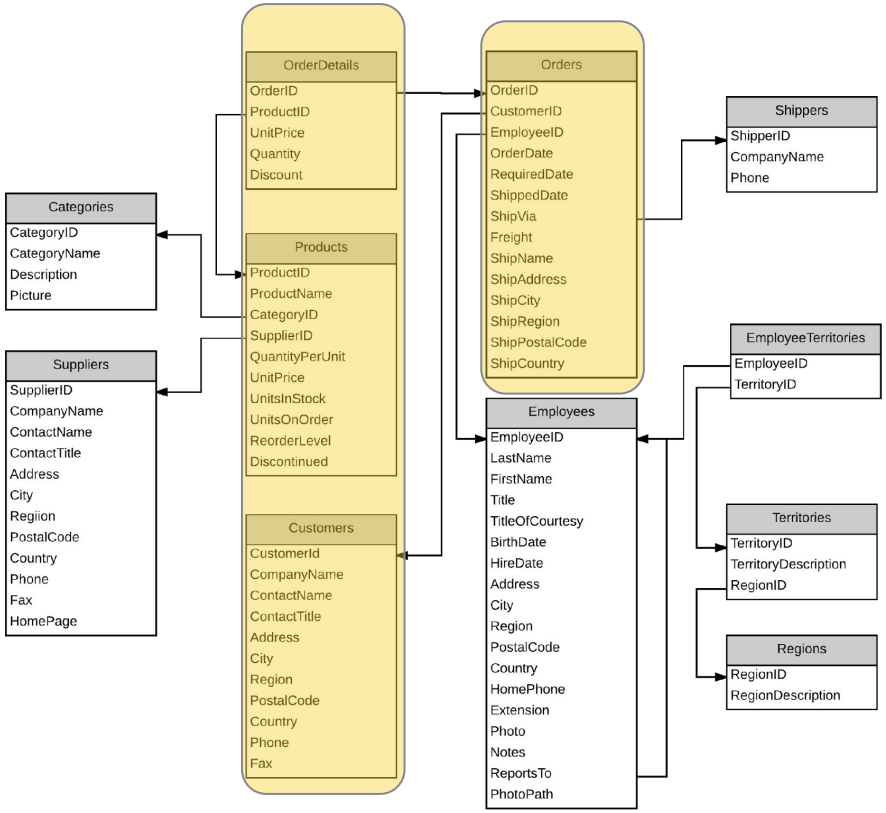
Load from relational database, either a full table or a sql statement

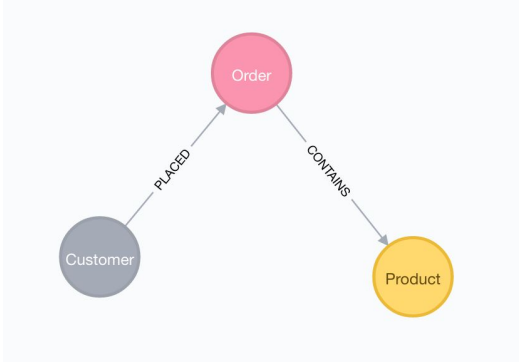
```
CALL apoc.load.jdbc('jdbc-url','TABLE') YIELD row
CALL apoc.load.jdbc('jdbc-url','SQL-STATEMENT') YIELD row
```

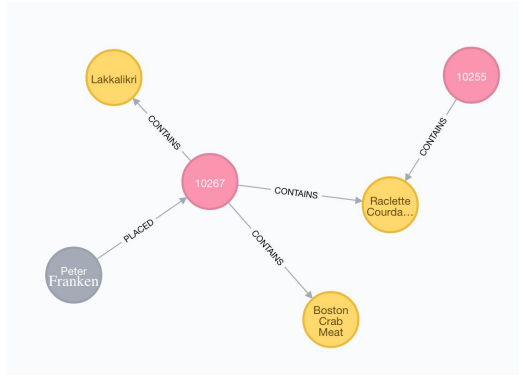
To simplify the JDBC URL syntax and protect credentials, you can configure aliases **in** conf/neo4j.conf:

```
apoc.jdbc.alias.url=jdbc:mysql://localhost:3306/<database>?user=<username>
CALL apoc.load.jdbc('alias','TABLE')
```

Load Northwind data from MySQL







Load Northwind data: Products and Orders

```
// Create Product nodes
cypher CALL apoc.load.jdbc("jdbc:mysql://localhost:3306/northwind?user=root","products")
YIELD row
CREATE (p:Product {ProductID: row.ProductID})
SET p.ProductName = row.ProductName,
    p.CategoryID  = row.CategoryID,
    p.SupplierID  = row.SupplierID
```

```
// Create Order nodes
cypher CALL apoc.load.jdbc("jdbc:mysql://localhost:3306/northwind?user=root","orders")
YIELD row
CREATE (o:Order {OrderID: row.OrderID})
SET o.CustomerID = row.CustomerID,
    o.EmployeeID = row.EmployeeID
```

Load Northwind data: Order Details

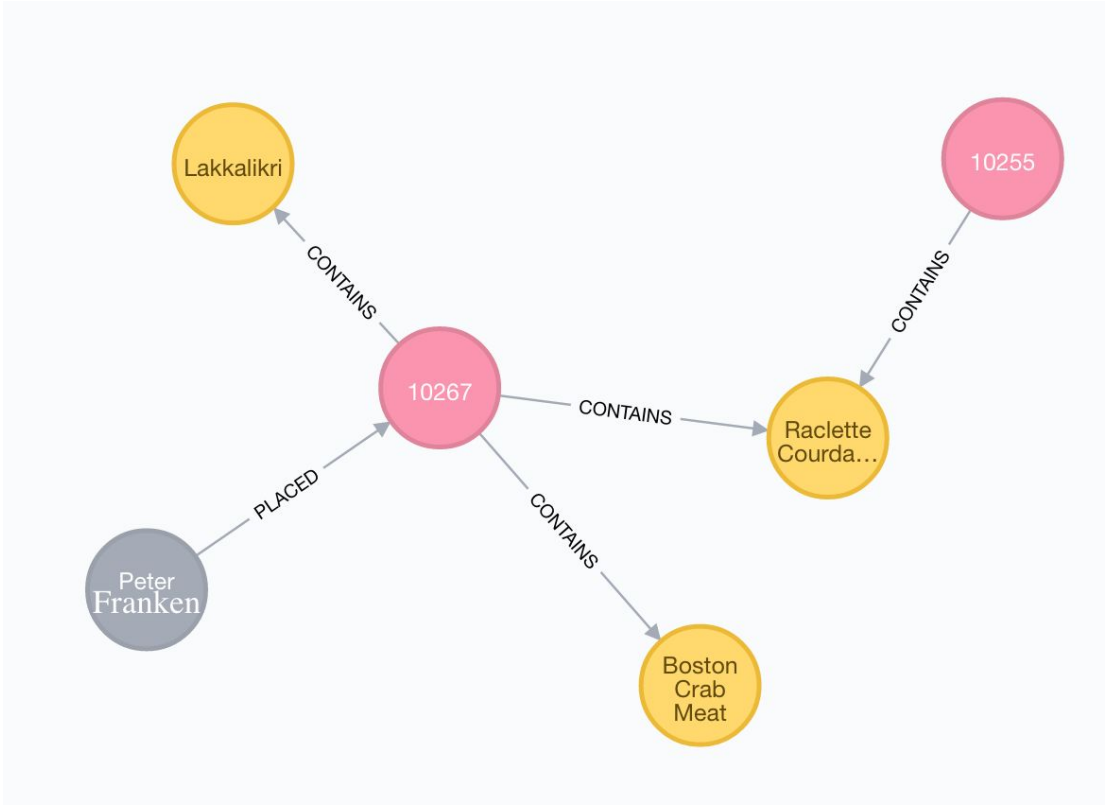
```
// Create CONTAINS relationships
cypher CALL apoc.load.jdbc("jdbc:mysql://localhost:3306/northwind?
user=root","OrderDetails") YIELD row
MATCH (p:Product {ProductID: row.ProductID})
MATCH (o:Order {OrderID: row.OrderID})
CREATE (o)-[r:CONTAINS]->(p)
SET r.UnitPrice = row.UnitPrice
    r.Quantity  = row.Quantity,
    r.Discount  = row.Discount
```

Load Northwind data: Customers

```
// Create Customer nodes
cypher CALL apoc.load.jdbc("jdbc:mysql://localhost:3306/northwind?
user=root","Customers") YIELD row
CREATE (c:Customer {CustomerID: row.CustomerID})
SET c.Companyname  = row.CompanyName,
    c.ContactName  = row.ContactName,
    c.ContactTitle = row.ContactTitle
```

```
// create PLACED relationships
MATCH (o:Order)
MATCH (c:Customer {CustomerID: o.CustomerID})
CREATE (c)-[:PLACED]->(o)
```

Load Northwind data: Graph Result



Product Recommendations

```
// simple collaborative filtering product recommendations
MATCH (c:Customer) WHERE c.ContactName = "Roland Mendel"
MATCH (c)-[:PLACED]->(o:Order)-[:CONTAINS]->(p:Product)
MATCH (p)<-[:CONTAINS]-(:Order)<-[:PLACED]-(other:Customer)
MATCH (other)-[:PLACED]->(:Order)-[:CONTAINS]->(p2:Product)
RETURN p2.ProductName, count(*) AS weight ORDER BY weight DESC
```

p2.ProductName	weight
Gorgonzola Telino	3933
Chang	3527
Guaran Fantstica	3509
Camembert Pierrot	3440
Raclette Courdavault	3396
Flotemysost	3379
Gnocchi di nonna Alice	3307
Jack's New England Clam Chowder	3179
Rhnbru Klosterbier	3151
Alice Mutton	3105
Pavlova	3075
Tarte au sucre	2969
Konbu	2773
Wimmers gute Semmelkndel	2766
Steeleye Stout	2689
Boston Crab Meat	2535

Loading data from JSON

```
// load from JSON URL (e.g. web-api) to import JSON as stream of values if the JSON was an array or a single value if it was a map
CALL apoc.load.json('json-url')
YIELD value as row
```

Loading data from XML

```
// import XML as single nested map with attributes and `_type`, `_text` fields
// and `_<childtype>` collections per child-element-type.
CALL apoc.load.xmlSimple('xml-url')
YIELD value as doc
```

Loading data from CSV

```
// load CSV fom URL as stream of values
CALL apoc.load.csv('csv-url',{config})
YIELD lineNo, list, map
// config contains any of:
{skip: 1,
 limit: 5,
 header: false,
 sep: 'TAB',
 ignore:['tmp'],
 arraySep: ';',
 mapping: {
  years: { type:'int', arraySep:'-', array:false, name:'age', ignore:false}
 }
```

Exercise: Import StackOverflow

Use APOC's **apoc.load.json** to import StackOverflow from this URI:

```
https://api.stackexchange.com/2.2/questions?pagesize=100&tagged=neo4j&site=stackoverflow
```

You can also find it in the documentation:

- neo4j-contrib.github.io/neo4j-apoc-procedures
- Search for "Load JSON StackOverflow Example"

Simple Example:
Write, Test, Use a Function

User Defined Functions

- `@UserFunction` annotated, named Java Methods
  - default name: package + method
- take `@Name`'ed parameters (with default values)
- return a single value
- are read only
- can use `@Context` injected `GraphDatabaseService`
- run within transaction of the Cypher statement

Simple Function (UUID)

```
@UserFunction("create.uuid")
@Description("creates an UUID (universally unique id)")
public String uuid() {
    return UUID.randomUUID().toString();
}
```

Simple Function (UUID)

```
@UserFunction("create.uuid")
@Description("creates an UUID (universally unique id)")
public String uuid() {
    return UUID.randomUUID().toString();
}
```

Calling the function from Cypher:

```
RETURN create.uuid();
CREATE (p:Person {id: create.uuid(), name:{name}})
```

Test Code

1. Deploy & Register in Neo4j Server via **`neo4j-harness`**
2. Call & test via **`neo4j-java-driver`**

https://github.com/neo4j-contrib/training/

Test Code

1. Deploy & Register in Neo4j Server via **neo4j-harness**
2. Call & test via **neo4j-java-driver**

```
@Rule
public Neo4jRule neo4j = new Neo4jRule().withFunction( UUIDs.class );

try( Driver driver = GraphDatabase.driver( neo4j.boltURI() , config ) {
   Session session = driver.session();
   String uuid = session.run("RETURN create.uuid() AS uuid")
                        .single().get( 0 ).asString();
   assertThat( uuid,....);
}
```

Deploying User Defined Code

- Build or download (shadow) jar
- Drop jar-file into `$NEO4J_HOME/plugins`
- Restart server
- Functions & Procedures should be available
- Otherwise check neo4j.log / debug.log

User Defined Procedures

- `@Procedure` annotated, named Java Methods
- additional mode attribute (Read, Write, Dbms)
- return a Stream of value objects with public fields
- value object fields are turned into columns

https://github.com/neo4j-examples/neo4j-procedure-template

Procedure Code Example (Dijkstra)


```
@Procedure
@Description("apoc.algo.dijkstra(startNode, endNode, 'KNOWS', 'distance') YIELD path," +
       " weight - run dijkstra with relationship property name as cost function")
public Stream<WeightedPathResult> dijkstra(
       @Name("startNode") Node startNode,
       @Name("endNode") Node endNode,
       @Name("type") String type,
       @Name("costProperty") String costProperty) {

   PathFinder<WeightedPath> algo = GraphAlgoFactory.dijkstra(
           PathExpanders.forType(RelationshipType.withName(type)),
           costProperty);
   Iterable<WeightedPath> allPaths = algo.findAllPaths(startNode, endNode);
   return Iterables.asCollection(allPaths).stream()
           .map(WeightedPathResult::new);
}
```

Test Procedure

```
@Rule
public Neo4jRule neo4j = new Neo4jRule()
            .withProcedure( Dijkstra.class );
try( Driver driver = GraphDatabase.driver( neo4j.boltURI() , config ) {
   Session session = driver.session();
   String query =
            "MATCH ... CALL apoc.algo.dijkstra(...) YIELD path RETURN ...";
   String pathNames = session.run(query)
                              .single().get( 0 ).asString();
   assertThat( pathNames, ....);
}
```

Periodic Execution & Transaction Control

Large operations? Consider TX batching


```
CALL apoc.periodic.iterate('
  CALL apoc.load.jdbc(
    "jdbc:mysql://localhost:3306/northwind?user=root", "company")',

  'CREATE (p:Person) SET p += value',

  {batchSize:10000, parallel:true})

RETURN batches, total
```

Exercise 2: Import 5 pages of StackOverflow

Use APOC's `apoc.cypher.iterate` with the **apoc.load.json** solution you wrote earlier.

You can also find information about in the documentation:
neo4j-contrib.github.io/neo4j-apoc-procedures

Graph Algorithms

Graph Algorithms

- Pathfinding: `apoc.algo.dijkstra, apoc.algo.aStar`
- PageRank: `apoc.algo.pageRankStats, apoc.algo.pageRankWithCypher`
- Centrality: `apoc.algo.betweenness, apoc.algo.closeness`
- Clustering: `apoc.algo.community, apoc.algo.wcc, apoc.algo.cliques`
- Node-Cover: `apoc.algo.cover`

Example Step-by-Step Optimization

Example Query - Genre Overlap

```
PROFILE
WITH ['Action','Drama','Mystery'] AS genreNames
UNWIND genreNames AS name
MATCH (g:Genre {name:name})<-[:GENRE]-(m:Movie)
WITH m, collect(g) AS genres, genreNames
WHERE size(genres) = size(genreNames)
RETURN m
```

Example Query - Genre Overlap - Optimized

```
PROFILE
WITH ['Action','Drama','Mystery'] AS genreNames
UNWIND genreNames AS name
MATCH (g:Genre {name:name})
WITH g ORDER BY size( (g)<-[:GENRE]-() ) ASC
WITH collect(g) AS genres
WITH head(genres) AS first, tail(genres) AS rest
MATCH (first)<-[:GENRE]-(m:Movie)
WHERE all(g IN rest WHERE (m)-[:GENRE]->(g))
RETURN m
```

Example Query - Genre Overlap - Optimized (2)

```
PROFILE
WITH ['Action','Drama','Mystery'] AS genreNames
UNWIND genreNames AS name
MATCH (g:Genre {name:name})
WITH g ORDER BY size( (g)<-[:GENRE]-() ) ASC
WITH collect(g) AS genres
WITH head(genres) AS first, tail(genres) AS rest
MATCH (first)<-[:GENRE]-(m:Movie)-[:GENRE]->(other)
WITH m, collect(other) AS movieGenres, rest
WHERE all(g IN rest WHERE g IN movieGenres)
RETURN m
```

Example - Genre Overlap - Procedure

- pass in list of genre names
- find genre with least movies (degree) as driver
- put other genres into **Set**
- iterate over movies of minimal genre
- for each movie
- for each genre that is in set increment counter
- if counter = Set size
  - add/stream movie to result

Example - Genre Overlap - Procedure - Code (1)

```
public class SharedGenres {
   @Context public GraphDatabaseService gdb;
   @Context public Log log;

   enum Types implements RelationshipType {
       GENRE;
   }

   enum Labels implements Label {
       Genre, Movie
   }
}
```

```
public static class MovieGenre {
   public String title;
   public String genre;
   public double weight;

   public MovieGenre(String title,
                     String genre,
                     double weight) {
       this.title = title;
       this.genre = genre;
       this.weight = weight;
   }
}
```

Example - Genre Overlap - Procedure - Code (2)

```
@Procedure public Stream<MovieGenre> sharedGenres(@Name("genreNames") List<String> genreNames) {
    int genreCount = genreNames.size();
    // lookup and order genres by degree
    List<Pair<Node, Integer>> genres =
        genreNames.stream().map(genreName -> {
            Node genre = gdb.findNode(Labels.Genre, "name", genreName);
            return Pair.of(genre, genre.getDegree(Types.GENRE, Direction.INCOMING));
        }).sorted((p1, p2) -> Integer.compare(p1.other(), p2.other())).collect(Collectors.toList());
    // find movies for the genre with the lowest degree
    Pair<Node, Integer> driver = genres.get(0);
    Map<Node, List<Pair<Node, Relationship>>> movies = new HashMap<>(driver.other());
    for (Relationship e0 : driver.first().getRelationships(Direction.INCOMING, Types.GENRE)) {
        Node movie = e0.getStartNode();
        List<Pair<Node, Relationship>> list = new ArrayList<>(genreCount);
        list.add(Pair.of(driver.first(), e0));
        movies.put(movie, list);
    }
```

Example - Genre Overlap - Procedure - Code (3)

```
// check and add to found movies
for (Pair<Node, Integer> genre : genres.subList(1, genres.size())) {
    for (Relationship e1 : genre.first().getRelationships(Direction.INCOMING, Types.GENRE)) {
        Node movie = e1.getStartNode();
        if (movies.containsKey(movie)) {
            movies.get(movie).add(Pair.of(genre.first(), e1));
        }
    }
}
```

Example - Genre Overlap - Procedure - Code (4)

```
    // transform map into result stream
    return movies.entrySet().stream().filter(entry -> entry.getValue().size() == genreCount)
        .flatMap(entry -> {
            String title = (String) entry.getKey().getProperty("title", null);
            return entry.getValue().stream().map(genreRel -> {
                String genreName = (String) genreRel.first().getProperty("name");
                Number weight = (Number) genreRel.other().getProperty("weight",0);
                return new MovieGenre(title, genreName, weight);
            });
        });
    }
}
```

End of Module: User Defined Procedures and Functions

Questions ?

Indexing

User-defined Procedures (Index Usage)

```
@Procedure("example.search")
@PerformsWrites // TODO: This is here as a workaround, because index().forNodes() is not read-only
public Stream<SearchHit> search( @Name("label") String label,
                                 @Name("query") String query )
{
    String index = indexName( label );

    // Avoid creating the index, if it's not there we won't be
    // finding anything anyway!
    if( !db.index().existsForNodes( index ))
    {
        // Just to show how you'd do logging
        log.debug( "Skipping index query since index does not exist: `%s`", index );
        return Stream.empty();
    }

    // If there is an index, do a lookup and convert the result
    // to our output record.
    return db.index()
            .forNodes( index )
            .query( query )
            .stream()
            .map( SearchHit::new );
}
```

https://github.com/neo4j-examples/neo4j-procedure-templat

Example in Groovy

```
cp $GROOVY_HOME/lib/groovy-2.*.jar $NEO4J_HOME/plugins/
$GROOVY_HOME/groovyc function.groovy && jar cf $NEO4J_HOME/plugins/uuid.jar UDF.class
```

```
@Grab(value="org.neo4j:neo4j:3.1.0-BETA1",initClass=false)

class UDF {
   @UserFunction("create.uuid")
   @Description("creates an UUID")
   def String uuid() { UUID.randomUUID().toString() }
}
```

Optional Exercise: Write your own Function

- Get Groovy 2.x
- Copy
  `$GROOVY_HOME/lib/groovy-2.*.jar`
  to
  `$NEO4J_HOME/plugins/`
- Put code in `udf.groovy`
- Compile code
  `$GROOVY_HOME/groovyc udf.groovy`
- Build jar
  `jar cf $NEO4J_HOME/plugins/udf.jar UDF.class`
- Restart Neo4j

```
@Grab(value="org.neo4j:neo4j:3.1.0-BETA1",
      initClass=false)


import org.neo4j.graphdb.*
import org.neo4j.procedure.*


class UDF {
    @Context public GraphDatabaseService db

    @UserFunction("create.uuid")
    @Description("creates an UUID")
    def String uuid() {
        UUID.randomUUID().toString()
    }
}
```