Graph-based!

Recommenders 101

Caroline Harbitz

This workshop is...

- A basic overview of traditional recommendation algorithms
- A hands-on introduction to graph-powered recommendations

This workshop is not about...

- In-depth analyses of recommendation algorithms
- Graph theory
- Implementing a graph database application
- Bashing RDBMS and aggregate stores
- nordstrom.com’s recommendations

Before we get started: some useful links

- Clone (or download) this workshop's github repository
  https://github.com/cterp/rec101-workshop
- Neo4j documentation
  http://neo4j.com/docs/stable/
- Intro to Cypher
  http://neo4j.com/developer/cypher-query-language/
- Neo4j ref card
  http://neo4j.com/docs/stable/cypher-refcard/

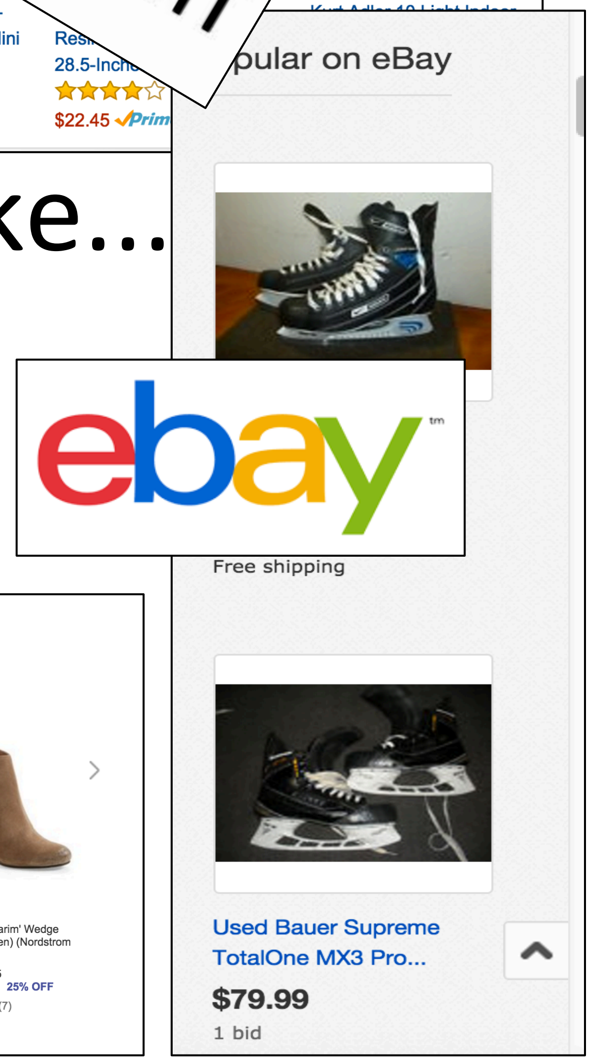
Workshop outline

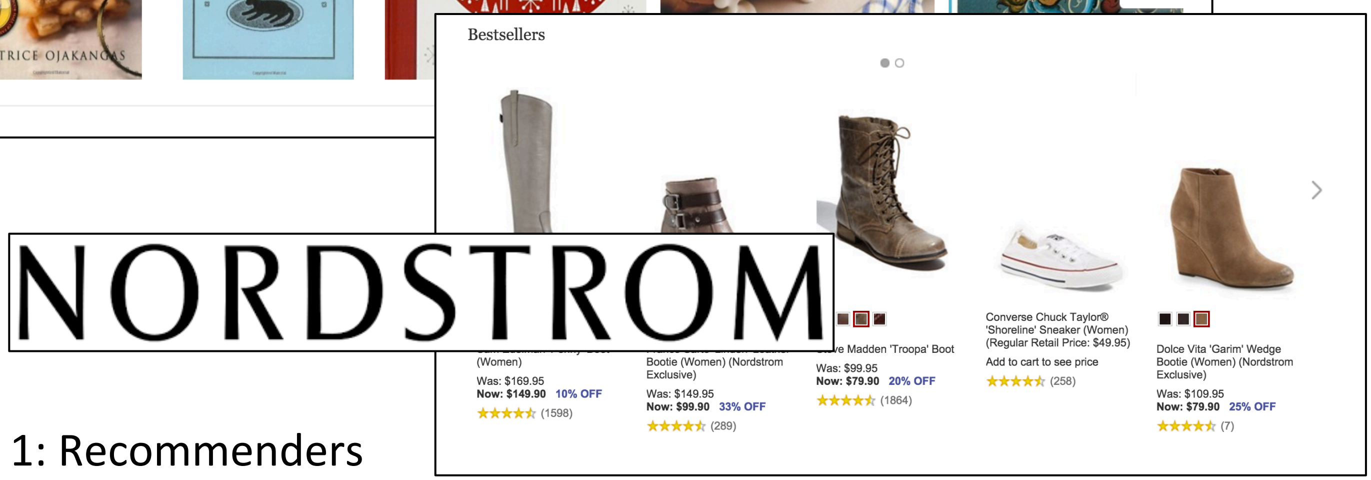
Part 1: Recommendation algorithms

Part 2: Graph database basics and queries

Part 3: Build your own simple recommender

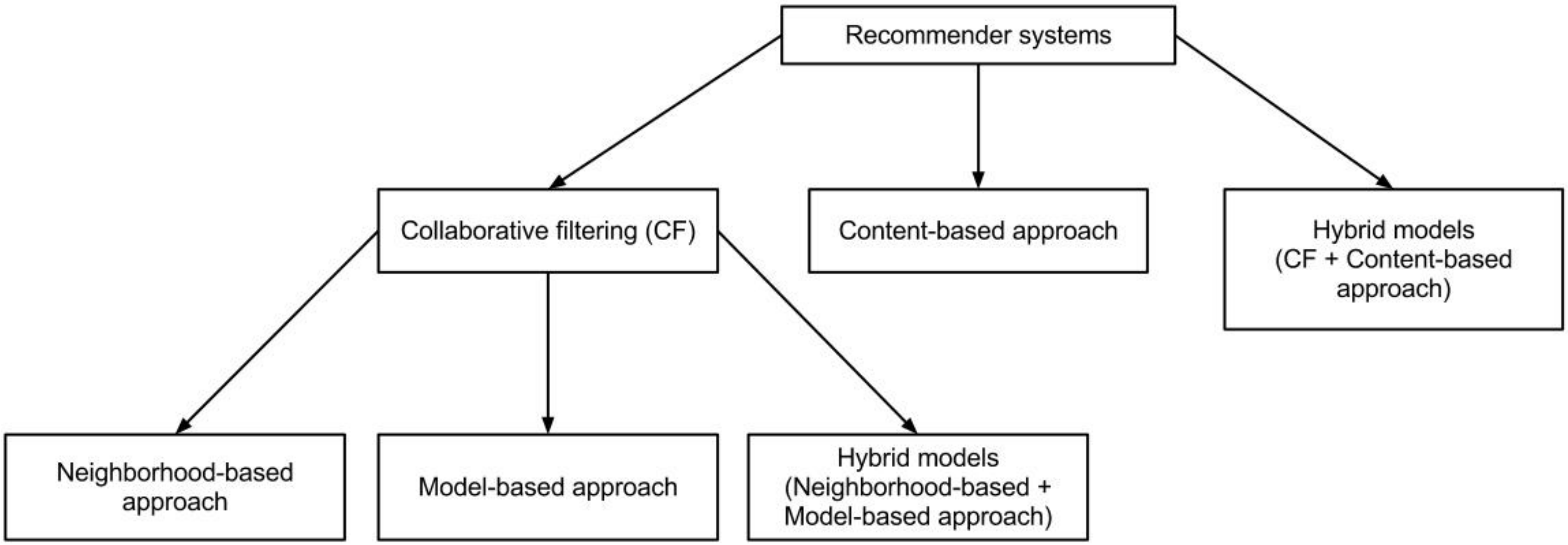
You might also like...





Part 1: Recommenders

Part 1: Recommenders



https://en.wikipedia.org/wiki/Collaborative_filtering

Recommendation process

Purpose: personalize each customer's online store.

1. Find potential recommendations
2. Narrow down potential recommendations
3. Hide irrelevant recommendations
4. Measure the quality of the recommendations

Collaborative filtering

Predict a person's affinity for something by connecting that person to other people with similar tastes.

Collaborative filtering: user-user

- Compute “distance” between all users
- Neighborhoods
- Pitfalls:
  - Neighborhood quality
  - Online calculations

Collaborative filtering: item-item

- Example: amazon.com
- Find items that customers tend to purchase together
- Calculate the similarity between a single product and all related products

Collaborative filtering: item-item

For each item in product catalog I_1

  For each customer C who purchased I_1

    For each item I_2 purchased by customer C

      Record that a customer purchased I_1 and I_2

  For each item I_2

    Compute the similarity between I_1 and I_2

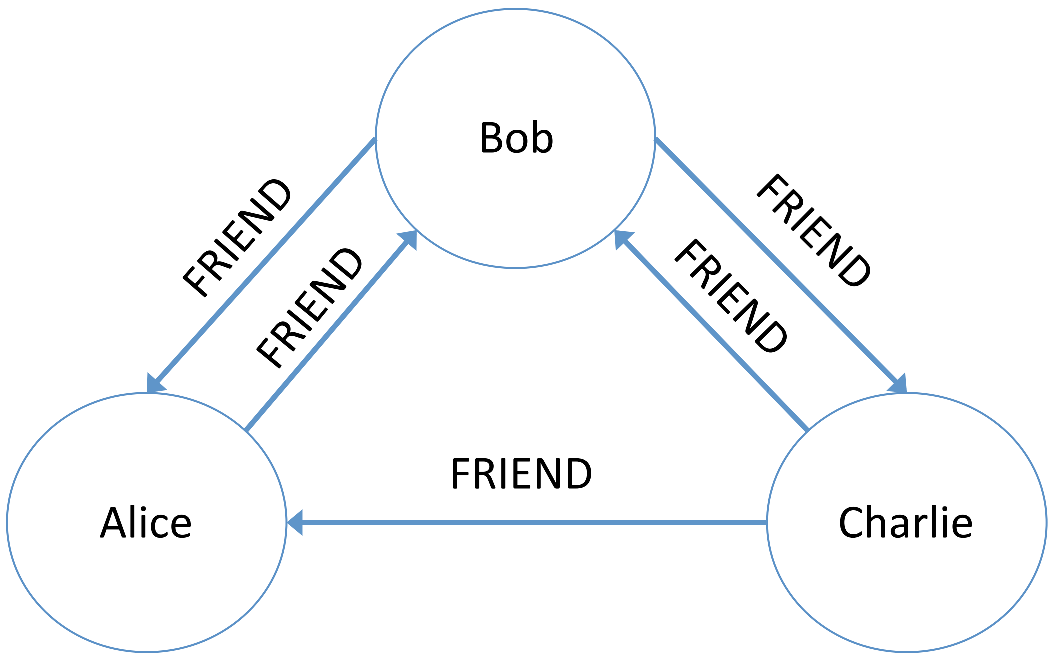
Linden, Greg, Brent Smith, and Jeremy York. "Amazon. com recommendations: Item-to-item collaborative filtering." Internet Computing, IEEE 7.1 (2003): 76-80.

Collaborative filtering: Limitations

- Rigid
- Scaling
- Recommendation quality
- Data sparsity

Part 2: Neo4j graphs

- A graph: a collection of *nodes* and *edges*

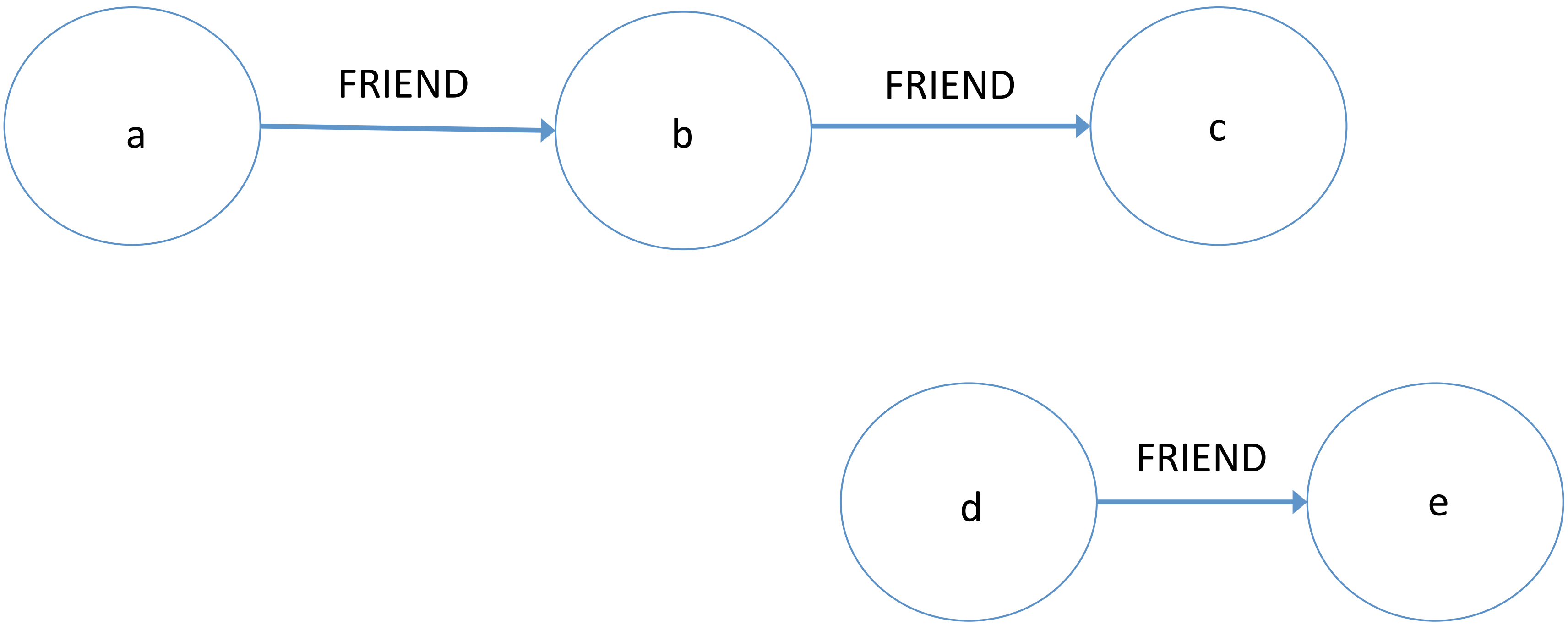


A simple property graph.

4 building blocks of a Neo4j graph

1. Nodes: entities
2. Relationships: connect nodes
3. Properties: entity attributes, relationship qualities, and metadata
4. Labels: group nodes by role

What is a traversal?



Part 2: Neo4j graphs

Ok, so what?

- Graphs let you forget about:
  - Primary and foreign key constraints
  - Expensive joins
  - Expensive reciprocal queries
  - Sparse tables with nullable columns

- Graphs are all about connectedness:
  - Store direct references to neighboring nodes
  - Relationships are first-class citizens of graph data model
  - Connect your data as the domain dictates rather than a schema imposed upfront

Limitations of this approach

- Your graph is only as good as your data model
- Neo4j's implementation optimizes traversals (OLTP) at the expense of non-traversal queries (OLAP)
- Not a standard industry tool

Why neo4j?

- Large user community
- Easy to use
- Documentation

Cypher

- Neo4j’s graph query language
  - DSL to query a Neo4j graph database
- In a nutshell: describe paths in the graph by using ASCII art

Cypher examples

- Nodes: ()
- Relationships: []
- Directions: -> and <-
- Graph pattern examples:

  ()-[]-()

  ()-[]->()

  ()<-[]-()

ASCII art describes the graph pattern of a Cypher query.

Frequently used statements

<u>MATCH</u>

- Draws the query pattern
- MATCH (node1:Label1)-[rel:REL_TYPE]->(node2:Label2)

<u>RETURN</u>

- Equivalent to SQL SELECT

<u>WHERE</u>

- Filter your pattern matching results

<u>WITH</u>

- Chain query parts; similar to Unix pipe. RETURN.

Start your Neo4j servers

From your Neo4j directory, run:

```
$ ./bin/neo4j start
```

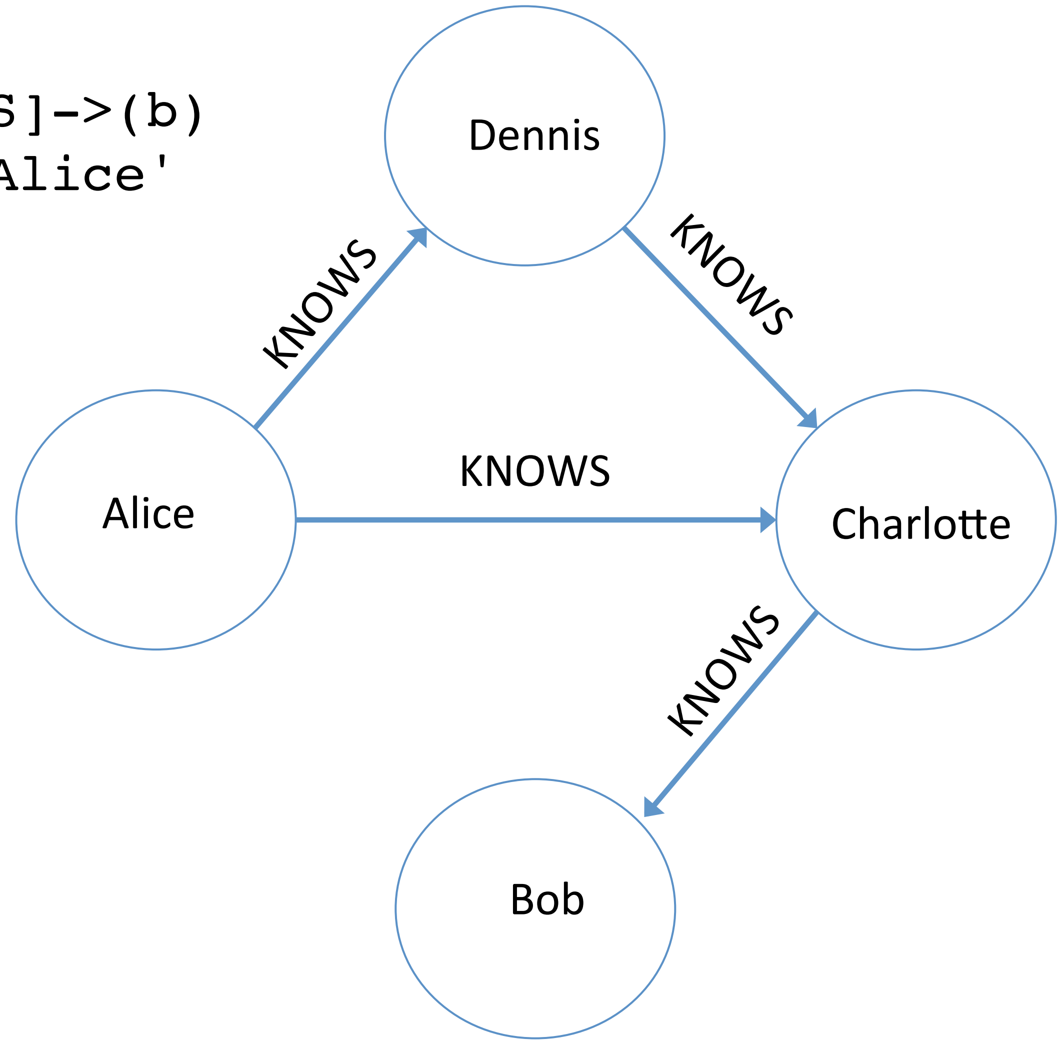
Then, in your browser go to:

http://localhost:7474/browser/

Run ‘part_2.cypher’ in the browser

Example 1: find Alice's friends

```
MATCH (a:Person)-[:KNOWS]->(b)
WHERE a.person_name = 'Alice'
RETURN b.person_name;
```



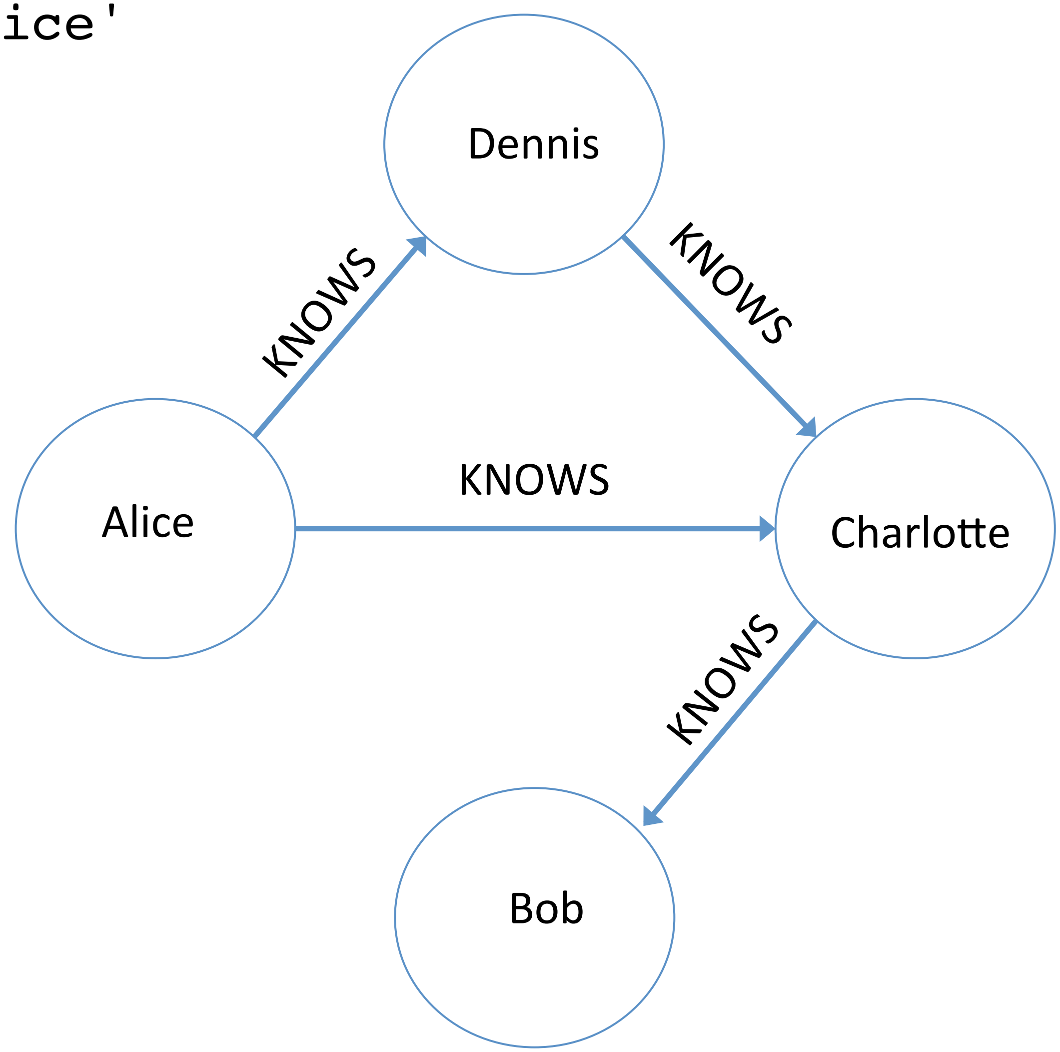
What happens when you return just 'b'?

Equivalent queries

```
MATCH (a:Person)-[:KNOWS]->(b)
WHERE a.person_name = 'Alice'
RETURN b.person_name;

MATCH (a:Person)-[:KNOWS]->(b:Person)
WHERE a.person_name = 'Alice'
RETURN b.person_name;

MATCH (a)-[:KNOWS]->(b)
WHERE a.person_name = 'Alice'
RETURN b.person_name;

MATCH (a:Person {person_name: 'Alice'})-[:KNOWS]
   ->(b:Person)
RETURN b.person_name;
```

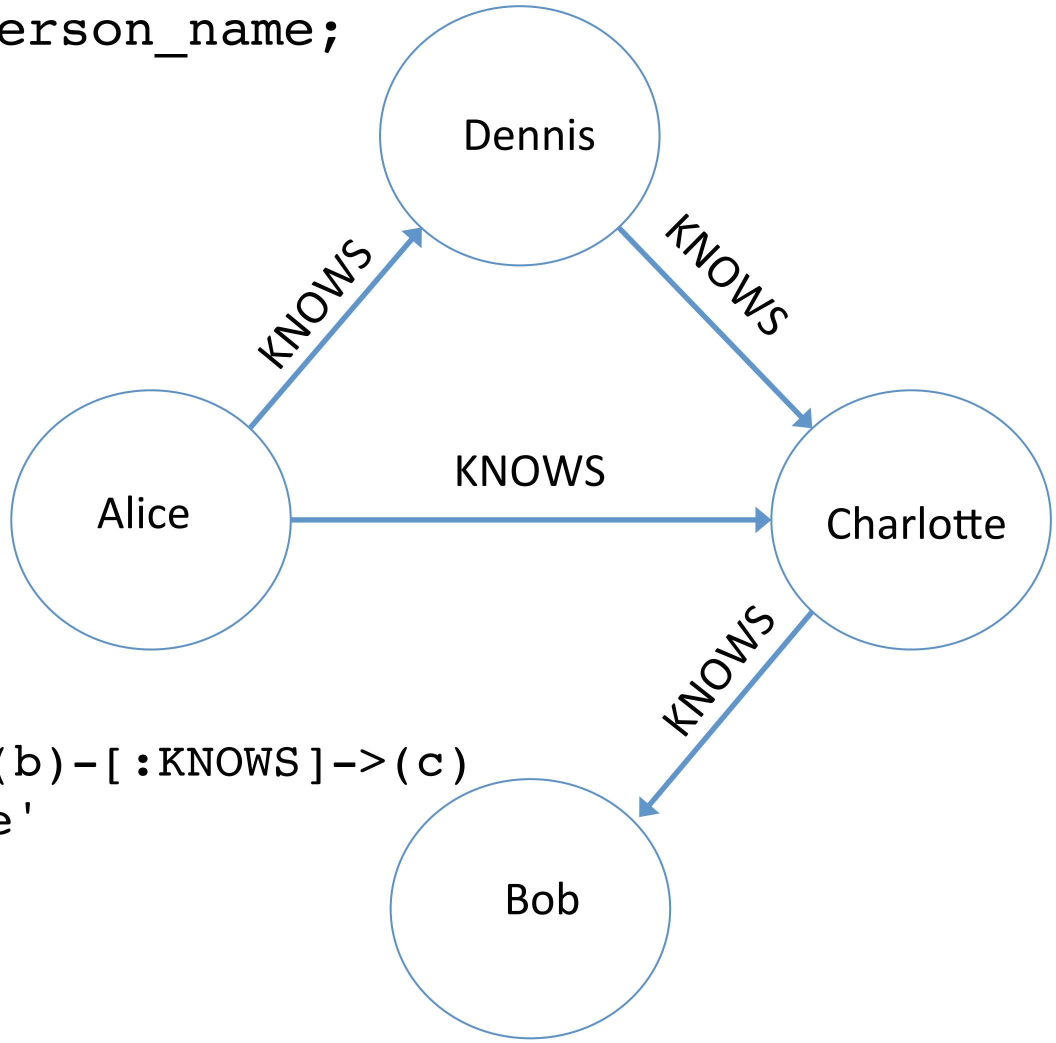
Example 2: find Alice's mutual friend

```
MATCH (a:Person)-[:KNOWS]->(b)-[:KNOWS]->(c),
(a)-[:KNOWS]->(c)
WHERE a.person_name = 'Alice'
RETURN b.person_name;
```



Example 3: find Alice's friend-of-friends

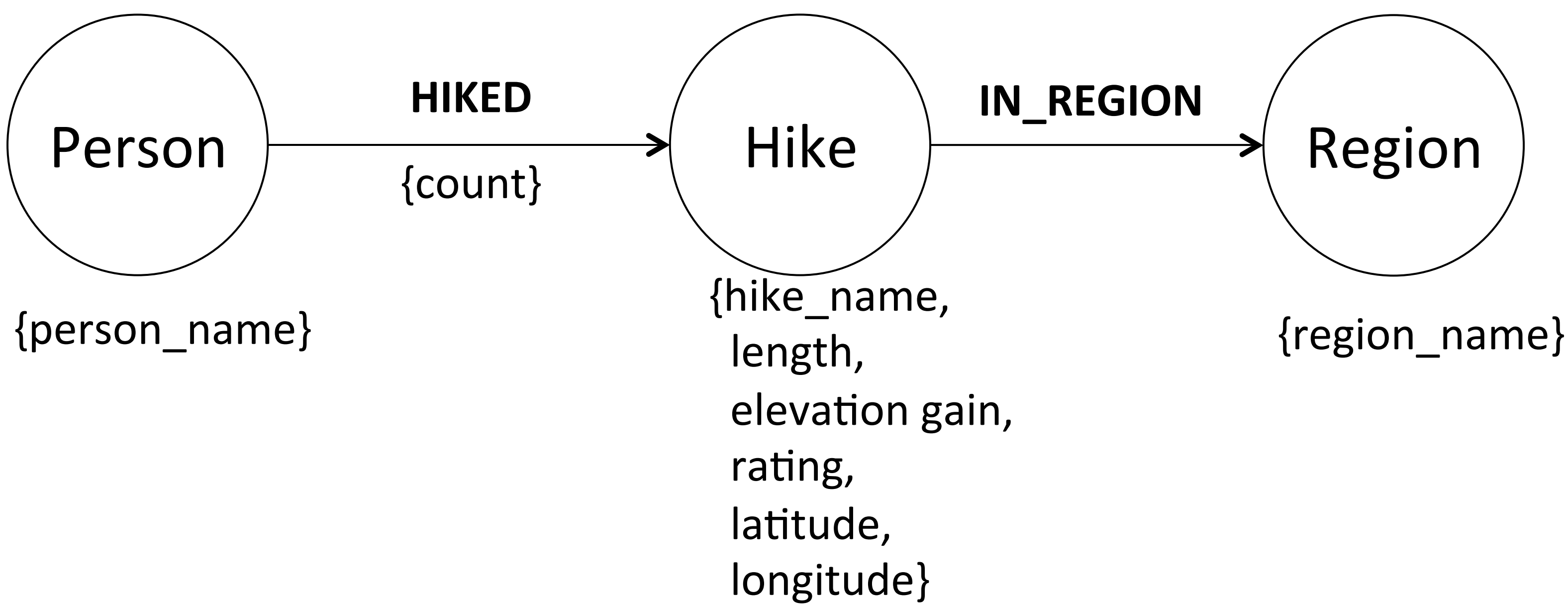
```
MATCH (p:Person {person_name: 'Alice'})-
   [:KNOWS*..2]->(friend_of_friend)
WHERE NOT (p)-[:KNOWS]-(friend_of_friend)
RETURN friend_of_friend.person_name;
```




```
// Equivalent query
MATCH (a:Person)-[:KNOWS]->(b)-[:KNOWS]->(c)
WHERE a.person_name = 'Alice'
AND NOT (a)-[:KNOWS]->(c)
RETURN c.person_name;
```

Part 3: Build your own recommender

Data model



Let’s gradually build up our recommender

1. Universally popular hikes
2. A person’s favorite hikes
3. Hikes that similar people liked

Take a moment to run these in the browser:

1. ‘clean_database.cypher’
2. ‘import_hikes.cypher’

Warm-up queries

Find...

- All the hikes on the Olympic Peninsula
- All the hikes with more than 4000 feet of elevation gain (**hint**: use toInt())

1. Universally popular hikes

Let's define "popular" as the hikes that our hikers hiked most often.

Write a query that finds the most popular hikes.

2. A robot's favorite* hikes

Write a query that lists Bender's top 5 favorite (most hiked) hikes.

3. Suggest new* hikes to Bender

Write a query that:

1. takes Bender's 5 favorite hikes,
2. finds other people that liked those hikes, and
3. returns the favorite hikes of those other people.

Hint: WITH will be useful.

3. Suggest hikes to Bender

- Take the favorite hikes,
- And find other hikers who hiked Bender’s favorite hikes:
- We need to take Bender out of his recommendations:
- Now find those hikers’ favorite hikes that are not Bender’s favorite hikes:
- Now take out Bender’s favorite hikes and return some recommendations:

Improvements

- Score recommendations and boost/penalize some scores
- Remove irrelevant or lowly rated recommendations
  - Blacklist, filter
- Measure the quality of recommendations
- Different graph implementation

What I hope you've gained

- Appreciation for a new data structure
- Insight into the complicated world of recommendations

Non-technical concerns

- $$$
- Computers are not creative:
  - Apple and Spotify hire DJs to compile some of their playlists
- Machines make educated guesses but can't understand music or appreciate the utility of an item