Realtime salesforce-GraphQL database integration

Why

Graph databases are a hot topic, and growing in adoption. This is due to some of the performance benefits available, namely that joining many tables in a query is not subject to the same performance impact in a Graph database as would be the case in a Relational database.

A customer of mine is a heavy user of graph db technology, and wanted to reduce the data being replicated into their salesforce instance, just so users could reference this information in the context of the salesforce Account and Contact records.

Given that creating a query to this graph database was unlikely to be a one time thing (addition integration use cases were likely to arise once this project is successful), putting a re-usable realtime query framework in place was the most attractive option. This can be done with a Salesforce Connect connection, using a custom adapter for the Graph database.

There is already another article on creating salesforce connect custom adapters, so this article will just highlight additions and differences to that process.

What

As a proof of concept of this integration, we chose to use some easily available resources and data. We spun up a trial environment of a Neo4j graph database, using the graphenedb.com add-on for Heroku. GraphenDB provides a handy graph database tutorial with sample data, as well as a public REST interface. The data available is actor and movies data, and their relationships (Acted In, Directed, Produced, etc...) .

For this POC, we'll focus just on a simple query of the people information in the Person “node” or table.

Overview of the integration - Swimlane
<add content>

Actors: user, salesforce platform, custom adapter, graphenedb
User:
Refreshes Listview on Person Tab

The code is here if you want to try it in your org -
https://github.com/eyewell/salesforce2Graph

How

- Spin up a sandbox
- Spin up a heroku app - provision grapheneDB add-on
    - Lookup the access and authentication information for this graph db. Add the Access URL to the whitelist of your sandbox's list of Remote Site Settings: https://.graphenedb.com:24780 -
    - Note: in order to use HTTPS, I had to use the $60/mo version of the GrapheneDB Add-on.
- Create the Custom Adapter Apex files
    - GraphQLDataSourceProvider class - see details below
    - GraphQLDataSourceConnection class - see details below
    - GraphQLParser - see details below
    - Note: as long as your high level class definitions compile, you'll be able to go on to the next step, and create the external data source.
- Create the External Data source in salesforce setup.
    - Once the sync() function is completed in GraphQLDataSourceConnection, you can sync the connection to create the person account
- Create the Custom Tab to make your testing life easy, via the salesforce setup menu

Pre-requisites & Handy Tools

- Knowledge of Apex, or at least java, and awareness of the resources at developer.salesforce.com
- An IDE & Debugger - I used Eclipse IDE + Apex Debugger
    - the Apex Debugger saved me hours and hours of time - it let me set breakpoints in apex code, step through code, look at complex variable values and their structure (key during parsing). link?
- A graph database - The graphenedb educational resources and environment. I was able to learn about basic queries, and then flip to the code/HTTP view, and see the header definitions, and the JSON output. link?
- A REST query tool - Curl was handy in figuring out basic graphenedb connectivity and REST queries

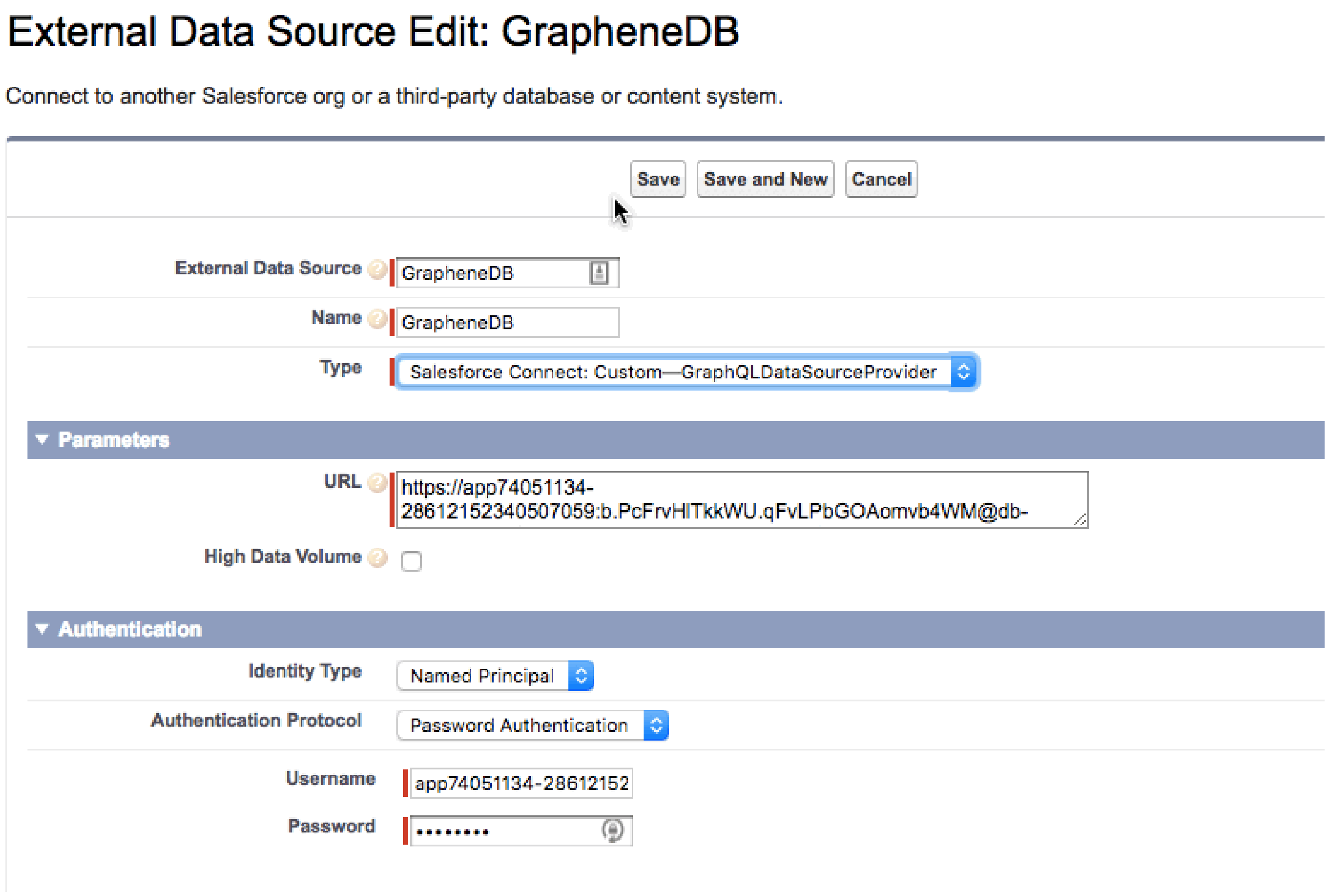
Config/Code Details

Creating the External data source:

Enter in the required values.

- External Data Source: give it a name
- Name: give it a technical name (no spaces)
- For Type, once you have saved the apex class files for the GraphQLDataSourceProvider, you'll find a new option in the type field for your custom adapter.
    - The URL value for graphenedb rest calls was: https://db-fatcbidnasczf5j0p2us.graphenedb.com:24780/db/data/transaction

- For URL, I had to play with a few curl calls to figure out what the URL should be. For my test environment, this is what I used:
    - https://db-fatcbidnasczf5j0p2us.graphenedb.com:24780/db/data/transaction
- For Identity: I chose Named Principal. This opens up the next 3 fields
- Auth Protocol: Authentication
- username: I used the username created for me by my trial neo4j environment
- password: I used the username created for me by my trial neo4j environment


External Data Source Edit: GrapheneDB

Connect to another Salesforce org or a third-party database or content system.

External Data Source	GrapheneDB
Name	GrapheneDB
Type	Salesforce Connect: Custom—GraphQLDataSourceProvider

Parameters

URL	https://app74051134-28612152340507059:b.PcFrvHlTkkWU.qFvLPbGOAomvb4WM@db-
High Data Volume	☐

Authentication

Identity Type	Named Principal
Authentication Protocol	Password Authentication
Username	app74051134-28612152
Password	••••••••

GRAPHQLDATESOURCEPROVIDER

This 1st short apex class defines some basic parameters about this salesforce connect connection, namely, the services provided, and the kind of authentication, as well as the various capabilities of this connection (read, write, etc...). I modified the template to enable BASIC authentication, more on that below.

```
global with sharing class GraphQLDataSourceProvider extends DataSource.Provider

    global GraphQLDataSourceProvider(){}
```

```
        override global List<DataSource.AuthenticationCapability> getAuthentica
        List<DataSource.AuthenticationCapability> capabilities = new List<DataS
            capabilities.add(DataSource.AuthenticationCapability.OAUTH);
            capabilities.add(DataSource.AuthenticationCapability.ANONYMOUS);
            capabilities.add(DataSource.AuthenticationCapability.BASIC);
        return capabilities;
    }

    override global List<DataSource.Capability> getCapabilities()
    {
    List<DataSource.Capability> capabilities = new
        List<DataSource.Capability>();
    capabilities.add(DataSource.Capability.ROW_QUERY);
    capabilities.add(DataSource.Capability.REQUIRE_ENDPOINT);
    //capabilities.add(DataSource.Capability.SEARCH);
    //capabilities.add(DataSource.Capability.ROW_CREATE);
    //capabilities.add(DataSource.Capability.ROW_UPDATE);
    //capabilities.add(DataSource.Capability.ROW_DELETE);
    return capabilities;
    }

    override global DataSource.Connection getConnection(DataSource.ConnectionPa
            return new GraphQLDataSourceConnection(connectionParams);
        }
}
```

GRAPHQLDATASOURCECONNECTION

This 2nd apex class contains the query function, as well as some custom query builder and parsing functions. After some basic variable definition, and the constructor, the first interesting function is sync():

Sync()
Sync is the function called when the connection is first created, and is used to define the External Objects in the salesforce org. In theory this can be made to be generic, able to discover the data model in the Neo4j database, and offer the user a choice of tables, and fields in those tables, to save a External objects.

To expedite the completion of this POC, I opted not to define a general purpose graph database sync () function. instead I explicitly specified which fields and entities I expect to expose to the salesforce admin. This function creates the external object table in salesforce once the salesforce admin validates and syncs() External Data Source connection.

```
override global List<DataSource.Table> sync(){

        List<DataSource.Table> tables = new List<DataSource.Table>();
        List<DataSource.Column> columns;
        columns = new List<DataSource.Column>();
        columns.add(DataSource.Column.text('Name', 255));
        columns.add(DataSource.Column.text('Born', 255));
        columns.add(DataSource.Column.text('ExternalId', 255));
        columns.add(DataSource.Column.url('DisplayUrl'));
        tables.add(DataSource.Table.get('Person', 'Name',columns));
        return tables;


    }
```

query()
Query gets called whenever 1 or more external records are requested.

The majority of the dev work was in the query() function.Here I built up the graphenedb REST query string (stored in the "payload" variable). Most of my time on this part was consumed learning Cypher, the query language for Neo4j.

```
override global DataSource.TableResult query(DataSource.QueryContext c){
        DataSource.Filter filter = c.tableSelection.filter;

        string entity = c.tableSelection.tableSelected;

        string payload = '{  "statements": [    { "statement": ';

        if(filter == null){
            //if no filter, the just retrieve all records
            if (entity == 'Person')
                {
                    payload = payload + '"MATCH (n:Person) RETURN n",';
                }


        }else if(filter != null){
            //if there are filters, then add them to the query

            String Operator;

            //set the operators that the graph database QL expects
            if (filter.type != null){
                if      (filter.type == DataSource.FilterType.GREATER_THAN)
                else if (filter.type == DataSource.FilterType.LESS_THAN)
                else if (filter.type == DataSource.FilterType.GREATER_THAN_OR_E
                else if (filter.type == DataSource.FilterType.LESS_THAN_OR_EQUA
                else if (filter.type == DataSource.FilterType.EQUALS)
                else if (filter.type == DataSource.FilterType.NOT_EQUALS)
            }

            String cName = filter.columnName;
            //now handle the queries
            //now handle the queries
            if (cName != null && cName.equals('ExternalId')){

                //This section used to query "All" records in the external obje
                payload = payload + '"MATCH (s) WHERE ID(s) = '+filter.columnVa

            }
            else if (cName != null && cName.equals('Born')){

                //This section used to apply filters to the external object's t
                payload = payload + '"MATCH (s) WHERE s.born '+ Operator + ' '

            }else{

                // in the case of no specific PRIMARY filters, and where an ORD
                // then look for secondary filters

                if(c.tableSelection.order != null){

                    //This section used for query of most recent items on the e
                    if (c.tableSelection.filter.subfilters != null && c.tableSe
                    {
                        //this section should be re-engineered to support multi
                        payload = payload + '"MATCH (s) WHERE ID(s) = '+filter.
                    }

                    //TODO: Add sort DESC and ASC logic

                }//end if
            }//end else
        }// end if(filter != null)

        //finish the rest call by adding the typical parameters
        payload = payload + '"parameters": {}, "resultDataContents": [ "row", "
```

Next, in the query function, we need make the callout to graphenedb's rest endpoint. This involves calling a helpful function I customized called getResponse(), detailed below. I copied this function from a previous article on salesforce connect custom adapters, and modified it to add support for basic authentication- this meant changing the HTTP headers, and also writing some code to format the username password into a single base64 encoded string, as basic authentication requires.

Aside: Basic Authentication

Once generated by the code below as it runs, the "Authorization" header will contain the username and password information, but will look like this (this base64 encoded string is just a sample):

Sample base64 encoded username and password: QWxhZGRpbjpPcGVuU2VzYW1l

getResponse()

```
private HttpResponse getResponse(String payload) {

        Http httpProtocol = new Http();
        HttpRequest request = new HttpRequest();


        //****added this section to determine the base64 encoded version of use

        //set auth header, in order to provide graphene with a hex variable tha
        //System.Debug('username:password: ' + username + ':' + password);
        Blob headerValue = Blob.valueOf(username + ':' + password);
        String authorizationHeader = 'Basic ' + EncodingUtil.base64Encode(heade
        //System.Debug('authorizationHeader: ' + authorizationHeader);

        request.setEndPoint(url);
        request.setMethod('POST');
        request.setHeader('Accept', 'application/json, text/plain, */*');
        request.setHeader('X-stream', 'true');
        request.setHeader('Content-Type', 'application/json;charset=utf-8');
        request.setHeader('Authorization', authorizationHeader);  //<-- ****thi
        request.setBody(payload);   //<-- ****passing in my REST call here

        HttpResponse response = new HttpResponse();

        try{
                response = httpProtocol.send(request);
        } catch(exception e)
        {
                System.Debug('bad send/response: ' + e.getMessage());
        }

        return response;

} // end getResponse()
```

PARSING MAGIC

The part of this project I was dreading the most was parsing the JSON results. ugh.
But it was the simplest! ... Once I found JSONtoAPEX! This website created the parsing code for me, based on the JSON I provided! Wow!
In the end, all I had to do was paste in a sample JSON result from my REST query, and a whole class was generated that I was able to use, almost 100% as-is.

I pasted this code into the GraphQLParser class, and starting using it.

I did make 1 modification - There is an internal class that is the same name as the wrapper class, and I slightly modified the name of that class - perhaps it wasn't necessary, but I ended up renaming the inner GraphQLParser class to "InnerGraphQLParser"

Once that was saved, I just called the Parse() function in my query() function, after I get my results back:

```
        // calling external service to get data
        HttpResponse response = getResponse(payload);

        //parse the JSON using an JSONtoApex class: GraphQLParser
        GraphQLParser.InnerGraphQLParser gqlp =  GraphQLParser.parse(response.g

        List<Map<String, Object>> rows = new List<Map<String, Object>>();

        // multiple rows
        if(gqlp.results[0].data.size() > 1){

            List<Object> m = (List<Object>) gqlp.results[0].data;

            //cycle through the results
            for(Integer i=0; i <m.size(); i++ ){

                try{
                    //extract a row of results
                    rows.add(createRow(m[i], display_url)); // <-- custom funct
                }catch (Exception e)
                {
                    System.Debug('Error: '+e);
                }
            }
        }else if(gqlp.results[0].data.size() == 1){
            // single row - in the case where a user clicks on an oData record
            // extract a single row of data
            rows.add(createRow(gqlp.results[0].data[0], display_url));
        }

        //send the data back out to salesforce
        return DataSource.TableResult.get(true, null, c.tableSelection.tableSel
    }
```

CREATEROW() HELPER FUNCTION

You'll notice a function called createRow above, called in 2 places. In this function, given the complexity of the GraphQLParser.InnerGraphQLParser object, I hard-coded the extraction of variables

```
    // Creates rows from JSON data
    public Map<String, Object> createRow(Object item, string DisplayUrl){

            //ceate a temp var to hold the first level of parsed results
            Map<String, Object> results = new Map<String, Object>();

            //cast the data back into a JSONtoAPEX generated class: GraphQLPars
            GraphQLParser.Data d = (GraphQLParser.Data) item;

            results.put('Born',d.graph.nodes[0].properties.born);
            results.put('Name',d.graph.nodes[0].properties.name);
            results.put('ExternalId',d.graph.nodes[0].id);

            return results;

        }
```

Wrap up

Getting this far took me about 20 hrs over a couple months - a better programmer could have done it faster. This was my first foray into Graph database, graphenedb, Cypher, and creating a custom adapter for salesforce connect.

Outstanding tasks that will still need to be developed include adding

- sort - the ability to click on a column in a results set, and re-query with a different sort order
- filtering on more fields, currently only filtering on year.
- adding support for additional tables and the relationships between them.

This would involve modifying query(), sync(), and using the JSONtoAPEX website to figure out how to grow the GraphQLParser class.

Hopefully this helps you create your own Salesforce Connect connection to your graph database.