UMD DATA605 - Big Data Systems

12.2: Neo4j

- **Instructor**: Dr. GP Saggese, gsaggese@umd.edu

- ***Querying graph data***

Query Languages for Graph Databases

- **Cypher**
  - Designed for Property Graphs
  - Data: vertices and edges with key-value properties
  - Declarative query language
  - Suitable for subgraph pattern matching
  - Struggles with reachability queries
  - Native to Neo4j
- **Gremlin**
  - Works with RDF and Property Graphs
  - Imperative query language
  - Describes graph traversal processes
- **SPARQL**
  - Similar to SQL in structure
  - Designed for querying RDF data
  - Standardized by the W3C for semantic web applications

```
MATCH (nicole:Actor
   {name:'Nicole Kidman'})-[[:ACTED_IN]->(movie
WHERE movie.year < 2007
RETURN movie
```

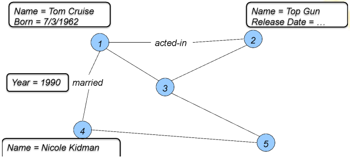
```
// calculate basic collaborative filtering for vertex 1
m = [:]
g.v(1).out('likes').in('likes').out('likes').groupCount(m)
m.sort{-it.value}

// calculate the primary eigenvector (eigenvector centrality) of a graph
m = [:]; c = 0;
g.V.as('x').out.groupCount(m).loop('x'){c++ < 1000}
m.sort{-it.value}
```

```
PREFIX  foaf: <[\textcolor{blue}{\underline{htt
SELECT ?name
       ?email
WHERE
  {
    ?person  a          foaf:Person.
    ?person  foaf:name  ?name .
    ?person  foaf:mbox  ?email .
  }
```

SCIENCE ACADEMY

Neo4j

- Graph DB storing data as Property Graph
  - Nodes, edges hold data as key-value pairs
- Focus is
  - On relationships between values
- Two querying languages
  - Cypher, Gremlin
- GUI or REST API
- Full ACID-compliant transactions
- High-availability clustering
- Incremental backups
- Run in small application or large server clusters

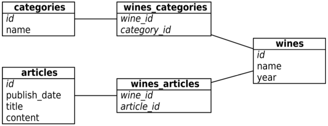


Graph DB: Example

Specs

- Create a wine suggestion engine
- Wines categorized by
  - Varieties (e.g., Chardonnay, Pinot Noir)
  - Regions (e.g., Bordeaux, Napa, Tuscany)
  - Vintage (year grapes harvested)
- Track articles describing wines by authors
- Users track favorite wines

Relational model



- The important relationships are `produced`, `reported_on`, `grape_type`
- Create various tables
  - `wines`: (id, name, year)
  - `wines_categories` (wine_id, category_id)
  - `category` table (id, name)
  - `wines_articles` (wine_id, article_id)
  - `articles` (id, publish_date, title, content)

Labeled Property Graphs in Neo4j

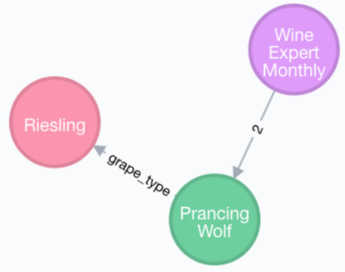
- **Nodes**
  - Main data elements
  - Connected via *relationships*
  - Have *properties* (key/value pairs)
- **Relationships**
  - Connect two *nodes*
  - Directional
  - Multiple relationships per node
  - Have *properties* (key/value pairs)
- **Properties**
  - Named values (key is a string)
  - Indexed and constrained
  - Composite indexes from multiple properties
- **Labels**
  - Group nodes into sets
  - Nodes may have multiple labels
  - Labels indexed for faster node retrieval
  - Native label indexes optimized for performance

Cypher Example

```
CREATE (w:Wine
    {name: "Prancing Wolf",
      style: "ice wine",
      vintage: 2015})
MATCH (n)
  RETURN n;
CREATE (p:Publication
    {name: "Wine Expert Monthly"})
MATCH (p:Publication
    {name: "Wine Expert Monthly"}),
    (w:Wine {name: "Prancing Wolf",
    vintage: 2015})
    CREATE (p)-[r:reported_on]->(w)
```

Cypher Example

```
MATCH (p:Publication {name: "Wine Expert Monthly"}),
    (w:Wine {name: "Prancing Wolf"})
    CREATE (p)-[r:reported_on {rating: 2}]->(w)

CREATE (g:GrapeType {name: "Riesling"})
MATCH (w:Wine {name: "Prancing Wolf"}),
  (g:GrapeType {name: "Riesling"})
  CREATE (w)-[r:grape_type]->(g)
```

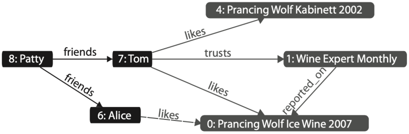


Cypher Example

```
CREATE (wr:Winery {name: "Prancing Wolf Winery"})
MATCH (w:Wine {name: "Prancing Wolf"}),
    (wr:Winery {name: "Prancing Wolf Winery"})
    CREATE (wr)-[r:produced]->(w)
CREATE (w:Wine
    {name:"Prancing Wolf", style: "Kabinett", vintage: 2002})
CREATE (w:Wine
    {name: "Prancing Wolf", style: "Spätlese", vintage: 2010})
MATCH (wr:Winery
    {name: "Prancing Wolf Winery"}),(w:Wine {name: "Prancing Wolf"})
    CREATE (wr)-[r:produced]->(w)
MATCH (w:Wine), (g:GrapeType {name: "Riesling"})
    CREATE (w)-[r:grape_type]->(g)
```

Cypher Example

- Add a social component to the wine graph
  - People preference for wine
  - Relationships with one another

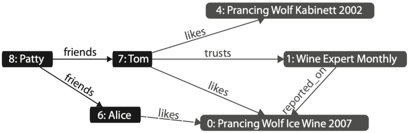



```
CREATE (p:Person {name: "Alice"})

MATCH (p:Person {name: "Alice"}),
    (w:Wine {name: "Prancing Wolf",
    style: "ice wine"})
    CREATE (p)-[r:likes]->(w)

CREATE (p:Person {name: "Patty"})

MATCH (p1:Person {name: "Patty"}),
    (p2:Person {name: "Tom"})
    CREATE (p1)-[r:friends]->(p2)
```

- The changes were made "superimposing" new relationships without changing the previous data

Cypher Example

```
MATCH (p:Person
  {name: "Alice"})-->(n)
  RETURN n;
```




```
MATCH (p:Person
  {name: "Alice"})-->(other: Person)
  RETURN other.name;
```

```
MATCH (fof:Person)-[:friends]-(f:Person)-[:friends]-(p:Person {name
  RETURN fof.name;
```

A general query structure

```
MATCH [Nodes and relationships]
WHERE [Boolean filter statement]
RETURN [DISTINCT] [statements [AS alias]]
ORDER BY [Properties] [ASC or DESC]
SKIP [Number] LIMIT [Number]
```

Simple query

- Get all nodes of type `Program` that have the name `Hello World!`

```
MATCH (a : Program)
WHERE a.name = 'Hello World!'
RETURN a
```

Query relationships

- Get all relationships of type `Author` connecting `Programmers` and `Programs`:


```
MATCH (a:Programmer)-[r:Author]->(b:Program)

RETURN r
```

Matching nodes and relationships

- Nodes
  - (a), (), (:Ntype), (a:Ntype),
  - (a { prop:'value' }) ,
  - (a:Ntype { prop:'value' })
- Relationships
  - (a)–(b)
  - (a)–>(b), (a)<–(b),
  - (a)–>(), (a)-[r]->(b),
  - (a)-[:Rtype]->(b), (a)-[:R1|:R2]->(b),
  - (a)-[r:Rtype]->(b)
- May have more than 2 nodes
  - (a)–>(b)<–(c), (a)–>(b)–>(c)
- Path
  - p = (a)–>(b)

More options

- Relationship distance:
  - `(a)-[:Rtype*2]->(b)`: 2 hops of type Rtype
  - `(a)-[:Rtype*]->(b)`: any number of hops of type Rtype
  - `(a)-[:Rtype*2..10]->(b)`: 2-10 hops of Rtype
  - `(a)-[:Rtype*..10]->(b)`: 1-10 hops of Rtype
  - `(a)-[:Rtype*2..]->(b)`: at least 2 hops of Rtype
- Could be used also as:
  - `(a)-[r*2]->(b)` r gets a sequence of relationships
  - `(a)-[*{prop:val}]->(b)`

Operators

- Mathematical
  - +, -, *, /,%, ^ (power, not XOR)
- Comparison
  - =,<>,<,>,>=,<=, =~ (Regex), IS NULL, IS NOT NULL
- Boolean
  - AND, OR, XOR, NOT
- String
  - Concatenation through +
- Collection
  - Concatenation through +
  - IN to check if an element exists in a collection

More WHERE options

- WHERE others.name IN [‘Andres’, ‘Peter’]
- WHERE user.age IN range (18,30)
- WHERE n.name =~ 'Tob.*‘
- WHERE n.name =~ '(?i)ANDR.*‘ - (case insensitive)
- WHERE (tobias)–>()
- WHERE NOT (tobias)–>()
- WHERE has(b.name)
- WHERE b.name? = ‘Bob’ (Returns all nodes where name = ‘Bob’ plus all nodes without a name property)

Functions

- On paths:
  - MATCH shortestPath((a)-[*]-(b))
  - MATCH allShorestPath((a)-[*]-(b))
  - Length(path) – The path length or 0 if not exists.
  - RETURN relationships(p) - Returns all relationships in a path.
- On collections:
  - RETURN a.array, filter(x IN a.array WHERE length(x)= 3) FILTER - returns the elements in a collection that comply to a predicate.
  - WHERE ANY (x IN a.array WHERE x = "one") – at least one
  - WHERE ALL (x IN nodes(p) WHERE x.age > 30) – all elements
  - WHERE SINGLE (x IN nodes(p) WHERE var.eyes = "blue") – Only one
- nodes(p) – nodes of the path p

With

- Manipulate result sequence before passing to following query parts
- Usage of WITH:
  - Limit entries passed to other MATCH clauses
  - Introduce aggregates for predicates in WHERE
  - Separate reading from updating the graph. Each query part must be read-only or write-only

Data access is programmatic

- REST API
- Through the Java APIs
  - JVM languages have bindings to the same APIs
    - JRuby, Jython, Clojure, Scala...
- Managing nodes and relationships
- Indexing
- Traversing
- Path finding
- Pattern matching