**1. Overview**

The goal of this project was to switch from our current MySQL database to a Neo4j NoSQL database while evaluating the querying capabilities and performance of the relational and graph models. In addition to that, we had to ensure a smooth data transfer that was consistent and didn’t have loss of information.

**2. System Description**

Neo4j and MySQL were locally installed for the migration. A Python script was used for data extraction and manipulation, which facilitated the export of data from MySQL into CSV files, who were then used for importation into Neo4j.

- Relational Database: MySQL, localhost

- NoSQL Database: Neo4j, Desktop version

- Programming Languages and Tools: Python3, CSV module for data extraction

**3. Data Model Description**

NoSQL Model Overview:

In Neo4j, the data was modeled as a graph consisting of nodes, properties, and relationships. Our model contains the track, creditor, content\_rating, trending\_date, insertion\_log nodes.

Nodes were composed of properties:

* **Node track**: TrackID, TrackName, TrackUri, Labels, Duration, PlayCount, Image
* **Node creditor**: CreditorID, Name, Role
* **Node content\_rating**: ContentRatingID, ContentRatingName
* **Node trending\_date**: TrendingID, PeakRank, AppearancesOnChart, ConsecutiveAppearancesOnChart, TrendingDate
* **Node insertion\_log**: LogID, TrackID, TrackName, InsertionTimeStamp.

And the relationships linking the nodes:

* **Relationship HAS\_RATING**: Track -[:HAS\_RATING]-> Content\_Rating
* **Relationship CONTRIBUTED\_TO**: Creditor -[:CONTRIBUTED\_TO]-> Track
* **Relationship TRENDING**: Track -[:TRENDING]-> Trending\_Date

**4. Data Transfer Methodology**

Data Extraction:

Data was extracted using a Python script that queried the MySQL database and wrote the outputs to CSV files. This included data for Tracks, Creditors, and various linking tables that represent relationships in the MySQL schema.

Data Import into Neo4j:

CSV files were imported into Neo4j using Cypher's `LOAD CSV` commands. The import scripts included commands to create nodes and relationships, applying some transformations such as trimming spaces and converting empty strings to null values.

**5. Query Performance and Optimization**

To evaluate the performance, several queries were benchmarked before and after applying indexes.

Here are some examples that demonstrated the change of performance when using indexes:

- Basic Search by Attribute: Searching tracks by name. Improved from 27ms to 7ms with indexing.

- Aggregate Data: Counting tracks with a PlayCount over 1000. Performance improved from 16ms to 10ms.

- Top N Entities: Listing top 5 tracks by PlayCount. Indexing initially increased the response time (18ms to 26ms).

- Group By: Averaging PlayCount by Label, which showed a consistent performance (23ms to 26ms).

Indexes were created for TrackName, PlayCount, and Labels, which enhanced the query responsiveness for some operations.

**6. Challenges encountered**

-Consistency: We had some problems ensuring consistency during the migration, as some columns that represented attributes were not filled at all in the CSV files, which initially hindered the integrity of our Neo4j database and our ability to perform queries. We later overcame this issue by improving the python script.

- Performance: We noticed indexing increased some query times, which we determined was likely due to the overhead of maintaining an index.