A Novel Query Language For Querying Graph Data

Hong Yang, Hao Tian, Rajshekhar Sunderraman
Computer Science Department
Georgia State University
Outline

- Motivation
- Goals
- Graph Data Model
- Graph Query Language
- Query Examples
- Implementation
- Conclusion
Motivation

- Data in many database applications are represented in the form of graph structures:
  - Geographical Information Systems
  - World Wide Web Searching
  - Social Networks
  - Biochemical Network
  - Neuron Network
  - etc.
- To deal with graph data in an effective manner, a well-designed *data model* and *query language* are needed.
Motivation

- General Purpose Data Models such as Relational/OO are capable of representing “graph” data.

- Disadvantages:
  - Difficult to express “graph” queries (dealing with path connectivity, shortest path, sub-graphs etc.)
  - Difficult to handle extensible attributes; in particular in the relational model.
  - Query optimizations are difficult.
Motivation

- Related Work
  - Gram (Amann and Scholl, 1992): A query language to specify sub-graphs in the database.
  - Graphlog (Consens and Mendelzon, 1990): A query language that uses graph representation for both data and queries (logic based).
  - GOQL (Sheng, Ozsoyoglu and Ozsoyoglu, 1999): Extension of OQL for graph data
  - GraphQL (He and Singh, 2007): A more recent work; Graph Algebra.

- Drawbacks with these systems:
  - Do not have the capability to specify various path conditions in the queries.
  - Cannot compose various graphs.
Goals

- Propose an object-oriented graph data model to represent the graph objects and their properties for various applications.
- Develop a graph query language which empowers users to query node, edge, path, or graph objects and their properties in a graph with various conditions.
Graph Data Model

- **Basic** classes in the graph data model:
  - **Node** class: represents nodes in graph.
  - **Edge** class: represents edges in graph.
  - **Path** class: represents paths; path objects are normally generated during the query processing.
  - **Graph** class: represents sub-graphs of larger graph; graph objects are normally generated during query processing.

- Application-specific **extended** classes in the graph data model:
  - The nodes in the graph may be of different data types, and each data type could be represented by a **extended subclass of Node** class.
  - The edges in the graph may be different data types as well, and each data type could be represented by a **extended subclass of Edge** class.
  - **Complementary** classes can be designed to help users to define objects referred by the Node subclasses or Edge subclasses.
Graph data model

Basic class schema:

class Node:
    ( name: string;
      other attributes);

class Edge:
    ( from: Node;
      to: Node;
      other attributes);

class Path:
    ( start: Node;
      end: Node;
      nodes: sequence<Node>;
      edges: sequence<Edge>;
      numberofnodes: integer;
      numberofedges: integer;
      other attributes);

class Graph:
    ( nodes: set<Node>;
      edges: set<Edge>;
      paths: set<Path>;
      numberofnodes: integer;
      numberofedges: integer;
      other attributes);
Graph Application Example
(Tritonia Neuron Network)

- The Tritonia Neuron Network consists of a set of nodes and directed edges:
  - Each node represents a neuron with a name
  - Each edge represents a synapse with a synaptic direction, and directed edges are colored differently to represent different types of synapse.

- Neurons communicate via synapses. One neuron can accept synaptic transmission from a pre-synaptic neuron and send synaptic transmission to a post-synaptic neuron.

- The synapse can be categorized into:
  - Chemical synapse
  - Electrical synapse
  - Modulation
  - Negative synapse

- The graph can be several unconnected sub-graphs or even a set of isolated nodes.

Figure 1. Tritonia Neuron Network:
Extended Subclasses for Neuron Network

• Nodes in neuron network represent only neurons, Neuron class is the only subclass of Node class:

```java
class Neuron: extends Node
    ( cell_count: integer;
    soma_size: integer;
    soma_location: enum('cerebral', 'pedal', 'buccal', 'unspecified');
    activity_resting: enum('silent', 'spiking', 'bursting', 'irregular');
    neuron_type: enum('interneuron', 'motor_neuron', 'sensory_neuron');
    function: enum('body_flexion', 'turning', 'arousal', 'withdrawal', 'escape');
    molecules: set<Molecule>;
    annotations: set<Annotation>;
    other attributes);
```

• Chemical_Synapse, Electrical_Synapse, Negative_Connection and Modulation classes are subclasses of Edge Class:

```java
class Chemical_Synapse extends Edge: ( connection_probability: Connection_Probablity;
    laterality: enum('bilateral', 'contralateral');
    synaptic_component: enum('excitatory', 'inhibition');
    molecule: set<Molecule>;
    annotations: set<Annotation>;
    other attributes);
```

```java
class Negative_Connection extends Edge: ( short_term_plasticity: enum('depressing', 'facilitating');
    connection_probability: Connection_Probablity;
    molecule: set<Molecule>;
    annotations: set<Annotation>;
    other attributes);
```

```java
class Electrical_Synapse extends Edge: ( connection_probability: Connection_Probablity;
    laterality: enum('bilateral', 'contralateral');
    annotations: set<Annotation>;
    other attributes);
```

```java
class Modulation extends Edge: (connection_probability: Connection_Probablity
    reversal_potential: enum('depressing', 'facilitating')
    annotations: set<Annotation>;
    other attributes);
```
Complementary Classes for Neuron Network

class Molecule:
( name: enum('5HT', 'peptide-unidentified', 'GABA', 'glutamate');
 type:  string;
 annotation: Annotation;
 other attributes);

class Reference extends Annotation:
( authors: set<Person>;
 pages: string
 PMID: integer;
 source: string;
 volume: integer;
 other attributes);

class Book extends Annotation:
( authors: set<Person>;
 city: string
 publisher: integer;
 year: string;
 other attributes);

class Book_Chapter extends Annotation:
( pages: string;
 source: string
 year: integer;
 other attributes);

class Example extends Annotation:
( image_type: enum('jpg', 'gif');
 image_url: string
 other attributes);
The GQL empowers users to query nodes, edges, paths and sub-graphs from graph with arbitrary conditions.

GQL queries consists of five elements:
- **Output** clause: specifies query outputs
- **Define** clause: specifies variables denoting collection of node objects, edge objects, path objects, and graph objects.
- **Where** clause: specifies query condition
- **Group by**
- **Order by**

The simple GQL can be combined through the set operators like:
- **UNION**
- **INTERSECT**
- **MINUS**
Graph Query Language

General Form of Query:

Output <Output-list>
Define <Node-formulas>|<Edge-formulas>|<Path-formulas>|<Graph-formulas>
Where <selection-predicate>
Group By <grouping-objects-attributes>
Order By <ordering-attributes>

The define clause allows the definition of one or more variables using node, edge, path, and graph formulas.

Most of the “expressibility” of the language comes from the define clause.

The output clause may output objects, their properties (dot-path expressions), or aggregates.
Formulas in Define Clause

**Node/Edge formula:**

Nodes $n[(\text{subclass}) \ <\text{predicate}>]$  
Edges $e[(\text{subclass}) \ <\text{predicate}>]$

where  
$n, e$: are variables; the bracketed expression is optional.  
$(\text{subclass})$ is optional and provides the ability to work with application specific classes.  
$<\text{predicate}>$ is a selection condition on the attributes of the Node/Edge objects.  
If $(\text{subclass})$ is mentioned, objects of that type satisfying the predicate are returned; Otherwise Node or Edge objects satisfying the predicate are returned.
Formulas in Define Clause

Simple Path Formula:

Paths

\[
\begin{align*}
&n_1[(\text{subclass}) \  \langle \text{predicate} \rangle]\n&e_1[(\text{subclass}) \  \langle \text{predicate} \rangle]\n&n_2[(\text{subclass}) \  \langle \text{predicate} \rangle]
\end{align*}
\]

where
n1, e1, n2: variables
Formulas in Define Clause

Simple Path Formula With Regular Expression:

Paths
n1[(subclass) <predicate>] _* n2[(subclass) <predicate>]

Paths
n1[(subclass) <predicate>] _N n2[(subclass) <predicate>]

Paths
n1[(subclass) <predicate>] _S n2[(subclass) <predicate>]

where
n1, n2: variables
*: one or more edges; N: N edges, S: shortest path
Formulas in Define Clause

General Path Formula:

Concatenate one or more simple path formulas by introducing edges between them:

\[ P_1 \ e_1[(\text{subclass}) \ <\text{predicate}>] \]
\[ P_2 \ e_2[(\text{subclass}) \ <\text{predicate}>] \]
\[ \ldots \]
\[ \ldots \]
\[ P_{n-1} \ e_{n-1}[(\text{subclass}) \ <\text{predicate}>] \]
\[ P_n \]
Formulas in Define Clause

Graph formula:

Graphs

\[ \langle \text{path-formula} \rangle \ p_1, \ldots, \langle \text{path-formula} \rangle \ p_n, \]
\[ \text{[graph-predicate]} \]

A graph object is constructed from a collection of paths based on the graph-predicate that specifies connecting points in the paths.

One path from each \( p_i \) is chosen and connected using the graph-predicate.
Informal Semantics

The informal operational semantics of a GQL query is described as follows:

- Each formula in the “define clause” produces a collection of objects. A Cartesian product of the various collections of objects is obtained.

- The tuple of objects that satisfy the where-predicate are retained.

- If the “group by” clause is present, the results are split into partitions based on partition attribute(s).

- If the “order by” clause is present, the results are sorted by the attribute defined in the “order by”.

- Output clause expressions are evaluated as query results.
Query Example 1

1. Find the neurons which are electrically coupled with each other.

```
Output p1.nodes
Define
    Paths n1[(Neurons)]e1[(Electrical_Synapse)]n2[(Neurons)] as p1,
    Paths n3[(Neurons)]e2[(Electrical_Synapse)]n4[(Neurons)] as p2
Where p1.nodes.elementAt[1] = p2.nodes.elementAt[2] and
    p1.nodes.elementAt[2] = p2.nodes.elementAt[1];
```
Query Example 2

2. Find synaptic outputs of the neurons which have molecule ‘5HT’

Output p1
Define
Paths n1[(Neurons) n1.molecule.name='5Ht'] e1
  n2[(Neurons)] as p1
Group by p1.start;
3. Find the shortest connection between neurons DSI and P17.

Output `p1`
Define

Paths `n1[(Neurons) n1.name='DSI'] _S`  
n2[(Neurons) n2.name='P17'] as p1;

4. Find the ganglion containing neurons that receive input from sensory neurons.

Output `p1.end.ganglion`
Define

Paths `n1[(Neurons) n1.neuron_type='Sensory_Neuron'] e1`  
n2[(Neurons)] as p1;
5. Find all processes that lead from node A to node B in less than 5 steps, and more than 3 steps in the Biochemical Network.

Output \( p_1 \)

Define

\[
\text{Paths } n1[n1\.name='A'] \_ \ast n2[n2\.name='B'] \text{ as } p_1
\]

Where \( p_1\.length \geq 3 \) and \( p_1\.length \leq 5 \);
6. Find sub-graphs that match the following pattern:

Output g1
Define
  Graphs
    n1[n1.name='A'] _2 n2[n2.name='B'] as p1,
    n3[n3.name='B'] _ n4[n4.name='C'] _ n5[n5.name='D'] as p2,
    n6[n6.name='A'] _* n7[n7.name='D'] as p3,
    [p1.end=p2.start and p2.end=p3.end and p1.start=p3.start] as g1;
7. Find sub-graphs that match the following pattern:

Output g1
Define
Graphs
n1[n1.name='A'] _ n2[n2.name='C'] _ n3[n3.name='D'] as p1,
n3[n3.name='B'] _ n4[n4.name='C'] as p2,
[p1.nodes.elementAt(2)=p2.nodes.elementAt(2)] as g1;
Implementation

- The query is sent to the GQL parser (using JFlex/JCUP) to verify the syntax of the query.

- The formulas in the define clause are evaluated by the GQL evaluator, then the outputs of the evaluator are inserted into OODB.

- The GQL translator translates the rest parts of the query into an OQL query, then the OQL query is used to query the OODB to get the desired results.

Architecture of Query System
Conclusion

Summary
- Presented an object-oriented graph data model used to represent various graph structure data.
- Proposed a graph query language empowering users to query objects and properties of the graph.

Future Work:
- Complete implementation
- Build graph applications (e.g. NeuronBank)
- Usability testing (e.g. neuro-scientists)