Introduction

In this report, I am going to present a (small) comparison of Amazon DynamoDB and Neo4J, two NoSQL database systems with different paradigm, and give some feedback on my personal experience with these two tools.

1 Quick presentation

Neo4J is a graph database system, i.e. representing data as a graph where nodes stores entries and edges relations between the entries (edges can also have a type and attributes). It is well-suited for storing data with strong structure and connexions between entries of different nature. Neo4J is implemented in Java, which makes it easily cross-platform.

DynamoDB is a document-oriented database system, a refinement of the key-value paradigm. Its main purpose is to provide a fast, efficient and scalable online service via the Amazon Web Services (AWS).

2 Setup and work environment

Neo4J is very easy to launch and setup: one just has to launch the server with the main program and open a browser at adress http://localhost:7474. It provides a web interface that allows to manage databases and execute queries using their own language: Cypher. In this interface, one can also visualize the data in an intuitive way, playing with the graph (or subgraph if result of a query), zooming etc. Tutorials and examples are directly available in this interface by default, which allows to quickly start learning about Neo4J and Cypher. Other tutorials can be found on the website, and one can even skip the setup part by directly launching an interface on the Neo4J website with a bunch of examples.

On the other hand, DynamoDB setup is less intuitive and more business-oriented. After launching the local server, one has to use one of the bindings (Java, C#, PHP, Python, Ruby...) to create databases and execute queries. I chose to use the Python binding, Boto3. When trying to access the DynamoDB
service in Python, I faced a surprising error message: I had not specify my credentials for connecting to the server, and I had not specify the server region. It turns out that the local version of DYNAMODB is not different from the online version, so that one has to specify everything as if connecting to the AWS, which I find rather counter-intuitive. I finally succeeded to connect to the DYNAMODB service with the following code:

```python
boto3.resource('dynamodb', aws_access_key_id = "fake",
              aws_secret_access_key = "credentials",
              region_name = "us-west-2",
              endpoint_url = "http://localhost:8000")
```

Screenshots of the two work environments are given in Figure 1.

### 3 Data types and representation

DYNAMODB database system relies on the key-value paradigm. When defining a table, one defines the key schema. Then one can add items in the table, having the defined attributes and other piece of information which are dependent on the item itself. For instance, a table products having primary key field “ProductID” may contain books with “Pages” attribute and televisions with “Weight” attribute (of course televisions does not specify the number of pages...) – see example codein Figure 1. This is clearly one of the advantages compared to SQL. One can also import data from a Json file.

When defining a database in Neo4j, one specifies the nodes which have types and attributes. Note that these types are not required to be explicitly specified beforehand: the first node defined with a certain type defines the type. By default, no primary key or index needs to be specified: one can directly begin entering data. Then one specifies the edges which also have types and attributes. All these definitions are quite intuitive in the Cypher language, for example given two nodes “Alice” and “Bob” one can add the social information with the command

```cypher
CREATE (Alice)-[:FRIENDS {meet:21122012}]->(Bob)
```

where FRIENDS is the type of the relation with an attribute meet.

### 4 Queries

As we have seen, Cypher is the language for Neo4j. The keyword are inspired by SQL but the queries format is adapted to the graph database paradigm. These queries are quite intuitive and easy to write. For instance in order to get the names of Alice’s friends and their meeting dates, a valid command is:

```cypher
MATCH (alice:Person {name:"Alice"})
-[:FRIENDS]->(friend)
RETURN friend.name, r.meet
```
As you can see, the MATCH command is similar to the CREATE command, where one can specify constraints on the attributes and types if nodes and edges. Moreover, it is possible to assign a variable name to any node and edge and then access the corresponding attributes. I find this way of writing queries quite simple and intuitive, one can even specify a number of relations between nodes (or any number).

The queries in DynamoDB for searching and updating items are intuitive in the key-value paradigm: one can search an item by its key, and of course add conditions to the query. The following code example prints all books and their prices in table products:

```python
response = table.scan(
    FilterExpression=Attr('Type').eq('book'))
for item in response['Items']:
    print(item['title'], ',', item['price'])
```

**Conclusion**

As a user discovering these two tools with no previous practice of NoSQL database systems, I feel like Neo4j is better than DynamoDB in all the aspects that I have treated: accessibility, local setup, work environment and data visualization. By providing their own language - Cypher - that allows for concise and intuitive description of data and queries, and with their web interface including interactive tutorials, examples and graph visualization, Neo4j is clearly easier to learn and more user-friendly. However, I did not treat here the questions of performances, scalability and integration. DynamoDB, through the AWS, seems designed more for developers and companies and therefore should have advantages in these aspects that I did not see.
(a) DynamoDB through Boto3 (in Emacs)

(b) Neo4j (browser interface)

Figure 1: Work environments