# Neo4j

## COMP421 special project

Ivan Petrov, Jérémie Poisson, Samy Zarour and Hans Christian Gregersen

# Disposition

- Motivation
- Model
- Query Language
- Internals
- API and Tools
- Application Demo
- Question Time

# Motivation

We want to model application domains that are **highly connected**, **variably structured** and **dynamically changing**

- Social Networks
  - e.g. Facebook, LinkedIn
- Telecom Networks
  - e.g. Bell, AT&T
- The Web
  - e.g. Search Engines like Google, Yahoo, DuckDuckGo
- Recommendation Systems
  - Amazon and our demo app :)

# Motivation cont.

- ***RDMS is a bad choice...***
  - A contrived model when we’re interested in connections, more than the datapoints themselves
  - Relationships are not conspicuous, but rather inferred from foreign keys and intermediary tables
  - Intermediary tables introduce unnecessary complexity
  - Struggles with highly connected domains as they require expensive join operations
  - Schemas are too rigid - hard to deal with variably structured and/or dynamically changing data.

# Queries on Reciprocal Relationships in RDMS are hard

<u>Bob's Friends</u>

```
SELECT p1.Person
FROM Person p1 JOIN PersonFriend
ON PersonFriend.FriendID = p1.ID
JOIN Person p2
ON PersonFriend.PersonID = p2.ID
WHERE p2.Person = 'Bob'
```

vs.

<u>Who is friends with Bob?</u>

```
SELECT p1.Person
FROM Person p1 JOIN PersonFriend
ON PersonFriend.PersonID = p1.ID
JOIN Person p2
ON PersonFriend.FriendID = p2.ID
WHERE p2.Person = 'Bob'
```

PERSON - Table

| ID | Name |
|---|---|
| 1 | John |
| 2 | Bob |
| ... | …. |
| N | Jeff |

PersonFriend - Table

| PersonID | FriendID |
|---|---|
| 1 | 2 |
| 2 | 1 |
| 2 | 20 |
| ... | ... |

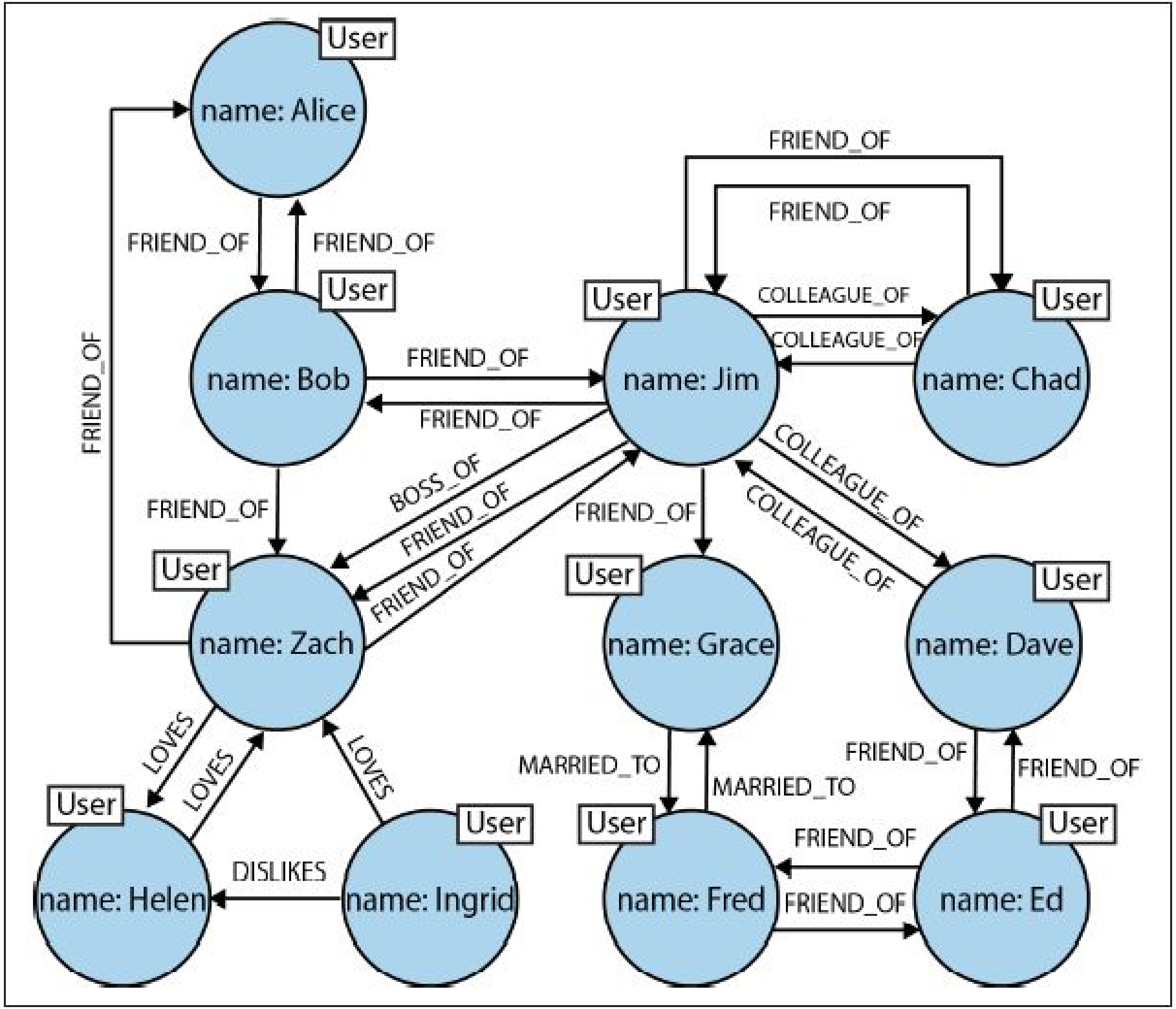
# Example continued...

◦ *Indexing is one Solution to this problem - but it's an expensive layer of indirection and requires careful consideration by the RDMS designer.*

◦ Even worse when asking queries such as:
*"Who are the friends of my friends?"*
or
*"Who are the friends of my friends of my ….?"*

In SQL this is done recursively joining tables, which significantly increases the syntactic and computational complexity of the query.

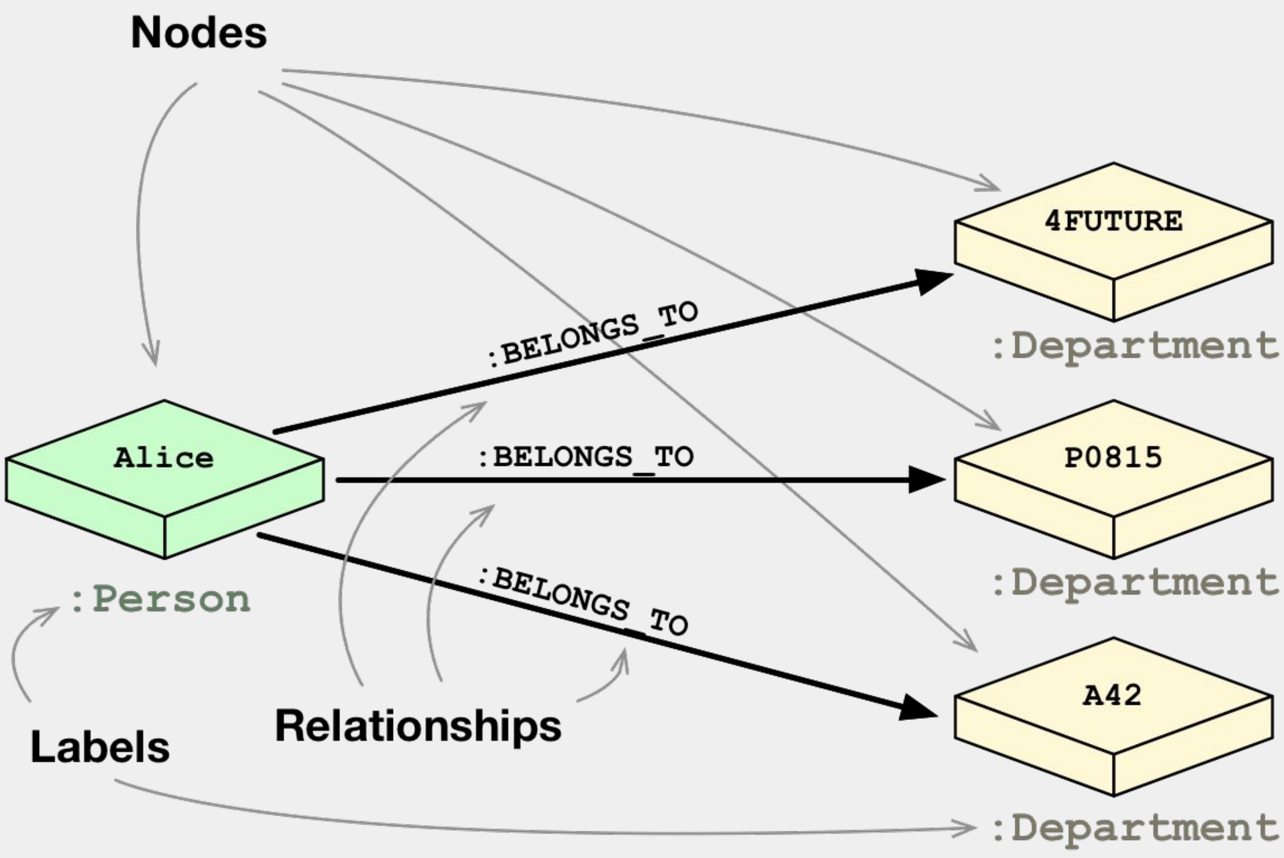
This is not practical in online scenarios.


User
name: Alice
FRIEND_OF
FRIEND_OF
User
name: Bob
FRIEND_OF
FRIEND_OF
FRIEND_OF
User
name: Jim
FRIEND_OF
FRIEND_OF
COLLEAGUE_OF
COLLEAGUE_OF
User
name: Chad
FRIEND_OF
BOSS_OF
FRIEND_OF
FRIEND_OF
FRIEND_OF
COLLEAGUE_OF
COLLEAGUE_OF
User
name: Zach
User
name: Grace
User
name: Dave
LOVES
LOVES
LOVES
MARRIED_TO
MARRIED_TO
FRIEND_OF
FRIEND_OF
User
name: Helen
DISLIKES
User
name: Ingrid
User
name: Fred
FRIEND_OF
FRIEND_OF
User
name: Ed

# The Labeled Property Graph Model

- A graph consists of nodes, relationships, properties and labels
- Nodes
  - contain properties
  - may contain one or more labels for categorization
- Relationships connect nodes.
  - has a direction,
  - has a single name,
  - a start and an end node.
  - may also have properties

- “Connected Data is Stored as Connected Data”


Nodes
4FUTURE
:Department
Alice
:BELONGS_TO
:BELONGS_TO
P0815
:BELONGS_TO
:Person
:Department
A42
Labels
Relationships
:Department

# Graph Solution Evaluated

Intuitive model

- Relationships are explicit, and so the model is easier to understand.

Better Performance

- Queries on connections are expressed as simple patterns, and the implementation is accomplished by walks on paths through the graph.

Flexibility and Agility:

- Graphs are additive: We can add new types of relationships and nodes to the graph without disturbing the outcome of existing queries.
- No schema!

# Partner & Vukotic: Comparing RDMS with Neo4j

Empirical Experiment:

- 1.000.000 people,
- avg. 50 friends each.
- evaluating “friend of a friend of a … “

| Depth | RDBMS ex. time(s) | Neo-4j ex. time(s) | Records returned |
|---|---|---|---|
| 2 | 0.016 | 0.01 | ~2500 |
| 3 | 30.267 | 0.168 | ~110,000 |
| 4 | 1543.505 | 1.359 | ~600,000 |
| 5 | unfinished | 2.132 | ~800,000 |
| | | | |

# QUERY LANGUAGE

CYPHER:

- Declarative Query Language
- Focus on the application domain instead of technicalities
    - *what* instead of *how*
    - Expressive
- Pattern oriented
    - Use patterns as a graph traversal
    - Placeholders to capture and manipulate info. within the query
- Small yet powerful

```
MATCH (js:User)-[:FRIEND_OF]-()-[:FRIEND_OF]->(surfer)
WHERE js.name = "Johan" AND surfer.hobby = "surfing"
RETURN DISTINCT surfer
```

# CYPHER - Graph Traversal / Data retrieval

## MATCH CLAUSE

- Find data that matches a specific pattern (specification by example)
  - Pattern is used for graph traversal
- Placeholders bundled with variable names
- Find data for specific nodes, relationship we specify the property values explicitly

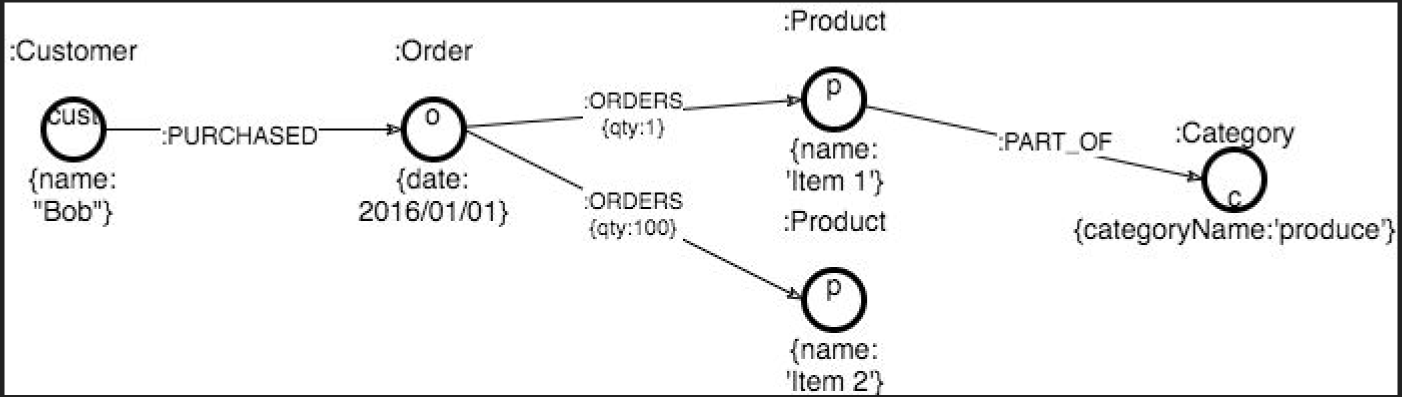
## WHERE CLAUSE

- Bundled with MATCH clause
- Optional
- Add constraints or filter results

```
MATCH (a:User {name:'Jim'})-[:FRIEND_OF]->(b)-[:FRIEND_OF]->(c), (a)-[:FRIEND_OF]->(c)
RETURN b, c
```

# CYPHER - Graph Traversal Examples

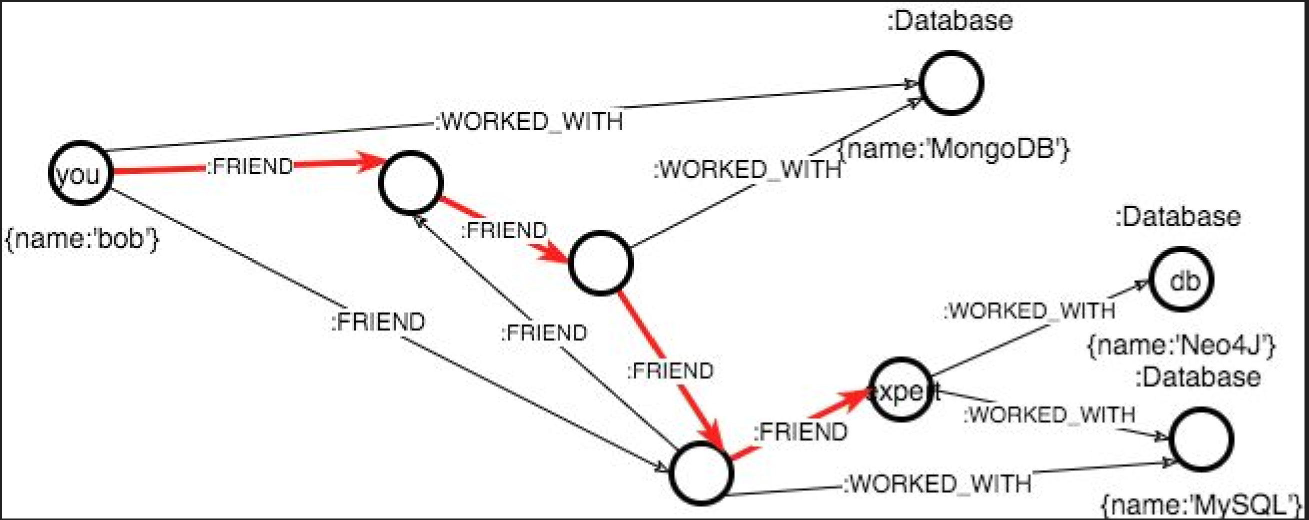
```
MATCH (cust:Customer)-[:PURCHASED]->(:Order)-[o:ORDERS]->(p:Product),
      (p)-[:PART_OF]->(c:Category {categoryName:"Produce"})
RETURN
     DISTINCT cust.contactName as CustomerName,
     SUM(o.quantity) AS TotalProductsPurchased
```




Selects the customers' name along with the number of product they purchased that are part of a category named 'Produce'

# CYPHER - Graph Traversal Examples

```
MATCH (you {name:"You"}), (expert)-[:WORKED_WITH]->(db:Database {name:"Neo4j"}),
  p = shortestPath( (you)-[:FRIEND*..5]-(expert) )
RETURN p,db
```




Selects the shortest path (succession of friends) to an expert that worked with Neo4J

# CYPHER - Graph Manipulation / Data retrieval

## CREATE CLAUSE / DELETE CLAUSE

- Creates all part of a pattern / Delete all part of a pattern

## MERGE CLAUSE

- will MATCH or CREATE (if record does not exist)

```
MATCH (n) WHERE id = 1 CREATE (n)-[:FOO]->(b:Bar)
```

(creates a new :Bar node and a :FOO relationship associated with every node having id = 1)

```
MATCH (ee:User) WHERE ee.name = "Emil"
CREATE (js:User { name: "Johan", from: "Sweden", learn: "Surfin'" }),
        (jp:User { name: "Jérémie", from: "Canada", learn: "Somethin'" })
       (ee)-[:FRIEND_OF { since: 2001 }]->(jp),
       (jp)-[:FRIEND_OF { since: 1994 }]->(js)
```

# CYPHER - Misc.

## AGGREGATION FUNCTION

- count(), sum(), avg(), min(), max()

## INDEXES

```
CREATE INDEX ON :Venue(name)
```

- Way of uniquely identify a node within a label
- Label / Properties combination
- Pick out node directly VS. letting Neo4J discover over the course of traversal
- Efficiency -- used in the internals

## CONSTRAINTS

```
CREATE CONSTRAINT ON (c:Country) ASSERT c.name IS UNIQUE
```

- Asserting uniqueness for specific property values

# CYPHER - Comparison with SQL and XPath

| Cypher | SQL | XPath |
|---|---|---|
| MATCH (:User)-[:FRIEND_OF]-()-[:FRIEND_OF]->(:USER) | N/A | |
| RETURN | SELECT ... | //node/path/pattern |
| WHERE cond = condval | WHERE cond = condval | //node/path/pattern [@attr=attrval] |
| CREATE ... | INSERT INTO ... | N/A |
| MATCH … SET n = x | UPDATE … SET n = x WHERE ... | N/A |

# CYPHER - Summary

## ADVANTAGES

- Expressive; bind naturally to application domain
- Easily understandable by developers, db professionals, and business stakeholders
- Specification by examples

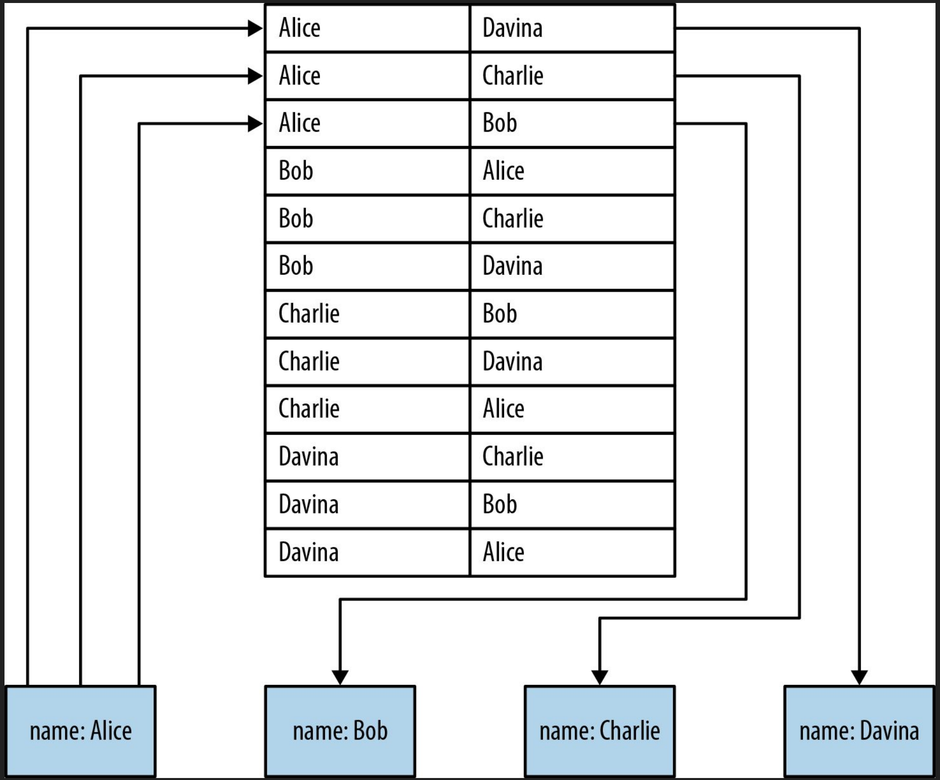
## DISADVANTAGES

- Not sure if a particular graph fits a query
- Easy to make mistakes
    - Unexpected result if misused

# Neo4J Internals

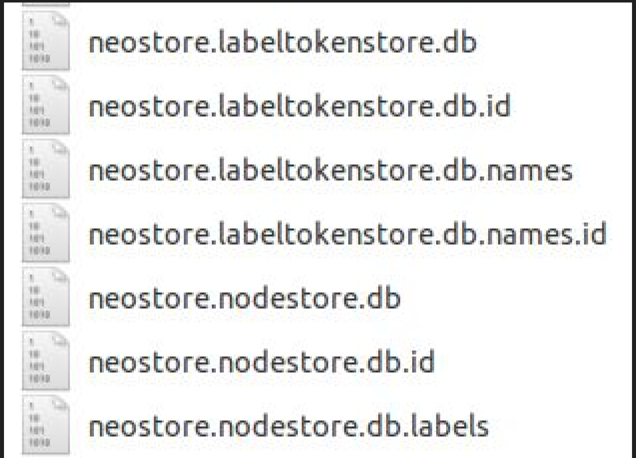
- Graph database has native processing capabilities if it has a property called **index-free adjacency**
  - Each node acts as micro-index of other nearby nodes
  - Query times are independent of the total size of the graph and are proportional to the amount of graph searched
  - Example:
    - In Relational db, we would scan file to find needed record, but in graph db we can do it in O(1) if we now node id

# Non native graph databases



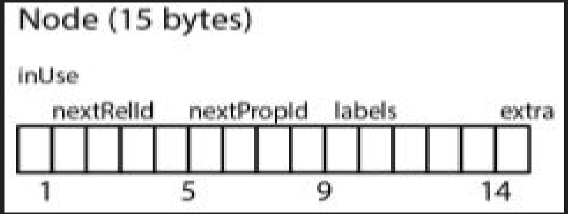

- We have **global index** which links nodes together
- It takes **O(log n)** to find Alice friends and **O(m log n)** to find who is friend with Alice

# Native graph storage files



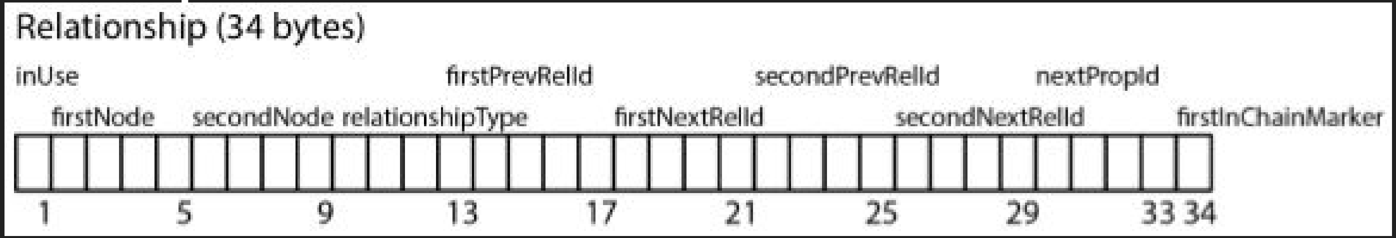

- Path to neo4j storage files neo4j-community-2.3.2/data/graph.db
- Store files
  - Nodes
  - Relationships
  - Labels
  - Properties

# Node store file




- **Neostore.nodestore.db**
  - Each record is 15 bytes long (depends on neo4j version)
  - **(1) inUse** tells if node record is being used
  - **(1-5)** Id of the first relationship connected to the graph
  - **(5-9)** Id of the first property of the node
  - **(9-14)** Id of the first label
  - **(14-15)** reserved for flags
    - One such flag is used to identify densely connected nodes
- org/neo4j/kernel/impl/nioneo/store/NodeStore.java

# Relationship store file




- **Neostore.relationshipstore.db**
- Fixed size records
- Note that relationships are implemented as doubly linked lists
- There is a pointer to a relationship type (which is in another neostore file)
- Note that node store and relationship store files are only concerned with a structure of the graph
- org.neo4j.kernel.impl.nioneo.store.RelationshipStore

# Relationship type store

- **Neostore.relationshiptypestore.db**
  - Relationship type name is stored in **.db.name**

| | |
|---|---|
| inUse 1 byte | Id of the name int -> 4 bytes |

# Property Store

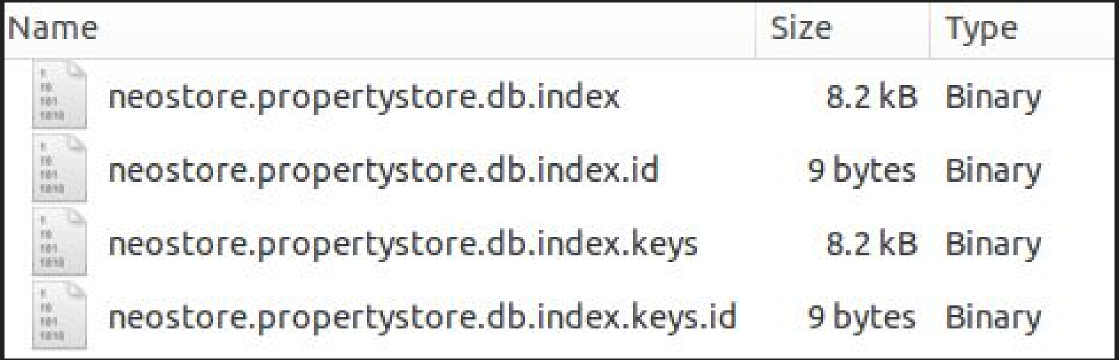
- **neostore.propertystore.db**
- Property records are of fixed size
- 4 property blocks and the ID of the next property in the property chain
    - Property is a {name: value} pair
        - Name - String
        - Value - Java primitive type, String, or their array

| | | | | | |
|---|---|---|---|---|---|
| inUse [1 byte] | Type [2 bytes] | keyIndexId [2 bytes] | dynamicStoreId Long [8 bytes] | propBlock [24 bytes] | nextPropId [4 bytes] |

# Neostore.property.db.index

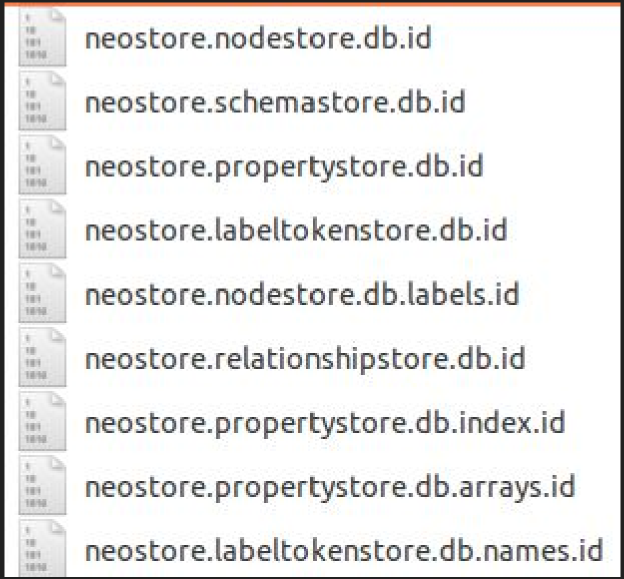
| | | |
|---|---|---|
| inUse<br>[1 byte] | Property count<br>Int -> [4 bytes] | id to dynamicStore<br>Int [ 4 bytes] |

- Dynamic store points to **neostore.proprety.db.index.keys**
    - It keeps property names



| Name | Size | Type |
|---|---|---|
| neostore.propertystore.db.index | 8.2 kB | Binary |
| neostore.propertystore.db.index.id | 9 bytes | Binary |
| neostore.propertystore.db.index.keys | 8.2 kB | Binary |
| neostore.propertystore.db.index.keys.id | 9 bytes | Binary |

# Manager of the available ids




- We need to plug the hole when we delete a node
- Manager is going to give recycled ids to new nodes and new ids when we don't have recycled ones
- Neo4J does not have to run forever so we need to store recycled ids
  - **.id** files

# .id file structure

- Id file has a flag at the beginning that is set to 1 when there is an open channel over it
  - It implies that data in id file is not up to date
- It is set to 0 after successful flush of ids and right before the close()
- After a sticky bit there is always at least one next highest available id
  - Id is implemented as long in Java (8 bytes long), so the minimum file size 9 bytes

# Dynamic Stores

- It is a doubly linked block structure within a dynamic store file

| | | | | |
|---|---|---|---|---|
| inUse | previousBlock | length | nextBlock | data |

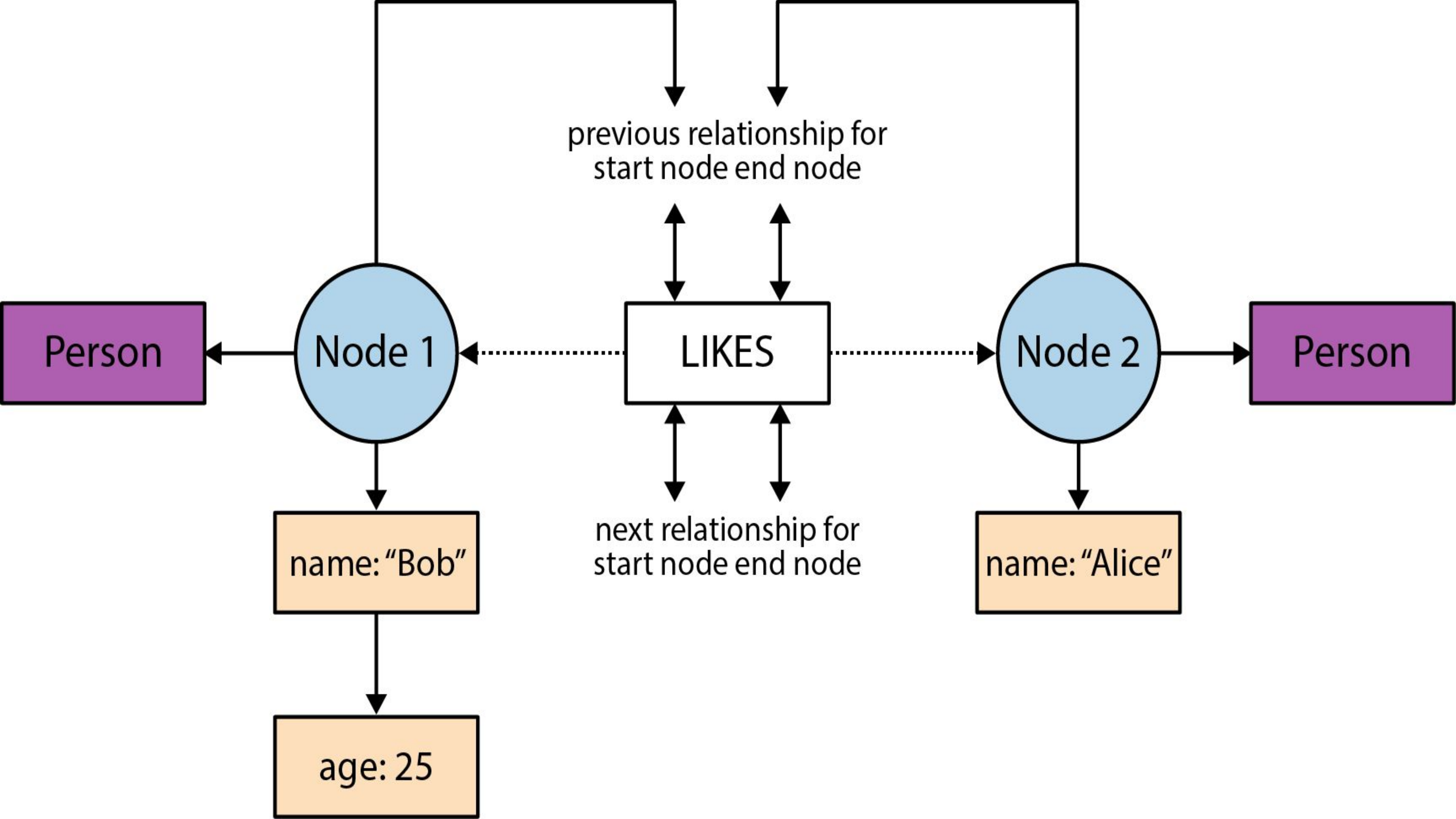
- previousBlock == -1 then it is the first block
- nextBlock == -1 is the last block
- Example array {“Hello”,”World”}:
  - In data segment, byte describing String[], lengths of array in int, size of the first string, Hello, size of next string, “world”

# Dynamic stores

| Name | Size | Type |
|---|---|---|
| neostore.propertystore.db | 2.4 MB | Binary |
| neostore.propertystore.db.id | 402.1 kB | Binary |
| neostore.propertystore.db.index | 8.2 kB | Binary |
| neostore.propertystore.db.arrays | 8.2 kB | Binary |
| neostore.propertystore.db.strings | 1.2 MB | Binary |
| neostore.propertystore.db.index.id | 9 bytes | Binary |
| neostore.propertystore.db.arrays.id | 9 bytes | Binary |
| neostore.propertystore.db.index.keys | 8.2 kB | Binary |
| neostore.propertystore.db.strings.id | 63.6 kB | Binary |
| neostore.propertystore.db.index.keys.id | 9 bytes | Binary |

# Execution Plan

EXPLAIN MATCH (nineties:Movie) WHERE nineties.released > 1990 AND nineties.released < 2000 RETURN nineties.title


previous relationship for
start node end node
Person
Node 1
LIKES
Node 2
Person
next relationship for
start node end node
name: "Bob"
name: "Alice"
age: 25

# NEO4J - APIs and Applications

Built-in tools and APIs for developers

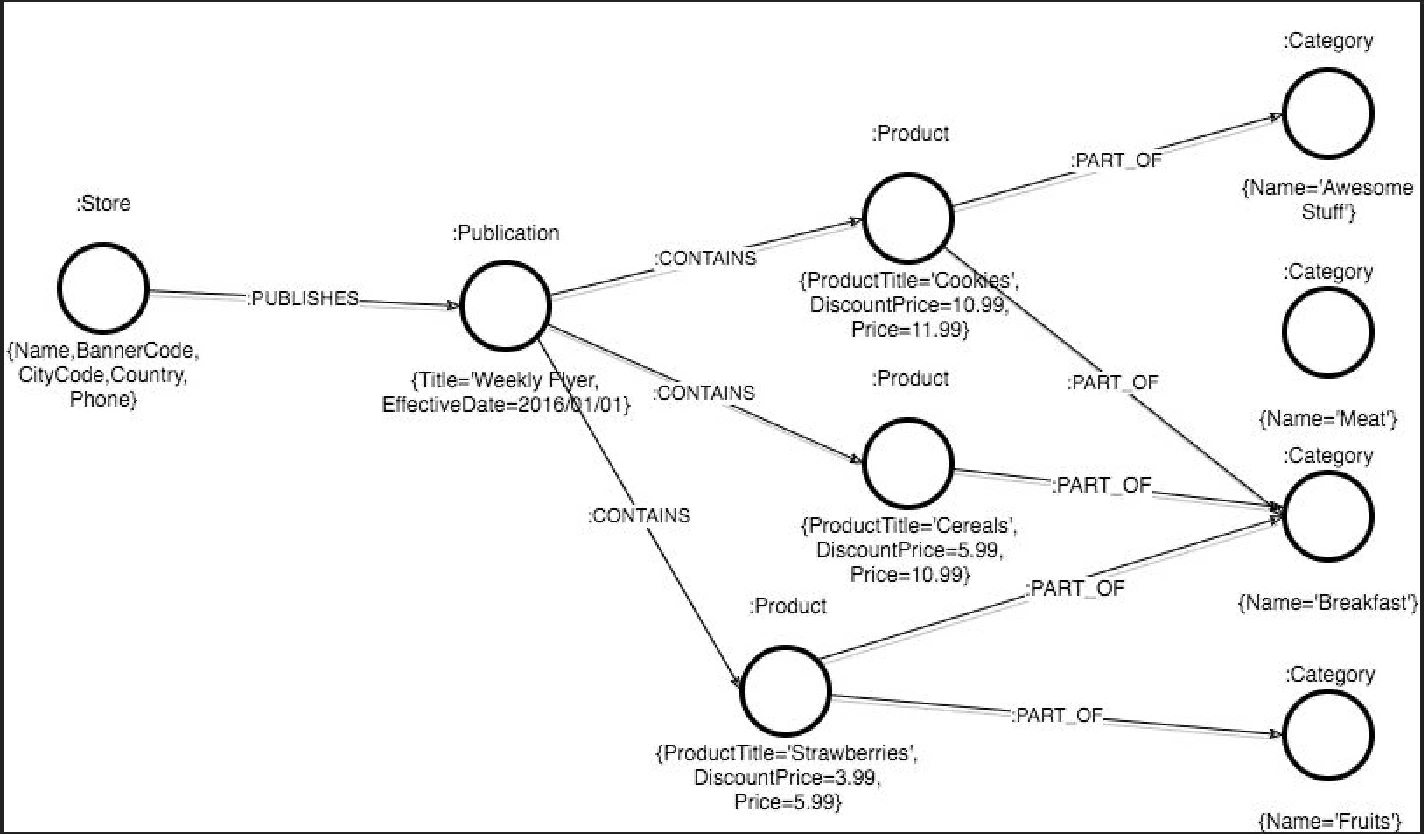
- Neo4J Browser
  - Test and visualize queries
- Built-in REST APIs
  - Query remotely over HTTP
  - Platform independence; easily extensible and many bindings available
  - Transaction support
  - Encapsulation & fixed format response
- Clustering
  - High-availability replication
  - Global Clusters (multi-region clustering)
- Shell Interface

# NEO4J - Demo

## Mobile application

- Huge dataset of currently available flyers & discounts
  - Stores : Metro, Super C, Provigo, IGA, Best-Buy
- With a postal code, locate nearby stores & extract best deals from flyers
- Works on iOS, Android
  - Phones and tablet
- Stores
  - Publishes a Publication
    - Publication contains Products
      - Products are associated with a Category

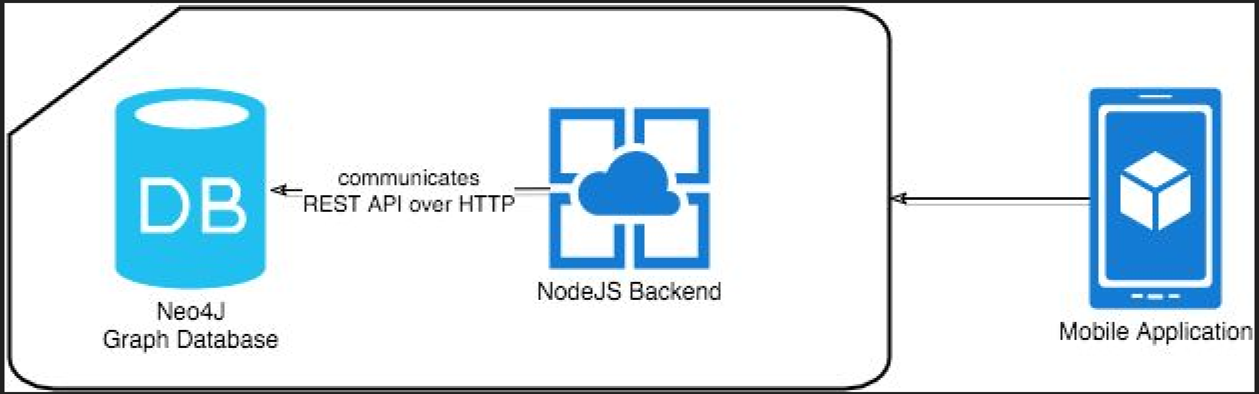
# NEO4J - Demo

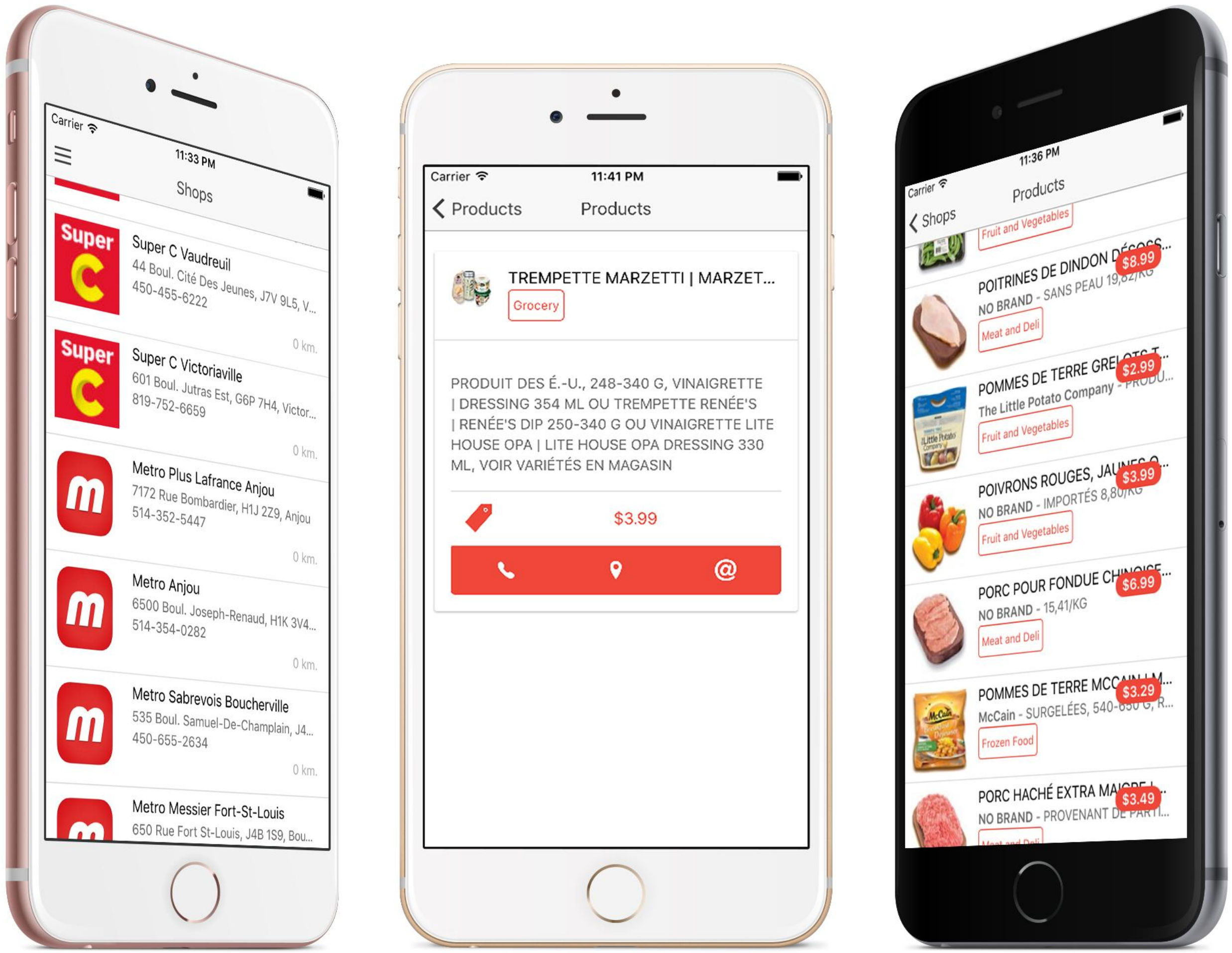


:Store
{Name,BannerCode, CityCode,Country, Phone}
:PUBLISHES
:Publication
{Title='Weekly Flyer, EffectiveDate=2016/01/01}
:CONTAINS
:CONTAINS
:CONTAINS
:Product
{ProductTitle='Cookies', DiscountPrice=10.99, Price=11.99}
:Product
{ProductTitle='Cereals', DiscountPrice=5.99, Price=10.99}
:Product
{ProductTitle='Strawberries', DiscountPrice=3.99, Price=5.99}
:PART_OF
:PART_OF
:PART_OF
:PART_OF
:PART_OF
:Category
{Name='Awesome Stuff'}
:Category
{Name='Meat'}
:Category
{Name='Breakfast'}
:Category
{Name='Fruits'}

# NEO4J - Technologies

- NodeJS Backend
  - Communicates with Neo4J REST API
  - Acts as an API; fronts some operation from the DB
- Mobile App
  - Ionic Framework
  - Communicate with the fronted NodeJS Backend





Carrier
11:33 PM
Shops
Super C
Super C Vaudreuil
44 Boul. Cité Des Jeunes, J7V 9L5, V...
450-455-6222
0 km.
Super C Victoriaville
601 Boul. Jutras Est, G6P 7H4, Victor...
819-752-6659
0 km.
Metro Plus Lafrance Anjou
7172 Rue Bombardier, H1J 2Z9, Anjou
514-352-5447
0 km.
Metro Anjou
6500 Boul. Joseph-Renaud, H1K 3V4...
514-354-0282
0 km.
Metro Sabrevois Boucherville
535 Boul. Samuel-De-Champlain, J4...
450-655-2634
0 km.
Metro Messier Fort-St-Louis
650 Rue Fort St-Louis, J4B 1S9, Bou...
Carrier
11:41 PM
Products
Products
TREMPETTE MARZETTI | MARZET...
Grocery
PRODUIT DES É.-U., 248-340 G, VINAIGRETTE | DRESSING 354 ML OU TREMPETTE RENÉE'S | RENÉE'S DIP 250-340 G OU VINAIGRETTE LITE HOUSE OPA | LITE HOUSE OPA DRESSING 330 ML, VOIR VARIÉTÉS EN MAGASIN
$3.99
Carrier
11:36 PM
Shops
Products
Fruit and Vegetables
POITRINES DE DINDON DÉSOSS...
$8.99
NO BRAND - SANS PEAU 19,82/KG
Meat and Deli
POMMES DE TERRE GRELOTS T...
$2.99
The Little Potato Company - PRODU...
Fruit and Vegetables
POIVRONS ROUGES, JAUNES O...
$3.99
NO BRAND - IMPORTÉS 8,80/KG
Fruit and Vegetables
PORC POUR FONDUE CHINOISE...
$6.99
NO BRAND - 15,41/KG
Meat and Deli
POMMES DE TERRE MCCAIN | M...
$3.29
McCain - SURGELÉES, 540-650 G, R...
Frozen Food
PORC HACHÉ EXTRA MAIGRE |...
$3.49
NO BRAND - PROVENANT DE PARTI...

# CYPHER - SQL vs Cypher

## SELECTION / JOINS / AGGREGATION

| | |
|---|---|
| `SELECT movie.title`<br>`FROM movie`<br>`WHERE movie.released > 1998;` | `MATCH (movie:Movie)`<br>`WHERE movie.released > 1998`<br>`RETURN movie.title;` |
| `SELECT DISTINCT co_actor.name`<br>`FROM person AS keanu`<br>`    JOIN acted_in AS acted_in1`<br>`      ON acted_in1.person_id = keanu.id`<br>`    JOIN acted_in AS acted_in2`<br>`      ON acted_in2.movie_id = acted_in1.movie_id`<br>`    JOIN person AS co_actor`<br>`      ON acted_in2.person_id = co_actor.id AND`<br>`         co_actor.id <> keanu.id`<br>`WHERE keanu.name = 'Keanu Reeves';` | `MATCH`<br>`  (keanu:Person)-[:ACTED_IN]->(movie:Movie),`<br>`  (coActor:Person)-[:ACTED_IN]->(movie)`<br>`WHERE`<br>`  keanu.name = 'Keanu Reeves'`<br>`RETURN`<br>`  DISTINCT coActor.name;` |
| `SELECT director.name, count(*)`<br>`FROM person keanu JOIN acted_in ON keanu.id = acted_in.`<br>`person_id JOIN directed ON acted_in.movie_id = directed.`<br>`movie_id JOIN person AS director ON directed.person_id =`<br>`director.id WHERE keanu.name = 'Keanu Reeves' GROUP BY`<br>`director.name ORDER BY count(*) DESC` | `MATCH`<br>`    (keanu:Person {name: 'Keanu Reeves'  })-[:ACTED_IN]->`<br>`(movie:Movie),`<br>`(director:Person)-[:DIRECTED]->(movie)`<br>`RETURN`<br>`    director.name, count(*)`<br>`ORDER BY count(*) DESC` |