Introduction to Data Management

Graph Databases

Alyssa Pittman
Based on slides by Jonathan Leang, Dan Suciu, et al
Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle
Announcements

- Final exam cancelled
  - More information about grading will come by Wednesday
- Everything else moved online
- HW8 will be released tomorrow
- Please fill out the course eval!
  - [https://uw.iasystem.org/survey/219314](https://uw.iasystem.org/survey/219314)
Recap: Graph Databases

- **Data model**
  - Entities (nodes)
  - Relationships (directed edges)
  - Both can have properties

![Graph Database Diagram]

- USER name: Al
- KNOWS since: 1/1/1970
- KNOWS since: 2/18/2018
- USER name: Bob
- USER name: Carl
- KNOWS

March 9, 2020
Recap: MATCH patterns

MATCH (tom:Person {name: "Tom Hanks"})
-[:ACTED_IN]->(movie:Movie)
RETURN movie
Recap: WHERE constraints

- Adds constraints to the MATCH pattern
  - Similar to a SQL WHERE
  - It can filter results
  - But adjusts the MATCH pattern instead of filtering after matching executes

```
MATCH (p:Person)-[:ACTED_IN]-> (:Movie {title: "Apollo 13" })
WHERE p.born < 1960
RETURN p
```

Can use:
- node or relationship properties
- boolean operators
- IN [...], EXISTS, IS NULL ...

And others you don’t need to know, such as subqueries
Recap: RETURN clause

- Neo4j uses the Cypher query language
  - Kind of like SELECT

- **Return nodes**
  
  MATCH (a:Person)-[r:ACTED_IN]->(b:Movie)
  RETURN a, b

- **Return relationships**
  
  MATCH (:Person)-[r:ACTED_IN]->(:Movie)
  RETURN r

- **Return properties**
  
  MATCH (a:Person)-[r:ACTED_IN]->(:Movie)
  RETURN a.name
Recap: misc clauses

- Many familiar SQL clauses:

MATCH (:Person {name:"Tom Hanks"})
  -[:ACTED_IN]->(:Movie)
  <-[:ACTED_IN]-(co:Person)
RETURN DISTINCT co.name
ORDER BY co.name
LIMIT 10
Outline

▪ Variable-length paths
▪ Solving SQL-like problems with Cypher
▪ Graph db case study
MATCH with variable length

- Neo4j uses the Cypher query language
  - Kind of like SELECT

- Variable length relationship match pattern:
  \[-[:TYPE*minHops..maxHops]->\]

  minHops, maxHops are optional
default to 1 and infinity, respectively
dots can be omitted if want a single length
MATCH with variable length

- **Within three hops**
  
  ```
  MATCH (tom { name: 'Tom Hanks' })
  -[:ACTED_IN*1..3]-(movie:Movie)
  RETURN movie
  ```

- **Exactly two hops (like a SQL self-join!)**
  
  ```
  MATCH (tom { name: 'Tom Hanks' })
  -[:ACTED_IN*2]-(movie:Movie)
  RETURN movie
  ```

- **Any number of hops (warning: huge! why?)**
  
  ```
  MATCH (tom { name: 'Tom Hanks' })
  -[:ACTED_IN*1..]- (movie:Movie)
  RETURN movie
  ```
MATCH with variable length

- What if we want to see the paths themselves?

- Give the match a variable:
  ```plaintext
  MATCH p=(tom { name: 'Tom Hanks' })
  -[:ACTED_IN*1..3]-(movie:Movie)
  RETURN p
  ```
MATCH shortest path

- Another nifty feature: find the shortest path between two nodes

- Give the path a variable:
  MATCH (tom:Person { name: 'Tom Hanks' }),
  (kevin:Person { name: 'Kevin Bacon' }),
  p = shortestPath((tom)-[*..15]-(kevin))
  RETURN p
OPTIONAL MATCH clause

- Matches patterns but fills in “null” for missing parts
  - Similar to OUTER JOIN

- In SQL:
  ```sql
  SELECT p.name, movie.name
  FROM Person p LEFT OUTER JOIN Movie m
  ON p.id = m.director;
  ```

- In Cypher:
  ```cypher
  MATCH (p:Person)
  OPTIONAL MATCH
  (p)-[[:DIRECTED]]->(movie:Movie)
  RETURN p, movie
  ```
OPTIONAL MATCH clause

- Matches patterns but fills in “null” for missing parts
  - Similar to OUTER JOIN

- In SQL:
  ```
  SELECT p.name, movie.name
  FROM Person p LEFT OUTER JOIN Movie m
  ON p.id = m.director;
  ```

- In Cypher:
  ```
  MATCH (p:Person)
  OPTIONAL MATCH
    (p)-[:DIRECTED]->(movie:Movie)
  RETURN p, movie
  ```

Matches all people

Either matches a movie that p has directed or fills with null
OPTIONAL MATCH clause

MATCH (p:Person)
OPTIONAL MATCH (p)-[:DIRECTED]->(movie:Movie)
RETURN p, movie

{ "name": "Steve Zahn", "born": 1967 }
{ "name": "Tom Hanks", "born": 1956

{ "title": "That Thing You Do", "tagline": "In every life there comes a time when that thing you dream becomes that thing you do", "released": 1996 }
MATCH (p:Person)
OPTIONAL MATCH
  (p)-[:DIRECTED]->(movie:Movie)
RETURN p.name, movie.name

| "Steve Zahn"   | null |
| "Tom Hanks"    | "That Thing You Do" |
| "Nora Ephron"  | "You've Got Mail" |
| "Nora Ephron"  | "Sleepless in Seattle" |
| "Rita Wilson"  | null |
| "Bill Pullman" | null |
| "Victor Garber"| null |
| "Rosie O'Donnell" | null |
| "John Patrick Stanley" | "Joe Versus the Volcano" |
OPTIONAL MATCH clause

Recall that the WHERE clause is part of the preceding MATCH clause...

MATCH (p:Person)
WHERE p.name = 'Tom Hanks'
OPTIONAL MATCH
    (p)-[:DIRECTED]->(movie:Movie)
RETURN p.name, movie.name

Only match people named ‘Tom Hanks’

<table>
<thead>
<tr>
<th>p.name</th>
<th>movie.title</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Tom Hanks&quot;</td>
<td>&quot;That Thing You Do&quot;</td>
</tr>
</tbody>
</table>
OPTIONAL MATCH clause

Recall that the WHERE clause is part of the preceding MATCH clause...

MATCH (p:Person)
OPTIONAL MATCH
  (p)-[:DIRECTED]->(movie:Movie)
WHERE p.name = 'Tom Hanks'
RETURN p.name, movie.name

<table>
<thead>
<tr>
<th>Name</th>
<th>Movie Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Steve Zahn&quot;</td>
<td>null</td>
</tr>
<tr>
<td>&quot;Tom Hanks&quot;</td>
<td>&quot;That Thing You Do&quot;</td>
</tr>
<tr>
<td>&quot;Nora Ephron&quot;</td>
<td>null</td>
</tr>
<tr>
<td>&quot;Rita Wilson&quot;</td>
<td>null</td>
</tr>
<tr>
<td>&quot;Bill Pullman&quot;</td>
<td>null</td>
</tr>
<tr>
<td>&quot;Victor Garber&quot;</td>
<td>null</td>
</tr>
<tr>
<td>&quot;Rosie O'Donnell&quot;</td>
<td>null</td>
</tr>
</tbody>
</table>
Aggregates

- Similar to SQL’s aggregate functions
  - Defaults to “full table” aggregation

MATCH (p:Person)
RETURN count(*), count(p.born),
min(p.born), max(p.born),
avg(p.born), collect(p.born)
Aggregates

- Similar to SQL’s aggregate functions
  - Defaults to “full table” aggregation

MATCH (p:Person)
RETURN count(*), count(p.born), min(p.born), max(p.born), avg(p.born), collect(p.born)

<table>
<thead>
<tr>
<th>count(*)</th>
<th>count(p.born)</th>
<th>min(p.born)</th>
<th>max(p.born)</th>
<th>avg(p.born)</th>
<th>collect(p.born)</th>
</tr>
</thead>
</table>
Grouping

- **Implicit grouping**
  - Once you use an aggregate, the non-aggregate fields become your grouping keys

MATCH (p:Person)
RETURN p.born, count(*)

<table>
<thead>
<tr>
<th>p.born</th>
<th>count(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1964</td>
<td>1</td>
</tr>
<tr>
<td>1967</td>
<td>7</td>
</tr>
<tr>
<td>1961</td>
<td>5</td>
</tr>
<tr>
<td>1960</td>
<td>4</td>
</tr>
<tr>
<td>1965</td>
<td>3</td>
</tr>
<tr>
<td>1970</td>
<td></td>
</tr>
</tbody>
</table>
### Implicit grouping

- Once you use an aggregate, the non-aggregate fields become your grouping keys

```
MATCH (p:Person)
RETURN p.born, count(*)
```

<table>
<thead>
<tr>
<th>p.born</th>
<th>count(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1964</td>
<td>1</td>
</tr>
<tr>
<td>1967</td>
<td>7</td>
</tr>
<tr>
<td>1961</td>
<td>5</td>
</tr>
<tr>
<td>1960</td>
<td>4</td>
</tr>
<tr>
<td>1965</td>
<td>3</td>
</tr>
<tr>
<td>1950</td>
<td></td>
</tr>
</tbody>
</table>
Matching (p:Person) 
RETURN p.born, count(*)

Counts the number of people born each birth year

<table>
<thead>
<tr>
<th>p.born</th>
<th>count(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1964</td>
<td>1</td>
</tr>
<tr>
<td>1967</td>
<td>7</td>
</tr>
<tr>
<td>1961</td>
<td>5</td>
</tr>
<tr>
<td>1960</td>
<td>4</td>
</tr>
<tr>
<td>1965</td>
<td>3</td>
</tr>
<tr>
<td>1970</td>
<td></td>
</tr>
</tbody>
</table>
Just like in SQL, RA, SQL++,
- After you use an aggregate, you can only use the aggregate keys or aggregate values

MATCH (p:Person)
RETURN p.born, count(*)
ORDER BY p.name
WITH/WHERE clauses

- Allows you to chain parts of your query - group and then filter
  - Enables you to do HAVING filters

MATCH (p:Person)
WITH p.born AS year,
    count(*) AS num_people_born
WHERE num_people_born > 5
RETURN year, num_people_born

<table>
<thead>
<tr>
<th>year</th>
<th>num_people_born</th>
</tr>
</thead>
<tbody>
<tr>
<td>1967</td>
<td>7</td>
</tr>
<tr>
<td>1956</td>
<td>6</td>
</tr>
</tbody>
</table>
WITH/WHERE clauses

- Allows you to chain parts of your query - group and then filter
  - Enables you to do HAVING filters

MATCH (p:Person)
WITH p.born AS year,
  count(*) AS num_people_born
WHERE num_people_born > 5
RETURN year, num_people_born

<table>
<thead>
<tr>
<th>year</th>
<th>num_people_born</th>
</tr>
</thead>
<tbody>
<tr>
<td>1967</td>
<td>7</td>
</tr>
<tr>
<td>1956</td>
<td>6</td>
</tr>
</tbody>
</table>
WITH/WHERE clauses

- Warning: make sure you remember when you’ve grouped

MATCH (p:Person)
WITH p.born AS year,
    count(*) AS num_people_born
WHERE num_people_born > 5
RETURN year, num_people_born,
    count(p.born)

Now the grouping is done in WITH

...we can only refer to our WITH aliases later

```
Neo.ClientError.Statement.SyntaxError

Neo<ClientError.Statement.SyntaxError: Variable `p` not defined (line 5, column 37 (offset: 134))
"RETURN year, num_people_born, count(p.born)"
```

March 9, 2020
WITH/WHERE clauses

- Warning: make sure you remember when you’ve grouped

MATCH (p:Person)
WITH p.born AS year,
  count(*) AS num_people_born
WHERE num_people_born > 5
RETURN year, num_people_born,
  count(*)

Now the grouping is done in WITH

...so this counts a single group rather than the rows in the group

<table>
<thead>
<tr>
<th>year</th>
<th>num_people_born</th>
<th>count(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1967</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>1956</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>
• Can create a sandbox at https://neo4j.com/sandbox/
• Today we’ll play with data from their movie database walkthrough
  • Enter `play movie-graph` in the query editor to use the data yourself
Case study: people data

▪ The year is 2012
▪ You’re maxing out your PostgreSQL database
▪ You’ve already moved your ETL process to a Hadoop cluster
▪ What do you do?
Case study: people data

- The year is 2012
- You’re maxing out your PostgreSQL database
- You’ve already moved your ETL process to a Hadoop cluster
- What do you do?

Try a graph db!
Case study: the good

- The benefits of using a graph db:
  - model was much more intuitive
  - easy to express joins
  - good for analytics
Case study: the trip wires

- Some problems were just new
  - supernodes
  - perf problems = cached results = what’s the use of the database?
Case study: the bad

- Cons:
  - large learning curve
  - couldn’t apply lessons from the relational world
    - especially operational concerns
Case study: was it worth it?

- End result: the company went back to RDBMS after the few people who knew the system left.

- It was reasonable to try, but...

- Learning a new system and operationalizing it can be much more expensive than optimizing what you already have

- The “shape” of their queries were the same, so they could easily be written in SQL
Case study: would it work for you?

- Graph dbs have evolved in the past decade
  - some perf problems probably gone
  - more people with experience & documentation
- Cloud hosting might alleviate operational burden
- A larger team might be able to carry through the project

Bottom line: NoSQL isn’t a silver bullet.
Takeaways

▪ Most query languages let us solve similar types of problems.
▪ Think and test carefully before moving to a new type of database.