Graphlite Documentation

Release 1.0.3

Eugene Eeo

Apr 12, 2017

Contents

Graphlite is a tiny graph datastore that stores adjacency lists similar to FlockDB but like conventional graph databases, allow you to query them with traversals (graph-walking queries), and works with datasets that you can fit into your SQLite database.

```
from graphlite import connect, V
graph = connect(':memory:', graphs=['knows'])

with graph.transaction() as tr:
    for i in range(2, 5):
        tr.store(V(1).knows(i))
    tr.store(V(2).knows(3))
    tr.store(V(3).knows(5))

# who are the friends of the mutual friends
# of both 1 and 2?
graph.find(V(1).knows)\
     .intersection(V(2).knows)\
     .traverse(V().knows)
```

If you are not familiar or new to the library perhaps you should check out *Introduction to Graphlite*. Else, you are most likely looking for the *API Documentation*.

CHAPTER 1

Introduction to Graphlite

What is Graphlite?

Graphlite is a social graph datastore. It doesn't store properties between relations, but it can store thousands of relations between integers, which are most likely representing objects in your other databases, such as the ID column of your users/statuses table.

Being based on a relational DBM, Graphlite supports very atomic transactions, similar to the transactions that SQLite offers even with transactions, performance isn't degraded because the SQLite library is very, very fast.

Graphlite aims to be performant, thread safe, and have a pleasant API for developer happiness. For example, to create a transaction:

```
with graph.transaction() as tr:
    for item in range(2, 6):
        tr.store(V(1).knows(item))
```

And one thing I'm very happy with is how querying works and the expressiveness of the querying "syntax":

```
graph.find(V(1).knows)\
     .union(V(3).knows)\
     .traverse(V().posted)
```

Cheatsheet

To connect to an existing SQLite database file (substitute `URI` and `GRAPHS` with the URI of your database file and the graphs that you want to query/insert to):

```
from graphlite import connect, V
graph = connect(URI, graphs=GRAPHS)
```

To insert (possibly multiple) relation(s), you must create a transaction and call methods of the transaction object:

```
with graph.transaction() as tr:
    tr.store(V(1).knows(2))
```

To query the graph, you can simply do:

```
graph.find(V(1).knows)    # people that 1 knows
graph.find(V().knows(1)) # people that knows 1
```

Querying has a few more "tricks", notably the powerful set operations that you can do:

```
graph.find(...).union(...)
graph.find(...).difference(...)
graph.find(...).intersection(...)
```

They should be quite familiar to you (remember the Venn diagrams from school?). If you are not familiar with set operations or would like a demo of how they work I would recommend looking at this diagram.

Graphlite also supports "graph-hopping" or graph traversal queries, in spite of the fact that it was inspired by FlockDB:

```
graph.find(V(1).knows).traverse(V().knows)
```

The above query states that "find all of the people that 1 knows, and then find all of the people that *they* know". You can also pass in a destination node to the second query, to select the source nodes:

```
graph.find(V(1).knows).traverse(V().knows(2))
```

Which means "find all of the people that one knows, that knows 2". This can also be expressed with the help of an intersection:

```
graph.find(V(1).knows).intersection(V().knows(2))
```

Note that you can traverse indefinitely, i.e. to find out who are the friends of friends of the people that know 2, you can do:

```
graph.find(V().knows(2)).traverse(V().knows)\
                        .traverse(V().knows)
```

You can also count the nodes returned by a query via the `count` method:

```
graph.find(V(1).knows).count()
```

To delete edges from the datastore, you have three options:

- Specific deletes
- Inverse & Forwards deletes
- Relation-wide deletes

To illustrate,

```
with graph.transaction() as tr:
    tr.delete(V(1).knows(2))
    # delete edges of type "1 knows ..."
    tr.delete(V(1).knows)

    # delete edges of type "... knows 1"
    tr.delete(V().knows(2))
```

```
    # delete edges of type "... knows ..."
    tr.delete(V().knows)
```

CHAPTER 2

Usage

Note: If you have read through the cheatsheet then this document for you, will just go deeper into the internals of Graphlite.

Initializing the graph

To initialize a graph object, you have two options- using the *graphlite.graph.Graph* object or the *graphlite.connect()* function. Usually you would use *graphlite.connect()* because it encapsulates anything that the codebase would want to do in the future.

```
from graphlite import V, connect
graph = connect(':memory:', graphs=['knows'])
```

Note that since Graphlite is based internally on SQLite (in fact it can be thought as a minimal wrapper around SQLite to give you a graph layer), you will need to pass in the graphs that you want to create and query because the appropriate tables need to be created.

Inserting edges

Graphlite represents edges as a row which contains a source node, a destination node, which is where the source node is "pointing to", i.e. in the edge "John knows Don", John is the source node and Don is the destination node. Graphlite also stores the nodes as unsigned integers, so you will need a separate backing store to store the documents, i.e. key-value.

```
with graph.transaction() as tr:
    for item in range(2, 5):
        tr.store(V(1).knows(item))
    tr.store(V(3).knows(1))
    tr.store(V(2).knows(6))
    tr.store(V(6).knows(7))
```

When your generator gets too large, it is often better to use the *graphlite.transaction.Transaction.store_many()* method because it's more efficient in terms of space:

```
with graph.transaction() as tr:
    tr.store_many(V(1).knows(n) for n in range(2, 200))
```

Tip: anything that modifies the graph (i.e. storage, removal) will be done within a transaction. This is partially because Graphlite is based on an SQLite backend and implementing transactions are quite straightforward this way.

Transactions are automatically committed at the end of the `with` block, so you don't have to hold a lock throughout the entirety of the block.

Querying

Querying can come in two flavours- you either do a forwards query, where you select the destination nodes and specify the source node, or an inverse query, where you get the source nodes but specify the destination node. Again, best explained by example:

```
>>> list(graph.find(V(1).knows()))
[2, 3, 4]
>>> list(graph.find(V().knows(1)))
[3]
```

You can also do queries which involve set operations, i.e. unions, differences, and intersections. They are all very efficient and does not require any data processing on our (Graphlite's) side because they can be represented easily by set operations. Possible queries:

```
graph.find(...).intersection(...)
graph.find(...).difference(...)
graph.find(...).union(...)
```

Graph traversal queries are also possible via Graphlite. For example to select the friends of friends of 1:

```
graph.find(V(1).knows).traverse(V().knows)
```

And you can also specify the destination node to the `traverse` query to select the source nodes that have the specific relation to the destination node. For example, to select the friends of friends of 1 that are friends with 2:

```
graph.find(V(1).knows).traverse(V().knows(2))
```

Perhaps you want to keep traversing and find out the friends of those people? You can do that as well:

```
graph.find(V(1).knows).traverse(V().knows(2))\
                      .traverse(V().knows)
```

You can also slice the query objects the same way you'd slice a slice object, but you will only get an iterable back. For example to get the first five people that 1 knows:

```
graph.find(V(1).knows)[:5]
```

Note that you can only iterate over the iterable, because internally Graphlite uses the `itertools.islice` function to generate an iterable that takes the slice into account. Basically, the need to do this is because we:

- Need to take `slice.step` into account
- Want to prevent people from doing queries like `graph.find(...)[1:].union(...)` because these are not allowed in SQLite, as only the rightmost select can contain a `LIMIT` statement.

Deleting Edges

Deleting edges can come in four flavours- you either do a specific delete of a specific edge, a forwards query, then delete all the rows (edges) matching it, an inverse query, or just wipe out everything from the table. Either way, an example would illustrate it best:

```
with graph.transaction() as tr:
    tr.delete(V(1).knows(2))

    # every edge with source node 1
    tr.delete(V(1).knows)

    # every edge with destination node 2
    tr.delete(V().knows(2))

    # everything within the knows table
    tr.delete(V().knows)
```

Similar to *graphlite.transaction.Transaction.store_many()* method, you should use the *graphlite.transaction.Transaction.delete_many()* method if you are deleting many specific nodes at once. For example:

```
with graph.transaction() as tr:
    tr.delete_many(V(1).knows(i) for i in gen())
```

Note that transactions are not locked, in a sense that the code within the `with` block is not ran in a thread lock. The lock will only be held during block exit, which is also when the transaction will be committed. Also, nested transactions are **not** recommended. They will not be treated as a single atomic operation since there is no way to enforce atomicity when we have multiple transactions within a transaction.

CHAPTER 3

API Documentation

graphlite.**connect**(*uri*, *graphs=()*)
Returns a Graph object with the given *uri* and created *graphs*.

Parameters

- **uri** – The URI to the SQLite DB.
- **graphs** – The graphs to create.

class graphlite.graph.**Graph**(*uri*, *graphs=()*)
Initializes a new Graph object.

Parameters

- **uri** – The URI of the SQLite db.
- **graphs** – Graphs to create.

close()
Close the SQLite connection.

find
Returns a Query object that acts on the graph.

setup_sql(*graphs*)
Sets up the SQL tables for the graph object, and creates indexes as well.

Parameters **graphs** – The graphs to create.

transaction()
Returns a Transaction object. All modifying operations, i.e. `store`, `delete` must then be performed on the transaction object.

class graphlite.transaction.**Transaction**(*db*, *lock*)
Represents a single, atomic transaction. All calls are delayed jobs- they do not execute until the transaction is committed.

Parameters

- **db** – An SQLite connection.

- **lock** – A threading.Lock instance.

abort()
Raises an `AbortSignal`. If you used the `Graph.transaction` context manager this exception is automatically caught and ignored.

clear()
Clears all the operations registered on the transaction object.

commit()
Commits the stored changes to the database. You *don't* have to call this function if the transaction object is used as a context manager. A transaction can only be committed once.

delete(*edge*)
Deletes an edge from the database. Either the source node or destination node *may* be specified, but the relation has to be specified.

Parameters **edge** – The edge.

delete_many(*edges*)
Delete multiple edge queries from the database. Best used when you have a fairly large generator that shouldn't be loaded into memory at once for efficiency reasons.

Parameters **edges** – An iterable of edges or `Graph.find` style edge queries to delete.

perform_ops()
Performs the stored operations on the database connection. Only to be called when within a lock and a database transaction by the `commit` method.

store(*edge*)
Store an edge in the database. Both the source and destination nodes must be specified, as well as the relation.

Parameters **edge** – The edge.

store_many(*edges*)
Store many edges into the database. Similar to the *graphlite.transaction.Transaction.delete_many()* method.

Parameters **edges** – An iterable of edges to store.

class graphlite.query.**Query**(*db*, *sql=()*, *params=()*)

count()
Counts the objects returned by the query. You will not be able to iterate through this query again (with deterministic results, anyway).

derived(*statement*, *params=()*, *replace=False*)
Returns a new query object set up correctly with the *statement* and *params* appended to the end of the new instance's internal query and params, along with the current instance's connection.

Parameters

- **statement** – The SQL query string to append.
- **params** – The parameters to append.
- **replace** – Whether to replace the entire SQL query.

difference
Compute the difference between the current selected nodes and the another query, and not a *symmetric difference*. Similar in implementation to *graphlite.query.Query.intersection()*.

intersection
Intersect the current query with another one using an SQL INTERSECT.

statement
Joins all of the SQL queries together and then returns the result. It is the query to be ran.

to(*datatype*)
Converts this iterable into another *datatype* by calling the provided datatype with the instance as the sole argument.

Parameters **datatype** – The datatype.

traverse(*edge*)
Traverse the graph, and selecting the destination nodes for a particular relation that the selected nodes are a source of, i.e. select the friends of my friends. You can traverse indefinitely.

Parameters **edge** – The edge query. If the edge's destination node is specified then the source nodes will be selected.

union
Compute the union between the current selected nodes and another query. Similar to the *graphlite.query.Query.intersection()* method.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index