AjguDB

Python GraphDB for exploring connected data

Amirouche Boubekki

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 Python

Getting started

```
$ git clone https://github.com/wiredtiger/wiredtiger
$ cd wiredtiger && git checkout develop
$ ./autogen.sh && ./configure --enable-python && make
$ make install
$ pip install ajgudb
```

Internals (1)

- wiredtiger is a storage engine
- kind of like leveldb but more powerful
- a tuple space is used to store vertex and edges

Internals (2)

There a table with an index.

```
(
    (1, '__kind__', 'vertex'),
    (1, 'person', 'Amirouche'),

    (2, '__kind__', 'vertex'),
    (2, 'movie', 'Alien 3'),
    (2, 'year', 1992),

    (3, '__kind__', 'edge'),
    (3, '__start__', 2),
    (3, '__end__', 1),
    (3, 'label', 'like'),
)
```

API

```
from ajgudb import *

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

Create a vertex

```
vertex = Vertex()  # Vertex inherit dict
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)  # Edge inherit dict too
graphdb.save(edge)
print(edge.uid)
```

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)

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

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

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

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

Other steps

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