F21BD - Big Data Management

"Coursework One: NoSQL Data Storage"

By David Hobbs, Michael King and Jordan Walker

Undergraduate Group 5

Brief

For this assignment, we have been tasked, as a group, to create a NoSQL system as a solution to the management of a music database that holds various information on items such as artists, songs, staff and sales. The current solution which is developed in a relational database has been reported with the issue that the users are complaining of “slower response time”.

Given this issue, we have been given a list of NoSQL systems including MongoDB, Neo4J and Cassandra, where we have picked two of these systems to contrast and compare their use to the company. This has been achieved through documentation and development with specific regards to the suitability of the task given.

Contents

1 - NoSQL Database Engines Overview

Cassandra Database Overview

Cassandra is a general purpose non-relational database. Cassandra differs from relational databases in that it offers a schema-free data model, high-velocity input data and row partitioning[1].

Cassandra makes use of the Cypher Query Language (CQL) - a query language in a similar type to SQL that enables a smooth transition from relational databases. It uses wide row design which means Cassandra only needs to read the columns required to complete queries[2]. The database model makes use of a ring architecture of peer nodes with all nodes playing an identical role and each communicating to one another through a gossip protocol. Gossip protocol involves nodes communicating randomly about themselves and the other nodes that they have information about, meaning all nodes can quickly learn about all other nodes[3].

Cassandra databases can be scaled to work with extensive data sets, meaning that their local server can support millions of instances. Expanding the capacity of a server is relatively simple, consisting of adding another node to an existing cluster without having to take any part of the network down first[2]. Clustering has the benefit of no single point of failure and can offer uninterrupted availability and uptime, while Cassandra can also apply data compression of up to 80% without any performance overheads[2].

Use Cases

- Retail application support, shopping cart protection, fast catalogue access[2].
- Applications that send large data bursts such as connected internet of things, activity tracking and extracting sensor data [2]
- Messaging, analytics, recommendation engines[2].

Storage

Data is written in four stages:

- Logging data in commit log[4].
- Writing data to memtable in main memory[4].
- Flushing data from the memtable[4].
- Writing data to disk in the form of immutable SS tables[4].

Data is written to a memtable in memory and a commit log on disk to provide a backup[4]. When the memtable buffer becomes full, the data is sorted and appended to an immutable SStable on disk. Data in SSTables are labelled with ‘tombstones’ when they are no longer needed as data is not deleted from an SSTable. More than one SSTable can be in use for a logical data table, and so over time, the distribution of data over multiple tables may slow down data reads. To solve this, SSTables can be compacted into one table to speed up the process, and any data marked with tombstones can be deleted as the tables are merged[3].

Consistency/Replication Model

Cassandra aims for availability and partition tolerance but is tuneable to perform towards consistent and partition tolerant. Replicas of data are stored on multiple nodes, meaning that information that is stored on a dead node can be found elsewhere in the topology[3].

Meeting the Needs of Chinook

Cassandra offers many perks that could be useful in the transition from the RDBMS model currently used by Chinook into a NoSQL solution. The main attraction could easily be that Cassandra works similarly to an RDBMS using rows to capture the data that is contained within the database. What could also be used as an advantage for using Cassandra is where it places itself in regards to the CAP theorem offering itself between high availability and partition tolerance (A, P). This allows the user (worker/client) to receive a "result" regardless of whether that is the right "result" due to its high availability but also allows for the user to access information even if some part of the database is not currently operating due to the duplication qualities of partition tolerance. Currently, the popular streaming service similar to what is being offered by Chinook Spotify uses Cassandra as their primary form of data storage which is something to be considered when continuing with this project.

Mongo Database Overview

MongoDB is the most popular open source NoSQL database available, with DB-Engines ranking it as 5th overall for database engines in February 2019[5]. MongoDB is a cross-platform, document-oriented database that boasts high performance, high availability, and easy scalability. MongoDB works on the concept of collection and documentation[6]. A document storage method consists of a set of key-value pairs and MongoDB allows for loosely typed values, where documents do not need to consist of the same structure as other values. This means documents have a dynamic schema and allows databases to hold heterogeneous data, making it ideal to use when you have evolving requirements for what you expect to use your database for[7].

Use Cases

Suitable for:

- Online shopping.
- Blogs and content management.
- Real-time analytics.
- Configuration management.
- Mobile and social networking sites.
- Evolving data requirements.

Unsuitable For:

- Highly transactional systems or where the data model is designed up front.
- Tightly coupled systems

Storage

MongoDB is highly-scalable with data able to be stored in shards, allowing for smaller and easier to manage collections of data[8]. Data representation in JSON or BSON also means that, although there is no strict schema, one can still gain an overview of the tables from observing the JSON schema. A downside of NoSQL is that most solutions are not as strongly ACID-compliant (Atomic, Consistency, Isolation, Durability) as the more well-established RDBMS systems. This results in an extremely complicated transaction in comparison to MySQL[7].

As opposed to storing data in rows and columns as one would with a relational database, MongoDB stores a binary form of JSON documents called BSON. BSON is stored closer in memory than standard JSON and also includes Data and binary data types[9]. MongoDB is built for scalability, performance and high availability. Sharding allows MongoDB to scale from single server deployments to larger and more complex multi-data architectures[8].

Consistency/Replication Model

MongoDB boasts built-in replication alongside automated failover, which enables high reliability and operational flexibility[6]. A replica set is a group of MongoDB instances that host the same data set. In a replica, one node is the primary node that receives all write operations[10]. These corresponding operations are copied into additional secondary nodes - usually two - which then take on an identical set of values as the primary node[10]. Replica sets can have only one primary node, meaning that secondary nodes can only replicate a single data set at a time.

As a result of this, MongoDB is consistent as writes are only made to a single primary set. Applications can read from secondary replicas in scenarios where it is acceptable for data to be slightly out of date, where the data will become eventually consistent when they are updated with the new write operations[6].

Meeting the Needs of Chinook

The Mongo database excels at scalability and flexibility, meaning that it can adapt quickly to the future needs of the company. Difficulties could arise when inserting new data into a mongo instance. Due to the schemaless nature of a document store, there could be a lot of inconsistent data entered over time leading to confusing results. The MongoDB query language has a steep learning curve which could result in trends between data going unnoticed. MongoDB focuses on consistency and partition tolerance, and this means that data in MongoDB will not be available instantly to all users as consistent data is given priority. For a music purchase company , consistency of data may not be the highest priority as data will change infrequently and demand for availability of data may be high.

Neo4J Database Overview

Neo4J is a graphical database system which features ACID compliance similar to SQL-style relational database management systems. Neo4J stores information as graphs. A graph is defined as any diagrammatic representation that consists of vertices (shown by circles) and edges (shown with intersection lines). Within these graphical representations, there are several types of graphs:

Undirected graphs - nodes and relationships are interchangeable, their relationship can be interpreted in any way.
Directed graphs - nodes and relationships are not bidirectional by default.
Graphs with weight - where graphic relationships between nodes have some kind of numerical assessment, normally to indicate the relative 'strength' of the relationship in comparison to other nodes. This allows operations to be subsequently performed.
Graphs with labels - these graphs have labels incorporated that can define the relationships between the numerous vertices.
Property graphs - this is a weighted graph with labels where properties can be assigned to both nodes and relationships[10].
The flexibility of a graph database allows the data captured to be easily changed and extended for additional attributes and objects. Relationship-based searches can be executed which can allow easy extraction of the properties linking different values together[11]. Graph databases are naturally indexed by relationships (the strength of the underlying model), providing faster access compared to relational data for information.

Storage

Neo4J consists of native graph processing and storage, meaning that the data storage is completely optimised for graphs and ensures that data is stored efficiently by writing nodes and relationships close to each other in memory. Neo4J databases are stored across numerous different files, with each record consisting of separate types of information, such as nodes, relationships or properties[12]. Native graph processing runs with index-free adjacency, meaning that there is no indexing for storage. A native graph process with index-free adjacency speeds up processing by ensuring that each node is stored directly to its adjacent nodes and relationships, allowing for a much faster retrieval period when the various tasks are executed. Neo4J supports using both no fixed schema at all and a strict schema that is defined by the user.

Consistency/Replication Model

As a NoSQL database management format, Neo4J has some operational advantages over a more common relational database management system. However, it still follows the ACID transaction format that a traditional SQL system would support. This ensures strong consistency with any data model in Neo4J.

Consistency and replication are achieved through Causal Clustering[14]. Causal Clustering follows the format master-slave approach of having one main server containing the original write operations and secondary servers that replicate these, similarly to MongoDB. A Causal Cluster uses the Raft protocol to provide consensus on a number of cluster operations, which include writes, cluster membership and leader information. An arbitrary number of read replica servers to allow the scaling out of graph queries[13].

Meeting the Needs of Chinook
Neo4J's advantage over many competitors is the ability to represent relationships in the form of a graph, making trends between data sets easily discoverable. Changing from a relational database to Neo4J is made easier by modelling in a similar fashion to MySQL using nodes instead of tables and relationships instead of join tables. The graphical user interface allows management to quickly visualise how data is related and can be used to inform strategic decisions within the organisation. Neo4j focuses on availability and consistency, and this means that the database will be highly available to customers wishing to download many tracks but will struggle if a database node fails as it is not partition-tolerant. Retrieval of data is made faster by relationships being stored on disk and not computed each time a query is made.

Chosen NoSQL solutions

As a group, it has been decided that we will carry forward the project using MongoDB and Neo4j as our solutions to the problems faced by Chinook. This has been decided due to our belief that the solution should have a focus on consistency meaning that every write is shown to the user either client or colleague. With this focus in mind, we can ensure that every piece of information used by either party is correct or an error is given. This is due to our belief that when there is money involved and transactions are an integral part of the companies model that the data storage has to be consistent and thus our reasoning for MongoDB and Neo4j.

Following the given reasoning of our NoSQL solution, we can now explore the benefits of having an available, consistent model (Neo4j) or a more partition, consistent model (Mongo). However, it is important to note that no system can be entirely Consistent, Available and Partition Tolerant tweaks can be made to move our chosen solutions towards a more overall encompassing solution however we are aware that by doing so some trade-offs have to be made.

2 - Database Schemas

MongoDB Schema

We designed the JSON schema to show the layout of the various documents for each collection. As MongoDB has no set schema for documents and no strict reference for relationships between them, we have treated all data as independent to ensure that the best data model was used to collate the information. To avoid extensive querying over several documents, we have decided to make use of an embedded data model when using MongoDB. This is achieved through loading instances of information from multiple separate MySQL tables into each different object in the collection, to ensure that all the relevant information can be stored and queried effectively. Where possible, we have aimed to avoid storing our embedded data as mutable arrays, meaning that our one-to-many relationships are stored as references to the single entity. This saves memory and can allow for more efficient querying.

Examples of the embedded data model can be shown through what information is stored in the Track document. In the relational model, information such as the artist and genre that are related to each track are stored as foreign keys and referenced appropriately. Since that is not the case when using MongoDB, all the information regarding those parameters is stored alongside the other properties that the track contains. Another design change that we have implemented during the transformation is that the invoice, customer, employee and 'invoiceLine' tables are combined into a single invoice document. This includes information about each track from the 'invoiceLine' table stored as a combination of properties within the invoice document.

/* JSON SCHEMA FOR MONGODB TRANSFORMATION */

```
/*playlist collection */
{
"$schema":"http://jsonschema.org/draft-07/schema#",
"bsonType": "object",
"PlaylistName":{
"bsonType":"string"
},
"tracks":{
"bsonType":"string"
},
},
"required":["PlaylistName",
"tracks"
]

}

/*tracks collection */
{"$schema":"http://jsonschema.org/draft-07/schema#",
"bsonType": "object",
"TrackId":{
"bsonType":"int"
},
"Track.name":{
"bsonType":"string"
},
"Album":{
"bsonType":"string"
},
"Artist":{
"bsonType":"string"
},
"Genre":{
"bsonType":"string"
},
"Track.milliseconds":{
"bsonType":"int"
},
"track.Composer":{
"bsonType":"int"
},
"Track.mediatype":{
"bsonType":"string"
},
"Track.unitPrice":{
"bsonType":"string"
},
"Track.Bytes":{
"bsonType":"number"
},
}

/*Invoice collection */
{"$schema":"http://jsonschema.org/draft-07/schema#",
"bsonType": "object",
"Invoiceline":{
"bsonType":"string"
},
"InvoiceId":{
"bsonType":"int"
```

```
},
"Invoice.date":{
"bsonType":"string"
},
"Invoice.BillingAddress":{
"bsonType":"string"
},
"Invoice.BillingCity":{
"bsonType":"string"
},
"Invoice.BillingState":{
"bsonType":"string"
},
"Invoice.BillingCountry":{
"bsonType":"string"
},
"Invoice.BillingPostalCode":{
"bsonType":"string"
},
"Invoice.BillingPostalCode":{
"bsonType":"string"
},
"InvoiceTotal":{
"bsonType":"decimal"
},
"Invoice.trackID":{
"bsonType":"int"
},
"Invoice.trackName":{
"bsonType":"string"
},
"Invoice.albumName":{
"bsonType":"string"
},
"Invoice.artistName":{
"bsonType":"string"
},
"Invoice.genre":{
"bsonType":"string"
},
"Invoice.composer":{
"bsonType":"string"
},
"Invoice.Milliseconds":{
"bsonType":"int"
},
"Invoice.Bytes":{
"bsonType":"int"
},
"Invoice.Quantity":{
"bsonType":"int"
},
"Invoice.Price":{
"bsonType":"decimal"
},
"Invoice.CustomerFirstName":{
"bsonType":"string"
},
"Invoice.CustomerLastName":{
"bsonType":"string"
},
"Invoice.company":{
```

```
"bsonType":"string"
},
"Invoice.phone":{
"bsonType":"int"
},
"Invoice.fax":{
"bsonType":"int"
},
"Invoice.email":{
"bsonType":"string"
},
"Invoice.employeeLastName":{
"bsonType":"string"
},
"Invoice.employeeFirstName":{
"bsonType":"string"
},
"Invoice.employeeTitle":{
"bsonType":"string"
},
"Invoice.employeeBirthDate":{
"bsonType":"string"
},
"Invoice.employeeHireDate":{
"bsonType":"string"
},
"Invoice.employeeAddress":{
"bsonType":"string"
},
"Invoice.employeeCity":{
"bsonType":"string"
},
"Invoice.employeeState":{
"bsonType":"string"
},
"Invoice.employeeCountry":{
"bsonType":"string"
},
"Invoice.employeePostalCode":{
"bsonType":"string"
},
"Invoice.employeePhone":{
"bsonType":"int"
},
"Invoice.employeeFax":{
"bsonType":"int"
},
"Invoice.employeeeEmail":{
"bsonType":"string"
},
```

We believe that taking the approach of using an embedded data model will help with writing the most efficient queries over the dataset and also format documents in a way that can give the user a clear overview of all the information for each collection.

Neo4J Relationship Overview

Figure 1, shows the node properties and types from each relation from the MySQL database. Joining tables have been dropped as they are represented by graph vertices stored alongside the data on disk. Each row in MySQL is represented as a node, and individual relationships (which can store their properties) can be constructed between them. In this case, each relationship line made between invoice and track nodes will store details of the unit price, and quantity of the track ordered.

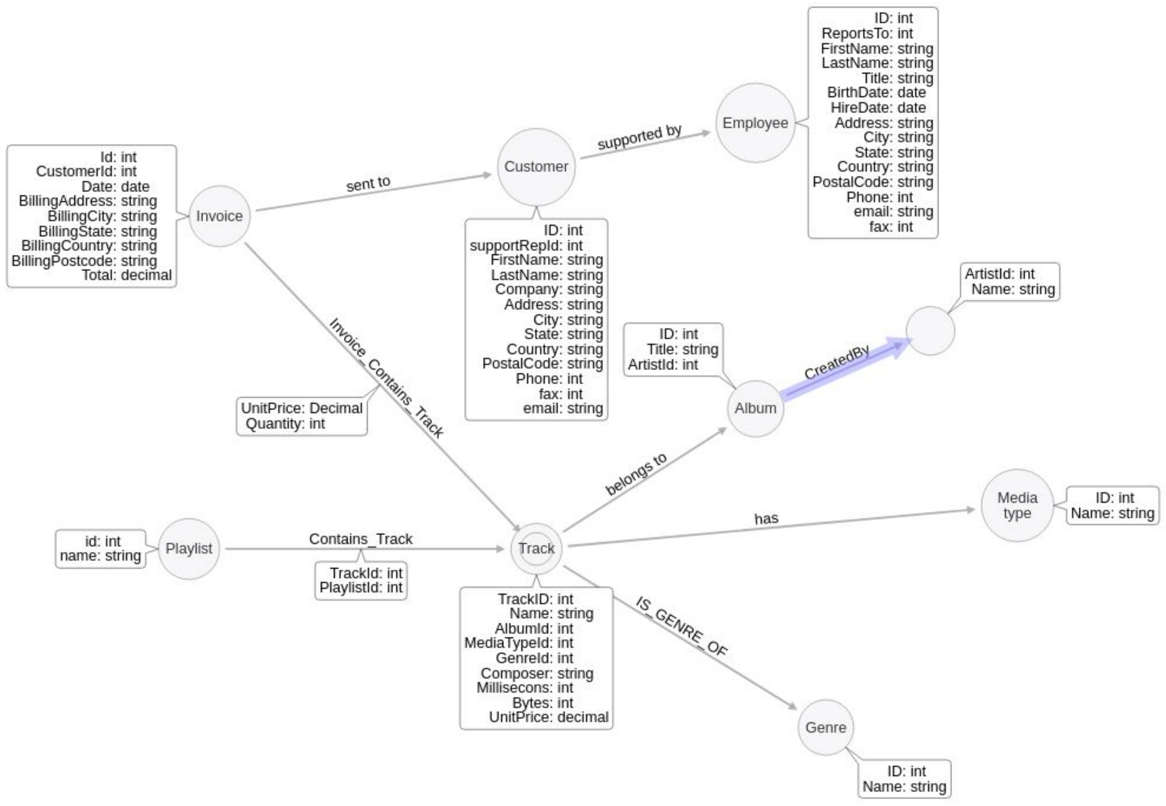


Figure 1 - Neo4J relationship overview.

Neo4J Schema

The data formatting for Neo4J is much more akin to the format of MySQL than the schema created for the documents in MongoDB. All the tables have been transformed into Neo4J in a similar structure to the Chinook database. We are aware Neo4j does not use a schema. However, the described schema is a representation of how Neo4j would be represented in a JSON format.

```
/* JSON SCHEMA FOR NEO4J TRANSFORMATION */

/*Playlist collection  */
{
        "$id":"#Playlist",
        "$schema":"http://jsonschema.org/draft-07/schema#",
        "Type": "object",
        "PlaylistId":{
        "Type":"int"
        },
        "name":{
        "Type":"string"
        },
        "required":["PlaylistId",
        "name"
        ]
        "properties" :{
         "contains_track":{  "$ref": "#TrackId"
                                   }
  }
}

/*Tracks collection  */
{"$id":"#Track",
"$schema":"http://jsonschema.org/draft-07/schema#",
"Type": "object",
        "TrackId":{
        "Type":"int"
        },
        "name":{
        "Type":"string"
        },
        "composer":{
        "Type":"string"
        },
        "milliseconds":{
        "Type":"int"
        },
        "bytes":{
         "Type":"int"
   },
   "genreID":{
         "Type":"int"
   },
   "mediaTypeID":{
         "Type":"int"
   },
   "albumID":{
         "Type":"int"
   },
        "unitPrice":{
            "Type":"decimal"
            },
            "required":["TrackId",
            "name",
            "bytes",
            "milliseconds",
            "genreID",
            "albumID",
            "mediaTypeID",
            "unitPrice"
            ]
            "properties" :{
             "has":{ "$ref": "#MediaType"},
             "belongs_to_Genre":{ "$ref": "#Genre"},
             "belongs_to":{ "$ref": "#Album"},
   }
}

/*Invoice collection  */
{"$id":"#Invoice",
"$schema":"http://jsonschema.org/draft-07/schema#",
"Type": "object",
            "InvoiceId":{
            "Type":"int"
            },
            "invoiceDate":{
            "Type":"date"
            },
            "billingAddress":{
            "Type":"string"
            },
            "billingCity":{
            "Type":"string"
            },
            "billingState":{
            "Type":"string"
            },
            "billingPostalCode":{
            "Type":"string"
            },
            "billingCountry":{
            "Type":"string"
            },
            "total":{
            "Type":"decimal"
            },
            "required":["InvoiceId",
            "invoiceDate",
            "billingAddress",
            "billingPostalcode",
            "billingState",
            "billingCountry",
            "total"
            ]
   "properties" :{
             "Invoice_contains_track":{ "UnitPrice": Float,
                                                  "Quantity": INT
                                 },
             "sent_to":{ "$ref": "#Customer"}
   }

}
```

```
/*MediaType collection */
{"$id":"#MediaType",
"$schema":"http://jsonschema.org/draft-07/schema#",
"Type": "object",
            "mediatype":{
            "Type":"string",
            "enum":"['MPEG audio file','Protected AAC audio file','Protected MPEG-4 video file','Purchased AAC audio file','AAC
audio file']"
            }
            "required":["mediatype"]
}

/*Genre Collection */
{"$id": "Genre",
"$schema":"http://jsonschema.org/draft-07/schema#",
"Type": "object",
            "GenreId":{
            "Type":"int"
            },
            "name":{
            "Type":"string"
            },
            "required":["GenreId",
            "name"
            ]
}

/*Album Collection */
{"$id":"#Album",
"$schema":"http://jsonschema.org/draft-07/schema#",
"Type": "object",
            "AlbumId":{
            "Type":"int"
            },
            "title":{
            "Type":"string"
            },
            "artistID":{
            "Type":"int"
            },
            "required":["AlbumId",
            "title",
            "artistID"
            ]
}

/*Artist Collection */
{"$id":"#Artist",
"$schema":"http://jsonschema.org/draft-07/schema#",
"Type": "object",
            "ArtistId":{
            "Type":"int"
            },
            "name":{
            "Type":"string"
            },
            "required":["ArtistID",
            "name"
            ]
}
```

```
/*Customer collection  */
{"$id":"#Customer",
"$schema":"http://jsonschema.org/draft-07/schema#",
"Type": "object",
            "CustomerId":{
            "Type":"string"
            },
            "firstName":{
            "Type":"string"
            },
            "lastName":{
            "Type":"string"
            },
            "company":{
            "Type":"string"
            },
            "address":{
            "Type":"string"
            },
            "city":{
            "Type":"string"
            },
            "state":{
            "Type":"string"

            },
            "country":{
            "Type":"string"
            },
            "PostalCode":{
            "Type":"string"
            },
            "phone":{
            "Type":"string"
            },
            "fax":{
            "Type":"int"
            },
            "email":{
            "Type":"string"
            },
            "supportRepID":{
            "Type":"int"
            },
            "required":["CustomerId",
            "firstName",
            "LastName",
            "address",
            "postCode",
            "email",
            "supportRepID"
            ]
}

/*Employee Collection  */
{"$id":"#Employee",
"$schema":"http://jsonschema.org/draft-07/schema#",
"Type": "object",
            "EmployeeId":{
            "Type":"int"
            },
            "lastName":{
```

```
            "Type":"string"
            },
            "firstName":{
            "Type":"string"
            },
            "title":{
            "Type":"string"
            },
            "birthDate":{
            "Type":"date"
            },
            "hireDate":{
            "Type":"date"
            },
            "address":{
            "Type":"string"
            },
            "city":{
            "Type":"string"
            },
            "state":{
            "Type":"string"
            },
            "country":{
            "Type":"string"
            },
            "postalCode":{
            "Type":"string"
            },
            "phone":{
            "Type":"string"
            },
            "fax":{
            "Type":"int"
            },
            "email":{
            "Type":"string"
            },
            "reportsTo":{
            "Type":"int"
            },
            "required":["EmployeeId",
            "firstName",
            "lastName",
            "birthDate",
            "hireDate",
            "address",
            "postCode"
            ]
  "properties" :{
             "supports":{ "$ref": "#Customer"},
             "reports_to":{ "$ref": "#Employee"}
  }
}
```

3 - ETL Pipelines

MongoDB

A full list of MySQL queries used to extract data can be found in appendix.

Extraction

As data will be stored in documents rather than relational tables as found in MySQL, related data can be stored in a single document. For example, the tracks table in MySQL will be normalised and have foreign keys that link to other attributes such as the album, mediatype, genre or artist related to it. With a document store, these attributes can be gathered into the same collection due to the low cost of disk space. As a result, there is less of a problem with repeating data compared to the relational database model.

Data is extracted from the MySQL database using queries that collate the data in the groups that will be represented in MongoDB. This involves designing queries that link together many tables using *JOIN WHERE* clauses to group together related items from different tables. An example of this could be an invoice that groups together many tracks, customer, and employee data. This means that the whole database can be represented using three MongoDB collections (Invoice, Playlist, Track) instead of eleven tables with MySQL.

An example MySQL command used to extract the required information:

```
mysql -u root -p < ~/Documents/mongoInvoiceQuery.sql > ~/Desktop/mongoInvoiceQuery.tsv
```

This command connects to MySQL using the root username and a password. The input file is a prewritten SQL query to extract data and send the output to a file on the desktop. File output by MySQL is in tab separated form. This format is accepted by MongoDB and does not require alteration.

Transformation

A full list of SQL commands used to create this database can be found in Appendix A.

Most of the transformation stage occurs in designing the SQL queries to join together data from many tables in the relational database to present data in the form that it will be stored in MongoDB. The key to a document store database is to denormalise as much of the data as possible, as is it more computationally expensive to perform lookups across different collections. The main collection ‘invoice’ gathers data from the bulk of the database and places them in one collection, with subdocuments used to categorise former relational tables.

Load

Loading the files into MongoDB is done by using a bash script consisting of terminal commands similar to the following:

```
mongoimport -u dah3 --authenticationDatabase admin --collection invoice --type tsv --file ~/Documents/invoice.txt
--headerline
```

This command inserts a *tsv* file into user *dah3* using a newly created invoice collection. --headerline option is used to ensure the headers in the first line of the tsv file as table headers for the collection.

A previous Iteration of our MongoDB design used five collections instead of three. This meant that we executed the ETL pipeline to transform the MySQL data into two formats; the first that has all the documents saved together and data stored under one collection and the second that utilises the normalised approach which is more akin to a traditional RDBMS. The decision was made to use the information in several documents due to the $lookup function which is supported from MongoDB version 3.6 onwards. This function implements a method of matching data from different collections in a similar way to how JOINS work in MySQL. We discovered that our querying required the use of lookup functions several times, meaning a different structure would be needed to execute more efficient queries.

Examples of MongoDB loaded data

Collections loaded into database

```
> show collections
invoice
playlist
track
```

Track from tracks collection using db.track.find().pretty()

```
{
	"_id" : ObjectId("5c642c6fe9232ffaa1bbe65b"),
	"trackID" : 24,
	"Track" : {
		"name" : "Love In An Elevator",
		"milliseconds" : 321828,
		"unitprice" : 0.99,
		"Bytes" : 10552051
	},
	"Album" : "Big Ones",
	"Artist" : "Aerosmith",
	"Genre" : "Rock",
	"track" : {
		"Composer" : "Steven Tyler, Joe Perry"
	},
	"MediaType" : {
		"name" : "MPEG audio file"
	}
}
```

After the uploads have been made, there are still a few conversion issues that need to be addressed within the new collections. NULL values have been carried over from the extraction process and instances of arrays of values, such as the collection of tracks made for each purchase, are loaded by MongoImport as String values. To sort this, several small MongoShell functions have been created to format the data in a more appropriate format. Example snippets for the NULL function and the JSON Parse implementation are shown below.

```
/*Removing NULL values from new tables - example: invoice*/
var coll = db.getCollection("invoice");
var cursor = coll.find();
while (cursor.hasNext()) {
   var doc = cursor.next();
   var keys = {};
   var hasNull = false;
   for ( var x in doc) {
      if (x != "_id" && doc[x] == "NULL") {
         keys[x] = 1;
         hasNull = true;
      }
   }
   if (hasNull) {
      coll.update({_id: doc._id}, {$unset:keys});
   }
}
```

Neo4J

Extraction

As the graph model of Neo4J is similar to the relational model of MySQL, all of the tables from MySQL can be exported using simple *SELECT * FROM* statements. The results of which can be exported into a tab separated file. This can be accomplished using a command like this:

```
mysql -u root -p < ~/Desktop/playlist.sql > ~/Desktop/playlist.tsv
```

Transformation

Files produced by MySQL are tab delimited files and Neo4J requires comma separated files. As such, the files will require alteration before being accepted by Neo4J.
The linux program *sed* can use regex to locate the offending characters and replace them.
Multiple sed commands can be strung together using the *-e* option.
E.g.

```
sed -e 's/,//g' -e 's/\t/,/g' -e 's/"//g'
```

's/,//g' This command removes any existing commas in the file as leaving the commas in will end up creating too many fields when tabs are replaced with commas.
's/\t/,/g' This command replaces tabs with commas to generate the comma separated fields.
s/"//g Neo4J does not like any quotation characters, this command removes all quotation characters from the file.

The whole process can be performed with one command using pipes and redirection.
This command sends an SQL file to MySQL server, the result is sent to sed via a pipe, then the result of that is sent to Neo4J's import folder in the form of *invoice.csv.*

```
mysql -u root -p < ~/Desktop/invoice.sql | sed -e 's/,//g' -e 's/\t/,/g' -e 's/"//g' >
~/Desktop/neo4j-community-3.4.4/import/invoice.csv
```

Load

Neo4J requires that any csv files that are loaded into the database are located in the import folder.
Nodes are created first by loading in properties only relevant to that particular node using a cypher command such as:

```
LOAD CSV WITH HEADERS FROM "file:///genre.csv" AS row CREATE (genre:Genre{GenreId: TOINT(row.GenreId)})
SET genre.Name = row.Name return genre;
```

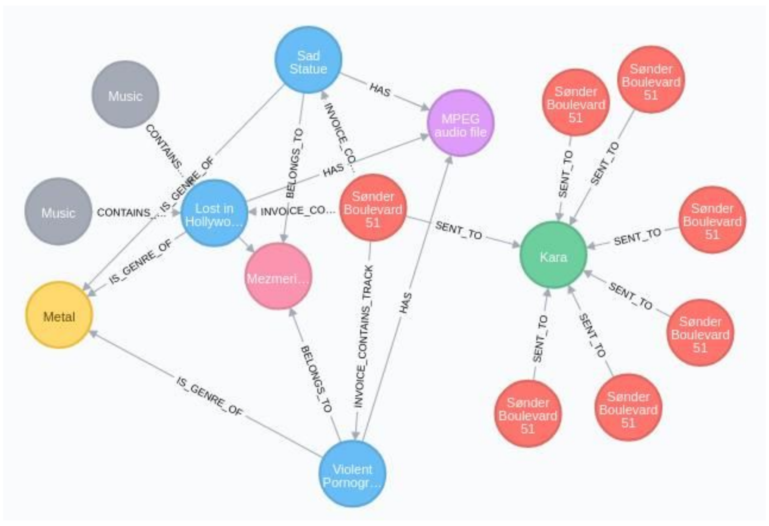
TOINT is used to convert strings into integers which is required for numerical values.

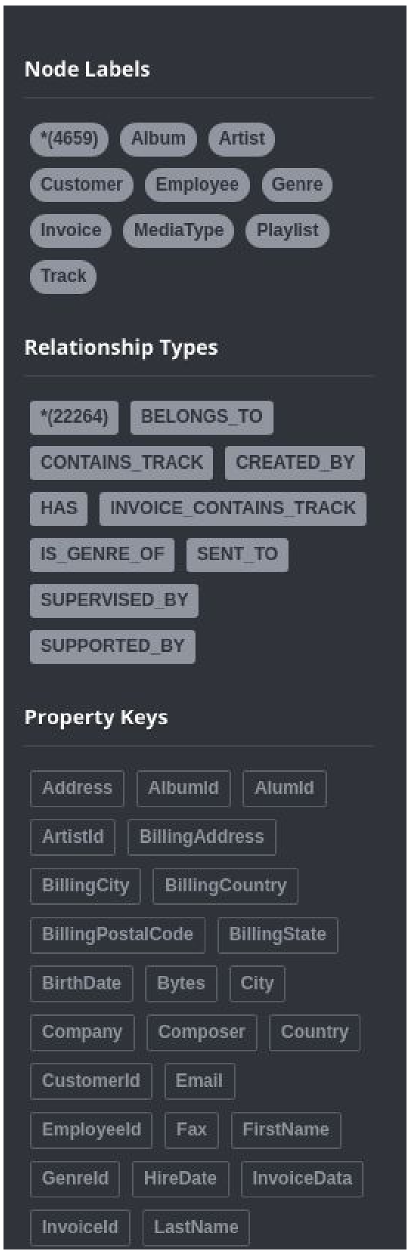
Relationships are created by using load CSV statements and MATCH commands are used to match ID lines in the CSV with ID lines in a corresponding node. Relationships are created using CREATE (a)-[:RELATIONSHIP]->(b). This creates a relationship between nodes a and b. A relationship is stored on disk alongside nodes and can contain their own properties. This enables Neo4J To remove the joining tables that MySQL uses to join a many to many relationship.
A relationship command looks like this:

```
LOAD CSV WITH HEADERS FROM "file:///track.csv" AS row MATCH(track:Track{TrackId: TOINT(row.TrackId)})
MATCH(genre:Genre {GenreId: TOINT(row.GenreId)}) CREATE (track)-[:IS_GENRE_OF]->(genre)
```

Examples of Loaded Data

Nodes showing a customer's various invoices, some of the songs they have purchased and their properties.





Node Labels

*(4659) Album Artist Customer Employee Genre Invoice MediaType Playlist Track

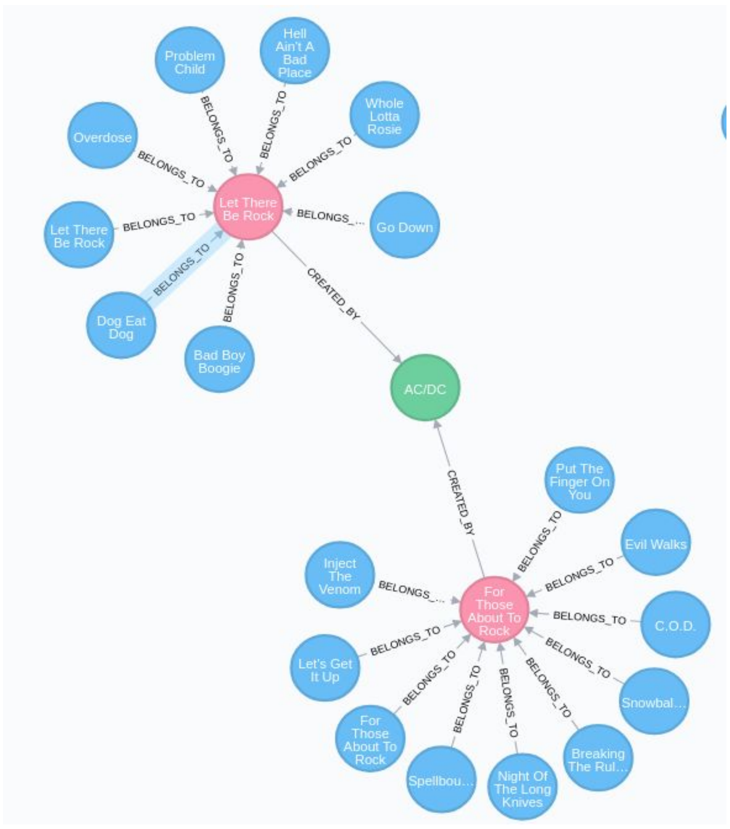
Relationship Types

*(22264) BELONGS_TO CONTAINS_TRACK CREATED_BY HAS INVOICE_CONTAINS_TRACK IS_GENRE_OF SENT_TO SUPERVISED_BY SUPPORTED_BY

Property Keys

Address AlbumId AlumId ArtistId BillingAddress BillingCity BillingCountry BillingPostalCode BillingState BirthDate Bytes City Company Composer Country CustomerId Email EmployeeId Fax FirstName GenreId HireDate InvoiceData InvoiceId LastName

Songs created by AC/DC



4 - NoSQL Database Queries

These queries were created to represent what we imagine would be “real world queries” that an online music service would hope to gain insight to. Because of this we imagined that many of the queries would center around the invoice and factors that relate to this part of the data set.
Also taking inspiration from how we as a group use music streaming services such as Spotify, Deezer, Amazon etc. we emphasised a strong focus on certain aspects such as understanding the most popular genre of the moment which could be used by Chinook to give recommendations of what songs and artists who encompass that genre to the user.

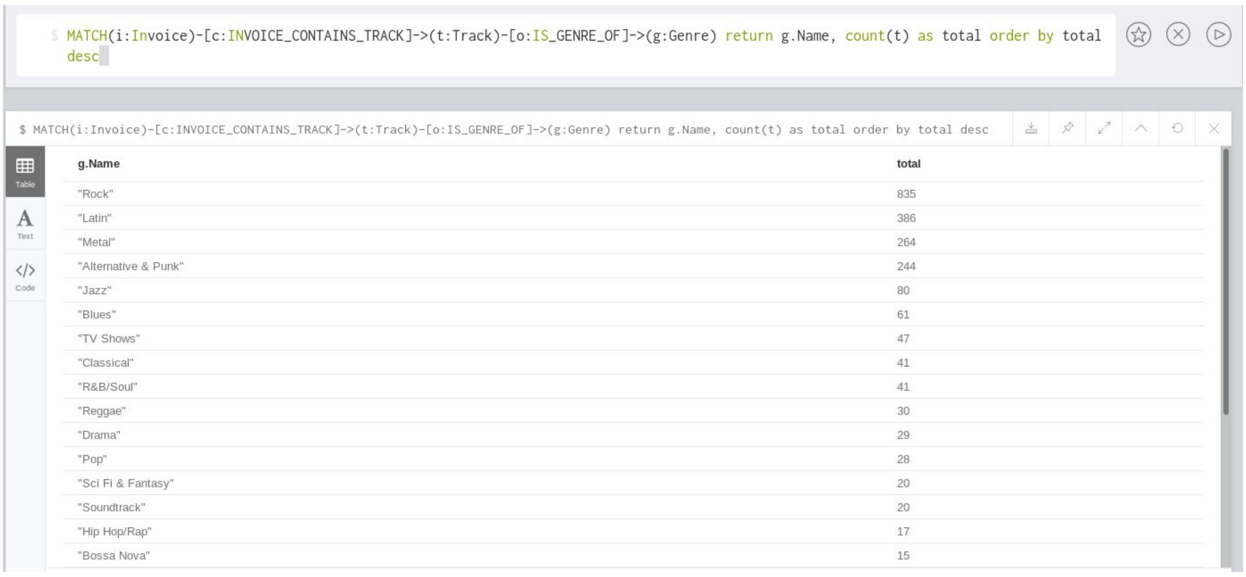
1) Order the popularity of Genre by how often it appears in the invoice of customers.

-Reasoning

Assuming that the company may want to drive more sales having an idea of the most commonly occuring genre involved with the invoice by a count of the total number of tracks and their accompanying genre.

-Neo4J

MATCH(i:Invoice)-[c:INVOICE_CONTAINS_TRACK]->(t:Track)-[o:IS_GENRE_OF]->(g:Genre) return g.Name , count(t) as total order by total desc



```
$ MATCH(i:Invoice)-[c:INVOICE_CONTAINS_TRACK]->(t:Track)-[o:IS_GENRE_OF]->(g:Genre) return g.Name, count(t) as total order by total
desc
```

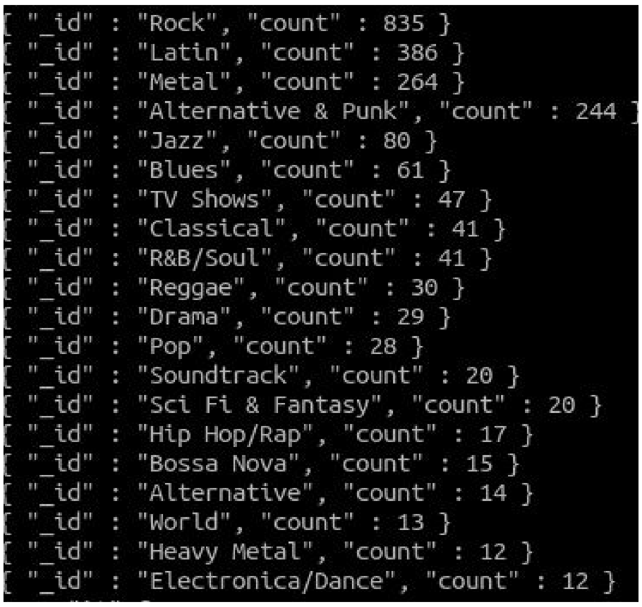
```
$ MATCH(i:Invoice)-[c:INVOICE_CONTAINS_TRACK]->(t:Track)-[o:IS_GENRE_OF]->(g:Genre) return g.Name, count(t) as total order by total desc
```

g.Name	total
"Rock"	835
"Latin"	386
"Metal"	264
"Alternative & Punk"	244
"Jazz"	80
"Blues"	61
"TV Shows"	47
"Classical"	41
"R&B/Soul"	41
"Reggae"	30
"Drama"	29
"Pop"	28
"Sci Fi & Fantasy"	20
"Soundtrack"	20
"Hip Hop/Rap"	17
"Bossa Nova"	15

-MongoDB

db.invoice.aggregate([{ $group:{ _id:"$Invoice.genre", count:{ $sum:1 } } },{$sort: {count:-1}}]);

Result:



```
{ "_id" : "Rock", "count" : 835 }
{ "_id" : "Latin", "count" : 386 }
{ "_id" : "Metal", "count" : 264 }
{ "_id" : "Alternative & Punk", "count" : 244 }
{ "_id" : "Jazz", "count" : 80 }
{ "_id" : "Blues", "count" : 61 }
{ "_id" : "TV Shows", "count" : 47 }
{ "_id" : "Classical", "count" : 41 }
{ "_id" : "R&B/Soul", "count" : 41 }
{ "_id" : "Reggae", "count" : 30 }
{ "_id" : "Drama", "count" : 29 }
{ "_id" : "Pop", "count" : 28 }
{ "_id" : "Soundtrack", "count" : 20 }
{ "_id" : "Sci Fi & Fantasy", "count" : 20 }
{ "_id" : "Hip Hop/Rap", "count" : 17 }
{ "_id" : "Bossa Nova", "count" : 15 }
{ "_id" : "Alternative", "count" : 14 }
{ "_id" : "World", "count" : 13 }
{ "_id" : "Heavy Metal", "count" : 12 }
{ "_id" : "Electronica/Dance", "count" : 12 }
```

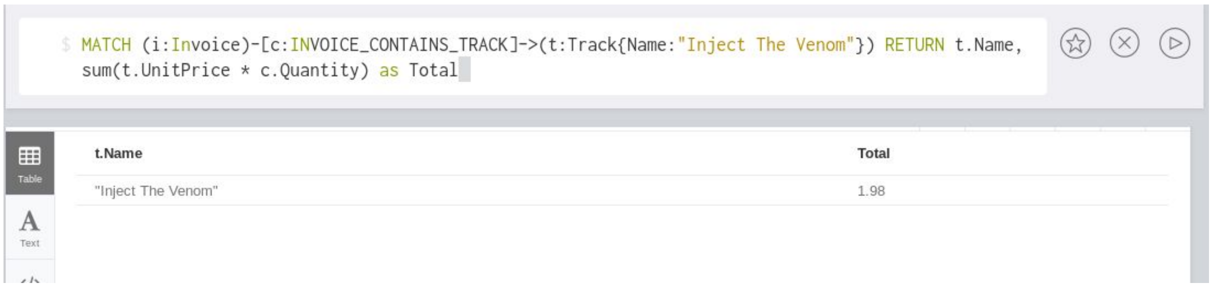
2) Given a specific "song", query how much revenue that song has generated.

<u>-Reasoning</u>

Given any service is used to generate income, having a query that the company can use to find the total income made by each song would be particularly useful. This like the previous query could be used to promote particularly popular songs to other users who may not have that song currently in their playlist.

<u>-Neo4J</u>

MATCH (i:Invoice)-[c:INVOICE_CONTAINS_TRACK]->(t:Track{Name:"Inject The Venom"}) RETURN t.Name, sum(t.UnitPrice * c.Quantity) as Total



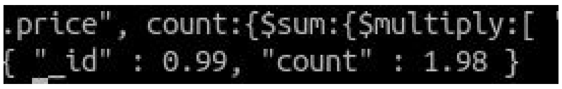
```
$ MATCH (i:Invoice)-[c:INVOICE_CONTAINS_TRACK]->(t:Track{Name:"Inject The Venom"}) RETURN t.Name,
  sum(t.UnitPrice * c.Quantity) as Total
```

t.Name	Total
"Inject The Venom"	1.98

<u>-MongoDB</u>

db.invoice.aggregate([{$match: {"Invoice.trackName":"Inject The Venom"}}, { $group:{ _id:"$Invoice.price", count:{$sum:{$multiply:["$Invoice.price","$Invoice.quantity"]}} } }]);

Result:



```
.price", count:{$sum:{$multiply:[
{ "_id" : 0.99, "count" : 1.98 }
```

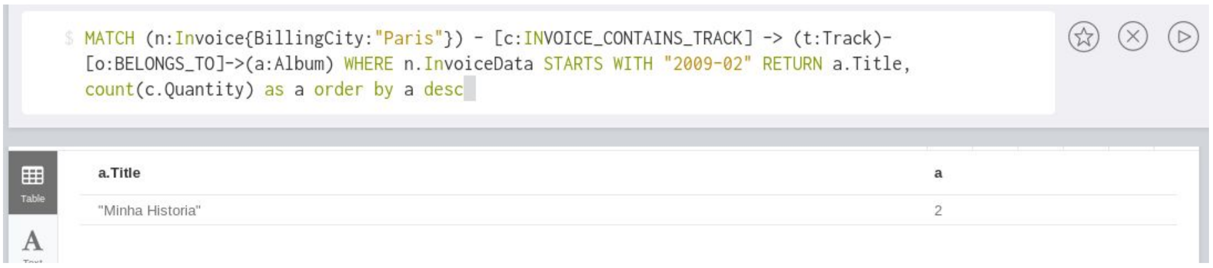
3) Realise the most popular album of the month by billing city.

-Reasoning

Like most services that provide music streaming to clients a popular recommendation is to promote the most “successful” album of that month to generate more sales. The inspiration for this query is that locations such as cities generally follow a similar trend for example Seattle in America is quite closely associated with Grunge. This query could then be used to market the album in certain locations to further drive sales.

-Neo4J

MATCH (n:Invoice{BillingCity: “Paris”}) - [c:INVOICE_CONTAINS_TRACK]-> (t:Track)-[o:BELONGS_TO]->(a:Album) WHERE n.InvoiceData STARTS WITH “2009-02” RETURN a.Title, count(c.Quantity) as a order by a desc



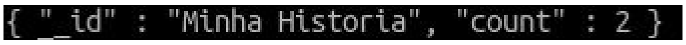
```
$ MATCH (n:Invoice{BillingCity:"Paris"}) - [c:INVOICE_CONTAINS_TRACK] -> (t:Track)-
[o:BELONGS_TO]->(a:Album) WHERE n.InvoiceData STARTS WITH "2009-02" RETURN a.Title,
count(c.Quantity) as a order by a desc
```

a.Title	a
"Minha Historia"	2

-MongoDB

db.invoice.aggregate([{$match:{$and:[{"Invoice.BillingCity":"Paris"},{"Invoice.date":/^2009-02/}]}}, { $group:{ _id:"$Invoice.albumName", count:{ $sum:1 } } }]);

Result:



```
{ "_id" : "Minha Historia", "count" : 2 }
```

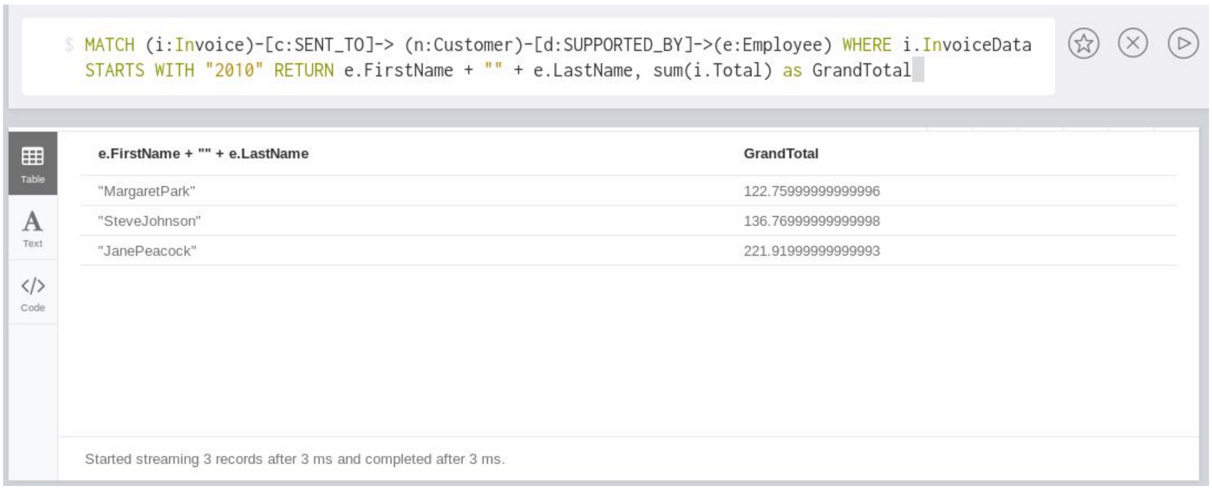
4) Find out how much each employee is making

-Reasoning

For any business to succeed it is important that the business knows the “worth” of that employee. This query will return all employees that have supported a customer and the total amount from the invoice they are responsible for maintaining for a given year.

-Neo4J

MATCH (i:Invoice)-[c:SENT_TO]->(n:Customer)-[d:SUPPORTED_BY]->(e:Employee) WHERE i.InvoiceData STARTS WITH "2010" RETURN e.FirstName + "" + e.LastName, sum(i.Total) as GrandTotal


```
$ MATCH (i:Invoice)-[c:SENT_TO]-> (n:Customer)-[d:SUPPORTED_BY]->(e:Employee) WHERE i.InvoiceData
STARTS WITH "2010" RETURN e.FirstName + "" + e.LastName, sum(i.Total) as GrandTotal
```

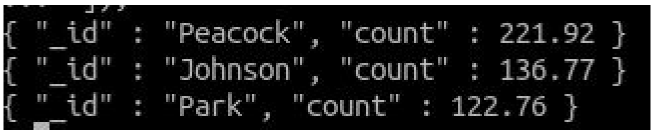
e.FirstName + "" + e.LastName	GrandTotal
"MargaretPark"	122.75999999999996
"SteveJohnson"	136.76999999999998
"JanePeacock"	221.91999999999993

Started streaming 3 records after 3 ms and completed after 3 ms.

-MongoDB

db.invoice.aggregate([{$match: {"Invoice.date":/^2010/}}, { $group:{ _id:"$Invoice.employeeLastName", count:{ $sum:"$Invoice.price" } } },{$sort: {count:-1}}]);

Result :


```
{ "_id" : "Peacock", "count" : 221.92 }
{ "_id" : "Johnson", "count" : 136.77 }
{ "_id" : "Park", "count" : 122.76 }
```

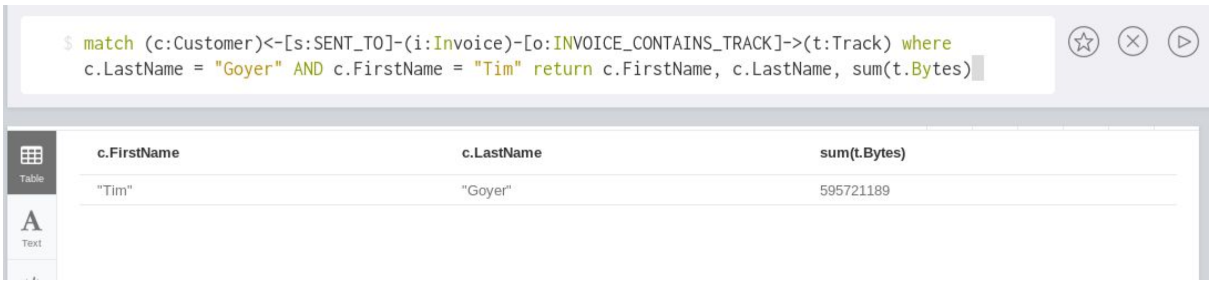
5) For a given customer find out how many bytes their invoice has created

-Reasoning

For both the customer and the service it is important that the total amount of bytes can be found. This would be especially useful if for instance the user was using the service through a mobile device and could potentially save their tracks to the device.

-Neo4J

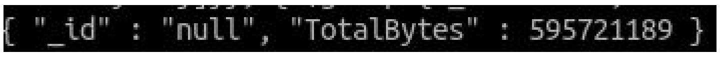
match(c:Customer)<-[s:SENT_TO]-(i:Invoice)-[o:INVOICE_CONTAINS_TRACK]->(t:Track) where c.LastName="Goyer" AND c.FirstName="Tim" return c.FirstName,c.LastName, sum(t.Bytes)


```
$ match (c:Customer)<-[s:SENT_TO]-(i:Invoice)-[o:INVOICE_CONTAINS_TRACK]->(t:Track) where
c.LastName = "Goyer" AND c.FirstName = "Tim" return c.FirstName, c.LastName, sum(t.Bytes)
```

c.FirstName	c.LastName	sum(t.Bytes)
"Tim"	"Goyer"	595721189

<u>-MongoDB</u>

db.invoice.aggregate([{$match:{$and:[{"Invoice.CustomerFirstName":"Tim"},{"Invoice.CustomerLastName":"Goyer"}]}}, { $group:{ _id:"null", TotalBytes:{ $sum:"$Invoice.Bytes" } } }]);

Result: 

```
{ "_id" : "null", "TotalBytes" : 595721189 }
```

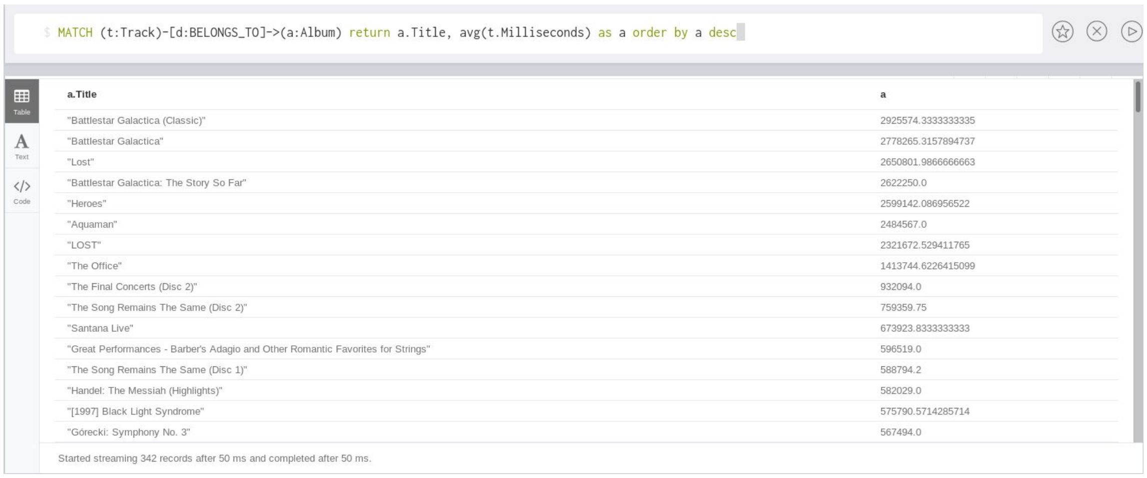
6) Obtain the average track length in minutes from each album.

<u>-Reasoning</u>

This could be used by the company if they tried to mimic an "About" feature which some services do such as Spotify. This would then show the business which albums have long tracks and could be used to market certain features about them. It could also be used as a preliminary indication how much space that album would take up on the users device if it was downloaded.

<u>-Neo4J</u>

MATCH (t:Track)-[d:BELONGS_TO]->(a:Album) return a.Title, avg(t.Milliseconds) as a order by a desc



```
$ MATCH (t:Track)-[d:BELONGS_TO]->(a:Album) return a.Title, avg(t.Milliseconds) as a order by a desc
```

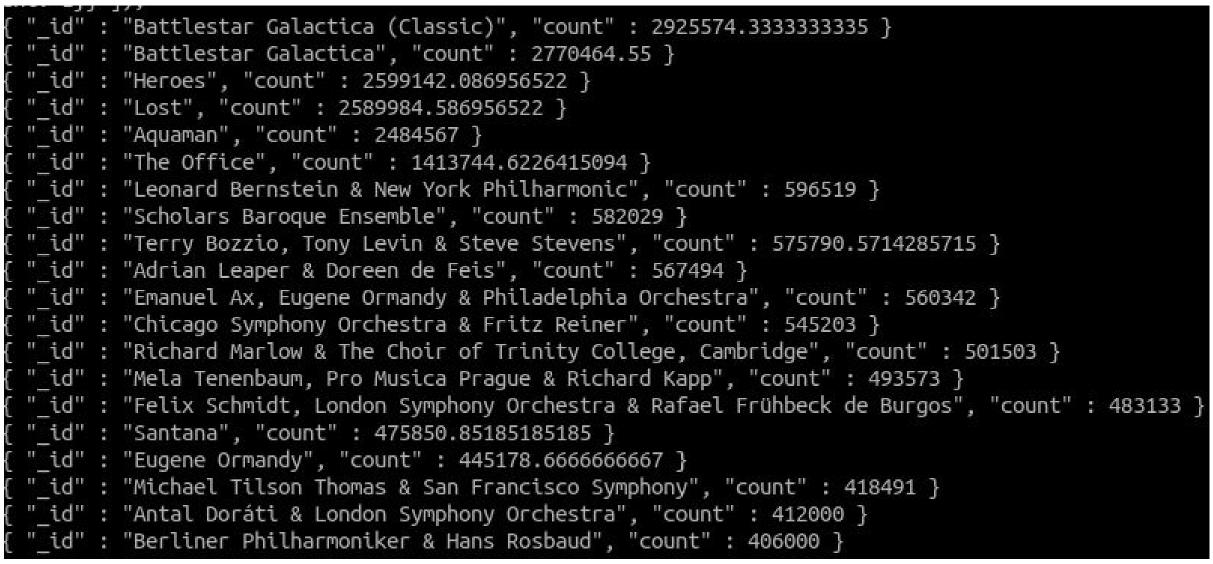
a.Title	a
"Battlestar Galactica (Classic)"	2925574.3333333335
"Battlestar Galactica"	2778265.3157894737
"Lost"	2650801.9866666663
"Battlestar Galactica: The Story So Far"	2622250.0
"Heroes"	2599142.086956522
"Aquaman"	2484567.0
"LOST"	2321672.529411765
"The Office"	1413744.6226415099
"The Final Concerts (Disc 2)"	932094.0
"The Song Remains The Same (Disc 2)"	759359.75
"Santana Live"	673923.8333333333
"Great Performances - Barber's Adagio and Other Romantic Favorites for Strings"	596519.0
"The Song Remains The Same (Disc 1)"	588794.2
"Handel: The Messiah (Highlights)"	582029.0
"[1997] Black Light Syndrome"	575790.5714285714
"Górecki: Symphony No. 3"	567494.0

Started streaming 342 records after 50 ms and completed after 50 ms.

<u>-MongoDB</u>

db.track.aggregate([{ $group:{ _id:"$Artist", count:{ $avg:"$Track.milliseconds" } } },{$sort: {count:-1}}]);

Result



```
{ "_id" : "Battlestar Galactica (Classic)", "count" : 2925574.3333333335 }
{ "_id" : "Battlestar Galactica", "count" : 2770464.55 }
{ "_id" : "Heroes", "count" : 2599142.086956522 }
{ "_id" : "Lost", "count" : 2589984.586956522 }
{ "_id" : "Aquaman", "count" : 2484567 }
{ "_id" : "The Office", "count" : 1413744.6226415094 }
{ "_id" : "Leonard Bernstein & New York Philharmonic", "count" : 596519 }
{ "_id" : "Scholars Baroque Ensemble", "count" : 582029 }
{ "_id" : "Terry Bozzio, Tony Levin & Steve Stevens", "count" : 575790.5714285715 }
{ "_id" : "Adrian Leaper & Doreen de Feis", "count" : 567494 }
{ "_id" : "Emanuel Ax, Eugene Ormandy & Philadelphia Orchestra", "count" : 560342 }
{ "_id" : "Chicago Symphony Orchestra & Fritz Reiner", "count" : 545203 }
{ "_id" : "Richard Marlow & The Choir of Trinity College, Cambridge", "count" : 501503 }
{ "_id" : "Mela Tenenbaum, Pro Musica Prague & Richard Kapp", "count" : 493573 }
{ "_id" : "Felix Schmidt, London Symphony Orchestra & Rafael Frühbeck de Burgos", "count" : 483133 }
{ "_id" : "Santana", "count" : 475850.85185185185 }
{ "_id" : "Eugene Ormandy", "count" : 445178.6666666667 }
{ "_id" : "Michael Tilson Thomas & San Francisco Symphony", "count" : 418491 }
{ "_id" : "Antal Doráti & London Symphony Orchestra", "count" : 412000 }
{ "_id" : "Berliner Philharmoniker & Hans Rosbaud", "count" : 406000 }
```

Note:
When trying to complete the implementation of query 6 what we noticed is that the same query displayed different results when ran on the opposing NoSQL system. Initially this left us confused as many of the same albums appear on the list however what we concluded was how the average works for both MongoDB and Neo4j.

This could be the result of one solution upon using addition only takes into consideration the integer value of the track as a collection i.e. 255 + 350 = 605 / 2 = 302.5 where as the other uses the full float value i.e. 255.36+350.74 = 606.1 / 2 = 303.0505 as seen here this gives a difference of 0.5505 which seems minute over two values but when there is multiple entities in the search that are associated with the query and given the much larger numbers being worked with (Milliseconds) this would seem to be the reasoning as to why we have received the results we have been given.

We tried different ways to combat this factor such as using a floor aggregation in both MongoDB and Neo4j, however, both times the result remained the same and the query returned the results in a different order as previously explained.

5 - Evaluation

Group Contribution

Throughout this project, a full group contribution has been given to make sure that the specification has been delivered on time and is to the standards expected of undergraduate students.
Part One saw all members find information about the relative NoSQL systems where we give an overview of all three possible models that could be chosen rather than just two asked for in the specification. This was done so we could highlight which NoSQL solutions would be the best to carry forward in the coursework.
In Part Two work was done by Michael and David in documenting and designing of the ETL pipelines for both MongoDB and Neo4j. This would ultimately be used to transfer the data correctly from the MySQL database to the given NoSQL database.
For Part, Three Jordan designed queries that could be seen as useful to a music streaming service and also implemented the queries that produced the output that was received as a result of implementation.
In conclusion, all group members contributed to the creation of the document and work was distributed equally amongst all members.

Conclusive Summary

During the duration of this coursework, we have been working with Neo4j and MongoDB both of which have quickly shown advantages and disadvantages when transitioning from a MySQL database into a NoSQL system.

Although as previously stated, both NoSQL solutions have their benefits and drawbacks what can easily be recognised is the learning curve when adapting a MongoDB solution to the given problem. This could be deciphered as preliminary inexperience within the group with working with Mongo and quickly trying to understand how the document driven database works compared to the graph solution provided by Neo4j. Given Neo4j is not necessarily a table relationship model like MySQL, transferring to the graph based system proved much simpler due to having a visual representation of the created database which was given through the web interface. This is not to say there was no learning curve involved with Neo, merely a graphical representation aided in understanding how the system worked when compared to Mongo where all the work was done on the terminal or the creation of scripts to assist in the transitioning. This often provided little to no error responses essentially leaving us working in the dark with some aspects with a significant emphasis on a brute force approach until a result was given where we could then develop further to create the structure envisioned for Mongo.

ETL- Mongodb proved very challenging in creating a database that could be used to perform efficient queries. Our first iteration contained five collections and was designed with too much normalisation to perform queries without performing lookups between at least three collections. In the first iteration of the design our ETL process involved regex modification of a plain tsv file to implement arrays and the removal of null values.
This design was abandoned as it became infeasible to pull any meaningful data from the database.
The current iteration involved extreme denormalization of data from the MySQL database. Most of the transformations are now performed in SQL itself to create three collections Tracks, Playlist, and Invoices so that all related information is stored in as few collections as possible. This enabled interesting queries to be performed without having to create expensive lookups across collections.
Creating the ETL in this way meant that we did not have to create any embedded arrays or perform any cleanup of tsv files and they could be loaded straight into Mongodb.

Neo4J ETL process was tricky as there was more cleaning of intermediate files involved before the data would be in the correct format for neo4J. MySQL outputs tab separated files by default so in order to convert them into comma separated files, the linux program sed was used to clean up the files in multiple stages. Another problem with Neo4J was modelling the initial designs too closely to the relational model of MySQL, as such, joining tables would crash the server as millions of relationships were being created. The solution to this was the realisation that relationships themselves could store the information that was previously held in joining tables of MySQL. The data was loaded effectively once relationships were correctly implemented.

Querying-

When implementing our chosen queries in both MongoDB and Neo4j it became apparent quite quickly that Neo4j was vastly easier to implement for. This was primarily due to the support given online, the GUI giving an error message when the query was implemented wrong and mostly because Neo4j acts so closely to a MySQL query.
Because of this familiarity shared with MySQL where we all have experience, completing the designed queries for Neo4j was done much faster with more time allocated to making sure the return from the queries was correct.
MongoDB however, was not as forgiving when trying to implement the given queries this was due mainly to writing the queries on the terminal and what was originally a text file but because of the parenthesis and bracketing conditions was moved to a json formatter to make some of the work easier. In general more time was spent trying to get queries to work with MongoDB than Neo4j due to the way queries are executed within Mongo, due to the design of our database and collections it also required more time to obtain the “correct” result when compared to Neo4j.

Final Recommendation-

Given the completion of the specification as a group we believe that the best solution for Chinook is to migrate from their current RDBMS to Neo4j. Our recommendation comes from a variety of reasoning including Neo4j’s easier learning curve, ease of maintenance (with regards to developing and distributing queries) and use of Neo4j’s GUI to help with having an interactive view over the database. Migration of data is also a much easier process and more enticing to companies who can see a clear improvement to the ease of which their data can be accessed. Neo4J’s ability to form complex queries with minimal amount of cypher code can allow organisations to observe trends between data that are much more difficult to create using mongodb’s extremely un-intuitive query language.

Improvements-

During the course of this project two main improvements have been established by the group. Firstly, more time should have been given to creating a suitable design for our MongoDB schema which does work to specification however we feel it could of been optimised to make querying and general use much easier for ourselves. This however, is more a statement of our previously mentioned comments about having little interaction with document storage databases and essentially working by trial and error with reliance on online libraries to help guide us in the correct direction.

The second improvement we could of implemented was one that we had originally designed but due to the reiteration of creating a suitable schema to work with for Mongo was to develop scripts that would better handle the ETL part of the coursework. What is meant by this is we could have created a more professional set of scripts that would more easily be suited for mass migration when considering this was only a portion of the data used by Chinook.

6 - References

[1] Soni, M. (2015). *An Overview of Cassandra.*
Available at: https://opensourceforu.com/2015/07/an-overview-of-apache-cassandra/
(Accessed 26th January 2019).

[2] DataStax. (2019). *A brief introduction to Apache Cassandra (Online Course).*
Available at:
https://academy.datastax.com/units/brief-introduction-apache-cassandra?resource=brief-introduction-apache-cassandra
(Accessed 28th January 2019).

[3] Jelasity, M. (2014). *Gossip Protocols.*
Available at: http://www.inf.u-szeged.hu/~jelasity/ddm/gossip.pdf
(Accessed 29th January 2019).

[4] DataStax. (2019). *How is data written? | Apache Cassandra.*
Available at:
https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlHowDataWritten.html
(Accessed 28th January 2019).

[5] DB-Engines. (2019). *DB-Engines Ranking*. Available at: https://db-engines.com/en/ranking [Accessed 31 January. 2019].

[6] Tutorialspoint (2019). *MongoDB Overview*. Available at:
https://www.tutorialspoint.com/mongodb/mongodb_overview.htm [Accessed 31 Jan. 2019].

[7] Dzone. (2016). *When to Use (and Not to Use) MongoDB - DZone Database*. Available at:
https://dzone.com/articles/why-mongodb [Accessed 30 Jan. 2019].

[8] MongoDB (2019). *Architecture Guide*. Available at:
https://www.mongodb.com/collateral/mongodb-architecture-guide [Accessed 1 Feb. 2019].

[9] Bsonspec. (n.d.). *BSON (Binary JSON) Serialization*. Available at: http://bsonspec.org/ [Accessed 3 Feb. 2019].

[10] Docs.mongodb. (2019). *Replication — MongoDB Manual*. Available at:
https://docs.mongodb.com/manual/replication/ [Accessed 3 Feb. 2019].

[11] NEWS BBVA. (n.d.). *Neo4j: What a graph database is and what it is used for- BBVA*. Available at:
https://www.bbva.com/en/neo4j-what-a-graph-database-is-and-what-it-is-used-for/ [Accessed 5 Feb. 2019].

[12] Tutorialspoint (2019). *Neo4j Quick Guide*. Available at:
https://www.tutorialspoint.com/neo4j/neo4j_quick_guide.htm [Accessed 6 Feb. 2019].

[13] InfoQ. (n.d.). *Neo4j 3.1 Supports Causal Clustering and Security Enhancements*. Available at: https://www.infoq.com/news/2016/12/neo4j-3.1 [Accessed 6 Feb. 2019].

[14] Neo4j.com. (n.d.). *Chapter 5. Clustering - The Neo4j Operations Manual v3.5*. Available at: https://neo4j.com/docs/operations-manual/current/clustering/ [Accessed 6 Feb. 2019].

7 Appendices

Appendix A - MySQL extraction and script for MongoDB

```
/**** Playlist Export ***/
use Chinook;
SELECT p.Name AS 'PlaylistName',t.Name AS 'TrackName'
FROM Playlist p, PlaylistTrack l, Track t
WHERE p.PlaylistId = l.PlaylistId AND l.TrackId = t.TrackId

/***Invoice Export***/
use Chinook;
SELECT l.InvoiceLineId AS 'InvoiceLine',l.InvoiceID AS 'InvoiceId', i.InvoiceDate AS 'Invoice.date', i.BillingAddress AS
'Invoice.BillingAddress', i.BillingCity AS 'Invoice.BillingCity', i.BillingState AS 'Invoice.BillingState', i.BillingCountry AS
'Invoice.BillingCountry'
,i.BillingPostalcode AS 'Invoice.BillingPostalCode', i.Total AS 'InvoiceTotal',
t.TrackId AS 'Invoice.trackID',t.Name AS 'Invoice.trackName',a.Title AS 'Invoice.albumName', r.Name AS
'Invoice.artistName',g.Name AS'Invoice.genre',
t.Composer AS 'Invoice.composer',t.Milliseconds AS 'Invoice.Milliseconds',t.Bytes AS 'Invoice.Bytes',l.quantity AS
'Invoice.quantity', l.UnitPrice AS 'Invoice.price'
 ,c.FirstName AS 'Invoice.CustomerFirstName' , c.LastName AS 'Invoice.CustomerLastName', c.Company AS
'Invoice.company', c.Phone AS 'Invoice.phone',
 c.Fax AS'Invoice.fax', c.Email AS'Invoice.email', e.LastName AS'Invoice.employeeLastName', e.FirstName AS
'Invoice.employeeFirstName',
 e.Title AS 'Invoice.employeeTitle', e.BirthDate AS 'Invoice.employeeBirthDate', e.HireDate AS
'Invoice.employeeHireDate', e.Address AS 'Invoice.employeeAddress',
 e.City AS 'Invoice.employeeCity', e.State AS'Invoice.employeeState', e.Country AS 'Invoice.employeeCountry',
e.PostalCode AS 'Invoice.employeePostalCode',
 e.Phone AS 'Invoice.employeePhone', e.Fax AS' Invoice.employeeFax', e.Email AS 'Invoice.employeeEmail'
FROM Invoice i, InvoiceLine l, Track t, Customer c, Employee e, Album a, Artist r, Genre g
WHERE l.InvoiceId = i.InvoiceId
AND l.TrackId = t.TrackId
AND c.CustomerId = i.CustomerId
AND c.SupportRepId = e.EmployeeId
AND t.AlbumId = a.AlbumId
AND r.ArtistId = a.ArtistId
AND t.GenreId = g.GenreId
order by l.InvoiceId, l.InvoiceLineid;
```

```
/**** Track Export ***/
use Chinook;
SELECT Track.TrackId AS trackID, Track.Name AS 'Track.name', Album.Title AS 'Album', Artist.Name as 'Artist',
Genre.Name as 'Genre', Track.Milliseconds AS 'Track.milliseconds', Track.Composer AS 'track.Composer',
MediaType.Name as 'MediaType.name', Track.UnitPrice AS 'Track.unitprice', Track.Bytes AS 'Track.Bytes'
FROM Track, Artist, Album, Genre, MediaType
WHERE Track.AlbumId = Album.AlbumId
AND Track.GenreId = Genre.GenreId
AND Track.MediaTypeId = MediaType.MediaTypeId
AND Album.ArtistId = Artist.ArtistId
```

Appendix B - Loading Script for MongoDB

```
mongoimport -u dah3 -p  --authenticationDatabase admin --collection invoice --type tsv --file ~/Desktop/mongoinvoice.tsv
--headerline
mongoimport -u dah3 -p  --authenticationDatabase admin --collection playlist --type tsv --file ~/Desktop/mongoplaylist.tsv
--headerline
mongoimport -u dah3 -p  --authenticationDatabase admin --collection track --type tsv --file ~/Desktop/mongotrack.tsv
--headerline
```

Appendix C - MySQL Extraction for Neo4J

```
mysql -u root -p < ~/Desktop/invoice.sql | sed -e 's/,//g' -e 's/\t/,/g' -e 's/"//g' >
~/Desktop/neo4j-community-3.4.4/import/invoice.csv

mysql -u root -p < ~/Desktop/customer.sql | sed -e 's/,//g' -e 's/\t/,/g' -e 's/"//g' >
~/Desktop/neo4j-community-3.4.4/import/customer.csv

mysql -u root -p < ~/Desktop/employee.sql | sed -e 's/,//g' -e 's/\t/,/g' -e 's/"//g' >
~/Desktop/neo4j-community-3.4.4/import/employee.csv

mysql -u root -p < ~/Desktop/playlist.sql | sed -e 's/,//g' -e 's/\t/,/g' -e 's/"//g' >
~/Desktop/neo4j-community-3.4.4/import/playlist.csv

mysql -u root -p < ~/Desktop/genre.sql | sed -e 's/,//g' -e 's/\t/,/g' -e 's/"//g' >
~/Desktop/neo4j-community-3.4.4/import/genre.csv

mysql -u root -p < ~/Desktop/artist.sql | sed -e 's/,//g' -e 's/\t/,/g' -e 's/"//g' >
~/Desktop/neo4j-community-3.4.4/import/artist.csv

mysql -u root -p < ~/Desktop/album.sql | sed -e 's/,//g' -e 's/\t/,/g' -e 's/"//g' >
~/Desktop/neo4j-community-3.4.4/import/album.csv

mysql -u root -p < ~/Desktop/PlaylistTrack.sql | sed -e 's/,//g' -e 's/\t/,/g' -e 's/"//g' >
~/Desktop/neo4j-community-3.4.4/import/PlaylistTrack.csv

mysql -u root -p < ~/Desktop/invoiceLine.sql | sed -e 's/,//g' -e 's/\t/,/g' -e 's/"//g' >
~/Desktop/neo4j-community-3.4.4/import/invoiceLine.csv

mysql -u root -p < ~/Desktop/MediaType.sql | sed -e 's/,//g' -e 's/\t/,/g' -e 's/"//g' >
~/Desktop/neo4j-community-3.4.4/import/MediaType.csv

mysql -u root -p < ~/Desktop/invoice.sql | sed -e 's/,//g' -e 's/\t/,/g' -e 's/"//g' >
~/Desktop/neo4j-community-3.4.4/import/invoice.csv
```

Appendix D - Neo4J Cypher Insert Queries

```
//load in non foreign key properties first, relationships will be made later
LOAD CSV WITH HEADERS FROM "file:///track.csv" AS row CREATE (track:Track{TrackId: TOINT(row.TrackId)})
SET track.Name = row.Name, track.Composer = row.Composer, track.Milliseconds =  TOINT(row.Milliseconds),
track.Bytes = TOINT(row.Bytes), track.UnitPrice = TOFLOAT(row.UnitPrice) return track;

LOAD CSV WITH HEADERS FROM "file:///genre.csv" AS row CREATE (genre:Genre{GenreId: TOINT(row.GenreId)})
SET genre.Name = row.Name return genre;

LOAD CSV WITH HEADERS FROM "file:///album.csv" AS row CREATE (album:Album{AlbumId:
TOINT(row.AlbumId)}) SET album.Title = row.Title return album;

LOAD CSV WITH HEADERS FROM "file:///artist.csv" AS row CREATE (artist:Artist{ArtistId: TOINT(row.ArtistId)})
SET artist.Name = row.Name return artist;

LOAD CSV WITH HEADERS FROM "file:///media.csv" AS row CREATE (mediatype:MediaType{MediaTypeId:
TOINT(row.MediaTypeId)}) SET mediatype.Name = row.Name return mediatype;

LOAD CSV WITH HEADERS FROM "file:///employee.csv" AS row CREATE (employee:Employee{EmployeeId:
TOINT(row.EmployeeId)}) SET employee.LastName = row.LastName, employee.FirstName = row.FirstName,
employee.Title = row.Title,employee.ReportsTo=row.ReportsTo ,employee.BirthDate = row.BirthDate, employee.HireDate
= row.HireDate, employee.Address = row.Address, employee.City = row.City, employee.State = row.State,
employee.Country = row.Country, employee.PostalCode = row.PostalCode, employee.Phone = row.Phone,
employee.Fax=row.Fax, employee.Email=row.Email return employee;

LOAD CSV WITH HEADERS FROM "file:///customer.csv" AS row CREATE (customer:Customer{CustomerId:
TOINT(row.CustomerId)}) SET customer.FirstName = row.FirstName, customer.LastName = row.LastName,
customer.Company = row.Company,customer.Address = row.Address, customer.City = row.City, customer.State =
row.State, customer.Country = row.Country, customer.PostalCode = row.PostalCode, customer.Phone = row.Phone,
customer.Fax=row.Fax, customer.Email=row.Email

LOAD CSV WITH HEADERS FROM "file:///invoice.csv" AS row CREATE (invoice:Invoice{InvoiceId:
TOINT(row.InvoiceId)}) SET invoice.InvoiceData = row.InvoiceDate, invoice.BillingAddress = row.BillingAddress,
invoice.BillingCity = row.BillingCity, invoice.BillingState = row.BillingState, invoice.BillingCountry = row.BillingCountry,
invoice.BillingPostalCode = row.BillingPostalCode, invoice.Total = TOFLOAT(row.Total)

LOAD CSV WITH HEADERS FROM "file:///playlist.csv" AS row CREATE (playlist:Playlist{PlaylistId:
TOINT(row.PlaylistId)}) SET playlist.Name = row.Name return playlist;
```

Appendix E - Neo4J Relationship Declarations

```
//relationships
LOAD CSV WITH HEADERS FROM "file:///track.csv" AS row MATCH(track:Track{TrackId: TOINT(row.TrackId)})
MATCH(genre:Genre {GenreId: TOINT(row.GenreId)}) CREATE (track)-[:IS_GENRE_OF]->(genre)

LOAD CSV WITH HEADERS FROM "file:///track.csv" AS row MATCH(track:Track{TrackId: TOINT(row.TrackId)})
MATCH(album:Album {AlbumId: TOINT(row.AlbumId)}) CREATE (track)-[:BELONGS_TO]->(album)

LOAD CSV WITH HEADERS FROM "file:///album.csv" AS row MATCH(album:Album{AlbumId:
TOINT(row.AlbumId)}) MATCH(artist:Artist {ArtistId: TOINT(row.ArtistId)}) CREATE
(album)-[:CREATED_BY]->(artist)
```

```
LOAD CSV WITH HEADERS FROM "file:///track.csv" AS row MATCH(track:Track{TrackId: TOINT(row.TrackId)})
MATCH(mediatype:MediaType {MediaTypeId: TOINT(row.MediaTypeId)}) CREATE (track)-[:HAS]->(mediatype)

LOAD CSV WITH HEADERS FROM "file:///invoice.csv" AS row MATCH(invoice:Invoice{InvoiceId:
TOINT(row.InvoiceId)}) MATCH(customer:Customer{CustomerId: TOINT(row.CustomerId)}) CREATE
(invoice)-[:SENT_TO]->(customer)

LOAD CSV WITH HEADERS FROM "file:///customer.csv" AS row MATCH(customer:Customer{CustomerId:
TOINT(row.CustomerId)}) MATCH(employee:Employee {EmployeeId: TOINT(row.SupportRepId)}) CREATE
(customer)-[:SUPPORTED_BY]->(employee) RETURN customer,employee;

LOAD CSV WITH HEADERS FROM "file:///employee.csv" AS row MATCH(employee:Employee{EmployeeId:
TOINT(row.EmployeeId)}) MATCH(supervisor:Employee {EmployeeId: TOINT(row.ReportsTo)}) CREATE
(employee)-[:SUPERVISED_BY]->(supervisor) RETURN employee,supervisor;

LOAD CSV WITH HEADERS FROM "file:///PlaylistTrack.csv" AS row MATCH (playlist:Playlist {PlaylistId:
TOINT(row.PlaylistId)}) MATCH (track:Track {TrackId: TOINT(row.TrackId)}) CREATE
(playlist)-[:CONTAINS_TRACK]->(track)

LOAD CSV WITH HEADERS FROM "file:///InvoiceLine.csv" AS row MATCH (invoice:Invoice {InvoiceId:
TOINT(row.InvoiceId)}) MATCH (track:Track {TrackId: TOINT(row.TrackId)}) CREATE
(invoice)-[i:INVOICE_CONTAINS_TRACK]->(track) SET i.UnitPrice=TOFLOAT(row.UnitPrice),
i.Quantity=TOINT(row.Quantity)
```