# Hadoop to Neo4J

Leading up to Graphconnect NY, I was distracting myself from working on my talk by determining if there was any way to import data directly from Hadoop into a graph database, specifically, Neo4j. Previously, I had written some Pig jobs to output the data into various files and then used the Neo4J batchinserter to load the data. This process works great and others have written about it. For example, this approach also uses the batchinserter while this approach uses some Java UDFs to write the Neo4J files directly.

Both of these approaches work great but I was wondering if I could use a Python UDF and create the Neo4J database directly. To test this out, I decided to resurrect some work I had done on the congressional bill data from Govtrack. You can read about the data and the java code I used to convert the files into single-line JSON files here. It's also a good time to read up on how to create an Elasticsearch index using Hadoop. Now that you're back from reading that link, let's look at the approach to try and go from Hadoop directly into Neo4J. From the previous article, you remember that recently Mortar worked with Pig and CPython to have it committed into the Apache Pig trunk. This now allows to take advantage of Hadoop with real Python. Users get to focus just on the logic you need, and streaming Python takes care of all the plumbing.

Nigel Small had written Py2Neo which is a simple and pragmatic Python library that provides access to the popular graph database Neo4j via its RESTful web service interface. That sounded awesome and something worth trying out. Py2Neo is easy to install using pip or easy_install. Installation instructions are located here.

The model that I was trying to create looks something like this:

The approach taken was to use Pig with a streaming Python UDF to write to the Neo4J database using its RESTful web service. I tested this out with Neo4J 2.0M6. I attempted to use Neo4J2.0RC1 but ran into several errors relating to missing nodes. The example code is below:

Since Neo4J uses an incrementing counter for each node, we have to create an id for each keyValue (node name) that we are creating. The keyValues are the congressional session, name of the congresswoman or congressman, billID or subject. Below is a simple Python code that creates that ID.

Once we have the id, we can use Py2Neo to create the nodes and relationships.

Of note is the ability to create the node and add the label in the createNode function. To create the relationship, we pass in the two node ids and the relationship type. This is passed via the REST API interface and the relationship is created.

**Performance** - Performance wasn't what I thought it would be. Py2Neo interacts with Neo4j via its REST API interface and so every interaction requires a separate HTTP request to be sent. This approach, along with logging, made this much slower than I anticipated. Overall, it took about 40 minutes on my MacBook Pro with 16GB ram and SSD to create the Neo4J database.

**Py2Neo Batches** - Batches allow multiple requests to be grouped and sent together, cutting down on network traffic and latency. Such requests also have the advantage of being executed within a single transaction. The second run was done by adding some Py2Neo batches. This really didn't make a huge difference as the log files were still being written.

Overall, it still took about 60 minutes on my MacBook Pro with 16GB ram and SSD to create the Neo4J database.

**Next Steps**
Hmmm....I should have known that the RESTful service performance wasn't going to be anywhere near as fast as the batchinserter performance due to logging. You could see the log files grow and grow as the data was added. I'm going to go back to the drawing board and see if a Java UDF could work better. The worst case is I just go back to writing out files and writing a custom batchinserter each time.