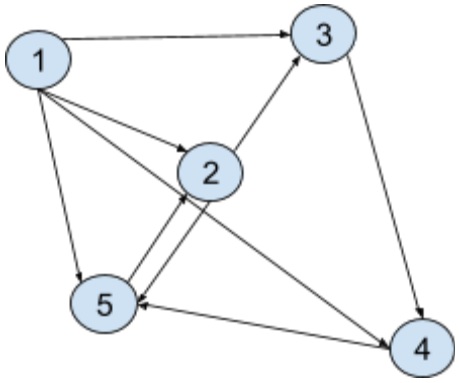
Problem Statement

Below is the detailed description of the task and that you should think about and implement before the interview.

The goal is to set up a web service that provides an API for clients to run algorithms and perform updates on the graphs. The graphs it will deal with are normal, directed graphs with a set of nodes and edges. There can be cycles in the graph. Each node and edge should have a primary ID and additional metadata like a name and other attributes. The web service needs functions to persist and query the basic graph structure (CRUD methods).

I want you to think about how the graph data structure and the method signatures of the API would look like to do the necessary operations and queries. These methods and data structure should be implemented as a service and provided to clients for consumption. If you run in to time trouble, it is OK to keep data in memory for the implementation and don't do a full data store implementation, but there should be a concept and the data structure ready how you will persist it.



A key requirement is that graphs can be very large, containing hundreds of millions of nodes and edges. Think for example of the Facebook or Twitter social graph. There will be questions in the interview how this can be scaled on a technology level and what an implementation would like like if certain aspects of the graph change. Also the data structure, interfaces and architecture should reflect this and take things like highly concurrent updates, conflicts, sub graph updates and querying, high read or write demands into account that you would expect from such an API. If not everything can be implemented there should be concepts and answers in place in your solution to take care of these challenges.

The second part of the task is to design a client that consumes the graph API to solve a graph problem. The problem is to find the shortest path between two nodes of the graph. Again, important are data structure and this time also how the algorithms works and performs in combination with the API you designed, specifically under the sizing criteria above. Consider if you have to provide more than basic CRUD operations and return data for your type of algorithm in a smarter way to reduce overhead of too many REST calls. Or maybe there are better ways to solve this problem with non naive algorithms or data models when storing the data.

Constraints

You can use any technology of choice for the implementation, just Python as a language is set, but you can choose any database, libraries or other tools around that you feel will help you to get the job done. You can write the client in another language than the server if it is easier and makes sense.

If anything in the description is not specified you can either ask for clarification or are welcome to make an assumption. As long as you can argue for your design choice and assumptions that is fine.

Other Notes

Besides the actual solution, it is also important for the interview that you keep notes for a presentation on how you approached the problem and what problems you encountered. During the interview we would like to get a walkthrough on how you solved the problem, a presentation with some notes and pictures on your thought process and architecture / data model are very helpful there. We will also iterate a bit on your solution and challenge certain assumptions to see how your solution holds up to changes and what you would need to change to make it work under the new assumptions.

Good luck and have fun with it. Get back to me in case anything is unclear or you have questions.

Solution Documentation

The problem statement requires a service based on graphs to be implemented to be used by an external clients. The requirement requires basic CRUD and other operations which is to be expected in such a data structure. The example on the type of graph shared is the Facebook or Twitter social graph. We have made some assumptions about the data structure and relationships.

Assumptions

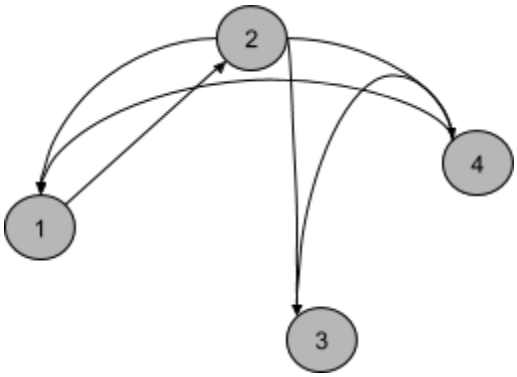


fig. 1

Facebook or Twitter or similar social graphs have nodes and edges. We are assuming the edges have non-negatives weights assigned to them, we will see how this might affect our approach later. An edge can begin from any node and can end at any node, but it must have a beginning node and an ending node. Due the example being Social graphs we are assuming an edge can't begin and end at the same node. Based on the example shared, we are assuming there can be at most only one edge can exist for each ordered pair of nodes. With the following properties set, let's look at the problem statement.

Problem Statement

PART A

Web Service to perform CRUD operations on the graph, query persistent data from the graph, highly concurrent updates, conflicts, sub graph updates and querying, high read or write.

PART B

Client to query the graph data from the web service and to calculate the shortest path between 2 nodes.

Considerations

The main points to be considered before we make a decision on the solution are

1. Ease of use for integrations with Web Frameworks
2. Future Scalability
3. Building blocks (Data Structures)

Ease of Use

The software should be easy to integrate with existing web frameworks to design the API to be served for this exercise.

Future Scalability

How scalable is the solution for large scale data. The test should include at least a sizable test dataset.

Building Blocks

How does the building blocks of the solutions affect our scalability question? Can we choose better solution based on the building blocks of a system.

Evaluation

Some of the solutions that were evaluated during this exercise are as follows:

1. Neo4j
2. Redisgraph
3. Apache Giraph
4. Self Implementation

Neo4j

Neo4j is what we call a graph database, which is a database designed around graph data. With core primitives designed to be efficient in querying, processing and modifying graph data.

Core Language	: Java
Python Driver bindings	: Available
Scalability	: High
Ease of Use	: High
Building Blocks	: Adjacency List (type of linked list with neighbors)

The solution is highly scalable and also has good python bindings for its drivers to be integrated with good python based web frameworks to implement the required business requirement.

Redisgraph

Redisgraph is redis based graph database built around fast matrix computations based on sparse matrix principles. This is a very interesting solution providing possible huge performance improvements by using GPU based acceleration for computation.

Core Language	: C
Python Driver bindings	: Available
Scalability	: Very High
Ease of Use	: Medium
Building Blocks	: Adjacency Matrix (stores data as sparse matrices)

The only issue with this package is that it's a great solution for querying graph data but more features are to be implemented in terms of graph processing. To implement a shortest path algorithm you would have to custom develop over the existing data structure of Adjacency Matrices, due to the large size of the data structure this would tight development over GraphBLAS which is the core library on which all the computations are running. So this may be worth to work on in the long term, this doesn’t fit well with the short term problem of submitting a solution soon.

Apache Giraph

Apache Giraph is a graph processing software developed based on the PREGEL Concept of Vertices based Graph processing system.

Core Language	: Java
Python Driver bindings	: Available through jython
Scalability	: High
Ease of Use	: Low
Building Blocks	: Bulk Synchronous Parallel

This is graph processing system, we would have to work on CRUD requirements separately using a graph db or nosql db. We will have to work with jython bindings and has less documentation on python examples and tutorials.

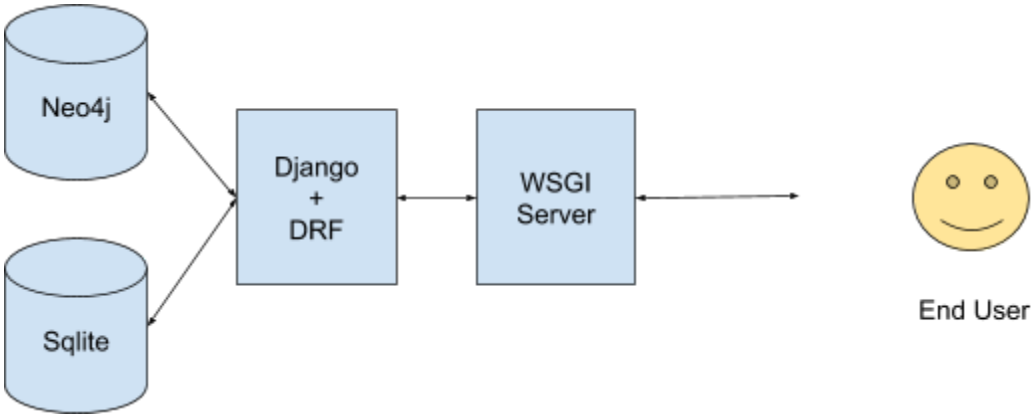
Self implementation

Usually winning strategies stand on the shoulders of giants who have done most of the work before. So in this case I think it's not efficient to work on a complete self implementation. If given the time I would rather work on the implementation based on Redisgraph, which I think is super cool especially the sparse matrix computation based calculation and the GPU based acceleration possibilities.

Implementation

For the current implementation I went with Neo4j based solution due to the time constraints at this moment. If given more time I would be tried the redisgraph based approach, I see a lot of potential there, but lack of good graph algorithm support left me with only one option.

Architecture



Database

For our implementation we are going with dual Databases Neo4j and Sqlite. Neo4j for the graph based requirements i.e. CRUD Operations and Shortest Path. Sqlite for the normal Django Requirements like User Management, Permissions and other stock tables.

Framework

I am choosing Django due by understanding of django and its flexible plug and play model. If we want to modify some components in future we can reimplement them again.

Server

For the server we are going with the stock WSGI server due to the time constraints. In future we can look at an implementation with NGINX as a proxy server for a more scalable deployment.

Models

```
class Person(StructuredNode):
  uid = UniqueIdProperty()
  name = StringProperty(unique_index=True)
  age = IntegerProperty(index=True, default=0)
  friends = RelationshipTo('Person', 'FRIEND', model=FriendRel)
  country = RelationshipTo(Country, 'IS_FROM')
```

Person class defines a Person as an object with a name, age, country and a relationship with other persons called friends.

Test Data

Dataset : https://snap.stanford.edu/data/soc-LiveJournal1.html

SocialNetwork of LiveJournal site with over 4 million users and 65 million relationship. For the test case I considered only the first million users and the relationships between them.

Data Loading took 2 hrs on 4vCPU, 16GB RAM system.

Average Time to query shortest path between users : < 3 secs

Code

Repo : https://github.com/aswincsekar/GraphAPI.git

API Methods

Shortest Path:

http://127.0.0.1:8000/persons/shortest_path/?start=a0cc4d975be74d0aa3e16daeb6920edf&end=5c8e63fc6ad943368bde261e91be552b

Query params : start, end
Type : Person UID

Response :
{
	Path_length : 2
}

CRUD Operations

GET, POST at URL : http://127.0.0.1:8000/persons/
PUT, DELETE at URL : http://127.0.0.1:8000/persons/{id}/

Possible Future Work

I am interested in a Redisgraph based implementation which seems to really a good implementation from ground up designed around sparse matrices as core data structure which allows high efficiency in computations and storage of the relationship data.