Introduction

For the assessment for the CSC7052 Database module we were asked to deliver a project in which we reverse engineered a database for the commercial event ticketing system www.ticketmaster.co.uk. The main aim of the project included designing and implementing a database system which would reflect the operation of chosen aspects of the ticketing system. Such aspects included; designing the database to allow the user, to create a Ticketmaster account, log into their account, search through the plethora of events Ticketmaster has to offer through various search options, see the many venues and ticket types on offer, select an event and book and pay for tickets for said event, then be able to have an order confirmation and see the event in their upcoming event section.

Given the large capacity of tickets that Ticketmaster can offer and the many ways to search through and access them, it was clear that it would take a lot of examining through the website. To accomplish this, at the beginning of the project, each member of the team decided to create a list of entities and attributes that they saw applicable for the project, then come together and combine ideas. Next, as a team we created a schema diagram that combined all our ideas together. The next few weeks was spent working on this diagram, adding and or deleting tables and fields, suggesting key attributes and entity relationships, until we all agreed on a finished version. This then allowed us to individually create our own diagram and to design and implement our own databases, picking and choosing which elements we thought individually we wanted to include in our databases as well as adding simple data that we deemed appropriate to show the functionality of the database.

To start the assignment, I will provide an example of a section of the original attribute and entities list that was based on reverse engineering Ticketmaster. I will also include the groups final completed schema diagram as well as my own individual final diagram, to explain and justify the important design decisions that I made during the creation of the finished version. Furthermore, I will explain the choice of foreign key constraints between the tables, data types chosen for the database and primary keys used within the database. Whilst discussing these decisions I will also cover the key design features chosen for the database, justifying why I made these decisions and how I applied the concept of normalisation throughout. I will also discuss improvements I could have made and further database design features that I could have implemented given I had more time and resources to do so. Finally, I will discuss and rationalise my report with evidence of SQL queries, to demonstrate the functionality of my implemented database.

Throughout the report I will continuously refer to different tables, foreign keys, primary keys, and non- key attributes. To understand these, it is important to note the naming conventions chosen for the database. The snake_case naming convention was used for the table names that had more than one word e.g. *ticket_type, delivery_costs*. All the key and non-key attributes have also been stored in snake_case e.g. *ticket_type_id, email, service_charge*. The table names and the attribute names appear in this naming convention as this is a widely recognised design within databases and makes for easier readability of the report and the SQL commands that will be included.

Early Developments – Group schema design

The very first step of starting the report began with identifying the entities and attributes through reverse engineering Ticketmaster. We began by individually noting what we classed as entities and their attributes and combined and updated our ideas collaboratively using a shared word document (Appendix I). This provided a good starting point for identifying which entities would be most important in capturing the operation of booking a ticket and which were irrelevant for the booking process itself.

Thereafter, as a group we made use of drawDB, an online tool to construct and export database diagrams, to draw and visualise a database schema diagram. Using drawDB allows you to insert tables, fields, data types, lengths/values, autoincrement fields, select primary keys and create foreign key relationships with one to one, one-to-many and many-to-one cardinality. As a group we focused on discussing the tables that would be needed for everything the user may interact with when researching for events, booking tickets and leaving reviews. We decided to add in tables which reflected this user experience whilst deciding that the other elements of the website including *looking for help, entertainment guides, be part of it* and *corporate* sections of Ticketmaster were not necessary as they take the user to various third-party websites, so are therefore out of scope for exclusively showing the ticket booking process itself.

Moving on from discussing entities and attributes, as a group we decided to add tables and fields into our schema diagram. We decided to add a field that would be autoincremented and set as the primary key for that table. According to MySQL reference manual 5.6.9, autoincrementing is an attribute which can be used to generate a unique identity for new rows. It is often used with primary keys so that each row can be uniquely identified without the need for manually specifying each value yourself when designing a database. A primary

key is a critical concept when designing a database. It is either a combination of columns (composite primary key; if one column alone is not enough to uniquely identify a row) or a single column, that uniquely identifies a record in each table. According to a Microsoft support article, a good candidate for a primary key has certain characteristics; primary keys are said to be unique, with no two rows in each table using the same foreign key and no duplicate records existing. They are also non-nullable, meaning they cannot accept null values, ensuring each row in a table is identifiable. The values that they contain also should ideally never change. Primary keys are used in databases to help create relationships between tables, a fundamental part of relational database structures. We choose to set the autoincremented field as the primary key as these could then be reliable references for other foreign keys in associated tables. In doing so this also made it easier to create foreign key relationships between the tables, ensuring that we maintained referential integrity.

This design decision was also made as it is good for simple scaling in one-to-many relationships. Take the field of *user_id* for example. This particular field appears in the *user* table and within the *booking* and *card* tables, connected by foreign key relationships with one-to-many cardinality. Each table has a *user_id*, however the user themselves will only have one unique user ID. This makes the different relationships between the tables more distinct and better organised and ensures that each field is consistent and unique across the related tables. We continued to develop a schema diagram by considering other relationships between our tables and attempting to connect them correctly using the rules of normalisation.

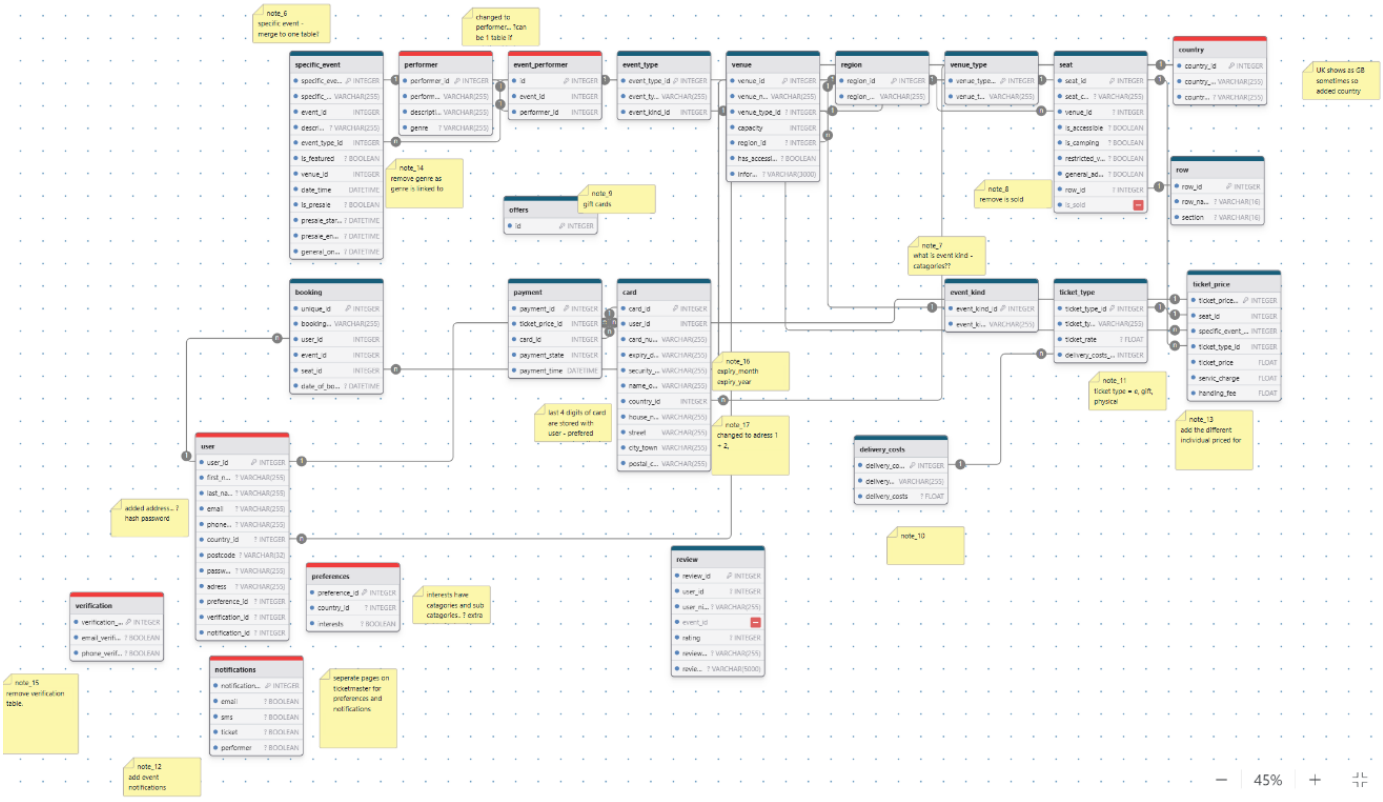


Figure 1- Groups final drawDB diagram with foreign keys and relationships

As a group we made use of surrogate keys quite often when adding columns to our tables. Surrogate keys are unique identifiers for records that are created artificially. They are usually numeric and have no meaning other than being there to uniquely identify the records. They are normally used when natural keys that are derived from the data aren't useful for guaranteeing uniqueness. Take *event_id* in the event table for example, this is a surrogate key which auto-increments. It is used to uniquely identify the event but doesn't actually have any meaning within the context of the events themselves. Similarly, *perfomer_id* in the performer table is a surrogate key, generated automatically to uniquely identify each performer. However, it doesn't have any business meaning outside of its role in identification. This design decision ensured that all the records could be uniquely identifiable without having to rely on complex data. They also provide simplicity by eliminating the need for use of composite keys. Surrogate keys also ensure that the data remains stable. Using natural keys would have meant that these could change over time, such as a user using a new different email address. Finally, choice of

using surrogate keys so often throughout the tables ensured that indexing and performing queries later on would be easier and more efficient than if we had have chosen to use composite keys.

We further developed the diagram by including new tables such as preferences, verification, and notifications, to show the concept of the user having preferential interests, verifying their account through email or phone number and wanting to receive notifications from Ticketmaster. We also discussed columns such as *is_presale*, presale start and end times and general on sale times in order to replicate all of the elements of buying tickets on Ticketmaster. Also, we discussed which entities would associate with other entities, for example, *payment* with *card* and *ticket_price* with *card, seat, specific_event* and *ticket_type* etc. In doing so we created the diagram shown above in Figure 1.

Individual schema design

Moving on from this group attempt I thought it was necessary to make improvements and create a final schema diagram that I was happy with, that I felt was easier to depict and encompassed only the key aspects of booking a ticket on Ticketmaster. Figure 2 shows my final schema diagram. Using my own final diagram also allowed me to then create my own database with sample data which shows the functionality of the database and how it reflects these key aspects of the user booking a ticket.

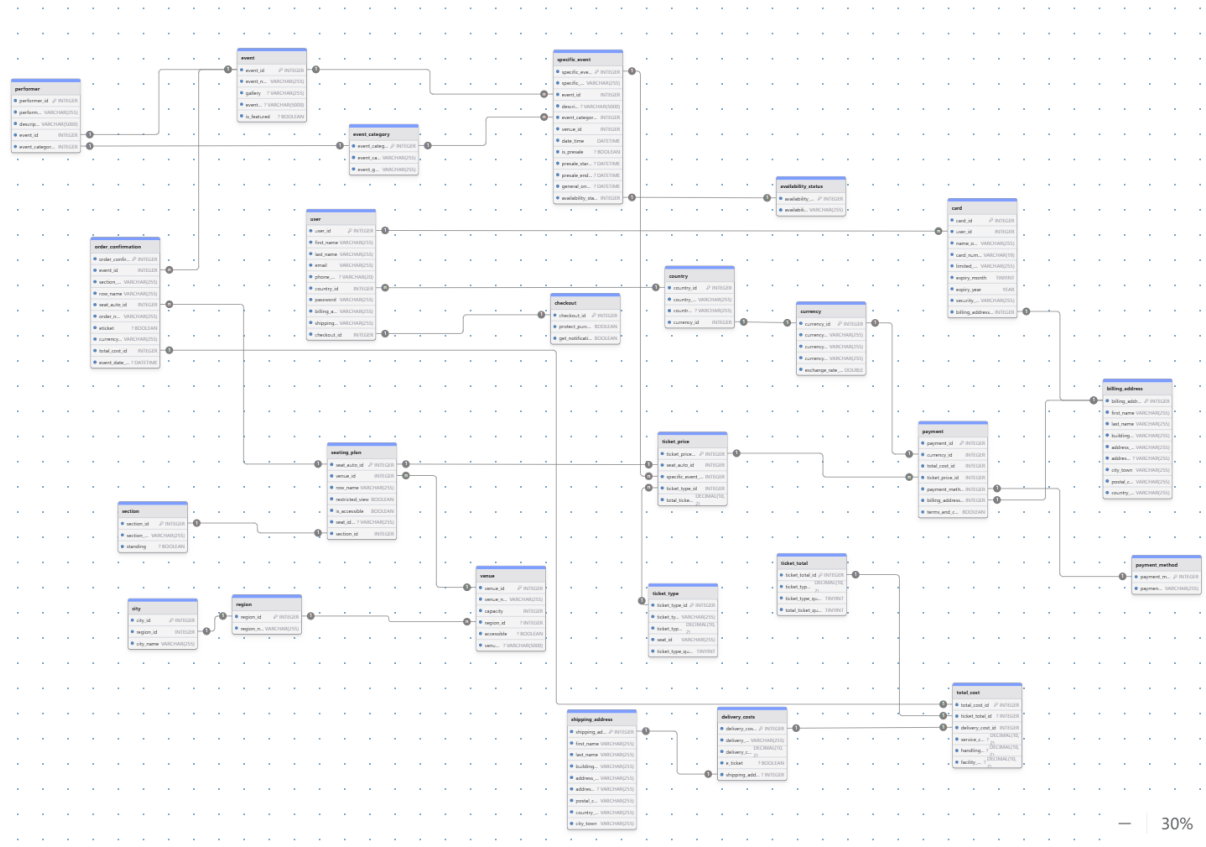


Figure 2 - Individual Schema Diagram

In order to create my individual schema diagram, I made changes to the groups diagram as I felt that some things were out of scope for the booking process and that I needed new or improved tables in order to show areas of the booking process I felt were missing. I also needed to work on ensuring the relationships and cardinality were all correct and that my diagram followed the rules of normalisation. I will cover these changes and why I made such changes in this section.

I felt it necessary to take time composing my schema diagram in a way which correctly followed the rules of normalisation. Data normalisation was developed by Codd, the creator of the Relational Database Model. He created various 'normal forms' to organise databases with minimal dependency, data redundancy and ensuring that data had integrity and consistency. When a table is in First normal form (1NF) all its values within its columns are atomic, each value has a single datatype, and each record is unique. I ensured that all my tables

complied with 1NF. Take my newly added *billing_address* table for example. Firstly, as a group we didn't have any tables for billing address or shipping address, multiple addresses values would have been within the one user address cell, therefore not complying with 1NF. In adding a *billing_address* table I kept each column holding only single atomic string values, with none of the columns having composite values, with multiple data entries in the one field. I set *billing_address_id* as the primary key, this ensured that each row could be uniquely identified. I also kept my data type choices consistent, using VARCHAR for columns that needed text and choosing INT for my primary key, ensuring that it was unique. I stuck to these rules throughout the rest of my tables, ensuring that they all followed 1NF.

For my database to comply to the rules of second normal form (2NF) I had to make sure that all the non-key attributes were fully functionally dependent and not only partially dependent on their primary key. To do this, I had to break down larger tables into separate smaller tables. One example of this can be seen in my individual schema design as I chose to take preferences and notifications out of the user table and create a separate *checkout* table. I added columns of BOOLEAN data types as the user can only click to agree, or not click to disagree with such options. Preferences and notifications originally had partial dependency; they were not dependent on the *user_id* alone as they are only relevant during the checkout process. This would have led to data redundancy as if the user updated their preferences for separate bookings, these would end up as duplicate data. Also, when a user hasn't checked out yet, these columns would have null data. By creating a separate checkout table I ensured that these columns had full functional dependency on the primary key - *checkout_id*, meaning that the user's preferences and choice of notifications would only exist when they selected these at checkout. Storing this information only when it is relevant ensures I kept data integrity, updating such preferences would now not affect any of the user specific information and ensure each table now had correct 2NF.

Furthermore, I ensured that my database followed all the rules of third normal form (3NF). Tables appear in 3NF when they are already in 2NF, and no non-key columns depend on other non-key columns i.e. having transitive dependency. When looking at the groups schema diagram I noticed that *section* was a column in the *row* table. This would mean that for every row entered, the section would have to be repeated for each row in that section. If the section name where to change, the section name for every row in that section would also have to be updated. This would lead to data redundancy and thus not comply to the rules of 2NF and 3NF. I thus decided to create a separate *section* table with columns *section_name* and *standing*, with BOOLEAN datatype to encompass if the section was standing or not. In doing so this meant that each section would only have to be defined once, eliminating the chance of data redundancy. This also meant that the row table no longer had transitive dependencies for section and that I maintained data integrity as any updates to section only need to happen within the *section* table itself rather than across multiple rows in the *row* table.

When looking at the groups diagram, I noticed that there was an issue with the *booking* and *ticket_price* tables. The way the relationships were initially set up give the impression that one ticket was one payment and one booking, meaning the user couldn't cumulate a price for multiple tickets before paying, so would therefore have to repeat the process one ticket at a time. Also, the process of physically booking a ticket and the user getting their booking confirmation was all merged together in the one booking table. In order to comply to the rules of normalisation and to better show the process of Ticketmaster where the user can select multiple tickets of varying prices, book and pay for these altogether then receive order confirmation of their booking I had to update the booking table and create new tables. I added in *ticket_total* and *total_costs* tables, connected by a one-to-relationship on the *ticket_total_id* column. Within the *ticket_total* table there are columns *ticket_type_price, ticket_type_quantity* and *total_ticket_quantity*. These symbolise when the user is selecting tickets, they can select different quantities of different ticket types that are priced separately e.g. 2 student tickets and 2 full price tickets, with 4 then being total ticket quantity. This then leads to the *total_cost* table where Ticketmaster would then add on the price of the extra fees, *service_charge, handling_fee, facility_fee*, before the user knows the full total price of everything they must pay for. I also updated the *booking* table to an *order_confirmation* table. This then depicts the full scope of what the user sees when they get sent their order confirmation after they have fully finished purchasing their tickets.

Another design decision I made was changing the *event_type* table to an *event* table and an *event_category* table in which I added columns *event_category_name* and *event_genre*. I then linked both the *event* and *event_category* tables to the *performer* and *specific_event* tables. This was because on Ticketmaster there is a section of different categories such as 'Music', 'Sport' etc. and within each category there is a subcategory such as 'All Concerts' or 'Ice Shows' (Appendix II) . In order to add specific events to my database that could be resulted by running a query to search for events in this way, I felt it was important to include these tables. Subdividing these tables out also continued to ensure that my database continued to comply to the rules of 3NF.

Through further inspection of Ticketmaster, I decided that I needed to add a *shipping_address* and a *billing_address* table into my database design. This is because this creates realistic modelling, mimicking real world scenarios where customers may want their tickets to be sent to their home but then use their office address for their billing information. Storing these addresses separately ensured my database had scalability as some customers may make multiple purchases and have their billing and shipping addresses as the same thing, therefore having them in separate tables makes it easier to scale and manage.

Another new table I added was currency, storing the *currency_code, currency_name, currency_symbol* and *exchange_rate_GBP.* I felt this was important to include as Ticketmaster sells tickets in many different countries, with users seeing the prices of the tickets in their own currency. Storing this kind of information in my database thus would make it able to handle such multi-currency transactions. Also, business practices for websites such as Ticketmaster would have to display the correct currency for customer receipts. My database then aligns more closely with Ticketmaster in this way and would also align with these standards over different regions.

When updating the schema diagram myself I noticed that the group had firstly set column *seat_id* to be a primary key. A characteristic of primary keys is that it is best practice for them to remain immutable and have stability. If foreign keys are to change then so does any relationships that have come from them, making it more difficult to retain referential integrity. This can also lead to errors and potential performance issues within the database. The group had originally chosen this column as the best choice to create the relationships between *seat, booking* and *ticket_price.* However, this ID could change frequently. For each event, Ticketmaster shows a seating plan that allows the user to see where their seat will be in relation to the exact floor plan for that specific event. For each event this seating plan changes, even within the same venue. For example, for one event in the SSE Arena, a ground floor seat may exist for one concert but not for a Belfast Giants game when the floor is an ice rink. For one event a seat could be one price whereas the exact same seat could be cheaper/ more expensive for another event. (Appendix III) Given the many different seating plans that come with the large amount of events Ticketmaster has to offer I subsequently thought it would be best to add a specific *seating_plan* table with surrogate key *seat_auto_id* to be the autoincremented primary key as this could remain stable. Within the *seating_plan* table I added columns such as *restricted_view* and *is_accesible* with BOOLEAN datatypes to demonstrate if the seats selected were or were not accessible or had a restricted view, as some seats would do in real world events. I also changed the relationships so that *seat_auto_id* could have a one-to-one relationship with the *ticket_price* table and a one-to-many relationship with the new *order_confirmation* table. I chose to then keep *seat_id* as a regular column so then when it changes, per seating plan, there will be no negative impact to the relationships created between the other tables. The user would still then be able to see the correct price per seat across all the different seating plans. This then helped to keep referential integrity and ensured the foreign key relationships remained stable.

When fixing the groups schema diagram I noticed that we had created a relationship between *specific_event* and *ticket_price* but no relationship that showed how the venue was important when buying a ticket. Doing further investigation of booking a ticket on Ticketmaster I noticed that after you selected for example, a concert that you wanted to see, you were then brought to the page that shows all the concerts in various venues, across various cities and regions. To book a ticket, you then must select the specific event shown at a venue to be able to see the availability of the tickets for that specific venue. In saying that, one venue can hold many events, and one event can be shown at many venues. This would mean that I would have to create a relationship with many-to-many cardinality. This is best shown by using the *specific_event* table as a bridge table in between, with the specific event table serving as the intermediary between the event and the venue. The user can then pick the specific event shown at a particular venue. This would then successfully manage the many-to-many relationship while also making it easier to run queries that will result in showing a specific venue or event.

Another change that I made was changing *expiry_date* to *expiry_month* – date type TINYINT 2 and *expiry_year*- data type YEAR. This is because when the user must enter their card details when they get to the payment section, they must enter their card expiration date in a MM/YY format, which is a standard format for card details. Having these as separate columns therefore aligns with these card standards and allows for simpler input validation, matching a format that is customary to users who have made online payments before. Choice of TINYINT and YEAR rather than keeping them VARCHAR was made as they use minimal space, meaning they are capable of storing these small values that fall within the correct ranges. Use of VARCHAR could have also led to making comparisons within the database less efficient. The new format chosen is also good for security and compliance reasons. It is important to follow PCI Compliance when handling payment information. Storing the month and year rather than a full date format keeps the minimum amount of data required, minimising the risk of storing too much identifiable payment information within the one column.

When looking at the card section I decided to also change the data type of *card_number* from INT to VARCHAR. Whilst at first the group thought as card numbers were typically 16-digit numbers, INT would be an appropriate data type, I done some further research to check how card numbers were actually stored. VARCHAR was chosen as the best data type for storing card numbers as it allows the card numbers to be saved as variable- length strings. Card numbers are usually long so could exceed the maximum value that could be stored by data type INT. Card numbers can also have 'leading zeros' which would be dropped and lost if storing them as a numeric value, therefore showing an incorrect representation of the card number. Using VARCHAR thus ensured that all the users card numbers would be preserved in their correct format.

Many columns in my database make use of the data type VARCHAR. This is a datatype used in databases to store strings which vary in length, without wasting any space. In some cases, it was important to set VARCHAR to a higher upper limit. When exploring Ticketmaster I noticed that for each event there was an 'About' section. (Appendix IV). These descriptions varied in length depending on the amount of information that was included. I decided it would therefore be beneficial to add an *information* column in the *venue* table to reflect this. I then set description to VARCHAR, length 5000. Some group members thought a TEXT data type would be better for description like columns. However, TINYTEXT is a fixed-length string type, meaning that it can only store up to 255 characters and a TEXT data type would give more capacity than needed: up to 65,535 characters. Through researching the lengths of different descriptions on Ticketmaster, I discovered that the length would vary in size but would rarely exceed 5000 characters. VARCHAR was then chosen as the most appropriate data type as it is a variable-length data type, so it only stores data for the number of bytes needed, meaning that when the description sizes are smaller, VARCHAR would be more space efficient than using TEXT. It was also chosen as a better fit as unlike TEXT, VARCHAR can provide a clear upper limit of 5000, avoiding having excess capacity, which would be wasteful, as the data length would stay between this set range.

That being said, there are many occasions within my database where I felt that a VARCHAR upper limit of less than 255 was necessary, as I could gage what the upper limits would be. An example of this are columns in the *user* table, *phone_number* and *post_code*, set to VARCHAR 20 and VARCHAR 10 respectively. This was done as I knew that phone numbers and various postcodes have a set length. It was therefore better practice to set the VARCHAR lengths to reflect the length of the expected data. This avoided over-allocating space and ensured that it is clearer to see the size of data that is expected. Another benefit being that that it reduces the likelihood of storing invalid data.

Finally, I removed the *is_sold* column from the *seat* table and added an *availability_status* table with a one-to-one relationship with an *availability_status_id* column in the *specific_event* table. This is because, on Ticketmaster when the user is looking at the different events, before clicking into the specific venue they will be shown 'low/limited availability' if tickets are running out. Once the user clicks into the venue, if there are no tickets left it will show 'no availability'. I felt this was an important to show as the ability to book a ticket is contingent on the obtainability of tickets, if no tickets can be found then the user cannot go any further in the booking process and will have to thus begin a brand new search for tickets that are available at a different venue and or time. Also, this prevents more than one user purchasing the exact same ticket. I added in these columns into my tables as I wanted to have queries that encompassed how this process works. *availability_status_name* will show available if there are enough tickets left, limited if the number of tickets drops below a certain threshold, no availability when ticket levels are at 0, otherwise when ticket levels are higher than 0, carry on with the booking. I could then run a query to update *availability_status* after each booking, if there are 0 tickets left, the status should be sold, if a certain ticket is sold, it should not be available for purchase by another user. With all these changes made I was then able to export the SQL from DrawDB into PhPMyAdmin and begin implementing my database itself.

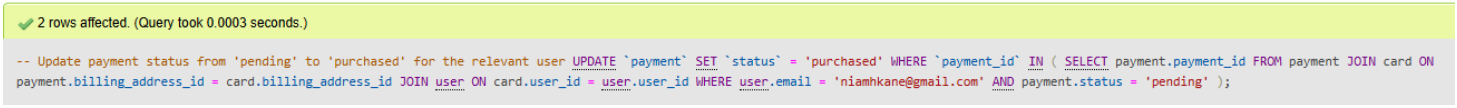
Individual database development

During the database development stage, I began by exporting my SQL into PhPMyAdmin, this created my tables, their columns and foreign key relationships that I had previously created. It became quickly apparent that there were many problems that came from setting up my database this way rather than building it from scratch. One being, that as the relationships were already formed in a very intrinsic way with every table relying on another one or more tables, inserting data into each column had to be done in a very specific order. To achieve this, I had to make sure to insert data into parent tables first. These tables were the ones that contained primary keys that referred to other tables via foreign keys. Secondly, I inserted data into child tables, those with the foreign keys. The child tables reference data that was already input into the parent tables, if I accidentally inserted data into a child table first, my SQL would throw a foreign key constraint error. This process thus needed extra time and attention to insert data in the correct way. The first steps of creating my database then consisted of me getting this order right so that I could ensure my database had referential integrity and avoided further foreign key constraint violations.

When inserting data into my database I noticed that the columns I had originally set to BOOLEAN were changed to TINYINT(1). After further researched I found out that this is because MySQL doesn't have a BOOLEAN datatype, BOOLEAN is an alias for TINYINT(1). An example of where this can be seen is in my *checkout* table. On Ticketmaster, the user must click a check box if they want to receive marketing information, agree to terms and conditions and or pay extra to protect their purchase from unforeseen circumstances such as adverse weather conditions. TINYINT(1) thus stands for 1 being true and 0 being false, with the implementation of these decisions being a check box being left up to this happening on a web design rather than database level.

Once I had populated my database with example data I began to run queries to see if I could do the same things as a user on the Ticketmaster website, such as creating an account, logging in, searching for events through different ways (searching by location, artist, venue, region, city, date, category, availability, price), purchasing different types of tickets like student or full price tickets, purchasing the ticket so that its availability changed to sold, and payment pending status changed to purchased and also viewing my order confirmation for tickets that I had purchased. Examples of these queries can be found in the appendix of my report. (Appendix V to XIII)

Through trial and error of running these queries from a user's perspective I was able to notice columns that were missing from my database. One example is the *status* column which I then inserted into the *order_confirmation* and *payment* tables. Firstly, adding status columns allowed me to query the total tickets sold. Searching where status = 'purchased' I could see the total amount of *total_tickets_sold* all added together. (Appendix XIV) Furthermore, when a user is purchasing tickets on Ticketmaster, they have a limited time to pay for their tickets, this time changes, with less time being given to tickets that are of higher demand. During this time the status of the ticket being purchased would be pending and once the money goes through it will be purchased. I thought it would be necessary to add these columns to then perform a query to display a user buying a ticket and the status then changing from 'pending' to 'purchased'. To perform this query, I used JOIN rather than LEFT JOIN, this is because LEFT JOIN results in showing all rows from the left table in the result, even when they do not match anything in the right table. However, JOIN results in specific matches from the tables involved, which is what I wanted for this query. Running this query allowed my database to better implement and demonstrate the business logic of buying a ticket and improved my query efficiency, through being able to directly query the status column itself rather than calculating derived statuses dynamically using many complex joins.



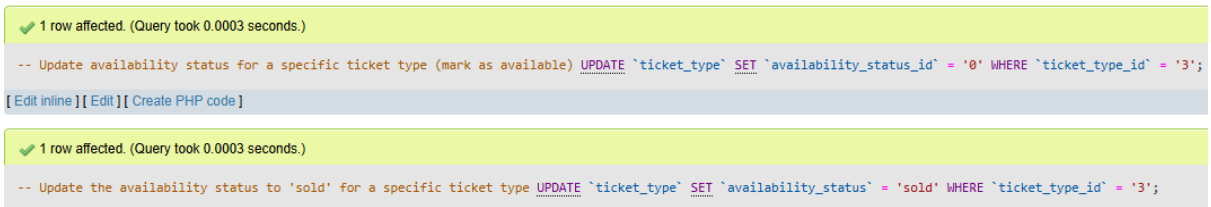
```
2 rows affected. (Query took 0.0003 seconds.)

-- Update payment status from 'pending' to 'purchased' for the relevant user UPDATE `payment` SET `status` = 'purchased' WHERE `payment_id` IN ( SELECT payment.payment_id FROM payment JOIN card ON
payment.billing_address_id = card.billing_address_id JOIN user ON card.user_id = user.user_id WHERE user.email = 'niamhkane@gmail.com' AND payment.status = 'pending' );
```

	payment_id	currency_id	total_cost_id	ticket_price_id	payment_method_id	billing_address_id	terms_and_conditions	status
Edit Copy Delete	1	1	1	0	1	1	1	pending
Edit Copy Delete	2	2	2	2	2	2	1	pending

	payment_id	currency_id	total_cost_id	ticket_price_id	payment_method_id	billing_address_id	terms_and_conditions	status
Edit Copy Delete	1	1	1	0	1	1	1	purchased
Edit Copy Delete	2	2	2	2	2	2	1	pending

Similarly to status, I thought it was important to capture the availability element of booking a ticket. When on Ticketmaster you can see the availability status of 'low availability' and 'limited availability' appear on specific events if the number of tickets left is running low. When these tickets fully sell out, they are then no longer available to be purchased by the user. To utilize this information in my database I created a query which shows this process, changing the availability of a ticket from 'available' to 'sold' when it is purchased. This replicates the real inventory management that Ticketmaster would carry out to prevent overselling and ensures my database has data consistency and operates as closely to Ticketmaster as possible.



```
1 row affected. (Query took 0.0003 seconds.)

-- Update availability status for a specific ticket type (mark as available) UPDATE `ticket_type` SET `availability_status_id` = '0' WHERE `ticket_type_id` = '3';

[ Edit inline ] [ Edit ] [ Create PHP code ]
```

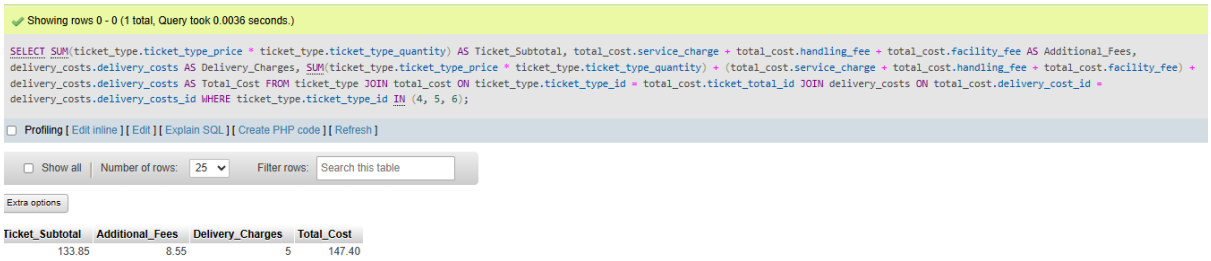
```
1 row affected. (Query took 0.0003 seconds.)

-- Update the availability status to 'sold' for a specific ticket type UPDATE `ticket_type` SET `availability_status` = 'sold' WHERE `ticket_type_id` = '3';
```

ticket_type_id	ticket_type_name	ticket_type_price	seat_id	ticket_type_quantity	availability_status_id	availability_status	specific_event_id
1	Full Price Ticket	20.50	Seat 77	1	1	Available	1
2	Possible Restricted View	38.95	Seat 15	1	2	Low Availability	2
3	Full Price Ticket	39.20	Seat 1	1	1	Available	3
4	Verified Resale Tickets	30.00	Seat 3	1	4	No Availability	4
5	Full Price Ticket	50.50	Seat 63	1	5	Presale	5
6	Full Price Ticket	53.35	Standing	1	6	Available	6
7	Student	45.00	Standing	1	2	Limitied availability	7

ticket_type_id	ticket_type_name	ticket_type_price	seat_id	ticket_type_quantity	availability_status_id	availability_status	specific_event_id
1	Full Price Ticket	20.50	Seat 77	1	1	Available	1
2	Possible Restricted View	38.95	Seat 15	1	2	Low Availability	2
3	Full Price Ticket	39.20	Seat 1	1	0	sold	3

When fixing my database design, I thought it would be important to remove some of the stored derived attributes. I originally included *total_ticket_type_price* to store the total of each type of ticket separately. This was removed in favour of just calculating an overall total dynamically. Storing derived attributes may have led to redundant data, also, if the elements of the attribute such as *ticket_type_price* were updated then the stored derived attribute would have become inaccurate unless it was recalculated, meaning that the data may also have become inconsistent. In terms of calculating totals dynamically I also added a *e_ticket* column to the *delivery_costs* table with TINYINT(1) and inserted data for 'Speedy Delivery' as a type of delivery. If the user wants an e-ticket then the delivery fee will be free, keeping this cost as 0 when calculating the *total_cost* of everything for the user, or charging them £5 if they chose speedy delivery. Furthermore, such derived attributes would have violated the rules of 3NF as they wouldn't have relied solely on the primary key but on the other attributes as well. Removing these was therefore important to ensure that I followed best practice, reduced redundancy and minimised the risk of having any data inconsistencies.



To further make my database reflect the operation of ticket master I thought it was important to encrypt the *password* column in the *user* table. Firstly, I inserted plain text passwords for every user and then I created a new column *salt* to store the salt. I used MySQL's UUID() to generate unique random salts for each of the users in the *user* table, using the query **UPDATE user SET salt = UUID().** Next I used the query **password = SHA2(CONCAT(password, salt), 256);.** This attached a salt to the original plain text passwords and using SHA-256 to hash the combined string, therefore replacing all the original passwords with new hashed versions. This is important to do because this then stored the passwords in a non-reversible format, meaning that its now impossible to get back to seeing the original passwords. This also complies to industry standards that Ticketmaster must follow. According to the 'Privacy Policy (2024)' section on the Ticketmaster website, they "anonymize the information…remove any information linking the sales back to individuals who made the purchases". Using similar measures to encrypt information in my own database thus further aligns my work with real-life practices that Ticketmaster follow.

user_id	first_name	last_name	email	phone_number	country_id	salt	password	billing_address	shipping_address	checkout_id
1	Niamh	Kane	niamhkane@gmail.com	07765048342 5	1	f897a117-a730-11ef-9390-c8348e758634	9d3128a07d48246b639bdc2699ed6efe120b7421e942cc6f0b...	59 Riverdale Gardens	59 Riverdale Gardens	1
2	Caoimhe	Fraser	cfraser@hotmail.com	07740623171	1	f897b5e1-a730-11ef-9390-c8348e758634	72ea22e8e2539e3db6c9744e7917529767fefc7717706dcb99...	7 Denewood Park	7 Denewood Park	2
3	Janet	McKeown	jmck@outlook.com	07626272781	1	f897b6c1-a730-11ef-9390-c8348e758634	e54d9ef50f9877ec07d7f10e6532776d96c1a8329a4a8c1e64...	6 Chopin Street	6 Chopin Street	3
4	David	Kane	daviddavid@hotmail.com	07750438171	2	f897b730-a730-11ef-9390-c8348e758634	ffe1ade35c04d30a459d2f80e41a1b4ea3ba1ca838b426ebff...	16 St James Drive	16 St James Drive	4
5	Julia	Smith	jsmith1995@hotmail.com	07654328765	1	f897b78d-a730-11ef-9390-c8348e758634	a0428b96b614665dfc5d58a470a9f7cd40fd2cb071060ba288...	9 Fountain Lane	9 Fountain Lane	5
6	Izabela	Lysakowska	ilys@hotmail.com	07760987651 2	2	f897b7e9-a730-11ef-9390-c8348e758634	e212b9af8ece5d59baf4ce247b11e5c3767cf67e4ff3e9bb46...	288 Evergreen Crescent	288 Evergreen Crescent	6

On Ticketmaster the user must log into their account to make a purchase. Another reason I used hash and salt on the passwords was to display a query which represented the user logging into their account. To do this, I matched the original password against the newly hashed version of the password within the user table, using the below query. Inserting the original plain text password for *user_id* 1, 'p@ssw0rd1!'. This amalgamated the user's password with the salt, as well as hashed the merged password and salt and compared the hash with the original stored password. The query then returned *user_id* 1 as the password was correct, thus accurately replicating the process of a user putting in their password and logging into their Ticketmaster account.

Another design decision I made when creating my database was removing the *security_code* column from the *payment* table. At first, I wanted to include all the details that the user types into Ticketmaster when they are paying for their tickets. However, through further research I found out that like other large websites, Ticketmaster only stores the last 4 digits of the user's card and when paying for tickets the user must manually enter their CVV number, as this is not stored and kept by Ticketmaster. PCI DSS guidelines clearly forbid storing security codes like CVC and CVV codes after they have been used in card transactions. Removing this column in my database therefore ensures it does not violate any of these real-world guidelines. Furthermore, I decided to use AES_ENCRYPT(), to encrypt data stored in the *card_number* column in my *card* table, but keeping the data in the *limited_card_number* visible, as these represent the last 4 digits of each users card number. Another reason I used the Advanced Encryption Standard was because this does comply to PCI DSS guidelines. According to a Basis Theory blog which outlines the PCI DSS Requirement 3: Protect Stored Account Data, Requirement 3.4 stipulates that the ability to copy cardholder data is restricted. Using AES thus complies with this requirement as it is an encryption algorithm which is industry standard. Making these design changes therefore further ensured that my database models the structure and functionality of Ticketmaster. All these individual design designs thus created my final database design (Appendix XV)

Run SQL query/queries on table ticketmaster.card:

```
1
2  SET @secretPassword = 'thisIsTooEasyToGuess';
3
4  UPDATE card
5  SET card_number = AES_ENCRYPT(card_number, @secretPassword)
6  WHERE user_id IN (2, 3, 4, 5, 6);
7
8  SELECT card_id, user_id, name_on_card,
9         card_number,
10        limited_card_number,
11        expiry_month,
12        expiry_year,
13        billing_address_id
14 FROM card
```

card_id	user_id	name_on_card	card_number	limited_card_number	expiry_month	expiry_year	billing_address_id
1	1	Niamh C Kane	□□r?;$??□7?h???^6?}	9123	12	27	1
2	2	C Fraser	?}"□?Qu?6?□:??_??T?	4321	10	26	2
3	3	J B McKeown	5Po?S?_쉽ㅂ??????)?	5437	12	28	3
4	4	David Kane	)D?#□-??□M??`q???"	8765	9	25	4
5	5	Julia E Smith	??_?:4!F7?B□??j□???	2467	3	27	5
6	6	I Lysakowska	@>?D??□???3Cs ??]9L	8675	2	26	6

Design Improvements

If I was given further time and resources to implement other areas of the Ticketmaster website, there are some design improvements that I could have made to the database. In this section I will discuss some of these additional improvements I would have made.

When doing further research into the Ticketmaster website I came across the 'Festivals' genre tab. Within the various options for each festival, it was apparent that there were many other ticket options available than those on sale for other events. Take the festival 'Cream Fields' for example. This festival offers a huge variety of ticket and camping options. (Appendix XVI) These include individual ticket and camping options for selected days, or multiple day camping tickets. They also offer premium ticket options including gold, silver and bronze ticket options, various luxury camping options, as well as the ability to book tickets for coaches, shuttle buses and car parking. If I had more time and additional resources to build my database, adding in these very detailed tickets and service options would require a comprehensive database that could handle these complex options. This would include expanding my *ticket_type* table to include many new ticket options, creating a separate camping table for all the camping packages and their attributes and creating tables for services such as parking and buses each with their own columns for pricing and availability. I would then need to create one-to-many relationships with the new ticket types and many-to-many relationships between the ticket types and different services as the user may want to book a combination of options together. However, given this would be a lot of tables just to cover one single festival, I felt it best to leave this out of my current database. Focusing on the elements I have included allowed me to reduce complexity, keep simplicity and have faster retrieval of queries. That being said, if I had more time and resources to expand on my database, I could see how adding in new elements for individual events would provide a more complete representation of Ticketmaster, further demonstrating its substantial scope of ticketing options.

Another thing I felt was out of scope for my database which I would improve upon in the future is adding tables to cover all the different images used in Ticketmaster. I have added a *gallery* column to my *event* table to show that the events have a gallery section of images of the event that the user can look through. However, it is likely that a database to cover all the images in Ticketmaster would include domain-specific tables for images related to specific areas, e.g. a separate events image table, artist image table, category image table etc, a centralised table to store global images like the website logo and regional images which would appear depending on the location the website would be viewed in. These would then use a tiered system to manage all the images, connecting the tables via junction tables and storing the images as URLs. Although I would like to implement

all the elements of Ticketmaster, like its use of many images, I thought it would be best to keep a similar design, as this is more practical for my use case.

Another design improvement that I was not able to demonstrate in my database was BCrypt. This cryptographic hash function is designed to hash and salt passwords for safe storing, before adding them to a database. BCrypt adds a salt to passwords before hashing them, making sure that one or more of the same passwords gets different hashes. This procedure requires logic that goes beyond the scope of MySQL. Although MySQL is good at manipulating data, and did allow me to use its built-in function of SHA2() for hashing users passwords, it is not capable of accomplishing such advanced cryptographic operations. If I were able to implement BCrypt this would be a beneficial improvement as "It is one of the most popular and powerful algorithms which is quite successful in restraining the password hacking and other unwanted attacks in the system." (Ertaul, Kaur and Gudise, 2016, p. 3). Keeping my data safe would be very important given the amount of data that Ticketmaster itself has had compromised. According to a recent BBC article, 'ShinyHunters', a hacking group, stole the personal details of 560 million customers and demanded a £400,000 ransom to stop them selling the data to other parties on the dark web. This hack being one of the biggest in history in terms of the amount of people effected globally and the extent of the amount of data that was stolen. Protecting my data would therefore be a beneficial improvement to my database.

Furthermore, if I had further resources, I would ensure that the data in my database further followed the Payment Card Industry's Data Security Standard. According to the book PCI compliance by Branden R. Williams and James Adamson (2022), "If you accept, process, transmit, or store payment card… you must comply with this lengthy standard". Although I was able to use the built in encryption function on MySQL, AES_ENCRYPT() to encrypt the card numbers, this is only a basic level of encryption and does not reach the standard of following PCI guidelines that large websites such as Ticketmaster would have to adhere to. Ticketmaster uses a trusted PCI compliant third party to handle customers payment data, limiting their direct exposure to data including card details. They verify user's names, addresses, phone numbers and the last 4 digits of the card used for each booking to protect users' information. A positive of Ticketmaster following PCI compliance is, that it can have global reach, operating internationally, selling tickets across many other countries where PCI compliance is mandatory for processing card details. However, using third-party payment processors puts dependency on these other parties, therefore issues with these external systems may have negative impacts on Ticketmaster's own customer experience. That being said, if I was able to, it would be a good improvement for my database to further adhere to such compliance rules.

Another improvement that I would make in the future would be, including SAVEPOINT and partial ROLLBACK within my database. When a user is trying to buy a ticket on Ticketmaster, a SAVEPOINT would be in place during the transaction to handle entry of incorrect card details. "A partial rollback of a transaction restores the state of the transaction and the database to the state in which they have been right after the savepoint to be rolled back has been established" (Kim *et al.*, 1999, p. 303). This would comply with ACID properties. Atomicity: if the payment process fails, the system would partially rollback to the SAVEPOINT, only undoing the payment steps, so that the user still has their selected tickets. Consistency: the tickets are either available or not available when they are purchased. Isolation: this would operate using a read committed isolation level, preventing dirty reads, like another user seeing the exact same tickets as available, during the transaction process. Even if the payment fails, the transaction won't be committed yet, so the tickets stay reserved and not visible to other users. Durability: when the payment is successful, this is permanently recorded in the database. This would be a great asset to include as this would simulate a robust checkout process which is critical in a high-traffic system such as Ticketmaster.

Conclusion

Taking my aforementioned points into consideration, my database operates successfully for storing various information that is included in the Ticketmaster website, in most regard to the user booking a ticket for an event. I have included and overview of my initial and final schema diagrams, describing the design assumptions I have made. My database makes use of primary keys and foreign key constraints, with the tables connected through foreign key relationships. I have complied to the rules of normalisation, specifically third normal form, reducing data redundancy and ensuring data consistency. I made key design decisions in terms of data types, relationships and cardinality, based on following the format and design of Ticketmaster. I have populated my database with real world data and used SQL queries to demonstrate the functionality of my database. My database is scalable and effective, but through further future improvements, such as further obeying PCI rules and adding in further tables, I could enhance data security and further the capability of my database, making it closer to the real-life design of Ticketmaster. However, for the use case of this database project, my database is a successfully designed and implemented database system which reflects the operation of chosen aspects of the ticketing system, particularly, but not limited to, a user being able to search, pay for, and book a ticket.

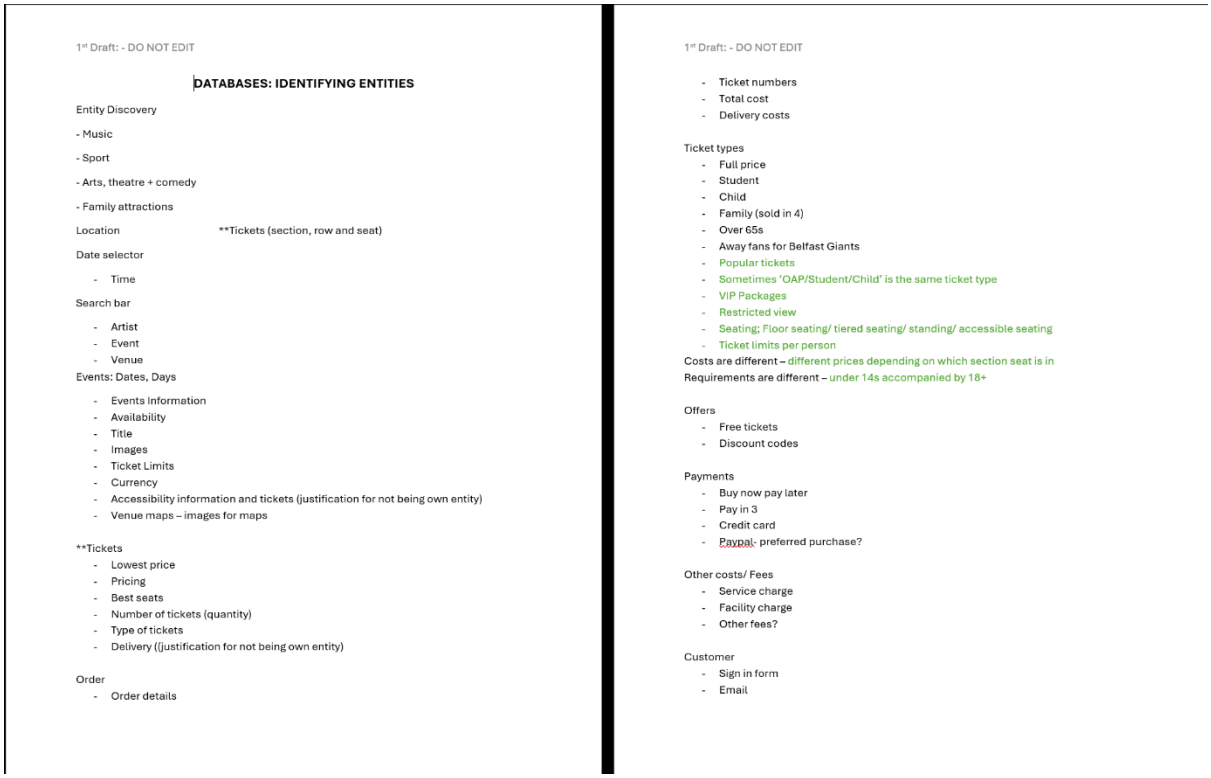
Bibliography

- APWilliams, B. and Adamson, J., 2022. *PCI Compliance: Understand and implement effective PCI data security standard compliance*. CRC Press.

- Basis Theory (2023) *PCI DSS requirement 3: Protect stored account data, Blog*. Available at: https://blog.basistheory.com/pci-dss-requirement (Accessed: 20 November 2024)
- Ertaul, L., Kaur, M. and Gudise, V.A.K.R., 2016. Implementation and performance analysis of pbkdf2, bcrypt, scrypt algorithms. In *Proceedings of the international conference on wireless networks (ICWN)* (p. 66). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp).
- Kim, S.H., Jung, M.S., Park, J.H. and Park, Y.C. (1999). A design and implementation of savepoints and partial rollbacks considering transaction isolation levels of SQL2. *Proceedings of the 6th International Conference on Advanced Systems for Advanced Applications*, Hsinchu, Taiwan, pp. 303-312. doi: 10.1109/DASFAA.1999.765764.

- MySQL (2024) *MySQL 8.4 Reference Manual :: 5.6.9 using auto_increment, MySQL*. Available at: https://dev.mysql.com/doc/refman/8.4/en/example-auto-increment.html (Accessed: 10 November 2024).
- Privacy Policy (2024). Available at: https://privacy.ticketmaster.co.uk/privacy-policy#looking-after-your-information (Accessed: 21 November 2024)
- Support, M. (2024) *Add or change a table's primary key in Access*, *Microsoft Support*. Available at: https://support.microsoft.com/en-us/office/add-or-change-a-table-s-primary-key-in-access (Accessed: 23 November 2024).

Appendix I – Select example of first list of entities and their attributes



1st Draft: - DO NOT EDIT

DATABASES: IDENTIFYING ENTITIES

Entity Discovery

- Music
- Sport
- Arts, theatre + comedy
- Family attractions

Location **Tickets (section, row and seat)

Date selector

- Time

Search bar

- Artist
- Event
- Venue

Events: Dates, Days

- Events Information
- Availability
- Title
- Images
- Ticket Limits
- Currency
- Accessibility information and tickets (justification for not being own entity)
- Venue maps – images for maps

**Tickets

- Lowest price
- Pricing
- Best seats
- Number of tickets (quantity)
- Type of tickets
- Delivery ((justification for not being own entity)

Order

- Order details

1st Draft: - DO NOT EDIT

- Ticket numbers
- Total cost
- Delivery costs

Ticket types

- Full price
- Student
- Child
- Family (sold in 4)
- Over 65s
- Away fans for Belfast Giants
- Popular tickets
- Sometimes 'OAP/Student/Child' is the same ticket type
- VIP Packages
- Restricted view
- Seating; Floor seating/ tiered seating/ standing/ accessible seating
- Ticket limits per person

Costs are different – different prices depending on which section seat is in

Requirements are different – under 14s accompanied by 18+

Offers

- Free tickets
- Discount codes

Payments

- Buy now pay later
- Pay in 3
- Credit card
- Paypal- preferred purchase?

Other costs/ Fees

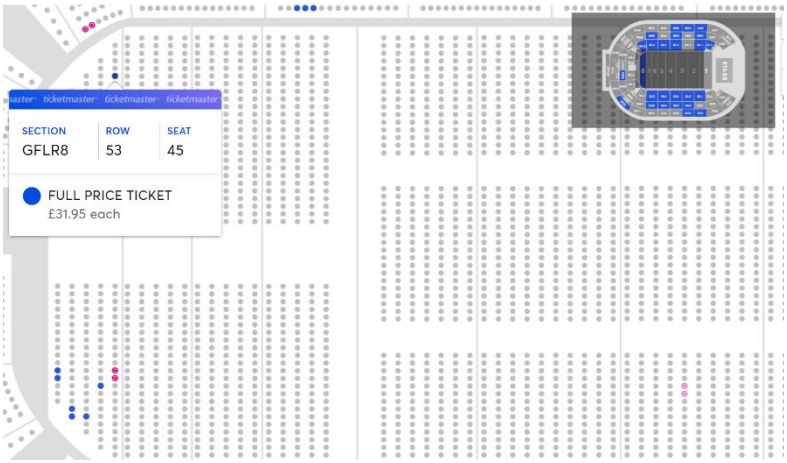
- Service charge
- Facility charge
- Other fees?

Customer

- Sign in form
- Email

Appendix II – Different categories on Ticketmaster

Appendix III – Various Ticketmaster seating plans



Appendix IV- example of an 'About' section on Ticketmaster

ABOUT

The Thirty, Rough and Dirty 2024 UK tour

Happy hardcore phenomenon Scooter is one of Germany's most successful recording artists ever, earning 23 Top 10 hits and over 80 Gold and Platinum certifications.

Formed in 1993 by H.P Baxter, Rick J. Jordan and Ferris Beuller, Scooter got their break in 1994 with their single 'Hyper Hyper', which flew off the shelves in Germany and across Europe.

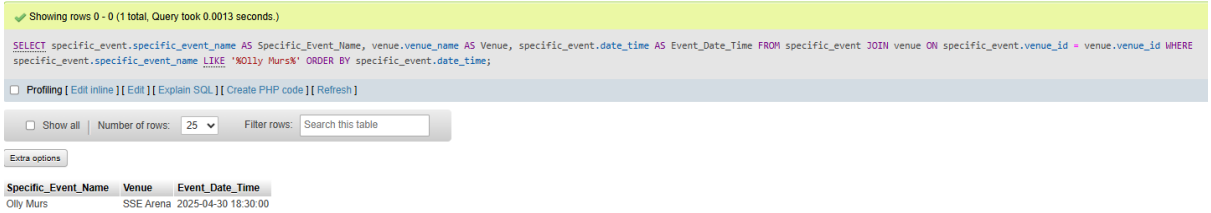
Successive singles 'Move Your Ass' and 'Friends' followed suit, charting highly on the continent, while in 1996 they had their UK breakthrough with 'Back in the UK'.

From their debut album *...and The Beat Goes On!* to later records including *Wicked!* (1996), *Sheffield* (2000), *Scooter Forever* (2017) and the forthcoming *Open Your Mind and Your Trousers*, Baxter and co. have continued to push the euphoria of dance music to its extremes that has incorporated the changing tides of the genre while always keeping their own instantly recognisable edge.

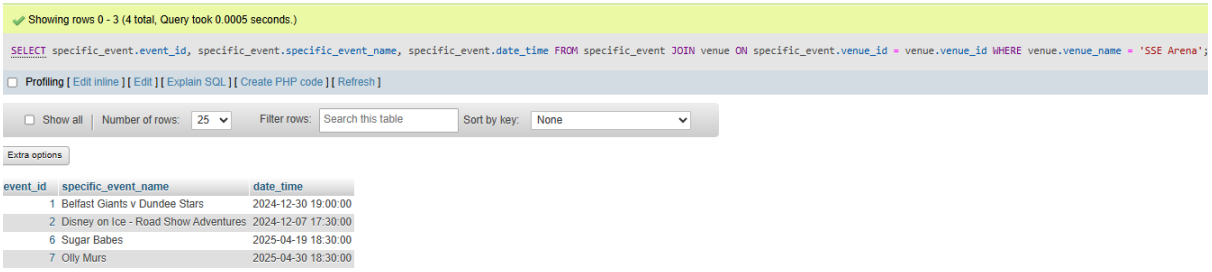
In April 2024, Scooter announced news of their 30th-anniversary tour – Thirty, Rough and Dirty – coming to venues including London's OVO Arena Wembley.

The anniversary tour news follows from the band's latest milestone which is the release of their 21st studio album – *Open Your Mind and Your Trousers*. Boasting over 30 million albums sold globally and more than 100 Gold and Platinum certifications, Scooter's latest offering is a testament to their enduring musical prowess. The album, featuring 15 tracks in their signature style, includes previously released singles such as 'Techno is Back', 'Rave & Shout', 'Waste Your Youth' and 'For Those About To Rave'.

Appendix V – Searching by specific event



Appendix VI – Searching by specific venue



Showing rows 0 - 3 (4 total, Query took 0.0005 seconds.)

```
SELECT specific_event.event_id, specific_event.specific_event_name, specific_event.date_time FROM specific_event JOIN venue ON specific_event.venue_id = venue.venue_id WHERE venue.venue_name = 'SSE Arena';
```

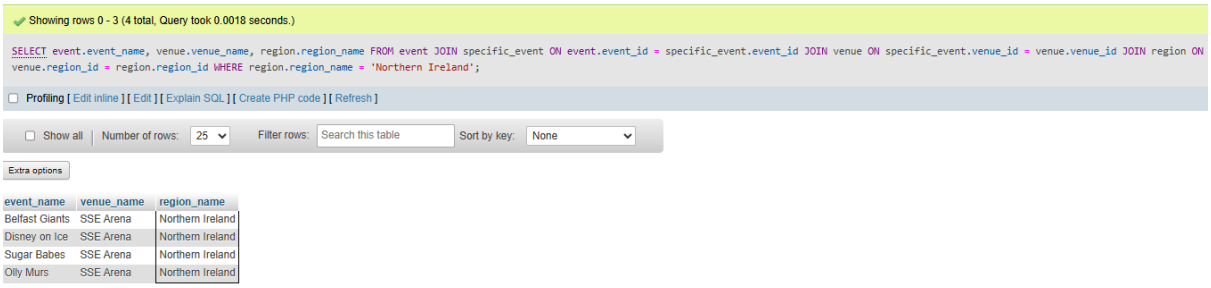
Profiling [Edit inline] [Edit] [Explain SQL] [Create PHP code] [Refresh]

Show all | Number of rows: 25 Filter rows: Search this table Sort by key: None

Extra options

event_id	specific_event_name	date_time
1	Belfast Giants v Dundee Stars	2024-12-30 19:00:00
2	Disney on Ice - Road Show Adventures	2024-12-07 17:30:00
6	Sugar Babes	2025-04-19 18:30:00
7	Olly Murs	2025-04-30 18:30:00

Appendix VII – Searching by specific region



Showing rows 0 - 3 (4 total, Query took 0.0018 seconds.)

```
SELECT event.event_name, venue.venue_name, region.region_name FROM event JOIN specific_event ON event.event_id = specific_event.event_id JOIN venue ON specific_event.venue_id = venue.venue_id JOIN region ON venue.region_id = region.region_id WHERE region.region_name = 'Northern Ireland';
```

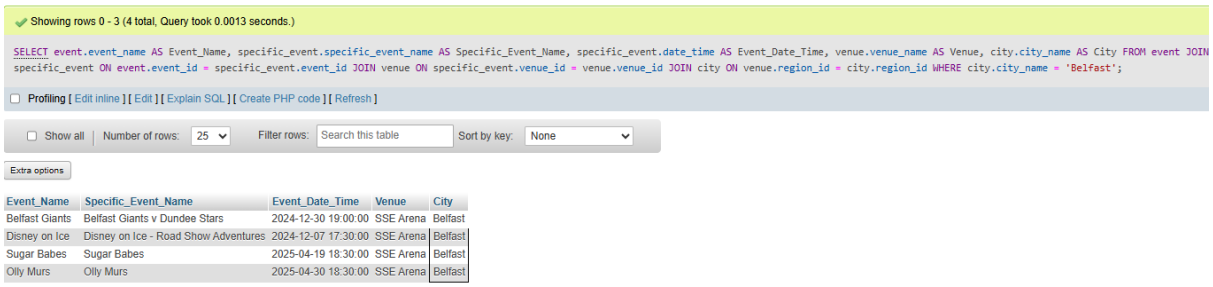
Profiling [Edit inline] [Edit] [Explain SQL] [Create PHP code] [Refresh]

Show all | Number of rows: 25 Filter rows: Search this table Sort by key: None

Extra options

event_name	venue_name	region_name
Belfast Giants	SSE Arena	Northern Ireland
Disney on Ice	SSE Arena	Northern Ireland
Sugar Babes	SSE Arena	Northern Ireland
Olly Murs	SSE Arena	Northern Ireland

Appendix VIII – Searching by specific city



Showing rows 0 - 3 (4 total, Query took 0.0013 seconds.)

```
SELECT event.event_name AS Event_Name, specific_event.specific_event_name AS Specific_Event_Name, specific_event.date_time AS Event_Date_Time, venue.venue_name AS Venue, city.city_name AS City FROM event JOIN specific_event ON event.event_id = specific_event.event_id JOIN venue ON specific_event.venue_id = venue.venue_id JOIN city ON venue.region_id = city.region_id WHERE city.city_name = 'Belfast';
```

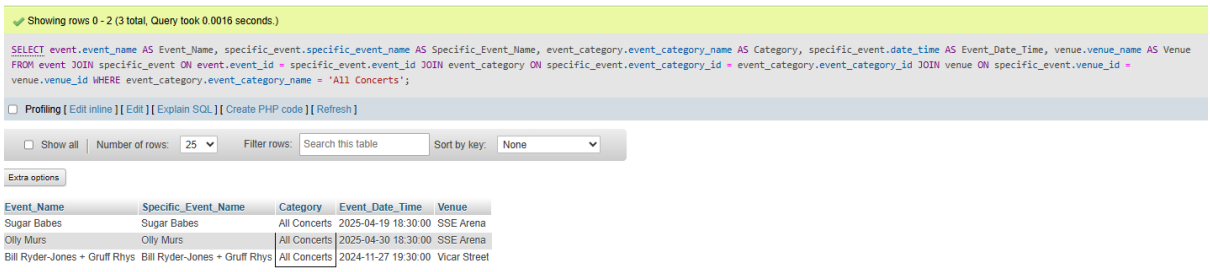
Profiling [Edit inline] [Edit] [Explain SQL] [Create PHP code] [Refresh]

Show all | Number of rows: 25 Filter rows: Search this table Sort by key: None

Extra options

Event_Name	Specific_Event_Name	Event_Date_Time	Venue	City
Belfast Giants	Belfast Giants v Dundee Stars	2024-12-30 19:00:00	SSE Arena	Belfast
Disney on Ice	Disney on Ice - Road Show Adventures	2024-12-07 17:30:00	SSE Arena	Belfast
Sugar Babes	Sugar Babes	2025-04-19 18:30:00	SSE Arena	Belfast
Olly Murs	Olly Murs	2025-04-30 18:30:00	SSE Arena	Belfast

Appendix IX – Searching by specific category



Showing rows 0 - 2 (3 total, Query took 0.0016 seconds.)

```
SELECT event.event_name AS Event_Name, specific_event.specific_event_name AS Specific_Event_Name, event_category.event_category_name AS Category, specific_event.date_time AS Event_Date_Time, venue.venue_name AS Venue FROM event JOIN specific_event ON event.event_id = specific_event.event_id JOIN event_category ON specific_event.event_category_id = event_category.event_category_id JOIN venue ON specific_event.venue_id = venue.venue_id WHERE event_category.event_category_name = 'All Concerts';
```

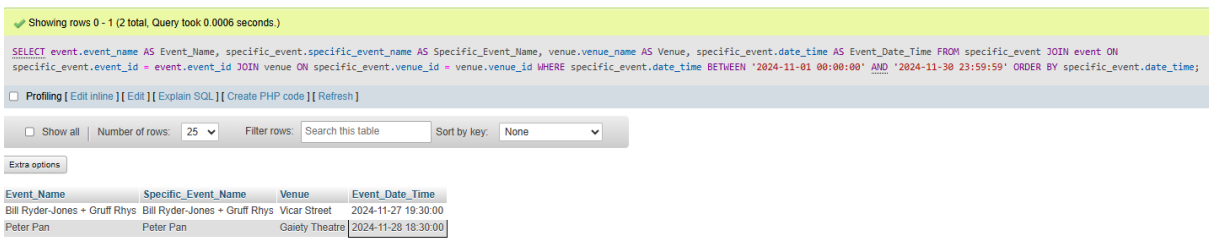
Profiling [Edit inline] [Edit] [Explain SQL] [Create PHP code] [Refresh]

Show all | Number of rows: 25 Filter rows: Search this table Sort by key: None

Extra options

Event_Name	Specific_Event_Name	Category	Event_Date_Time	Venue
Sugar Babes	Sugar Babes	All Concerts	2025-04-19 18:30:00	SSE Arena
Olly Murs	Olly Murs	All Concerts	2025-04-30 18:30:00	SSE Arena
Bill Ryder-Jones + Gruff Rhys	Bill Ryder-Jones + Gruff Rhys	All Concerts	2024-11-27 19:30:00	Vicar Street

Appendix X - Searching by specific date



Showing rows 0 - 1 (2 total, Query took 0.0006 seconds.)

```
SELECT event.event_name AS Event_Name, specific_event.specific_event_name AS Specific_Event_Name, venue.venue_name AS Venue, specific_event.date_time AS Event_Date_Time FROM specific_event JOIN event ON specific_event.event_id = event.event_id JOIN venue ON specific_event.venue_id = venue.venue_id WHERE specific_event.date_time BETWEEN '2024-11-01 00:00:00' AND '2024-11-30 23:59:59' ORDER BY specific_event.date_time;
```

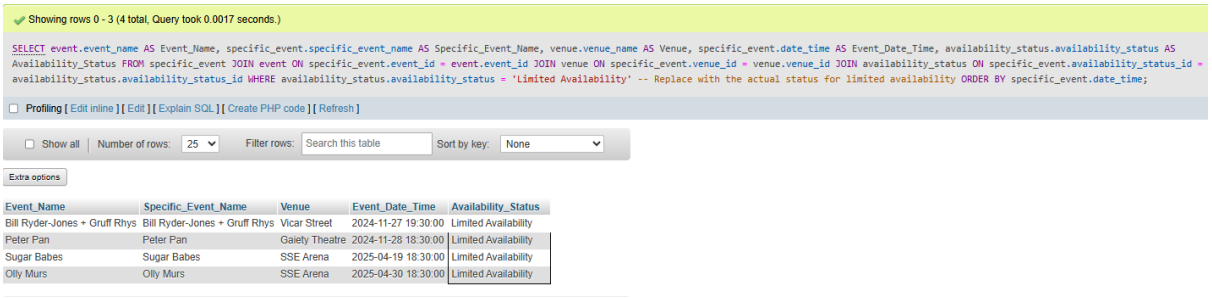
Profiling [Edit inline] [Edit] [Explain SQL] [Create PHP code] [Refresh]

Show all | Number of rows: 25 Filter rows: Search this table Sort by key: None

Extra options

Event_Name	Specific_Event_Name	Venue	Event_Date_Time
Bill Ryder-Jones + Gruff Rhys	Bill Ryder-Jones + Gruff Rhys	Vicar Street	2024-11-27 19:30:00
Peter Pan	Peter Pan	Gaiety Theatre	2024-11-28 18:30:00

Appendix XI - Searching by specific availability



Showing rows 0 - 3 (4 total, Query took 0.0017 seconds.)

```
SELECT event.event_name AS Event_Name, specific_event.specific_event_name AS Specific_Event_Name, venue.venue_name AS Venue, specific_event.date_time AS Event_Date_Time, availability_status.availability_status AS
Availability_Status FROM specific_event JOIN event ON specific_event.event_id = event.event_id JOIN venue ON specific_event.venue_id = venue.venue_id JOIN availability_status ON specific_event.availability_status_id =
availability_status.availability_status_id WHERE availability_status.availability_status = 'Limited Availability' -- Replace with the actual status for limited availability ORDER BY specific_event.date_time;
```

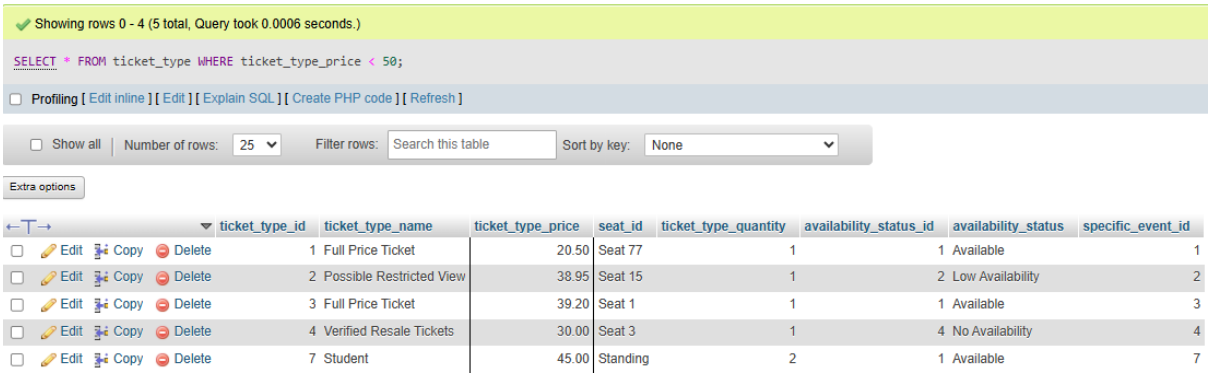
Profiling [Edit inline] [Edit] [Explain SQL] [Create PHP code] [Refresh]

Show all | Number of rows: 25 | Filter rows: Search this table | Sort by key: None

Extra options

Event_Name	Specific_Event_Name	Venue	Event_Date_Time	Availability_Status
Bill Ryder-Jones + Gruff Rhys	Bill Ryder-Jones + Gruff Rhys	Vicar Street	2024-11-27 19:30:00	Limited Availability
Peter Pan	Peter Pan	Gaiety Theatre	2024-11-28 18:30:00	Limited Availability
Sugar Babes	Sugar Babes	SSE Arena	2025-04-19 18:30:00	Limited Availability
Olly Murs	Olly Murs	SSE Arena	2025-04-30 18:30:00	Limited Availability

Appendix XII - Searching by specific price



Showing rows 0 - 4 (5 total, Query took 0.0006 seconds.)

```
SELECT * FROM ticket_type WHERE ticket_type_price < 50;
```

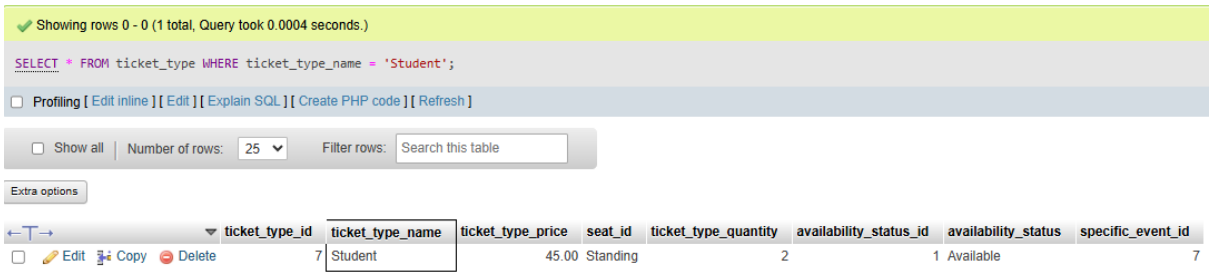
Profiling [Edit inline] [Edit] [Explain SQL] [Create PHP code] [Refresh]

Show all | Number of rows: 25 | Filter rows: Search this table | Sort by key: None

Extra options

	ticket_type_id	ticket_type_name	ticket_type_price	seat_id	ticket_type_quantity	availability_status_id	availability_status	specific_event_id
Edit Copy Delete	1	Full Price Ticket	20.50	Seat 77	1	1	Available	1
Edit Copy Delete	2	Possible Restricted View	38.95	Seat 15	1	2	Low Availability	2
Edit Copy Delete	3	Full Price Ticket	39.20	Seat 1	1	1	Available	3
Edit Copy Delete	4	Verified Resale Tickets	30.00	Seat 3	1	4	No Availability	4
Edit Copy Delete	7	Student	45.00	Standing	2	1	Available	7

Appendix XIII - Searching by specific ticket type



Showing rows 0 - 0 (1 total, Query took 0.0004 seconds.)

```
SELECT * FROM ticket_type WHERE ticket_type_name = 'Student';
```

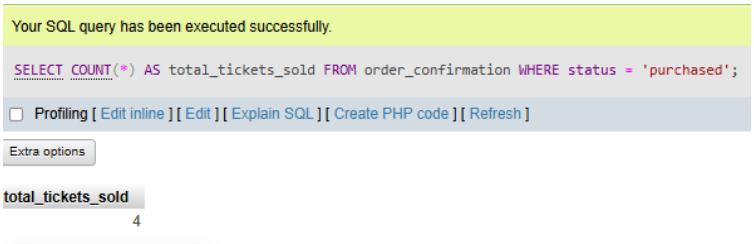
Profiling [Edit inline] [Edit] [Explain SQL] [Create PHP code] [Refresh]

Show all | Number of rows: 25 | Filter rows: Search this table

Extra options

	ticket_type_id	ticket_type_name	ticket_type_price	seat_id	ticket_type_quantity	availability_status_id	availability_status	specific_event_id
Edit Copy Delete	7	Student	45.00	Standing	2	1	Available	7

Appendix XIV – Total tickets sold



Your SQL query has been executed successfully.

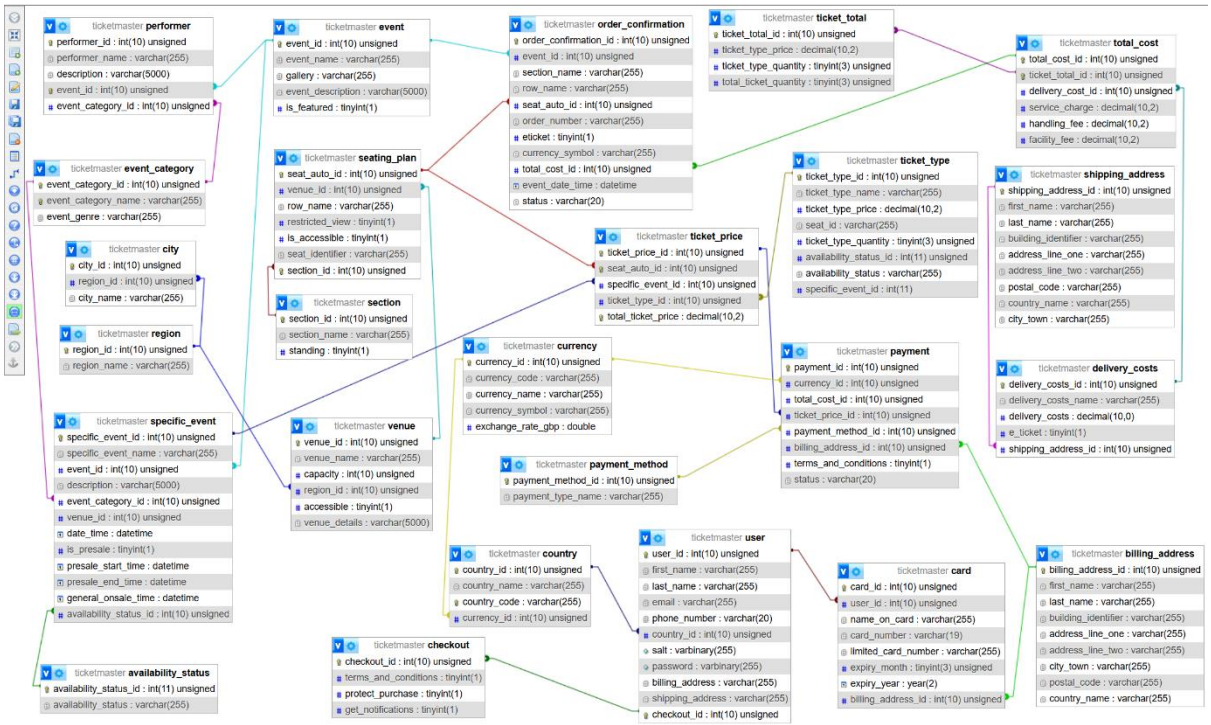
```
SELECT COUNT(*) AS total_tickets_sold FROM order_confirmation WHERE status = 'purchased';
```

Profiling [Edit inline] [Edit] [Explain SQL] [Create PHP code] [Refresh]

Extra options

total_tickets_sold
4

Appendix XV – Database designer view



Appendix XVI - Example of large variety of ticket options for selected festival Creamfields

STANDARD DAY TICKETS

Friday Day Ticket - Standard PAYMENT PLAN
Friday 22 August 2025

Saturday Day Ticket - Standard PAYMENT PLAN
Saturday 23 August 2025

Sunday Day Ticket - Standard PAYMENT PLAN
Sunday 24 August 2025

2 Day Non Camping (Fri / Sat) - Standard PAYMENT PLAN
Friday 22 August - Saturday 23 August 2025

2 Day Non Camping (Sat / Sun) - Standard PAYMENT PLAN
Saturday 23 August - Sunday 24 August 2025

3 Day Non Camping (Fri / Sat / Sun) - Standard PAYMENT PLAN
Friday 22 August - Sunday 24 August 2025

Friday Day Ticket - Standard
Friday 22 August 2025

Saturday Day Ticket - Standard
Saturday 23 August 2025

Sunday Day Ticket - Standard
Sunday 24 August 2025

2 Day Non Camping (Fri / Sat) - Standard
Friday 22 August - Saturday 23 August 2025

2 Day Non Camping (Sat / Sun) - Standard
Saturday 23 August - Sunday 24 August 2025

3 Day Non Camping (Fri / Sat / Sun) - Standard
Friday 22 August - Sunday 24 August 2025

GOLD CAMPING

4 Day Camping (Thurs / Fri / Sat / Sun) - Gold PAYMENT PLAN
Thursday 21 August - Sunday 24 August 2025

3 Day Camping (Fri / Sat / Sun) - Gold PAYMENT PLAN
Friday 22 August - Sunday 24 August 2025

2 Day Camping (Sat / Sun) - Gold PAYMENT PLAN
Saturday 23 August - Sunday 24 August 2025

4 Day Camping (Thurs / Fri / Sat / Sun) - Gold
Thursday 21 August - Sunday 24 August 2025

3 Day Camping (Fri / Sat / Sun) - Gold
Friday 22 August - Sunday 24 August 2025

2 Day Camping (Sat / Sun) - Gold
Saturday 23 August - Sunday 24 August 2025