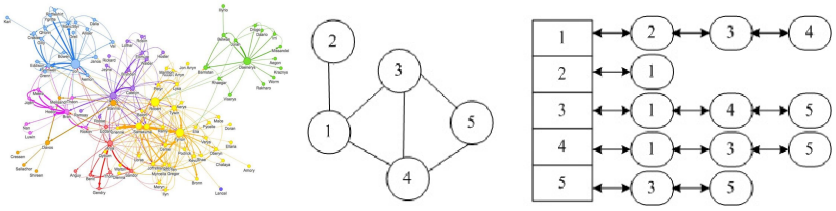
Graph Data Models before Graph Databases

DS4300: Large-Scale Storage and Retrieval

Prof. Rachlin



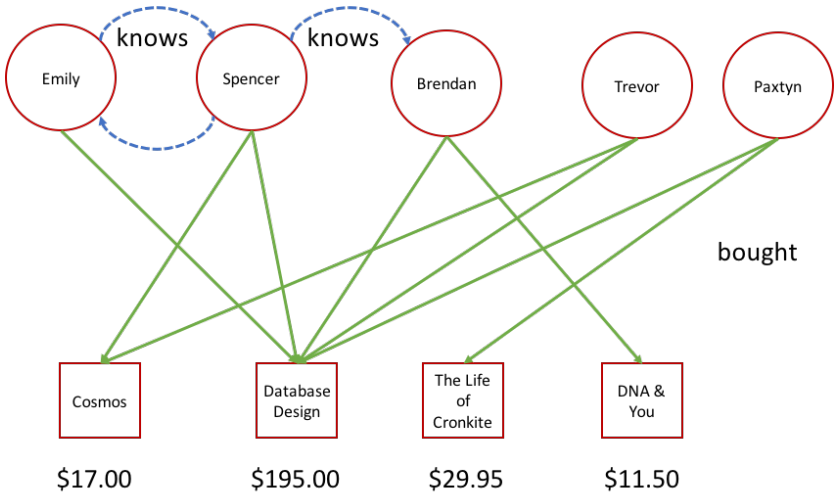
The NoSQL movement is being driven by two factors: 1) The need for greater scalability in managing large datasets in a distributed environment, and 2) the need for greater schema flexibility necessary to overcome the inherent limitations of the traditional relational model. In this homework, we will model generic graphs in both a relational database and using redis – everyone's favorite NoSQL database! We'll explore graph databases in class in the coming weeks.

Part I: Graphs in a Relational Database

The modeling of graphs and networks is critical for social networks, biological datasets, and for implementing recommendation engines commonly used by ecommerce giants like Amazon, Netflix, and Spotify. Consider the graph below, depicting people, other people they know, and the books they bought. The people have attributes (name) as do the books (title and price). There are different types of relationships (knows and bought). You could model this information pretty easily with a table for people and books. But your schema wouldn't be very flexible. You'd have to add columns to support new product attributes, and what if you sell millions of different products the way Amazon does? You can't support millions of different tables!

So instead, you decide to create a relational model to store ANY graph. Instead of tables for people and books, you know have tables for nodes, edges, and node properties supporting both strings (VARCHAR) and numeric (DOUBLE) properties. (We'll ignore relationship properties.) Run the **graph_start.sql** file to build your graph schema. (Note, this script was specifically tested with MySQL but should work with little or no modification with other databases.)

Congratulations, you managed to create a pretty flexible solution for storing graphs! *But at what cost?* Write SQL queries to answer each of the questions below. You will discover for yourself why relational modeling imposes certain challenges for this type of problem.



QUERIES (10 pts each): Write SQL queries that answer the following questions:

a. What is the sum of all book prices? Give just the sum.

b. Who does spencer know? Give just their names.

c. What books did Spencer buy? Give title and price.

d. Who knows each other? Give just a pair of names.

e. Demonstrate a simple recommendation engine by answering the following question with a SQL query: What books were purchased by people who Spencer knows? Exclude books that Spencer already owns. *Warning: The algorithm we are using to make recommendations is conceptually simple, but you may find that your SQL query is rather complicated. This is why we need graph databases!*

Part II: Graphs in Redis

Create an API for storing graphs (nodes and relationships) in Redis. Your API should support the following core graph functions:

`add_node(name, type)`
Add a node to the database of a given name and type

`add_edge(name1, name2, type)`
Add an edge between nodes named name1 and name2.
Type is the type of the *edge* or *relationship*.

`get_adjacent(name, node_type=None, edge_type=None)`
Get the names of all adjacent nodes. User may optionally specify that the adjacent nodes are of a given type and/or only consider edges of a given type. For example, the books that Spencer bought might be determined with a call similar to the following:

```
get_adjacent('Spencer', node_type='Book', edge_type='bought')
```

Congratulations! You now have a way of managing graph models in a scalable NoSQL database. Now implement ONE application-specific API to make book recommendations:

`get_recommendations(name)`
Get all books purchased by people that a given person knows but exclude books already purchased by that person.

Demonstrate your methods by re-building the person/friend/book graph above programmatically (via API calls) and demonstrating that you can obtain book recommendations for Spencer with a single API call.

What to submit:

1. **hw3_queries.sql:** A file containing your queries for part I.

2. **hw3_output.txt:** A textfile containing the output of each of your queries above (Or you may embed the output as comments in the .sql file.)

3. **hw3_api.py (.java, .c, etc.):** Your code for implementing the graph API in Redis (Part II).

4. **hw3_recommend.py (.java, .c, .etc):** Your code for demonstrating the correctness of your API. It builds the person/book network and obtains book recommendations for Spencer.