Part 1: Data Retrieval Queries

1. Simple Queries

1. Write a query to return the Name of every Track in the database.

[**SELECT**](http://weeb01.cems.uwe.ac.uk/phpmyadmin/url.php?url=http%3A%2F%2Fdev.mysql.com%2Fdoc%2Frefman%2F5.5%2Fen%2Fselect.html&token=65abab681351e8e129ab199cd0808891) name

**FROM** Track

2. Write a SQL query that returns album and track information for tracks where the composer information is missing.

[**SELECT**](http://weeb01.cems.uwe.ac.uk/phpmyadmin/url.php?url=http%3A%2F%2Fdev.mysql.com%2Fdoc%2Frefman%2F5.5%2Fen%2Fselect.html&token=65abab681351e8e129ab199cd0808891) Album.Title, Track.Name

**FROM** Track

**INNER** **JOIN** Album **ON** Track.AlbumId = Album.AlbumId

**WHERE** Composer **IS** **NULL**



3. Select everything from Invoice orderd by InvoiceDate, newest to oldest

[**SELECT**](http://weeb01.cems.uwe.ac.uk/phpmyadmin/url.php?url=http%3A%2F%2Fdev.mysql.com%2Fdoc%2Frefman%2F5.5%2Fen%2Fselect.html&token=65abab681351e8e129ab199cd0808891) \*

**FROM** Invoice

**ORDER** **BY** InvoiceDate **DESC**

4. List all the Track Names for PlayListID = 17

[**SELECT**](http://weeb01.cems.uwe.ac.uk/phpmyadmin/url.php?url=http%3A%2F%2Fdev.mysql.com%2Fdoc%2Frefman%2F5.5%2Fen%2Fselect.html&token=65abab681351e8e129ab199cd0808891) Name

**FROM** Track

**INNER** **JOIN** PlaylistTrack **ON** Track.TrackID = PlaylistTrack.TrackID

**WHERE** PlaylistTrack.PlaylistID =17



5. Write a SQL Query that returns the following from the Chinook Database:

Artist Name

Album Title

Track Name

Genre Name

Composer

Unit Price

[**SELECT**](http://weeb01.cems.uwe.ac.uk/phpmyadmin/url.php?url=http%3A%2F%2Fdev.mysql.com%2Fdoc%2Frefman%2F5.5%2Fen%2Fselect.html&token=65abab681351e8e129ab199cd0808891) Artist.Name, Album.Title, Track.Name, Genre.Name, Track.Composer, Track.UnitPrice

**FROM** Track

**INNER** **JOIN** Album **ON** Album.AlbumId = Track.AlbumId

**INNER** **JOIN** Genre **ON** Genre.GenreId = Track.GenreId

**INNER** **JOIN** Artist **ON** Artist.ArtistId = Album.ArtistId

1. Built in Function Queries

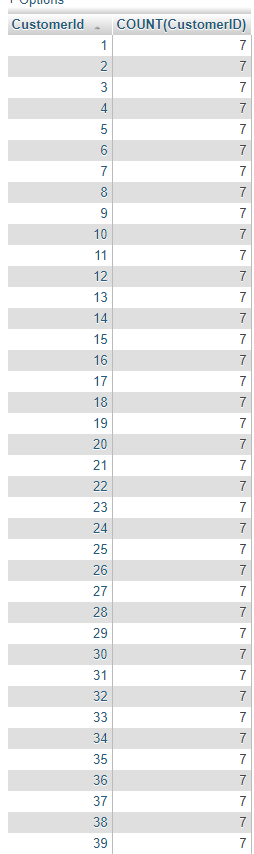
1. Write a query will return the total number of invoices grouped by customerid.

[**SELECT**](http://weeb01.cems.uwe.ac.uk/phpmyadmin/url.php?url=http%3A%2F%2Fdev.mysql.com%2Fdoc%2Frefman%2F5.5%2Fen%2Fselect.html&token=ee36c08d2d571e8c35f0637fba722b32) CustomerId, COUNT( CustomerID )

**FROM** Invoice

**GROUP** **BY** CustomerId

**ORDER** **BY** CustomerId



2. Write a query which will return total number of tracks per AbumID

[**SELECT**](http://weeb01.cems.uwe.ac.uk/phpmyadmin/url.php?url=http%3A%2F%2Fdev.mysql.com%2Fdoc%2Frefman%2F5.5%2Fen%2Fselect.html&token=ee36c08d2d571e8c35f0637fba722b32) Album.AlbumId, Album.Title, COUNT( TrackId )

**FROM** Track

**INNER** **JOIN** Album **ON** Album.AlbumId = Track.AlbumId

**GROUP** **BY** AlbumId

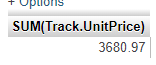
**ORDER** **BY** AlbumId



3. If you bought one copy of every track, how much would you spend in total?

**SELECT** SUM( Track.UnitPrice )

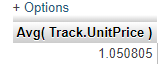
**FROM** Track



4. What is the average track price in the store?

[**SELECT**](http://weeb01.cems.uwe.ac.uk/phpmyadmin/url.php?url=http%3A%2F%2Fdev.mysql.com%2Fdoc%2Frefman%2F5.5%2Fen%2Fselect.html&token=ee36c08d2d571e8c35f0637fba722b32) AVG( Track.UnitPrice )

**FROM** Track

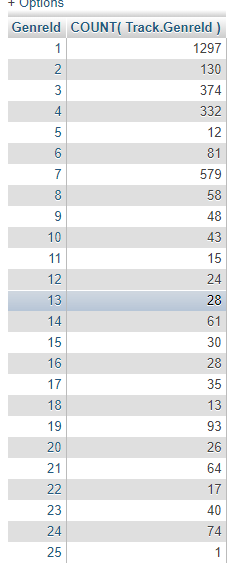


5. How many distinct genres of music are represented by tracks sold in the store?

[**SELECT**](http://weeb01.cems.uwe.ac.uk/phpmyadmin/url.php?url=http%3A%2F%2Fdev.mysql.com%2Fdoc%2Frefman%2F5.5%2Fen%2Fselect.html&token=ee36c08d2d571e8c35f0637fba722b32) GenreId, COUNT( Track.GenreId )

**FROM** Track

**GROUP** **BY** GenreId



The Tracks in the store cover all 25 genres listed in the genre table

6. How many artists have albums in the store?

[**SELECT**](http://weeb01.cems.uwe.ac.uk/phpmyadmin/url.php?url=http%3A%2F%2Fdev.mysql.com%2Fdoc%2Frefman%2F5.5%2Fen%2Fselect.html&token=ee36c08d2d571e8c35f0637fba722b32) COUNT( **DISTINCT** Album.ArtistId )

**FROM** Album



7. Create a view (named “cheap\_tracks”) for all tracks where UnitPrice < 1

[**CREATE VIEW**](http://weeb01.cems.uwe.ac.uk/phpmyadmin/url.php?url=http%3A%2F%2Fdev.mysql.com%2Fdoc%2Frefman%2F5.5%2Fen%2Fselect.html&token=ee36c08d2d571e8c35f0637fba722b32) cheap\_tracks [**AS**](http://weeb01.cems.uwe.ac.uk/phpmyadmin/url.php?url=http%3A%2F%2Fdev.mysql.com%2Fdoc%2Frefman%2F5.5%2Fen%2Fselect.html&token=ee36c08d2d571e8c35f0637fba722b32)

[**SELECT**](http://weeb01.cems.uwe.ac.uk/phpmyadmin/url.php?url=http%3A%2F%2Fdev.mysql.com%2Fdoc%2Frefman%2F5.5%2Fen%2Fselect.html&token=ee36c08d2d571e8c35f0637fba722b32) Track.Name

**FROM** Track

**WHERE** UnitPrice < 1

C. Advanced Queries

1. Find the FirstNames of the employees that are older than their corresponding supervisor. Output should have only one column named "FirstName".

[**SELECT**](http://weeb01.cems.uwe.ac.uk/phpmyadmin/url.php?url=http%3A%2F%2Fdev.mysql.com%2Fdoc%2Frefman%2F5.5%2Fen%2Fselect.html&token=ee36c08d2d571e8c35f0637fba722b32) Employee.FirstName

**FROM** Employee

**WHERE** **EXISTS** (

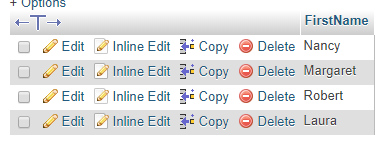
[**SELECT**](http://weeb01.cems.uwe.ac.uk/phpmyadmin/url.php?url=http%3A%2F%2Fdev.mysql.com%2Fdoc%2Frefman%2F5.5%2Fen%2Fselect.html&token=ee36c08d2d571e8c35f0637fba722b32) \*

**FROM** Employee **AS** temp

**WHERE** temp.EmployeeId = Employee.ReportsTo

**AND** temp.BirthDate > Employee.BirthDate

)



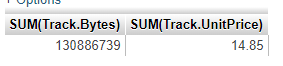
2. What is the space, in bytes, occupied by the playlist “Grunge", and how much would it cost? (Assume that the cost of a playlist is the sum of the price of its constituent tracks).

[**SELECT**](http://weeb01.cems.uwe.ac.uk/phpmyadmin/url.php?url=http%3A%2F%2Fdev.mysql.com%2Fdoc%2Frefman%2F5.5%2Fen%2Fselect.html&token=ee36c08d2d571e8c35f0637fba722b32) SUM( Track.Bytes ) , SUM( Track.UnitPrice )

**FROM** Track

**INNER** **JOIN** PlaylistTrack **ON** Track.TrackId = PlaylistTrack.TrackId

**WHERE** PlaylistId =16



3. Find audio tracks which have a length longer than the average length of all the audio tracks;

Get Average Length for error checking

[**SELECT**](http://weeb01.cems.uwe.ac.uk/phpmyadmin/url.php?url=http%3A%2F%2Fdev.mysql.com%2Fdoc%2Frefman%2F5.5%2Fen%2Fselect.html&token=ee36c08d2d571e8c35f0637fba722b32) AVG( Track.Milliseconds )

**FROM** Track



Actual Query for question

[**SELECT**](http://weeb01.cems.uwe.ac.uk/phpmyadmin/url.php?url=http%3A%2F%2Fdev.mysql.com%2Fdoc%2Frefman%2F5.5%2Fen%2Fselect.html&token=ee36c08d2d571e8c35f0637fba722b32) Name, Milliseconds

**FROM** Track

**WHERE** Track.Milliseconds > (

[**SELECT**](http://weeb01.cems.uwe.ac.uk/phpmyadmin/url.php?url=http%3A%2F%2Fdev.mysql.com%2Fdoc%2Frefman%2F5.5%2Fen%2Fselect.html&token=ee36c08d2d571e8c35f0637fba722b32) AVG( Track.Milliseconds )

**FROM** Track

)



4. Which playlists do not contain any tracks for the artists “Black Sabbath" nor “Chico Buarque"?

[**SELECT**](http://weeb01.cems.uwe.ac.uk/phpmyadmin/url.php?url=http%3A%2F%2Fdev.mysql.com%2Fdoc%2Frefman%2F5.5%2Fen%2Fselect.html&token=ee36c08d2d571e8c35f0637fba722b32) **DISTINCT** Playlist.PlaylistId, Playlist.Name

**FROM** Playlist

**INNER** **JOIN** PlaylistTrack **ON** Playlist.PlaylistID = PlaylistTrack.PlaylistId

**INNER** **JOIN** Track **ON** Track.TrackId = PlaylistTrack.TrackId

**INNER** **JOIN** Album **ON** Album.AlbumId = Track.AlbumId

**INNER** **JOIN** Artist **ON** Artist.ArtistId = Album.ArtistId

**WHERE** **NOT** Artist.Name = "Black Sabbath"

**AND** **NOT** "Chico Buarque"

**ORDER BY**  PlaylistId



Part 2: Entity Relationship, Relational Modelling and NoSQL solution

1. ER Diagram

Assumptions made in the following ER Diagram:

1. There is only a single stage in the theatre. Based on the sentence “box office staff are able to confirm the play being performed that evening” key words being “the play” singular.
2. Checking “which day of the week is any given date” isn’t within the scope of a database and would be handled by the UI. The primary key of the Performance table Performance\_Date would be used to either calculate the day of the week or input into an API.
3. Whether a performance is matinee or evening is determined from the Performance\_Date DATETIME in the Performance Table. I.e. if time in the DATETIME is before 17:00 on a given day it is matinee and evening if it is after.
4. “When a caller pays with a credit card, it must be recorded whether they were there in person or on the telephone and an authorization code must be obtained and recorded.” Is taken to mean when a “customer” pays with a credit card since a “caller” could mean a person calling but then it would be pointless to record whether they were in person or on the telephone.

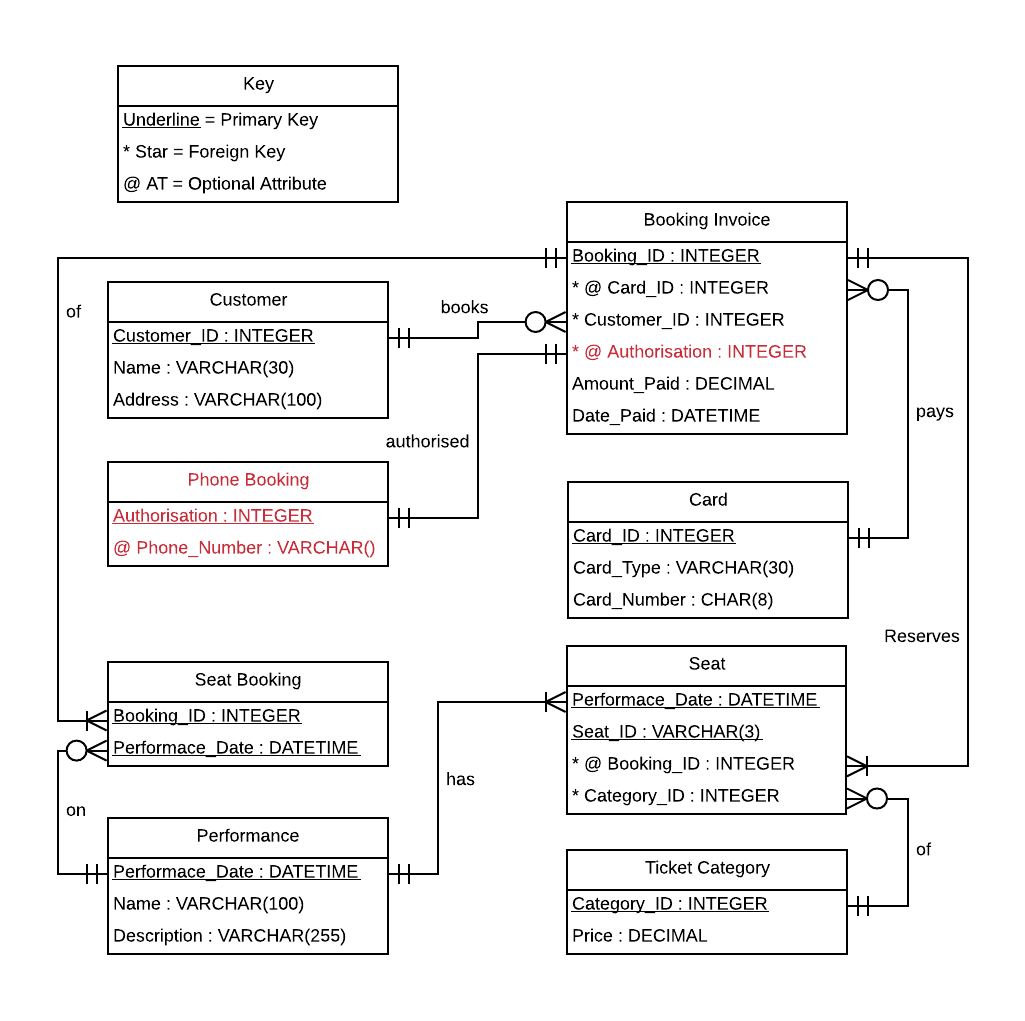
If a customer is in person and pays with cash the foreign keys Card\_ID and Authorisation in Booking Invoice remain NULL

If a customer is in person and pays with card the foreign key Card\_ID, in Booking Invoice, is populated with the Card\_ID of the relevant card in the Card Table and Authorisation, in Booking Invoice, remain NULL

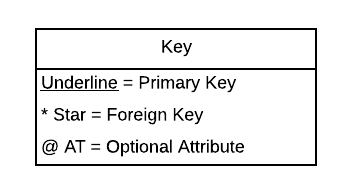
If a customer is on the phone they can not pay in cash so no booking is made

If a customer is on the phone and pays with card the foreign key Card\_ID, in Booking Invoice, is populated with the Card\_ID of the relevant card in the Card Table and Authorisation, in Booking Invoice and Phone Booking, is populated with the authorisation obtained. The Phone\_Number, in Phone Booking, is optionally recorded to allow contacting the customer.

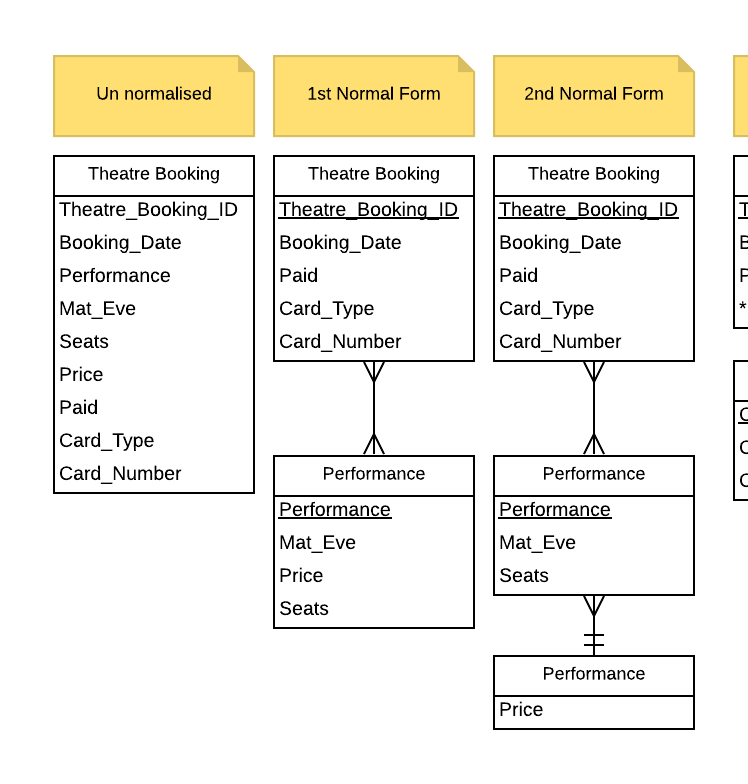
1. As per Card No in the table for the normalisation section Card\_Number in Card Type are assumed to all be 8 digits, possibly beginning with 0. They are thus are stored as CHAR to have a fixed length and maintain the preceding zero.
2. A Seat\_ID is up to 3 character in a VARCHAR since they are stored as A to Z 0 to 50. The shortest ID is A0 and the longest Z50

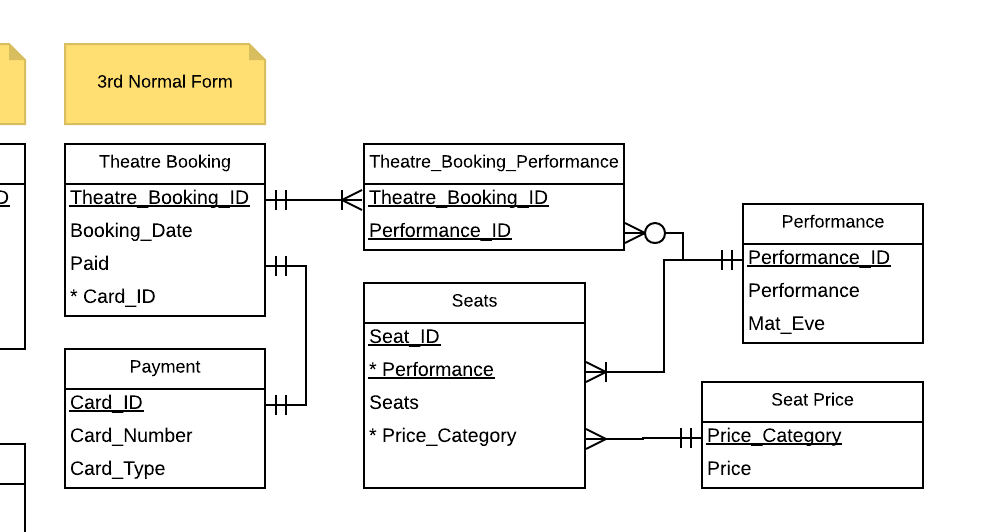
****

1. Normalisation



|  |  |  |  |
| --- | --- | --- | --- |
| Un-Normalised Data | 1st Normal Form | 2nd Normal Form | 3rd Normal Form |
| Choose a suitable key and list all data | Separate repeating data into different groups. Groups have keys compound or not  (Assuming that two performances can occur on a single day the key must be compound) | Separate into new tables items identified by only part of the key  (The price of seats is not related to the performance but the category of the seats themselves) | Separate into new tables items identified by non key items and resolve any many to many relations |
| Theatre\_Booking\_ID  Booking\_Date  Performance  Mat\_Eve  Seats  Price  Paid  Card\_Type  Card\_Number | Theatre\_Booking\_ID  Booking\_Date  Paid  Card\_Type  Card\_Number  Performance  Mat\_Eve  Price  Seats | Theatre\_Booking\_ID  Booking\_Date  Paid  Card\_Type  Card\_Number  Performance  Mat\_Eve  Seats  Price | Theatre\_Booking\_ID  Booking\_Date  Paid  \* Card\_ID  Card\_ID  Card\_Number  Card\_Type  Performance\_ID  Theatre\_Booking\_ID  Performance\_ID  Performance  Mat\_Eve  Seat\_ID  \* Performance  seats  \* Price\_Category  Price\_Category  Price |





1. Extension to ER Diagram

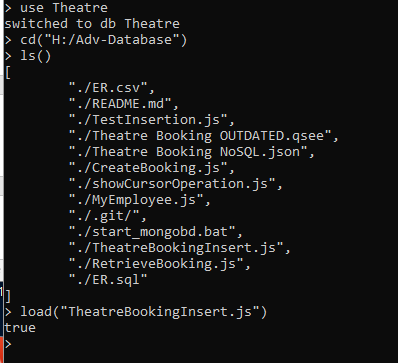
The extension to allow for telephone booking is highlighted in red in the ER Diagram in section A. The assumption being that details about the booking are entered by the staff member making the call in a similar manner to how detail would be entered if the person was in front of the member of staff. The difference being the detail as highlighted in red are entered.

It is also assumed that the Authorisation code is unique for each booking.

1. NoSQL Solution

Building the Database

The layout for the database was created with test data then that layout was imported using the following javascript file. After running use Theatre in the command line then using cd() to navigate to the file location.



db.Booking\_Collection.insert(

{

\_id: 1,

"Customer" : 1,

"Amount\_Paid" : 240,

"Date\_Paid" : "2019-11-27",

"Card\_Type" : "MASTERCARD",

"Card\_Number" : "84108529",

"Authorisation" : "",

"Phone\_Number" : "",

"Performances\_Booked" :

[

{

"id" : 1,

"Seats" : "A1,A2"

},

{

"id" : 2,

"Seats" : "C2-C5"

}

]

})

db.Performance\_Collection.insert(

[

{

\_id: 1,

"Date" : "2019-12-12 11:00:00",

"Free\_Seats" : "A3-Z50",

"Name" : "Wet Dog",

"Description" : "A dog gets wet",

"Mat\_Eve" : "Mat"

},

{

\_id: 2,

"Date" : "2019-12-13 18:00:00",

"Free\_Seats" : "A1-B30,C1,C6-Z50",

"Name" : "Late Night",

"Description" : "A person stays up late",

"Mat\_Eve" : "Eve"

},

{

\_id: 3,

"Date" : "2019-12-14 12:00:00",

"Free\_Seats" : "A1-Z50",

"Name" : "Early Morning",

"Description" : "A person wakes up early",

"Mat\_Eve" : "Mat"

}

])

db.Categories\_Collection.insert(

[

{

\_id: 1,

"Begins" : "A",

"Ends" : "K",

"Price" : "40",

"Number\_In\_Row" : "30"

},

{

\_id: 2,

"Begins" : "L",

"Ends" : "S",

"Price" : "30",

"Number\_In\_Row" : "40"

},

{

\_id: 3,

"Begins" : "T",

"Ends" : "Z",

"Price" : "25",

"Number\_In\_Row" : "50"

}

])

db.Customer\_Collection.insert(

[

{

\_id: 1,

"First\_Name" : "Nott",

"Last\_Name" : "Abbot",

"Address" : "23 not a street"

},

{

\_id: 2,

"First\_Name" : "Carne",

"Last\_Name" : "Age",

"Address" : "n/a"

},

{

\_id: 3,

"First\_Name" : "Linton",

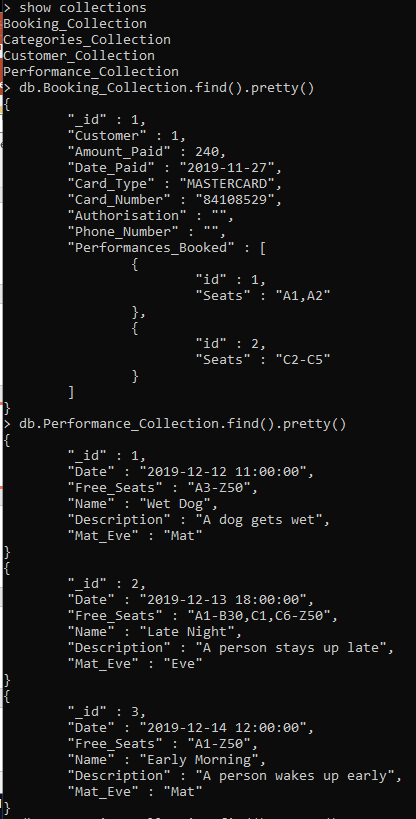
"Last\_Name" : "Lash",

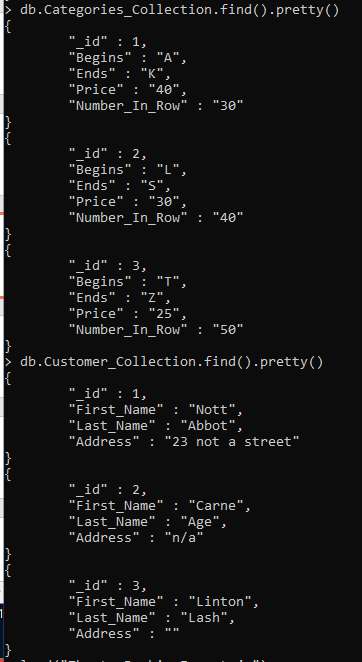
"Address" : "123 Street"

}

])

This adds the following collections to the Theatre database





Creating a Booking

The exact process would be implementation specific but would involve a script that constructs the following:

db.Booking\_Collection.insert(

{

\_id: 2,

"Customer" : 4,

"Amount\_Paid" : 400,

"Date\_Paid" : "2019-10-20",

"Card\_Type" : "VISA",

"Card\_Number" : "92219013",

"Authorisation" : "201",

"Phone\_Number" : "01664012012",

"Performances\_Booked" :

[

{

"id" : "2019-12-14 12:00:00",

"Seats" : "A1,A2,C3-C5"

}

]

})

The booking process would involve retrieving details like available seats and their prices to inform the customer and to calculate the total price of the booking..

It would also involve searching the Customer\_Collection to determine whether the customer already exists or not to determine whether they should be added to the list. Also to find the highest customer \_id to increment it to be used as the \_id.

In this case the customer is new so they are added to the Customer\_Collection

db.Customer\_Collection.insert(

{

\_id: 4,

"First\_Name" : "Clinton",

"Last\_Name" : "East",

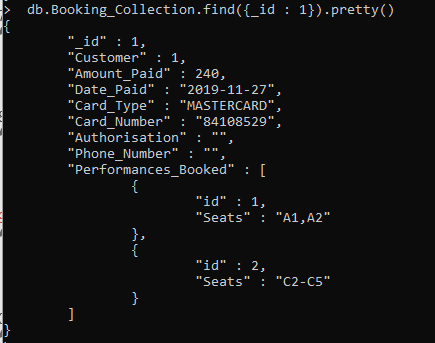
"Address" : "West"

})

Retrieving a Booking

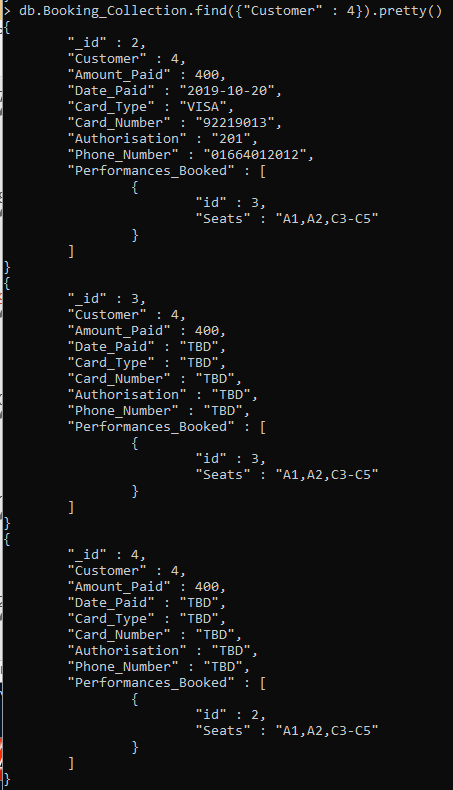
To find a single booking the following command can be used

db.Booking\_Collection.find({\_id : 1}).pretty()



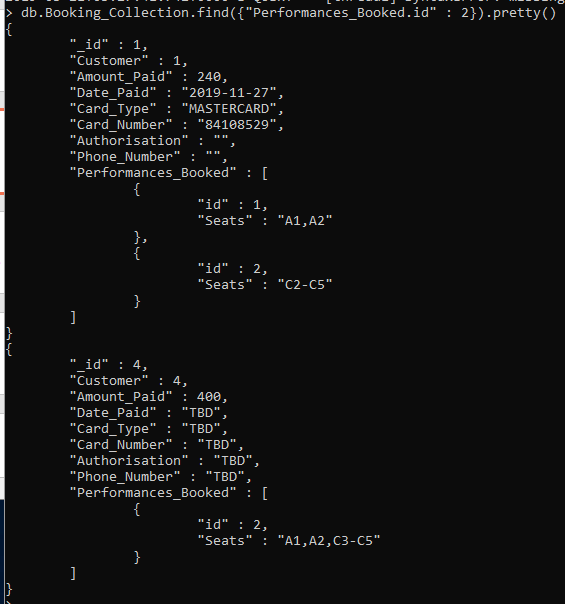
To retrieve all bookings made by a certain customer:

db.Booking\_Collection.find({"Customer" : 4}).pretty()



To retrieve all bookings for a certain performance

db.Booking\_Collection.find({"Performances\_Booked.id" : 2}).pretty()



To retrieve customer, performance, and/or pricing details, say to be shown on an invoice, would again be implementation specific but would involve retrieving the relevant booking in Booking\_Collection then using the id’s to retrieve the corresponding items from Customer\_Collection, Performance\_Collection, and Categories\_Collection

1. Compare and Contrast Reflection

Creating the relational solution before the non-relational one meant that determining important information for the situation was already completed and the largest work was determining how to order that information into a form that fully utilised the benefits of a non-relational database. This involved deciding which sections of the data belong in their own separate collections and which could simply be embedded documents.

In the non-relational solution the elements relating to payment card and authorisation of card payments over phone bookings are embedded documents because the degree of duplicated data is minimal and by doing so it reduces the need for the complex application code that is needed to resolve the relational queries.

The relational solution keeps data duplication to a minimum since it is fully normalised however in doing so it complicates the queries that are required to access that data.

Given that there are one thousand seats in the theatre, (11 \* 30) + (8 \* 40) + (7 \* 50) = 1000, the relational solution introduces another thousand rows in the seats table for every performance. This allows for the categories of seats to be assigned and their relation to performances and bookings to be resolved. However this is likely to be a cause of performance issued when querying the table. In contrast the non-relational solution stores which seats are free and which are booked in separate items in the database as a string of comma separated values where single seats are stored as their assigned letter and number and a range of seats without any gaps is stored by the first and last values. For example a customer wants to book seats 1 , 3,4,5,6 , 29,30 in row A and seats 1,2,3 in row B.

That would be stored as A1 , A3-A6 , A29-30, B1-B3

While doing so compresses a thousand rows into two values it adds an extra level of complexity in regards to retrieving and utilising the data in the application layer. It also means that all bookings must access the same value to update which seats are booked which could cause issues if a queue is formed leading to a slow response to the application making the request. This would not occur in the relational solution since each seat on a performance can be accessed separately.

In conclusion I am unsure if my solutions fully demonstrate the advantages and disadvantages of either method however the work involved in solving the same problem with different methods did help to contrast the differences and similarities between the two.