AjguDB

Python GraphDB for exploring connected data

Amirouche Boubekki

Getting started

```
> pip install ajgudb
```

Kesako AjguDB?

- graph database written in Python
- with functional querying similar to Tinkerpop's gremlin
- handles bigger than RAM dataset thx to wiredtiger

Why?

Graph awesomness without leaving the confort of you favorite language

API

```
from ajgudb import *
```

Create a graph database

```
graphdb = AjguDB('/tmp')
```

Create a vertex

```
vertex = Vertex()
vertex['movie'] = 'Alien 3'
vertex['year'] = 1992
graphdb.save(vertex)
print(vertex.uid)
```

Create an edge

```
a = graphdb.save(Vertex())
b = graphdb.save(Vertex())
edge = a.link(b)
graphdb.save(edge)
print(edge.uid)
```

Get

Get object with uid 42

```
vertex_or_edge = graphdb.get(42)
```

Querying

Querying is done using gremlin!

Figure 1: gremlin

Kesako gremlin? (1)

- `gremlin(*steps)` is a composition of *step* functions
- a *step* function takes an iterator as input and returns another iterator
- values generated by steps are chained to be able to go back to previous results

Kesako gremlin? (2)

```
def gremlin(*steps):
    """Gremlin pipeline builder and executor"""

    def composed(ajgudb, iterator=None):
        # ... some magic happens here
        # to accept various things as `iterator`
        for step in steps:  # compose
            iterator = step(ajgudb, iterator)
        return iterator

    return composed
```

Kesako gremlin? (3)

Steps most of the steps look like this:

```
def step(ajgudb, iterator):
    for item in iterator:
        # do something with item
        yield out
```

Kesako gremlin? (4)

- map
- filter
- reduce
- navigate the graph (which is a map actually)

and `back` to navigate the results...

Kesako gremlin (5)

- querying is dynamic
- you have to be aware of what goes through each step
- no checks are done so the query might be succesful but the result garbage

Kesako gremlin (6)

The steps provided by AjguDB operate on vertex and edge **uids** to save memory.

Seed steps

Must be at the start of query.

- `VERTICES` generate uids for every vertex in the database
- `EDGES` generate uids for every edges in the database
- `FROM(key=value)` generate uids for **things** where `key == value`

Navigation steps (1)

When the pipe contains the uid of a vertex you can:

- `outgoings` yields *a list* of edge uids that starts at the current vertex
- `incomings` yields *a list* of edge uids that ends at the current vertex

You will most-likely use `scatter` after those function to flatten the generator.

Navigation steps (2)

When the pipe contains the uid of an edge you can:

- `start` yield the edge's start vertex
- `end` yield the edge's end vertex

`gmap(func)`

yields application of `func(ajgudb, value)` to every value found in the iterator

`gfilter(func)`

Keep value if `func(ajgudb, value)` returns `True`

back

Retrieve the previous result. Useful after a `gfilter` was applied.

path(count)

Return the path of values that leads to the current value.

`key(name)`

- Retrieve the value associated with `name`
- This expects vertex or edge uids as input.

`where(**kwargs)`

- The generator must contain vertex or edge uids.
- Only keep elements which match the `kwargs` specification.
- Similar to SQL `WHERE` clause

Helpers

- `skip`
- `limit`
- `paginator(count)`
- `count`
- `value`
- `get`
- `sort(key, reversed)` (consume the iterator)
- `unique` (lazy)
- `mean`
- `group_count`