

Some Reminders for a Seamless Online Class...

- Please turn on your video
- Mute yourself (press and hold spacebar when you'd like to talk)
- Don't do anything you wouldn't do in an in-person class
- I will occasionally check the chat for messages if you'd like to share there instead
- Please say your name before you speak



Recap

- Data-savviness is the future!
- “Classical” relational databases
 - Notion of a DBMS
 - The relational data model and algebra: bags and sets
 - SQL Queries, Modifications, DDL
 - Database Design
 - Views, constraints, triggers, and indexes
 - Query processing & optimization
 - Transactions
- Non-classical data systems
 - Data preparation:
 - Semi-structured data and document stores
 - Unstructured data and search engines
 - Data Exploration:
 - Cell-structured data and spreadsheets
 - Dataframes and dataframe systems
 - OLAP, summarization, and visual analytics
 - Batch Analytics:
 - Compression and column stores
 - Parallel data processing and map-reduce
 - Streaming, sketching, approximation
 - Special Topics:
 - **Graph processing systems**



Today's Lecture

- Let's talk about graphs!



A lot of real-world datasets can be modeled as graphs...

- Social network datasets
 - A follows B, A is a friend of B
- Internet dataset
 - Page A links to Page B
- Road network datasets
 - Road A is connected to road B
- Scientific literature datasets
 - Paper is written by author, cites another paper
- Scientific interaction datasets
 - Disease treated by drug, drug A and drug B have interactions



So how do we manage and process graphs?

- Graph databases:
 - We'll briefly describe two different types of models for representing graphs
 - Property graph model
 - Systems like Neo4J, Titan, InfiniteGraph
 - Triple-store
 - Systems like Datomic, AllegroGraph, ...
 - Contrast with RDBMS representation
 - Talk about querying languages: Cypher and SQL for RDBMS representations
- Graph processing systems:
 - Graphlab and Pregel (briefly)

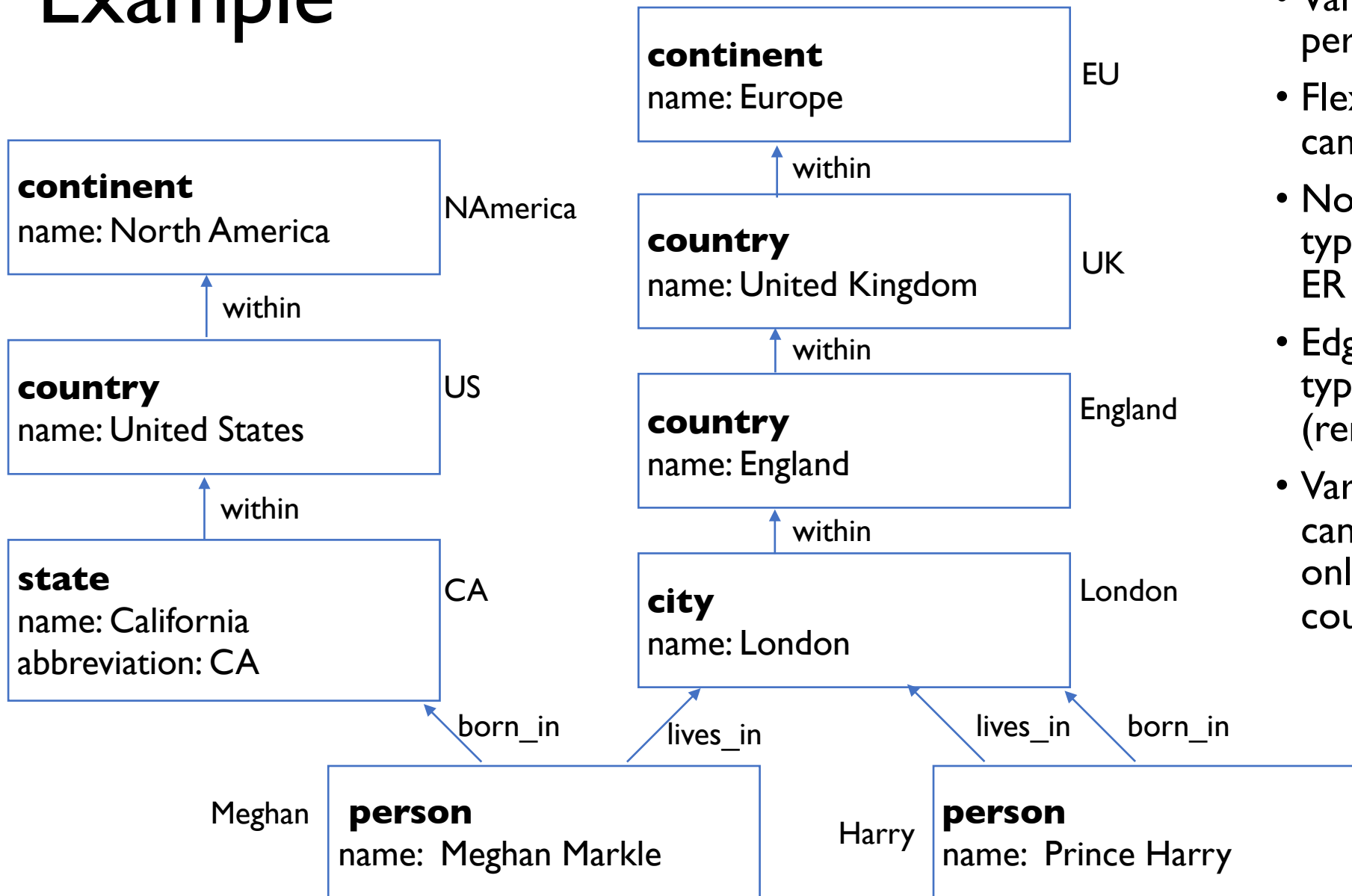


Graph Databases: The Property-Graph Model

- Only two different types of entities:
 - Node and Edges
- Each Node consists of:
 - A unique identifier
 - A type for the node
 - A collection of properties (key, value pairs, like in JSON)
- Each edge consists of
 - A unique identifier
 - A head and a tail node id
 - A type for the relationship
 - A collection of properties (key, value pairs, like in JSON)



Example



- Varying number of properties per node
- Flexibility in edges: a country can be within a country
- Nodes encompass a variety of types of entities (remember ER diagrams!)
- Edges encompass a variety of types of relationships (remember ER diagrams!)
- Varying granularities: **born_in** can be a state if that is the only information known, or could be a city if that is known



Why is this powerful?

- At a high level, there are only two “relations” — nodes and edges, as opposed to the relational schema where you may have many different entities and relationships, each encoded into its own relations
 - Can easily evolve by adding new nodes, edges, or adding properties to existing nodes or edges
- Unlike pure json which nests everything making it hard to operate on/query, the property graph data model surfaces the graph itself as an important structure to “traverse” and walk across forward or backward
- To see this, we’ll look at some examples of queries
- But before that, how do we represent this data model in an RDBMS?



Supporting Property Graphs in RDBMSs

CREATE TABLE Nodes

(node_id STRING PRIMARY KEY, type STRING, properties JSON)

CREATE TABLE Edges

(edge_id STRING PRIMARY KEY,
head_id INTEGER REFERENCES Nodes (node_id),
tail_id INTEGER REFERENCES Nodes (node_id),
type STRING, properties JSON)

Property graphs allow rapid traversal on both head and tail nodes, so

CREATE INDEX edge_tails ON Edges (tail_id);

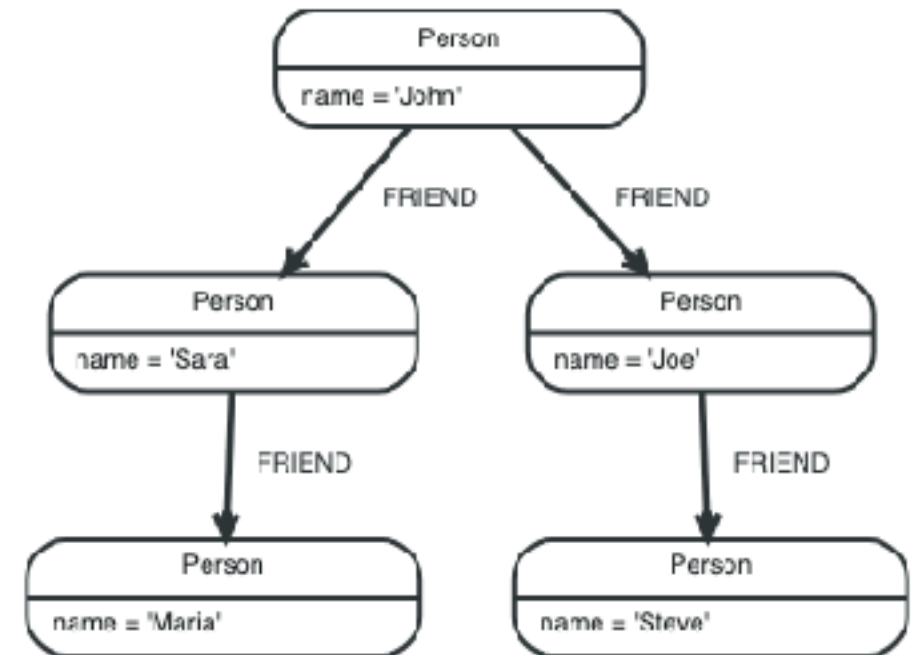
CREATE INDEX edge_heads ON Edges (head_id);



Cypher Query Language (from Neo4j)

- A declarative query language for property graphs
- First let's consider a simple example that we construct...

```
CREATE (john:Person {name: 'John'})
CREATE (joe:Person {name: 'Joe'})
CREATE (steve:Person {name: 'Steve'})
CREATE (sara:Person {name: 'Sara'})
CREATE (maria:Person {name: 'Maria'})
CREATE (john)-[:FRIEND]->(joe)-[:FRIEND]->(steve)
CREATE (john)-[:FRIEND]->(sara)-[:FRIEND]->(maria)
```



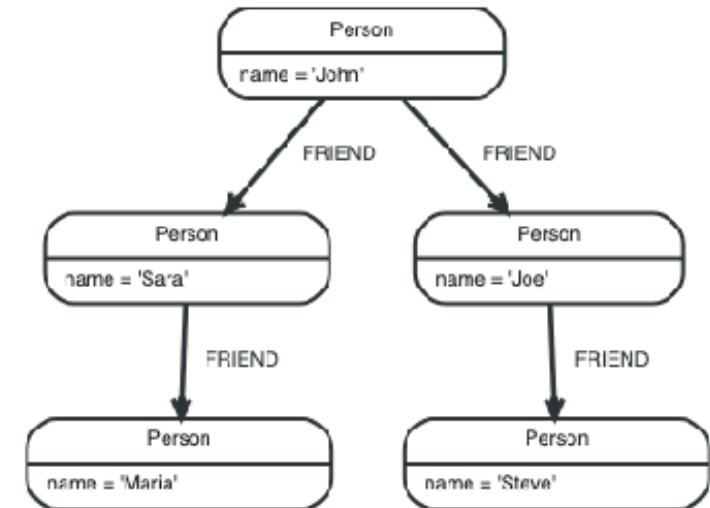
Cypher Query Language (from Neo4j)

```
MATCH (john {name: 'John'})-[:FRIEND]->()-[:FRIEND]->(fof)
RETURN john.name, fof.name
```

- What do we think this query returns?

```
MATCH (user)-[:FRIEND]->(follower)
WHERE user.name IN ['Joe', 'John', 'Sara', 'Maria', 'Steve'] AND follower.name =~ 'S.*'
RETURN user.name, follower.name
```

- What do we think this query returns?

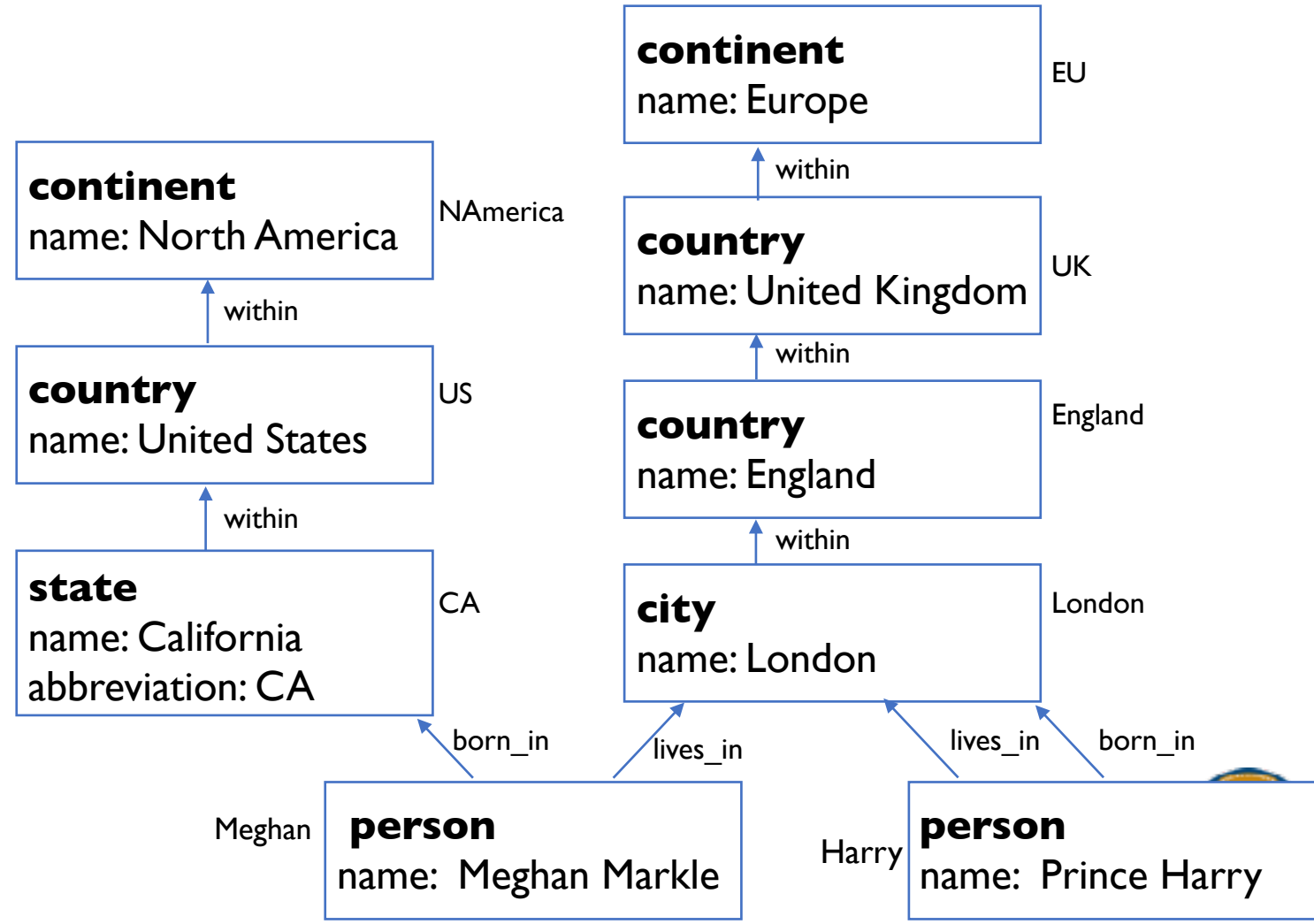


More complicated query

MATCH

```
(person)-[:BORN_IN]->()-[:WITHIN*0..]->(us:Country {name: 'United States'})  
(person)-[:LIVES_IN]->()-[:WITHIN*0..]->(eu:Continent {name: 'Europe'})
```

RETURN person.name



How would we issue the same query in SQL?

- Query: find people who were born in the US and live in Europe.
 - Let's focus on a subquery: we just want to find all the locations within the US.
 - Nodes (node_id, type, properties)
 - Edges (edge_id, head_id, tail_id, type, properties)
- We use *recursive common table expressions*
 - WITH defines a CTE; WITH RECURSIVE defines a CTE where the relation can refer to itself.

```
WITH RECURSIVE in_usa (node_id)
```

```
AS (
```

```
  SELECT node_id FROM Nodes WHERE properties->>'name' = 'United States'
```

```
  UNION
```

```
  SELECT tail_id FROM Edges JOIN in_usa ON Edges.head_id = in_usa.node_id
```

```
  WHERE Edges.type = 'WITHIN'
```

```
)
```

```
SELECT * FROM in_usa;
```



How would we issue the same query in SQL?

- We use *recursive common table expressions*
- WITH defines a CTE; WITH RECURSIVE defines a CTE where the relation can refer to itself.
- Semantics:
 - Start by adding the contents from the first subquery
 - Then, in each round, we “add” new content using the second subquery
 - We stop when we have no more to add

```
WITH RECURSIVE in_usa (node_id)
```

```
AS (
```

```
  SELECT node_id FROM Nodes WHERE properties->>'name' = 'United States'
```

```
  UNION
```

```
  SELECT tail_id FROM Edges JOIN in_usa ON Edges.head_id = in_usa.node_id
```

```
  WHERE Edges.type = 'WITHIN'
```

```
)
```

```
SELECT * FROM in_usa;
```



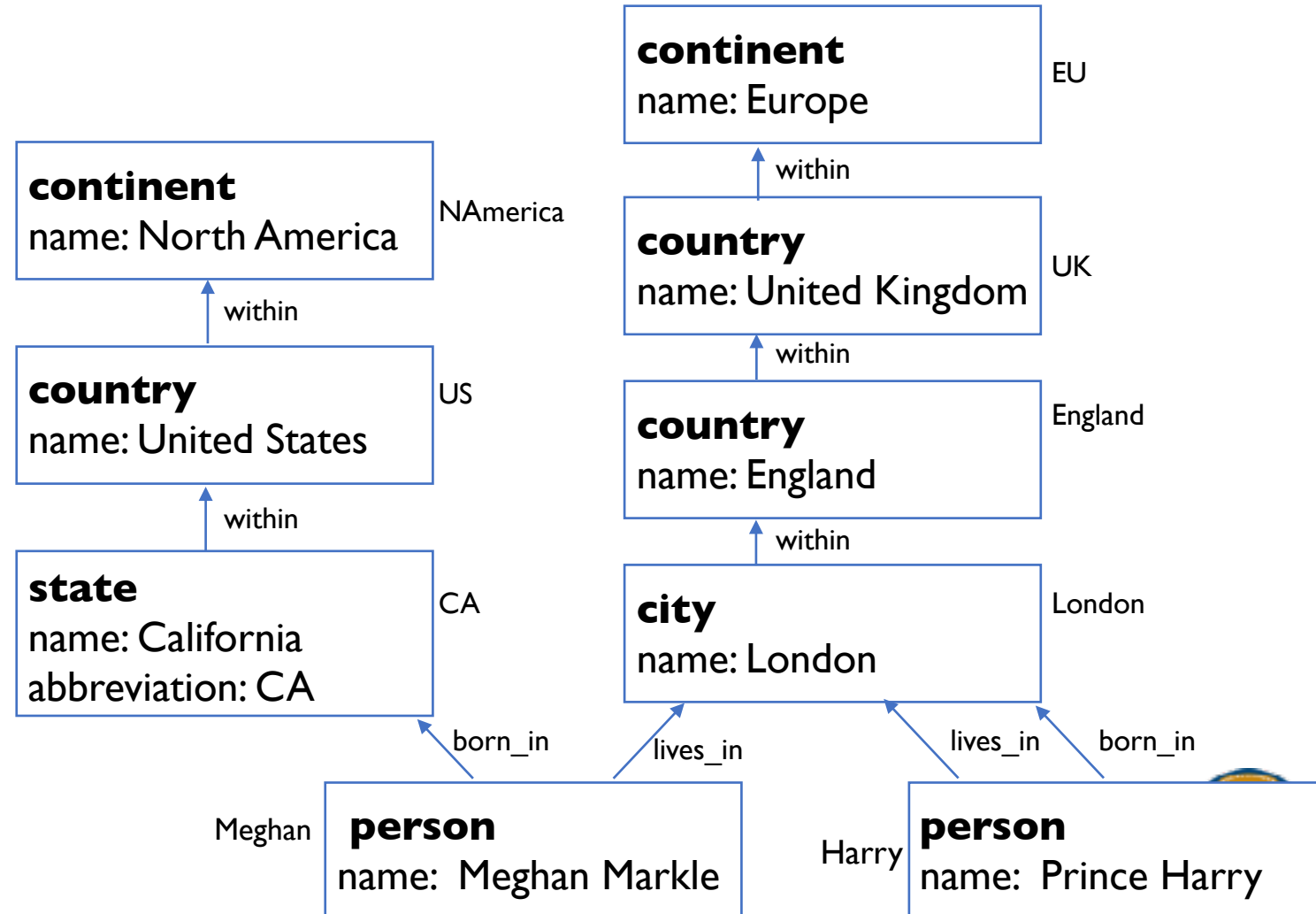
Constructing the query in SQL

- The full query is even more complicated: this was just one of four query fragments
- In general, really hard to do graph traversal type queries via SQL



Triple-Stores

- A different model to represent the same information as property graphs
- Everything is a triple
 - (subject, predicate, object)
- For example:
 - (Meghan, type, person)
 - (Meghan, name, Meghan Markle)
 - (Meghan, born_in, CA)
 - (Meghan, lives_in, London)
 - (CA, type, state)
 - (CA, name, California)
 - ...



Triple-Stores

- A different model to represent the same information as property graphs
- Everything is a triple
 - (subject, predicate, object)
- RDF: Resource Description Format: is a mechanism for encoding data in triple stores
- SPARQL: Query language, like Cypher, that operates on RDF



Graph Processing Systems

- Another type of systems for graphs focus on analytics on graphs as opposed to management of graph data
- For example:
 - Computing connected components of a graph
 - Computing pagerank of nodes
- Also many ML algorithms!



Large Scale ML



Map Reduce

Feature Extraction Cross Validation
Computing Sufficient Statistics

Graphical Models
Gibbs Sampling
Belief Propagation
Variational Opt.

Collaborative Filtering
Tensor Factorization

Semi-Supervised Learning
Label Propagation
CoEM

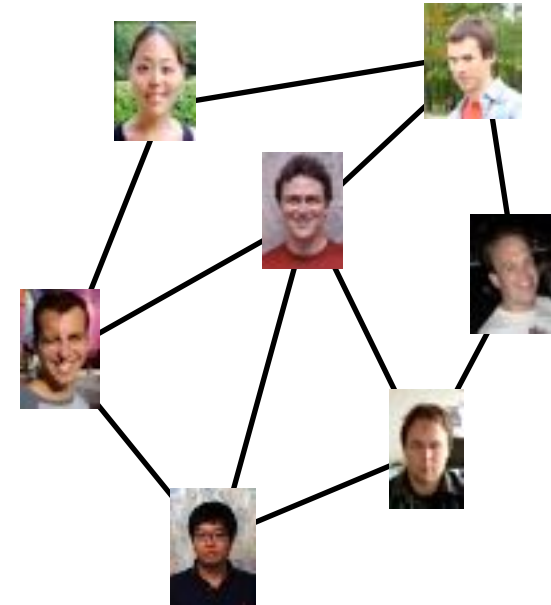
Graph Analysis
PageRank
Triangle Counting

Label Propagation for Content Recommendation

- Recurrence Algorithm:

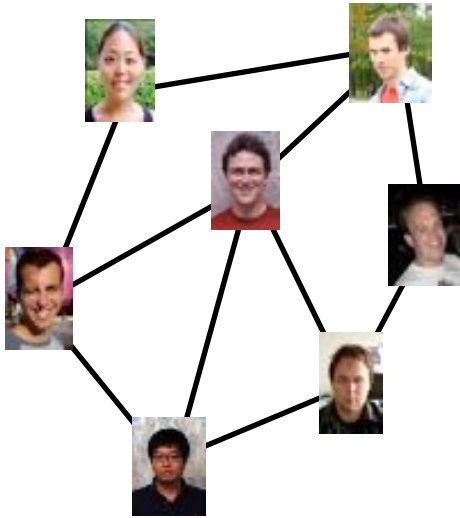
$$Likes[i] = f \left(\sum_{j \in Friends[i]} g(w_{ij}, Likes[j]) \right)$$

- iterate until convergence
- Parallelism:
 - Compute all $Likes[i]$ in parallel

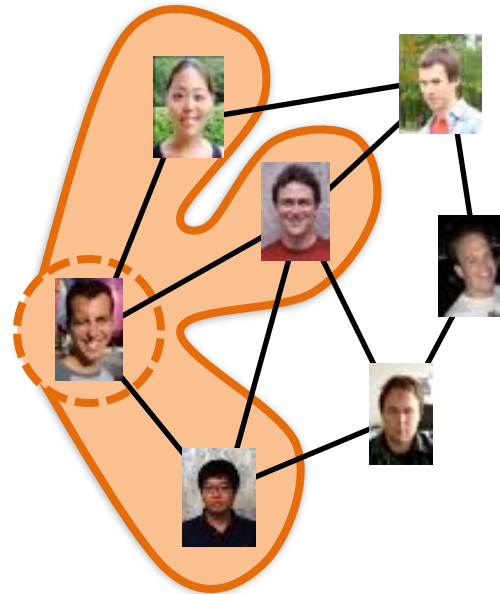


Properties of Graph-Parallel Algorithms

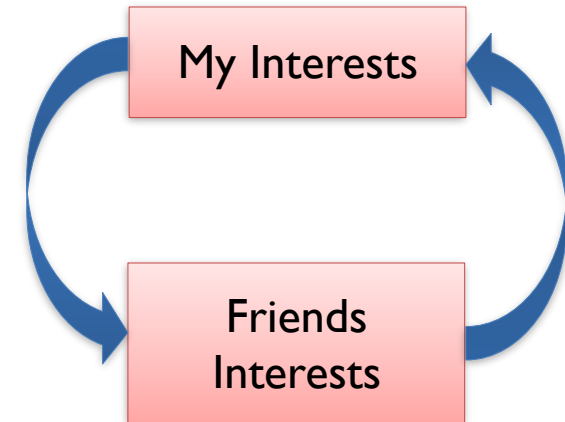
Dependency
Graph



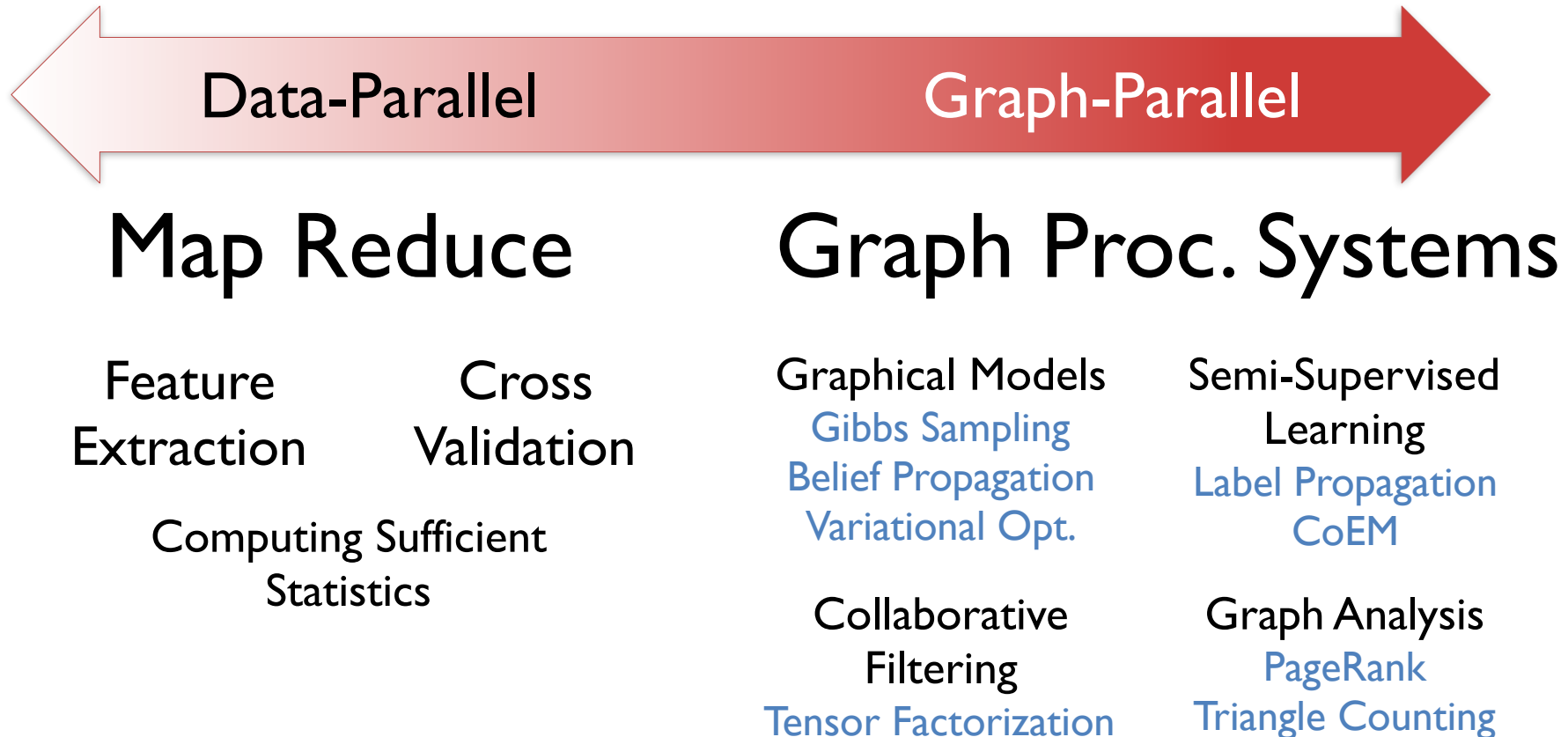
Local
Updates



Iterative
Computation

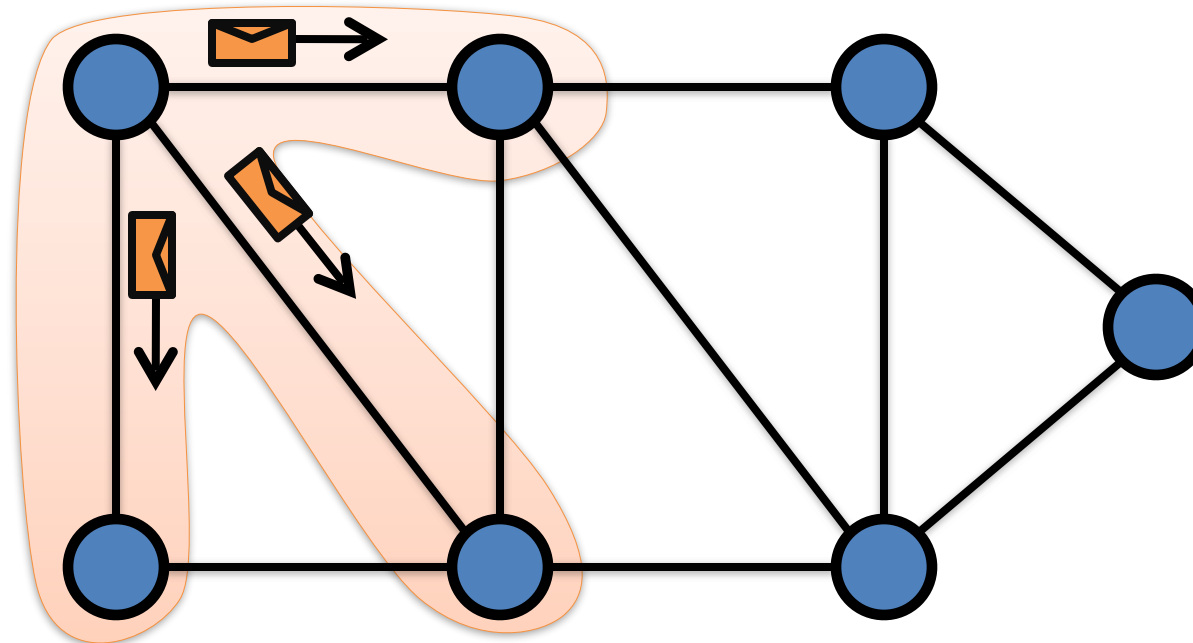


Large Scale ML



Graph-Parallel Abstractions

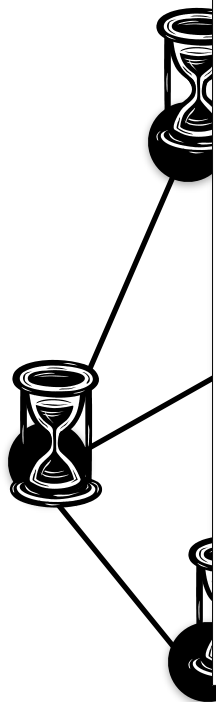
- **Vertex-Program** associated with each vertex
- Graph constrains the interaction along edges
 - Pregel: Programs interact through Messages: synchronous, exact
 - GraphLab: Programs can read each-others state: asynchronous, approximate



The Pregel Abstraction

Compute

Communicate



```
Pregel_LabelProp(i)
  // Read incoming messages
  msg_sum = sum (msg : in_messages)

  // Compute the new interests
  Likes[i] = f( msg_sum )

  // Send messages to neighbors
  for j in neighbors:
    send message(g(wij, Likes[i])) to j
```

Barrier



The Pregel Abstraction

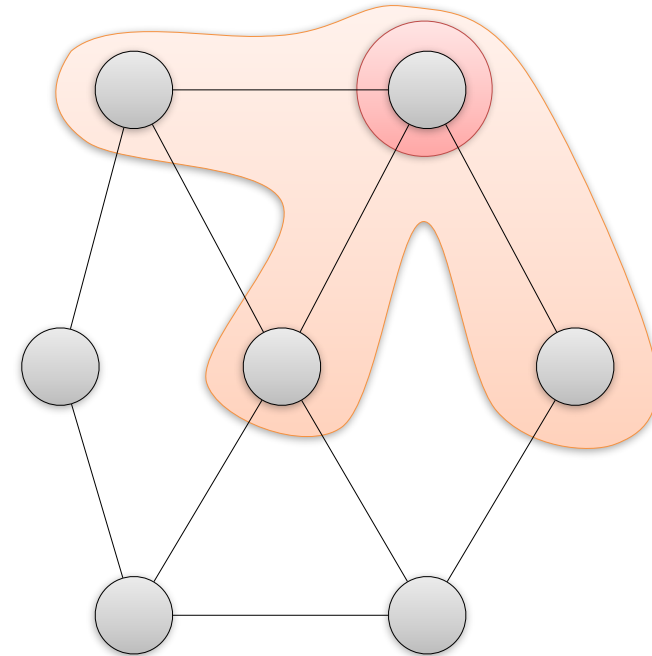
- “Thinking like a vertex”
- *Bulk Synchronous Parallel* (BSP) of parallel programming
- Each vertex maintains its own state and logic, and exchanges messages with neighboring vertexes in each round
 - After a point, if no vertexes receive any new messages, they can opt to terminate
- Q: Let’s say I want to compute connected components using pregel, how would I do it?



The GraphLab Abstraction

Vertex-Programs are executed asynchronously and directly read the neighboring vertex-program state. So Approximate BSP

```
GraphLab_LblProp(i, neighbors
Likes)
  // Compute sum over neighbors
  sum = 0
  for j in neighbors of i:
    sum = g(wij, Likes[j])
  // Update my interests
  Likes[i] = f( sum )
  // Activate Neighbors if needed
  if Like[i] changes then
    activate_neighbors();
```



Activated vertex-programs are executed *eventually* and can read the new state of their neighbors

Graph Processing Systems Today

- Apache Giraph, GraphX within Spark, Naiad (but also supports streaming applications), GPS, ...
- More recently, ML systems have become more tensor centric thanks to deep learning
 - TensorFlow, PyTorch, ... support distributed tensor manipulation in a semi-declarative interface
 - A lot to say here but we don't have time



Takeaways

- Graph databases: the property graph and triple store models
 - Can leverage structure for querying, while also not being tied to a schema
 - Can be approximated via relational databases, but SQL is not a convenient language for graph querying
- Graph processing:
 - Many systems to support efficient processing on graphs
 - The BSP modality is convenient for implementing graph algorithms
- Graph databases and graph processing still very much a work in progress. Jury is still out about their long-term effectiveness

