MicroDB Graph Database

Sreenivas Appasani
Ajay Mandlekar
Ruthwick Pathireddy
Sathwick Pathireddy

CS123 Database Systems Project
California Institute of Technology

June 4, 2015

1 Motivation

Information is connected in the world and relationships are a central aspect of modeling such data. Traditional Relational Database Management Systems (RDBMS) model relationships through join-like operations, which are computationally intensive. There needs to be an efficient way to express relationships between large sets of data. Graph databases fulfill such a need and are a powerful alternative to relational databases.

2 Query Language

MicroDB's Query Language supports both basic and commonly-used graph queries as listed below:

- Create, modify, and delete nodes.
- Create, modify, and delete edges.
- Query for arbitrary length node-edge path relationships.
- Determine if a path exists between two nodes.
- Find the shortest path between two nodes.

2.1 Internal Data Structures

Our graph database utilizes the networkx package [1] to maintain a graph representation in memory during an active database session. The graph infrastructure provided by this package is used to create and manipulate nodes and edges during an interactive user session.

2.2 Nodes

Internally, each node is assigned a unique identifier, which is just a number. Each node's attributes is stored in a dictionary with attribute names as keys and attributes as values. When a query returns a set of nodes, the nodes are returned as a list of tuple pairs which contain its unique identifier and its dictionary of attributes.

2.2.1 Example Queries

The following query creates a node which has attributes "Label", "Name", and "Salary".

CREATE n: id1 Label:Boss Name:Donnie Salary:1000000;

In this query, "id1" is simply a local identifier that can be used to refer to the created node by the user. The user can view the results that are stored in any given identifier by using the RETURN command (for example, RETURN id1;). Another useful command for viewing the current contents of the in-memory graph is the SHOW command. It might be a good idea to run a SHOW command after every one of the following commands.

The following query modifies the existing node's attributes.

MODIFYNODE n: prev Name:Donnie n: new Name:Ruthwick b: new val:1;

In this case, only the "Name" attribute is modified. It should be noted that this command will match on all nodes whose "Name" attribute is "Donnie" and update their "Name" attribute to be "Ruthwick". This allows for a simple mechanism to update several related nodes at once. The important part of the "b:" portion of the query is the value of 1. This value is a boolean flag that indicates whether the attribute is to be updated or deleted. If this flag had been set to 0, the "Name" attribute would have been deleted (the new "Name" value would have been irrelevant). The other fields present in the "b:" part of the query are dummy fields that are necessary for the parser. Note that both "prev" and "new" are both identifiers just as "id1" in the create node query above.
The following query deletes the node.

DELETENODE n: fst Name:Donnie;

In particular, the query deletes all nodes whose "Name" attribute is "Donnie". Once again, "fst" is used as an identifier. However, the identifier serves no purpose here. It is included for ease of parsing.

2.3 Edges

Internally, each edge is stored in a dictionary that uses the two node identifiers as a key and the relationship attributes as a value. When an edge is queried, the result is represented as a tuple triple which consists of two node identifiers and a dictionary of attributes that describe the relationship. The keys in the dictionary are edge attribute names and the values are the edge attributes.

2.3.1 Example Queries

Let's continue the previous example. We first create a second node.

CREATE n: id2 Label:Employee Name:Sreeni Salary:50000;

Then, we create an edge between the nodes.

CREATEEDGE n: id1 Label:Boss e: id3 Relation:Manager n: id2 Label:Employee;

The command creates an edge between all nodes whose "Label" attribute is "Boss" and all nodes whose "Label" attribute is "Employee". The edge has an attribute called "Relation" and the value is set to "Manager". Note that this is a directed edge from the first node to the second node.
The following query modifies the edge by changing its attribute.

MODIFYEDGE e: fst Relation:Manager e: snd Relation:Co-worker b: oth val:1;

This command works just like the MODIFYNODE command. It updates all edges with "Relation" fields equal to "Manager" to set their fields to "Co-worker".
The following query deletes the edge we just created.

DELETEEDGE e: a Relation:Co-worker;

This command deletes all edges whose "Relation" field is equal to "Co-worker".

2.4 Match Query

Match queries are used to query for arbitrary length node-edge-node relationships. It can also be used to query for a single type of node or a single type of edge. The internal process used to match a single node or single edge simply involves a networks dictionary lookup. For arbitrary length relationships, we implemented an algorithm that repeatedly utilizes set intersection on node-edge-node triples in the path until we collapse the nodes to our desired result.

2.4.1 Example Queries

The following query matches a single node with the following attributes.

MATCH n: id1 Animal:Dog;

The command finds nodes with the attribute key "Animal" whose value is "Dog".

The following query matches a single edge with the following attributes.

MATCH e: id2 Relation:Enemy;

The command finds edges that contain an attribute key "Relation" whose value is "Enemy".

The following query matches a node-edge-node path with some attributes.

MATCH n: id1 Animal:Dog e: id2 Relation:Enemy n: id3 Age:10;

The command finds nodes that not only have the attribute dictionary "Animal" : "Dog" but is also connected to an edge with "Relation" type Enemy which is connected to another node with an attribute dictionary "Age":10.

2.5 Graph Functionality

In addition to basic creation, deletion, and matching of nodes and edges, MicroDB provides functionality to perform common graph operations. Most of these operations are implemented as an abstraction on top of the graph library operations the networkx package provides. Other operations are provided to ease user experience during an interactive session.

2.5.1 Example Operations

The following operation determines is a path exists between two nodes specified by the given attributes.

HASPATH n: id1 Name:Ruff n: id2 Name:Furry;

In this query, we want to find out if there is a path between a node with name "Ruff" and a node with name "Furry".

The following operation displays the shortest path between the two nodes if it exists. If no path exists, a message is displayed to the user.

SHORTESTPATH n: id1 Name:Ruff n: id2 Name:Furry;

In this query, we display the shortest path between a node with name "Ruff" and a node with name "Furry".

The following operation returns a all neighbors of nodes that contain some specific attributes.

NEIGHBOR n: id1 Name:Ruff;

In this query, we want to know all the neighbors of the node with name "Ruff".

The following operation determines if an edge exists between two nodes specified by the given attributes.

HASEDGE n: id1 Name:Ruff n: id2 Name:Furry;

In this query, we want to know if there is a directed edge from the node with name "Ruff" to the node with name "Furry".

The following operation displays the common neighbors, or mutual nodes, between two nodes specified by the given attributes.

COMMONNEIGHBORS n: id1 Name:Ruff n: id2 Name:Furry;

The query displays nodes that are connected to both nodes with the name "Ruff" and nodes with the name "Furry".

The following operation returns the value stored in an identifier.

RETURN n: id1;

The operation displays the value stored in the identifier id1.

The following operation returns the value stored in an identifier.

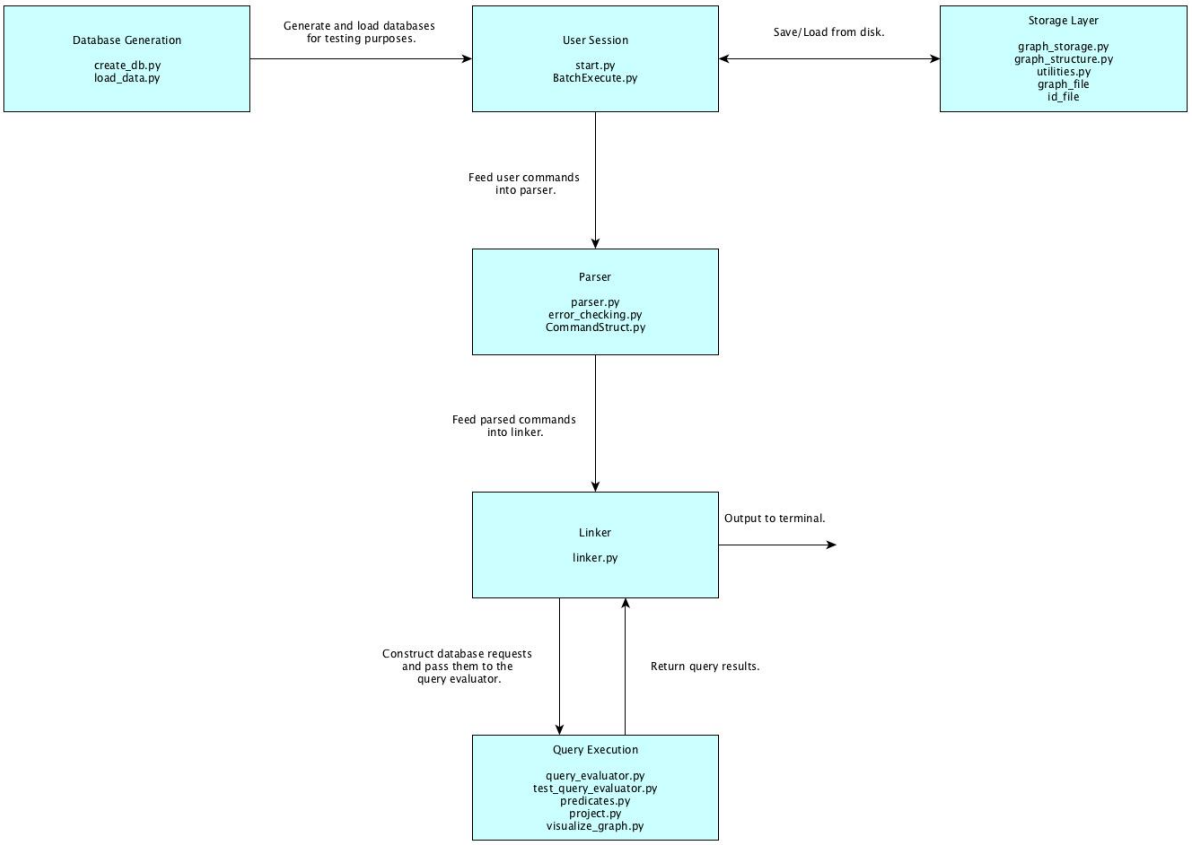
RETURN n: id1;

The operation displays the value stored in the identifier id1.

3 Database System Architecture

3.1 Overview

The following diagram describes interactions between major components of the system architecture as well as the files that pertain to each component.



3.2 User Interface

3.2.1 Files

- start.py - the main program that starts the database and handles all user interaction.
- BatchExecute.py - allows the user to run a batch of commands from a text file.

3.3 Parser

3.3.1 Files

- parser.py - contains the main parser implementation.
- error_checking.py - error checking functionality used by the parser.
- CommandStruct.py - wrapper around commands for the parser, it also details all of the commands and their syntax.

3.4 Linker

3.4.1 Files

- linker.py - contains the entire linker implementation.

3.5 Query Evaluator

3.5.1 Files

- query_evaluator.py - responsible for servicing all queries.

- test_query_evaluator.py - unit testing framework for servicing queries.
- predicates.py - responsible for processing predicates.
- project.py - responsible for executing project operations.
- visualize_graph.py - produces a neat visualization of the database. Essentially responsible for executing the visualize command.

3.6 Graph Storage

3.6.1 Files

- graph_storage.py - contains the basic storage layer for loading the database from disk and writing the database to disk.
- graph_structure.py - contains the in-memory graph structure implementation, with the help of the networkx package.
- utilities.py - miscellaneous useful functions for file I/O.
- graph_file - file that contains the entire database on disk.
- id_file - file that saves the last unique identifier generated by the database on disk.

3.7 Testing

4 Results

5 References

References

[1] Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, "Exploring network structure, dynamics, and function using NetworkX", in Proceedings of the 7th Python in Science Conference (SciPy2008), Gel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11-15, Aug 2008