Neo4j

Brian DuSell
What is Neo4j?

- Neo4j is a “graph database management system”
  - cf. relational database management system
  - Like SQL for graphs
  - Instead of defining tables and columns, we define nodes and relationships
- Uses a SQL-like language called Cypher Query Language to query and update graphs
Background

- Designed to deal with databases of graph-structured data
  - Often more natural representation than relational tables
  - Claims significant speedups over RDBMS
- Relatively new
- Developed by Neo4j, Inc.
  - Founded in 2007
  - Based near San Francisco, Sweden, and elsewhere
Using Neo4j

- Open-source community edition and closed-source enterprise edition
- Can be run as a server or embedded in an application
- Implemented in Java
- Drivers exist for major languages (Python, JavaScript, Java, etc.)
- Driver communicates with server via “bolt” protocol
  - HTTP is also an option

Using Neo4j with Python

```python
# pip install neo4j-driver

from neo4j.v1 import GraphDatabase, basic_auth

driver = GraphDatabase.driver(
    "bolt://54.162.76.69:33079",
    auth=basic_auth("neo4j", "whistles-contract-home"))

session = driver.session()

cypher_query = ""
MATCH (n)
RETURN id(n) AS id
LIMIT $limit
"

results = session.run(cypher_query, parameters={"limit": 10})

for record in results:
    print(record['id'])
```
Property Graph Model

- **Nodes**: graph vertices
  - Nodes have one or more *labels* that specify node type

- **Relationships**: directed edges between nodes
  - Each one has exactly one relationship type
  - Can have multiple edges between same nodes

- **Properties**: key-value pairs that can be attached to both nodes and relationships
  - Values have their own type system with ints, floats, strings, etc.

**Example**

- **User**: name: “Brian”
- **User**: name: “Justin”
- **Post**: message: “Graphs are cool!”
- **Post**: posted_at: “Oct 2, 2018 11:30 AM”
Property Graph Model - Movie Database Example

```graph
[Diagram showing nodes and edges for a movie database example, including movie titles, actors, director, genres, and ratings.]
```
Cypher Query Language

- Declarative language inspired by SQL
- Intentionally similar to SQL and best learned by example
- Unlike SQL, the language includes data types for lists, maps, and paths
- Standardization attempt via openCypher

```
MATCH (m:Movie)<-[:RATED]-(u:User)
WHERE m.title CONTAINS "Matrix"
WITH m.title AS movie, COUNT(*) AS reviews
RETURN movie, reviews
ORDER BY reviews DESC
LIMIT 5;
```

“How many reviews does each Matrix movie have?”
Pattern Syntax

- Describes patterns of nodes, relationships, and attributes in graphs with ASCII art
- Nodes are in (parentheses), relationships are in [brackets], properties are like [key: value]
- Arrows can be written in either direction or omitted
- Binds data to variable names like n
- Names $likeThis$ are named parameters
- Same syntax used for both matching and creating data

```
CREATE (n {name: $value})
Create a node with the given properties.

CREATE (n $map)
Create a node with the given properties.

UNWIND $listOfMaps AS properties
CREATE (n) SET n = properties
Create nodes with the given properties.

CREATE (n)-[r:KNOWS]->(m)
Create a relationship with the given type and direction; bind a variable to it.

CREATE (n)-[:LOVES {since: $value}]->(m)
Create a relationship with the given type, direction, and properties.
```
(n:Person)
Node with Person label.

(n:Person:Swedish)
Node with both Person and Swedish labels.

(n:Person {name: $value})
Node with the declared properties.

()-[:r {name: $value}]->()  
Matches relationships with the declared properties.

(n)-->(m)
Relationship from n to m.

(n)--(m)
Relationship in any direction between n and m.

(n:Person)-->(m)
Node n labeled Person with relationship to m.

(m)<-[:KNOWS]-:(n)
Relationship of type KNOWS from n to m.

(n):[:KNOWS|LOVES]->(m)
Relationship of type KNOWS or of type LOVES from n to m.

(n)-[:r]->(m)
Bind the relationship to variable r.

(n)-[*1..5]->(m)
Variable length path of between 1 and 5 relationships from n to m.

(n)-[*]->(m)
Variable length path of any number of relationships from n to m. (See Performance section.)

(n)-[:KNOWS]->(m) {property: $value}
A relationship of type KNOWS from a node n to a node m with the declared property.

shortestPath((n1:Person)-[*..6]-(n2:Person))
Find a single shortest path.

allShortestPaths((n1:Person)-[*..6]->(n2:Person))
Find all shortest paths.

size((n)--()>())
Count the paths matching the pattern.
Basic Query Syntax

MATCH a pattern and bind variable names

WHERE filters results using a Boolean expression

WITH (1) assigns values to variable names, and (2) computes aggregate functions like COUNT; explicitly separates query parts

```
MATCH (m:Movie)<-[[:RATED]]->(u:User)
WHERE m.title CONTAINS "Matrix"
WITH m.title AS movie, COUNT(*) AS reviews
RETURN movie, reviews
ORDER BY reviews DESC
LIMIT 5;
```

“How many reviews does each Matrix movie have?”
Basic Query Syntax

Additional MATCH-WHERE clauses can be inserted here to further filter results.

RETURN determines what the query returns much like SQL SELECT.

Fully analogous to SQL.

```
MATCH (m:Movie)<-[[:RATED]]-(u:User)
WHERE m.title CONTAINS "Matrix"
WITH m.title AS movie, COUNT(*) AS reviews
RETURN movie, reviews
ORDER BY reviews DESC
LIMIT 5;
```

“How many reviews does each Matrix movie have?”
Execution Model

- Supports ACID transactions
- Queries are analyzed and decomposed into an execution plan AST
  - Vertices are low-level operations like “filter rows” or “sort”
  - Operations input and output sets of rows and pipe into each other
  - Some support lazy evaluation
  - Leaves extract data from the database
- Indexes can speed up queries
- Cypher query planner optimizes execution plans using four pre-computed statistics including
  - Number of nodes with label X
  - Number of relationships by type
Parallelism

- Declarative API decouples query from execution model
- Only Enterprise Edition of Neo4j supports multi-machine clustering ("Causal Clustering") and "Massively Parallel Graph Algorithms" library
- Clusters consist of Core Servers and Read Replicas
- Read replicas allow large-scale graph queries to be widely distributed
Parallelism

- Core Servers maintain synchronized version of data
- Applications communicate only with Core Servers
- Only Core Servers handle writes
- Data is asynchronously copied to Read Replicas
- Read-only queries can be processed in parallel among Read Replicas
What movies are most similar to Inception based on jaccard similarity of genres?

MATCH (m:Movie {title: "Inception"})-[[:IN_GENRE]->(g:Genre)<-[:IN_GENRE]-(other:Movie))
WITH m, other, COUNT(g) AS intersection, COLLECT(g.name) AS i
MATCH (m)-[:IN_GENRE]->(mg:Genre)
WITH m, other, intersection, i, COLLECT(mg.name) AS s1
MATCH (other)-[:IN_GENRE]->(og:Genre)
WITH m, other, intersection, i, s1, COLLECT(og.name) AS s2
WITH m, other, intersection, i, s1, s2
WITH m, other, intersection, i, s1 + FILTER(x IN s2 WHERE NOT x IN s1) AS union, s1, s2
RETURN m.title, other.title, s1, s2, ((1.0 * intersection) / SIZE(union)) AS jaccard ORDER BY jaccard DESC LIMIT 100

<table>
<thead>
<tr>
<th>m.title</th>
<th>other.title</th>
<th>s1</th>
<th>s2</th>
<th>jaccard</th>
</tr>
</thead>
</table>