CSC7052 Databases
Individual Database Design Report

This report presents a comprehensive analysis and critical evaluation of the design and implementation of a database, grounded in Peter Chen's entity-relationship (ER) modelling framework. The database was designed to illustrate selected aspects of Ticketmaster, a ticket sales and distribution service that facilitates transactions between event organisers and consumers.

Chen's ER modelling theory[1] is based on three fundamental components: entities (representing objects), attributes (describing object properties), and relationships (defining associations between entities). These principles have been applied using phpMyAdmin, an open-source web-based tool for managing MySQL databases.

The objective of this report is to deliver a qualitative assessment that caters to both laypersons and academics by clearly articulating the design and implementation choices made throughout the process. The report is supported by an Appendix containing screenshots of database tables and MySQL queries, providing detailed insight into the entity structure, data types, and foreign key constraints.

Additionally, this report distinguishes my approach from those of my peers through a critical evaluation of the design and implementation decisions. It aims to contribute both a practical and theoretical perspective on database modelling and development.

I. Initial Discovery

This section outlines the initial discovery process, emphasizing the collaborative and individual contributions to the project. The group collaboration was limited to developing the initial schema, while the subsequent design and implementation were conducted independently.

The project began with a group review of the Ticketmaster booking website. This exploratory exercise involved navigating the platform as a customer, simulating the process of purchasing a ticket for an event. The purpose of this activity was to reach a consensus on three key aspects:

1) Identifying the core entities,
2) Summarising their attributes,
3) Understanding the relationships between these entities.

In my initial analysis, I identified the following core entities based on the essential components of Ticketmaster's service:

- **Customers:** Individuals using the website to purchase tickets.
- **Orders:** Transactions initiated by customers.

[1] Chen, P. (1977) 'The Entity-Relationship Model: A Basis For The Enterprise View of Data', *Library of the Massachusetts Institute of Technology*. Available at: https://dspace.mit.edu/bitstream/handle/1721.1/47625/entityrelationshx1977chen.pdf (Accessed: 15 November 2024).

- **Tickets:** The items being purchased.
- **Events:** The occasions for which tickets are sold.
- **Venues:** Locations where events are held.
- **Categories:** Classifications of events.

I independently compiled a preliminary snapshot of these entities and their attributes, as illustrated in Figure 1. This early schema, including appropriate data types, served as a foundation for subsequent implementation within phpMyAdmin.

table	key type	data type		table	key type	data type
customer				event		
customer_id	**pk**	int		**event_id**	**pk**	int
email		string		event_name		string
first_name		string		event_description		string
sur_name		string				
user_name		string				
password		string		venue		
				venue_id	**pk**	int
order				venue_name		string
order_id	**pk**	int		venue_address		string
order_price		double		capacity		string
order_date		int				
billing_address		string		category		
delivery_type		string		**category_id**	**pk**	int
delivery_address		string		category_name		string
				description		string
ticket						
ticket_id	**pk**	int		location		
event_id	**fk**	int		**location_id**	**pk**	int
customer_id	**fk**	int		location_name		string
availability		int		description		string
ticket_price		int				
ticket_type		string				

Figure 1: Initial Entity and Attribute findings

Following this individual analysis, the group transitioned from examining Ticketmaster as customers to adopting the perspective of developers. This shift encouraged us to consider the broader range of services provided by Ticketmaster, such as its role in hosting events and collaborating with event organizers. However, these services primarily addressed on a separate domain (business.ticketmaster.com) were deemed beyond the scope of our shared objectives.

Once the core entities were identified, the next step was a collaborative data normalization process. This phase aimed to organize entities into well-structured tables that minimized redundancy and ensured data integrity, following the first two principles of Edgar F. Codd's relational database model.

- **First Normal Form (1NF):** Ensured each table contained atomic values and eliminated repeating groups by allowing only one value per cell.
- **Second Normal Form (2NF):** Introduced primary keys for each table, ensuring all non-key attributes were functionally dependent on the primary key.

Third Normal Form was not prioritised at this stage of the project, as further changes to the database structure were anticipated

II. Individual Design and Implementation

During this process, I diverged from the group in certain design decisions. While we collectively agreed that each entity should include an auto-incrementing unique identifier as its primary key, I approached the relationships and attributes with a focus on eventual user and developer needs. Initially, my assumptions regarding some relationships and attributes were tentative, as the priority was to establish a developer-oriented perspective on the entities rather than solely reflecting the customer experience.

The designer view in phpMyAdmin of my database is located at Figure 2. It highlights the final entities and attributes implemented within phpMyAdmin. This design reflects a balance between user-centric functionality and technical efficiency, focusing on selected aspects of Ticketmaster's operations. This approach allowed for a robust database design tailored to the project's objectives

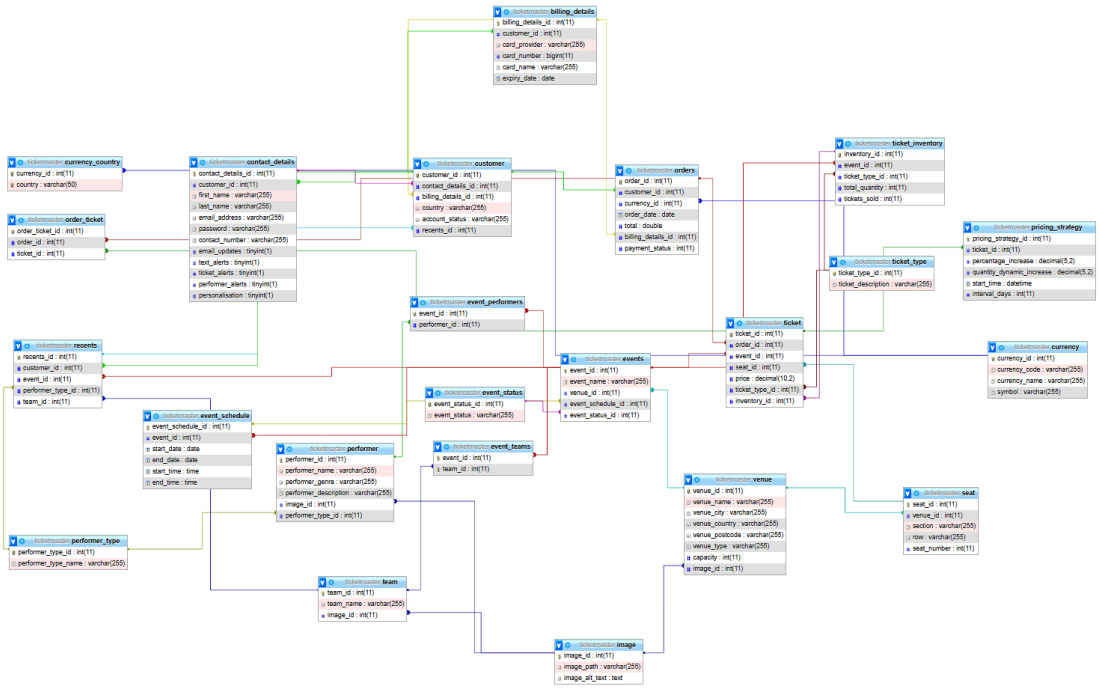

Figure 2 Final designer view in phpMyAdmin

The initial discovery phase involved a collaborative brainstorming session where the group worked together to establish key assumptions about Ticketmaster's operations. This served as a foundation for the next stage: independently designing our individual interpretations of the Ticketmaster system. Creating a relational database for a system as intricate as Ticketmaster posed several significant challenges, which can be summarised as follows:

- Complex Relationships between many Entities
- Dynamic Pricing Rules
- Geographic complexity
- Compliance with regulations and encryption standards
- Scaling and Performance

In the next sections I will provide an explanation of the key entities I have created alongside their attributes and how they address the above challenges.

A. Customer and Contact details

My objective was to build a database that reflected the core aspects of Ticketmaster, starting with the customer entity. A user interacts with Ticketmaster, while a customer is a specific type of user who makes purchases. In an ERD, a customer entity would link to a user entity with a one-to-one relationship. I deviated from group consensus by choosing to have separate customer and contact details entities. In Figure 6 (in Appendix) I opted to use a contact_details table to store the information expected when registering an account on Ticketmaster. The naming convention used is snake casing. I chose to use contact_details instead of user_details because it more accurately describes the attributes and does not imply that an individual is actively using the website.

The customer table is linked to the contact_details table via a one-to-one relationship, using customer_id as the primary key in the customer table and as a foreign key in the contact_details table. The separation of contact_details from the main customer table aligns with best practices for data normalisation. Isolating contact information (like contact numbers, email addresses) into its own table reduces redundancy and improves data integrity. The SQL code I used in figure 48 (in Appendix) demonstrates that if a customer needs to update their email address or contact information, the update only is required in the contact_details table, without affecting other customer related data. This structure is easier to manage in a real-world booking system in Ticketmaster.

In the contact_details table, I introduced columns to capture the communication preferences a customer selects when creating an account on Ticketmaster. Following standard practices for many online platforms, these columns use tinyint datatypes to indicate whether a customer opts to receive email updates, ticket alerts, performer alerts, or a personalised homage. This feature was designed to enhance the user experience by ensuring customers receive tailored communications based on their preferences.

B. Billing Details

Similarly, the customer table is linked to the billing_details table using a one-to-one relationship, as highlighted in figure 3 (see Appendix). The billing_details table has a foreign key, customer_id, referencing the customer table. Isolating billing information into a separate billing_details table is beneficial from both a security and performance standpoint. In real world systems such as Ticketmaster, sensitive data like credit card numbers, billing address and payment details require higher levels of security and compliance, such as PCI DSS compliance for handling payment information. The separation facilitates stricter access controls and encryption (which we will cover in more detail later) without impacting other parts of the customer profile.

This design is essential to make the Ticketmaster system more flexible. If a customer needs to update their billing information, those changes are confirmed to the billing_details table, leaving the customer data untouched. Additionally, having billing details in a dedicated table means that multiple billing records can be easily linked to the same customer if needed in the future.

C. Orders

The orders table connects to the customer table using the customer_id foreign key, creating a one-to-many relationship where an individual customer can have more than one order. Each order is associated with its corresponding billing details through reference to the billing_details table. This design accommodates the real-world scenario where a customer can place multiple orders over time and have unique billing details.

In 2024, customers have various payment providers to choose from. I consciously chose to keep the orders table linked to customers rather than embedding customers details within it. This means that if a customer's contact details change, only one update is needed in the relevant record in the contact details table. This approach preserves data consistency by ensuring that changes are immediately reflected across all past and future orders linked to that customer.

It took several iterations to select the attributes within the tables discussed so far. This was the first significant stumbling block I encountered in designing and implementing my database. It became clear from my initial schema that I had not fully grasped the importance of ensuring historical data in the order table remains accurate. For example, if a customer's address changes, previous orders should still show the original address used at the time of purchase and that is crucial for accurate record keeping. This was not obvious in our initial group discussions when navigating Ticketmaster as a customer.

The modularity created in the database creates potential for an easier integration of customer relationship management tools because customer profiles are kept separate from transactional data.

D. Events

My second priority in the individual design phase was to clearly define the Event entity. The project group did not reach a consensus on how to approach this. In my own database, (see figure 12 in Appendix) the events table is the core entity that defines each event that could be advertised on Ticketmaster. The events_schedule table captures detailed information when and where these events take place. This is supported by the event_status table which provides additional metadata on the status of events through a foreign key relationship with the events table.

The three interconnected tables were my response to the challenge of preventing data redundancy while capturing the key attributes of events. Separating data in events from specific scheduling details ensures that multiple instances of the same event can be managed efficiently. Events have multiple showtimes and multiple venues, such as in the context of a touring concert or a sports season. The event_status table attempts to track an event's lifecycle, which is important in real world ticketing applications such as Ticketmaster, where statuses can change.

I chose to link event_status_id to events to enable real-time updates to event statuses without affecting other event data. This approach aims to support better event management and customer communication.

The third priority was to focus on ticketing. On Ticketmaster, a customer places an order to purchase a ticket. I did not consciously decide to create another triad of tables.

My final database incorporated a ticket_inventory table that would manage the overall stock of tickets available for each event, while the ticket table captures individual ticket details for orders. The ticket_type table classifies tickets into categories and would be linked using a foreign key constraint via ticket_type_id.

It was necessary to separate the ticket_inventory from the ticket table to make a distinction between stock management and individual ticket sales. This separation allows the system to track available tickets in real time, ensuring that ticket availability is accurately reflected on the front end. The ticket_type table was introduced to provide a layer of flexibility for tiered tickets, which offer additional levels of access for a higher premium. Our initial entity discovery process indicated that offering a variety of ticket options help maximise revenue.

E. Performer, Teams

To accurately capture the diverse types of performers and teams featured on Tickemaster, I designed several tables in phpMyAdmin: performer, event_performers, performer_type, team and event_tables. This structure reflects a nuanced understanding of how events are organised, accommodating individual performers, multiple performers and teams.

The performer table serves as the primary repository for information for individuals participating in events. Each performers page on Ticketmaster includes their name, a biography or description, and an image. To manage the many-to-many relationship between performers and events, I created the event_performers junction table.

The performer_type table categorises performers into types that such as Music, Sport, Arts, Family, as seen on the Ticketmaster homepage. While reviewing the website as a customer, it was clear that junction tables were not necessary. However, as a system developer, I recognised that these tables are essential for managing the complexity of many-to-many relationships, which are common in events like sporting matches and concert tours. The tables also support complex SQL queries as seen in figure 49 (see in Appendix)[2].

To ensure scalability without disrupting the existing database structure, I created a team table to store information about groups participating events, such as sports teams. I mirrored the junction table concept for teams with the event_teams table, facilitating many-to-many relationships between team and events. This supports complex event configurations where multiple teams can be associate with a single event (see figure 20 in Appendix).

Recognising that both teams and performers are represented by images on Ticketmaster, I created an images table to centralise image management. This table stores all performers, team and venue-related images, eliminating redundancy. Linking images to performers and teams enhances the user experience, leading to a more engaging platform. I selected an image_alt_text attribute to support accessibility, and this has potential for ensuring that users with visual impairments can access descriptive information through screen readers. Ticketmaster uses HTML as an attribute within image tags to provide a textual description of

[2] ChatGBPT (2024) "Demonstration of SQL query code involving Performers and Teams in a database". Chat conversation

images on a web page which enhances search engine optimisation, user experience, and increase the discoverability of events.

F. Ticketing, Pricing Strategy, Venue and Seats

The third priority was to focus on ticketing. On Ticketmaster, a customer places an order to purchase a ticket. My final database incorporated a ticket_inventory table that would manage the overall stock of tickets available for each event, while the ticket table captures individual ticket details for orders. The ticket_type table classifies tickets into categories and would be linked using a foreign key constraint via ticket_type_id.

It was necessary to separate the ticket_inventory from the ticket table to make a distinction between stock management and individual ticket sales. This separation allows the system to track available tickets in real time, ensuring that ticket availability is accurately reflected on the front end. The ticket_type table was introduced to provide a layer of flexibility for tiered tickets, which offer additional levels of access for a higher premium. Our initial entity discovery process indicated that offering a variety of ticket options help maximise revenue.

I next considered how to organise venue seating using a venue, seat and ticket entities. The venue table defines the event locations, each of which has an associated collection of seats managed by the seat table. I had opted not to use a separate seat_allocation table, as seats assignments are directly handled within the ticket table where each ticket references a specific seat. This ensures the direct linkage of seats to ticket orders, ensuring that each ticket sold is associated with a particular seat. The seat table is related to the venue table through a foreign key (venue_id), allowing each venue to maintain its own set of seats with attributes like section, row and seat number. The ticket table connects to specific events via event_id and seat_id, ensuring that each ticket corresponds to a particular seat for a designated event. This was the most challenging aspect of Ticketmaster to recreate. I chose an approach that prioritises organisation over performance, which could encounter more issues if this database were operating at Ticketmaster's large scale. The lack of a seat allocation table means that seat management is coupled with ticket sales and does not accommodate advanced functions such as block bookings or for more dynamic seat management. My phpMyAdmin database currently lacks the volume of data present in Ticketmaster's database and does not yet include the functionality to dynamically manage seating, such as holding a seat for a set period before purchase. When a customer cancels their ticket, the system needs to release the seat and make it immediately available for others. During high-demand periods, when many customers are trying to book seats simultaneously, this would naturally impact the speed of seat selection and ticket purchasing processes.

Additionally, Ticketmaster increasingly relies on dynamic pricing based on demand, seat location and booking team. For this project demonstration, my database supports static seat assignments. Introducing a table that associates different pricing tiers with specific seats or sections could be introduced to facilitate dynamic pricing. However, implementing the complex algorithms driving these prices would require a behind the scenes look at Ticketmaster's system.

Given the complexity of this aspect, I chose to utilise phpMyAdmin to devise a method of implementing dynamic pricing rules. I created a pricing strategy table as the centralised repository of the numeric values used in the dynamic pricing logic (see figure 33 for pricing_strategy and figure 55 for dynamic pricing logic in Appendix). The primary key, pricing_strategy_id, is set to auto-increment. This table includes a ticket_id column with a foreign key constraint to link it to the relevant tickets.

Additionally, I created a column named percentage_increase with a decimal (5,2) datatype to capture the set percentage used in my dynamic pricing logic, allowing for future edits. Another column, quantity_dynamic_increase, was added to handle dynamic pricing rules based on ticket quantity thresholds (see Appendix figure 52). Finally, I included a start_time using the datetime datatype and an interval_days column to define the frequency of the dynamic pricing.

In my database, I successfully implemented basic dynamic pricing logic for the Coldplay: Music of the Spheres concert. Using the event scheduler, I developed code that schedules ticket price increases from a specified start date. This code integrates the tickets, pricing_strategy, and events_schedule tables, increasing ticket prices by a set percentage when the current date falls within the defined range and interval. This set up ensures that ticket prices adjust dynamically over time, reflecting demand and other factors.

I also developed dynamic pricing logic to apply a premium to tickets when their quantities fall below certain threshold, aiming to reflect high demand. While this is a speculative approach and not necessarily indicative of Ticketmaster's practices, it demonstrates a potential capability that can be facilitated within this relational database.

G. Currency and Currency-Country relationship

I created a multi-currency system, essential for a global ticketing platform like Ticketmaster. For this project, the database was tailored for customers in United Kingdom and Ireland, who use Pounds and Euros respectively.

I constructed a currency table that stores details about the available currencies. Additionally, a currency_country table defines which currencies are allowed for customers based on their country of location. This relationship ensures that only valid currencies are accepted for orders based on the customers location. The customer table, discussed earlier in the report includes information such as country, which is crucial for determining the valid currencies they can use. Finally, the orders table contains details of the ticket orders including the currency_id used for the transaction.

This approach to handling currency is justified for several reasons. By tying the customer's country to the allowed currency, it provides a personalized experience where users transact in their local currencies. This also facilities the easy addition (or removal) of currencies for specific countries, making the system adaptable to changes in the marketplace. For example, if a country starts accepting a new currency, it can be added to the currency_country table without altering the existing logic.

Within phpMyAdmin, I utilised the trigger functionality to create a trigger named validate_currency. This trigger executes before inserting a new row into the orders table (see figure 53 in Appendix.)

To summarise, the trigger declares an integer variable named valid_currency_count to store the count of valid currencies. It then checks if the currency selected for an order is valid for the customers country by performing a JOIN between the customer and currency_country tables. If the count of valid currencies is zero, it implies that the selected currency is not allowed for the customer based on their country. If no valid entry is found, it raises an error with SQL state 45000, effectively preventing the insertion of the order.

The trigger enforces business rules at the database level, ensuring that only valid currency transactions are processed, which adds an extra layer of data integrity and reduces the risk of incorrect or fraudulent transactions. It also automatically validates currency rules without requiring additional checks in the application layer, streamlining the order process.

In terms of error handling, the use of signal sqlstate provides immediate feedback to the console, which can be useful for debugging. However, there are potential issues if implemented on a larger scale. This could create a significant performance overhead, and the ticketing platform could create a bottleneck with thousands of errors. I have not added an audit log for failed transactions to provide more valuable insights as to why certain orders were rejected however this is something that could be used in future iterations.

H. Encryption

Encryption has not been explored in great depth in the design and implementation of this database for several reasons. Bcrypt, a password-hashing function based on the Blowfish cipher, is widely considered to be the industry standard for hashing passwords. For the purposes of this project, bcrpyt itself cannot be directly implemented as it is typically used within application code, such as PHP. Since we are using web-based interface for managing MySQL databases, I have briefly included a password update mechanism in SQL code (see figure 54 in the Appendix). This mechanism utilises salting and hashing techniques to secure user passwords. What I have replicated in my own databases is purely for demonstrative purposes and is not fit for purpose in an industry setting.

Bcrypt is preferred for several reasons. It includes a work factor that can be adjusted to increase the hashing complexity, slowing down brute-force attacks as computational power increases over time. It also handles its own generation and storage of salts, reducing the risk of incorrect salting implementation. Finally, it’s design makes it resilent to common attacks like rainbow tables, which could easily crack SHA2 hashes if they are not salted correctly.

While the salting and hashing techniques I have implemented rely on basic security practices, given the real time and scale of Ticketmaster, bcrypt would offer a more secure and maintainable solution, protecting passwords against a wider range of attacks.

III. Future Improvements

I have developed presents a well-conceived relational database structure that I feel encapsulates the core components of Ticketmaster. However, despite its functionality, there are notable limitations and complexities that hinder its ability to replicate the robust features of Ticketmaster.

Dynamic seating allocation is a particularly challenging aspect to replicate, as the exact operation isn't available to the public. The current database is designed for static seating arrangements, which limits the system's ability to handle dynamic configurations. This limitation is most evident if we attempted to book an event with flexible seating due to venue layouts which change a lot. Future iterations of the database should incorporate dynamic mapping logic for real-time seat reservations. This also includes a more robust method for removing a seat so that it cannot be double booked but also keeps a historical record of seats that have been booked.

Additionally, greater support for dynamic pricing algorithms would be beneficial. During our initial project group discussions, we acknowledged this as a major issue, but no one could offer a foolproof solution. In my individual design, I utilised the event schedule function to trigger a marginal increase in ticket prices as the event date approaches. This was for demonstration purposes only, and it is understandable why Ticketmaster would prefer to keep such algorithms confidential.

Ticketmaster employs encryption to secure passwords and payment information. In this relational database, passwords are hashed and salted to illustrate the necessity of encryption, though this approach alone would not meet industry standards. While the encryption and tokenisation of sensitive data during transactions are crucial, these aspects have been deemed out of scope for this project. The use of event logs in phpMyAdmin would also have been beneficial to create daily reports to identify problems whilst the database is in operation

IV. Conclusion

In conclusion, I feel the database represents a strong foundation for building selected aspects of Ticketmaster in phpMyAdmin. It effectively models the core components required for such a system. Through normalisation and relational design, the system ensures efficient data storage, reduced redundancy, and logical relationships between the core entities we had initially agreed as a group. The key features such as international currency handling, event scheduling and ticket relationship with seating demonstrates the systems' potential to cater to other ticket-based platforms.

The relationships between events, venues, and tickets accurately reflect the complexities of managing multiple events across various locations. Additionally, the databases' handling of currency through the currency_country and currency tables adds flexibility for international operations, accommodating Ticketmaster's ability for global pricing. The billing_details table lays the groundwork for payment processing, while the contact_details and customer tables support user-specific preferences like email or text updates, enabling personalised communication. I have addressed key imitations and gaps in the database, highlighting areas for future development to better emulate an advanced platform like Ticketmaster. With further enhancements, the database could evolve into a highly capable system.

Appendix

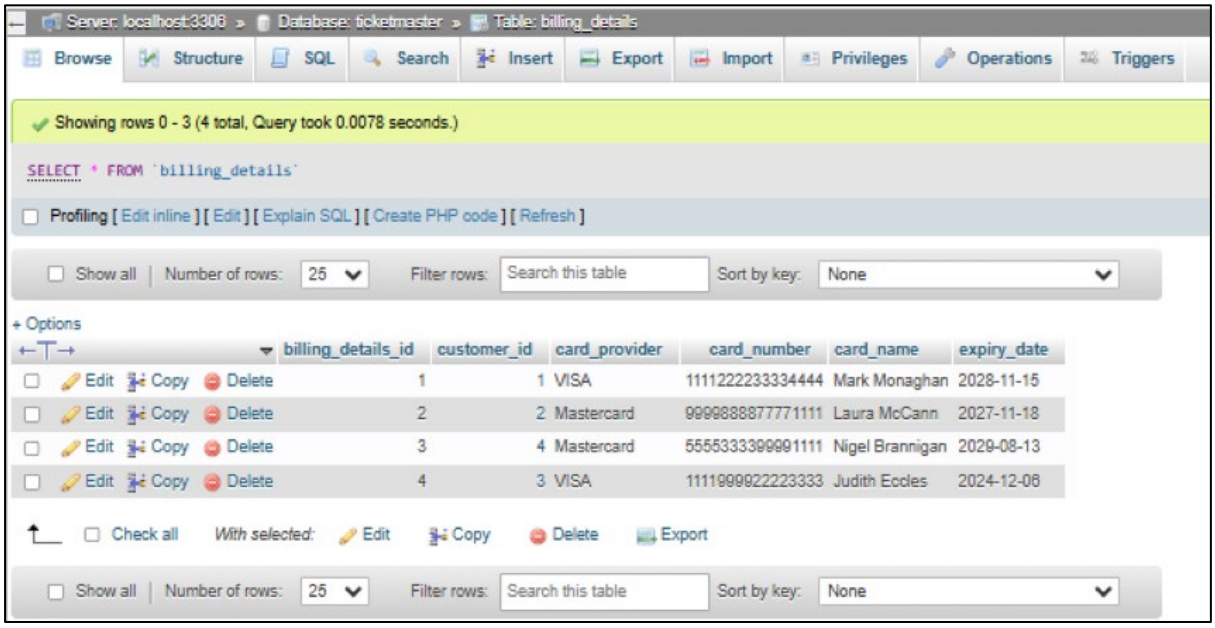

Figure 3 billing_details

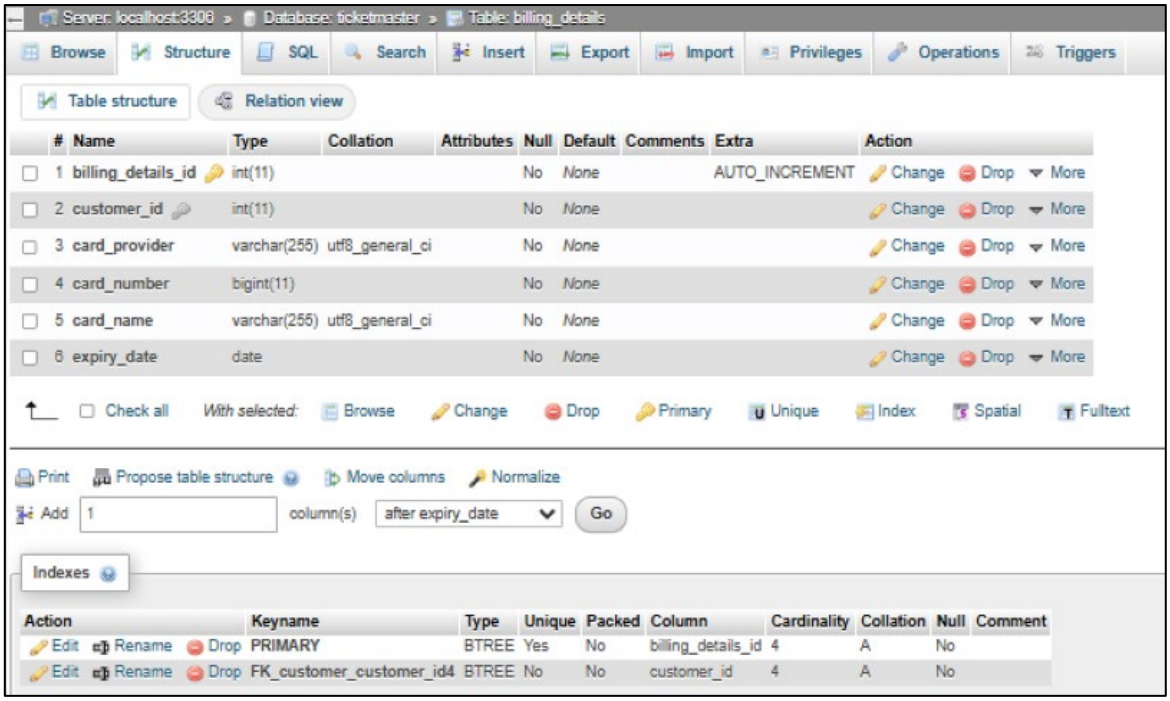

Figure 4 billing_details structure

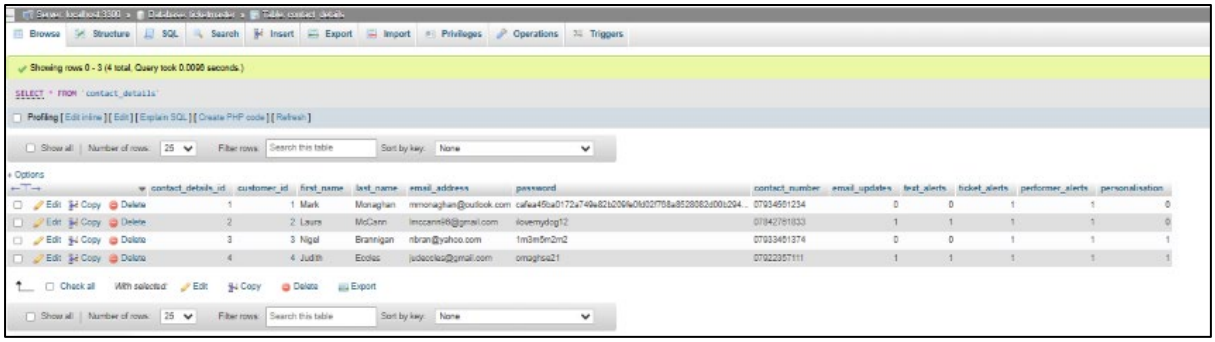

Figure 5 contact_details

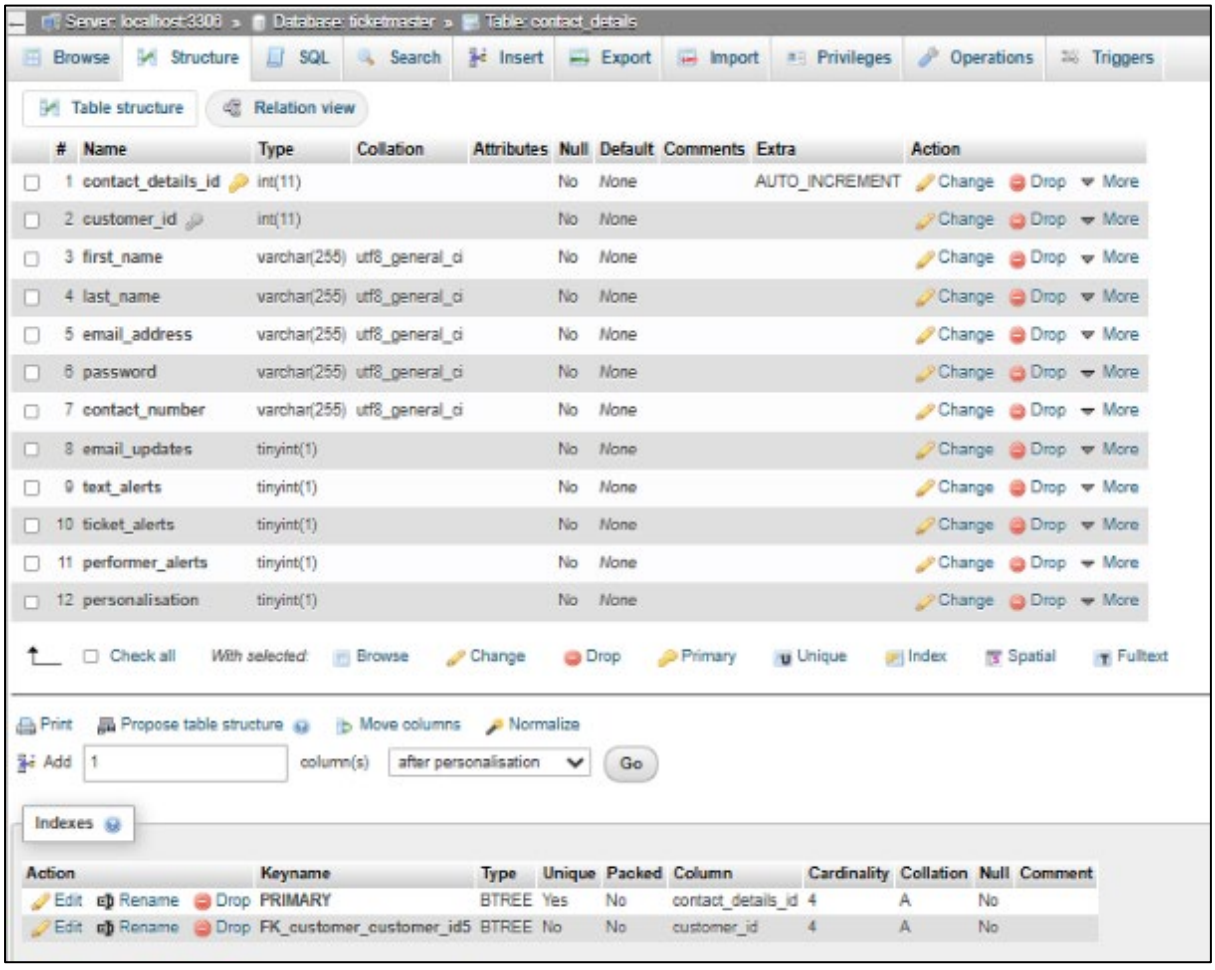

Figure 6 contact_details structure

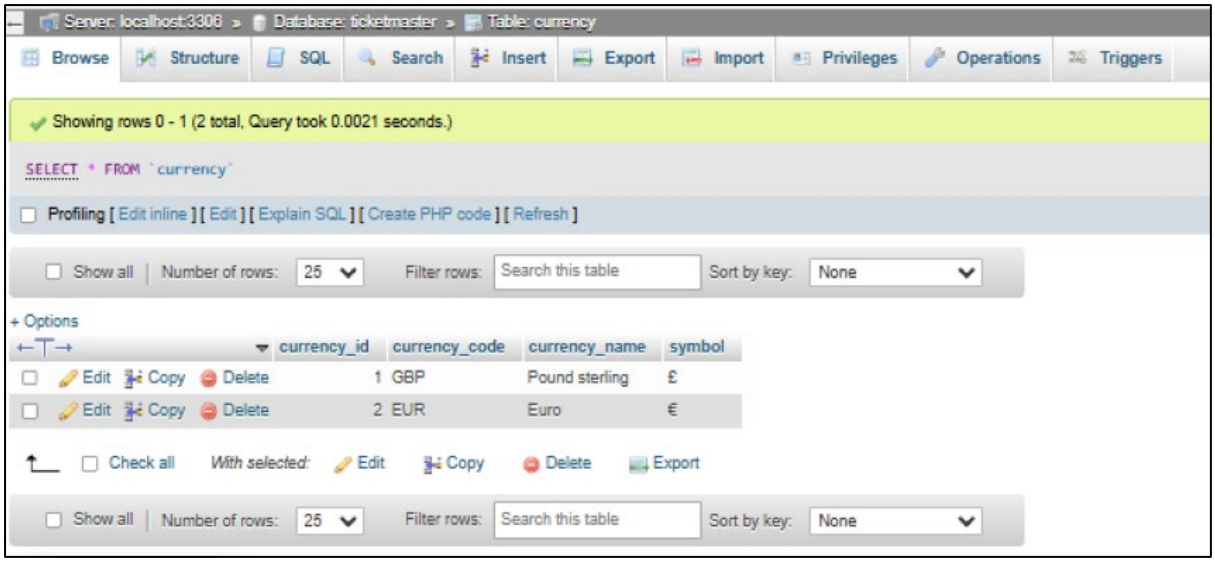

Figure 7 currency

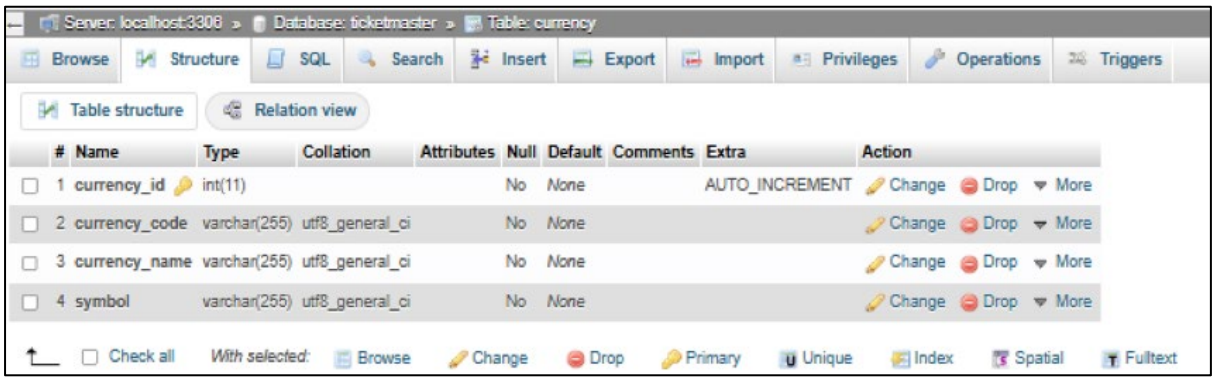

Figure 8 currency structure

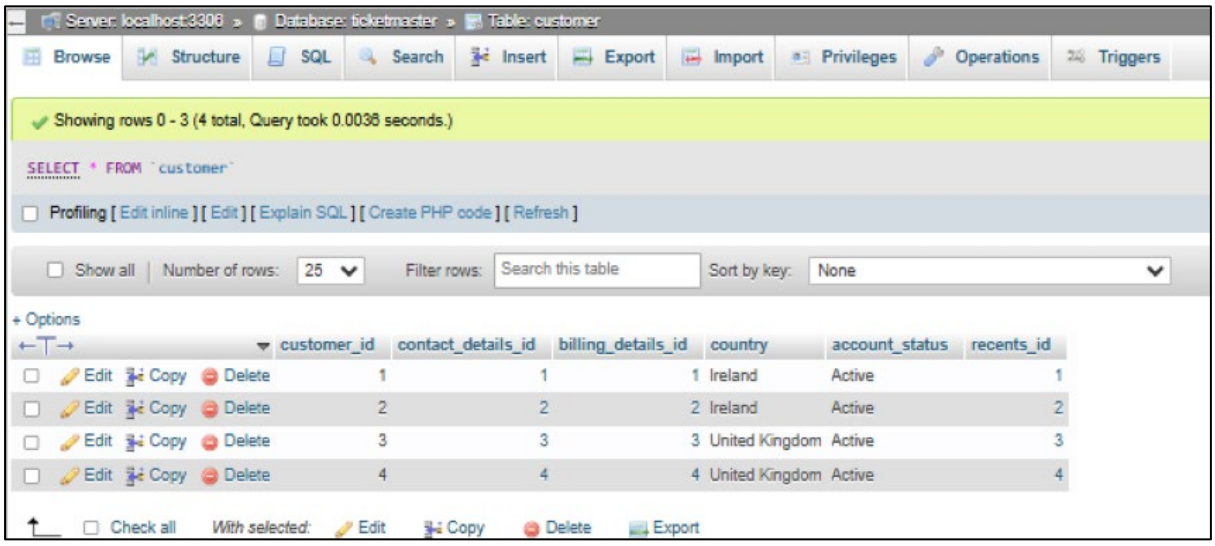

Figure 9 customer

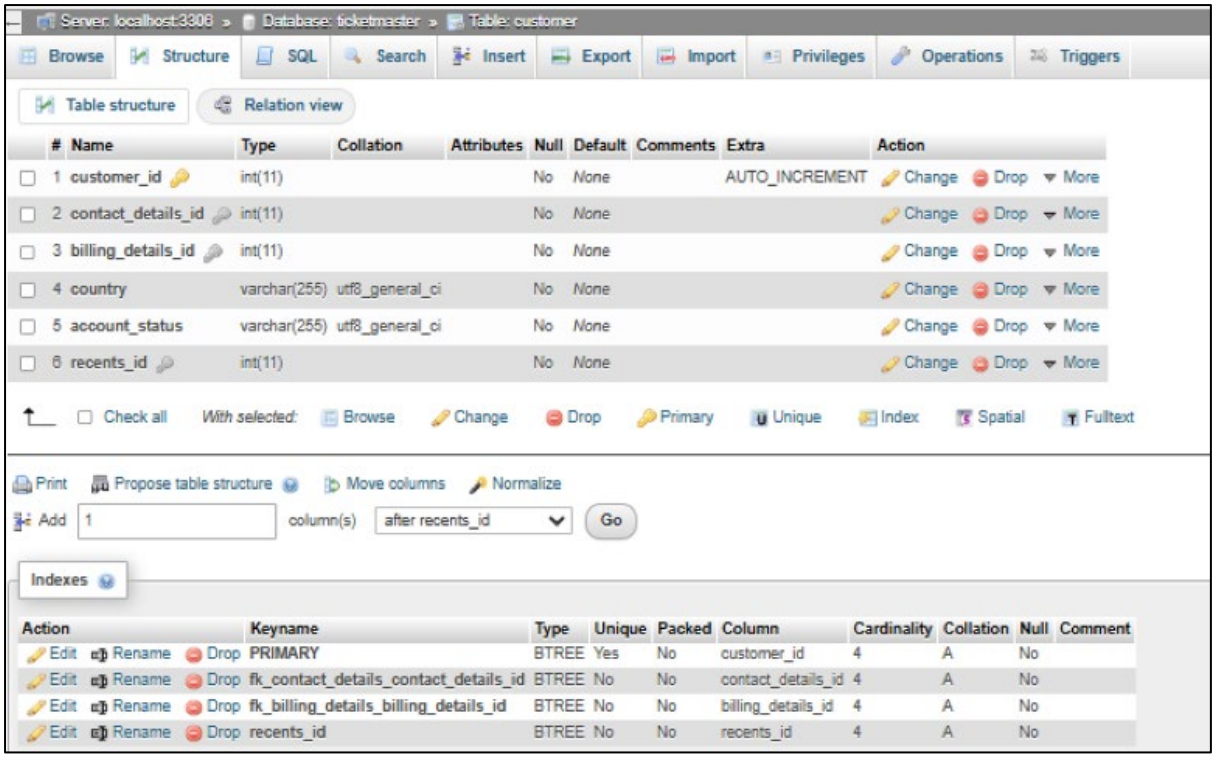

Figure 10 customer structure

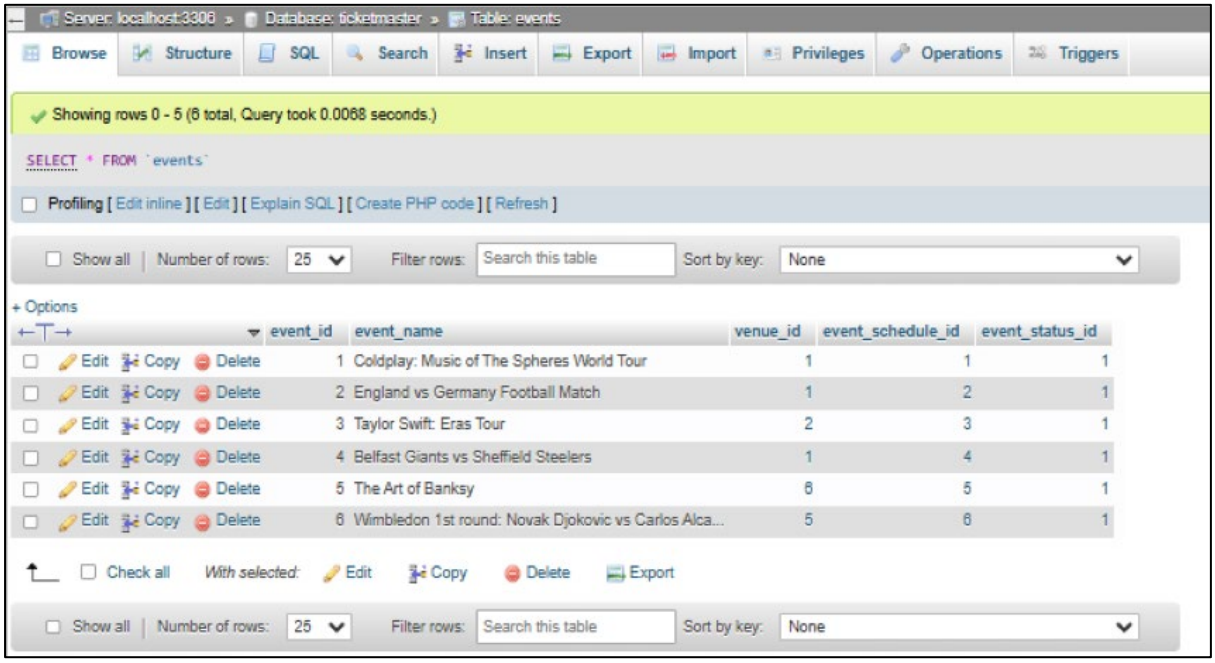

Figure 11 events

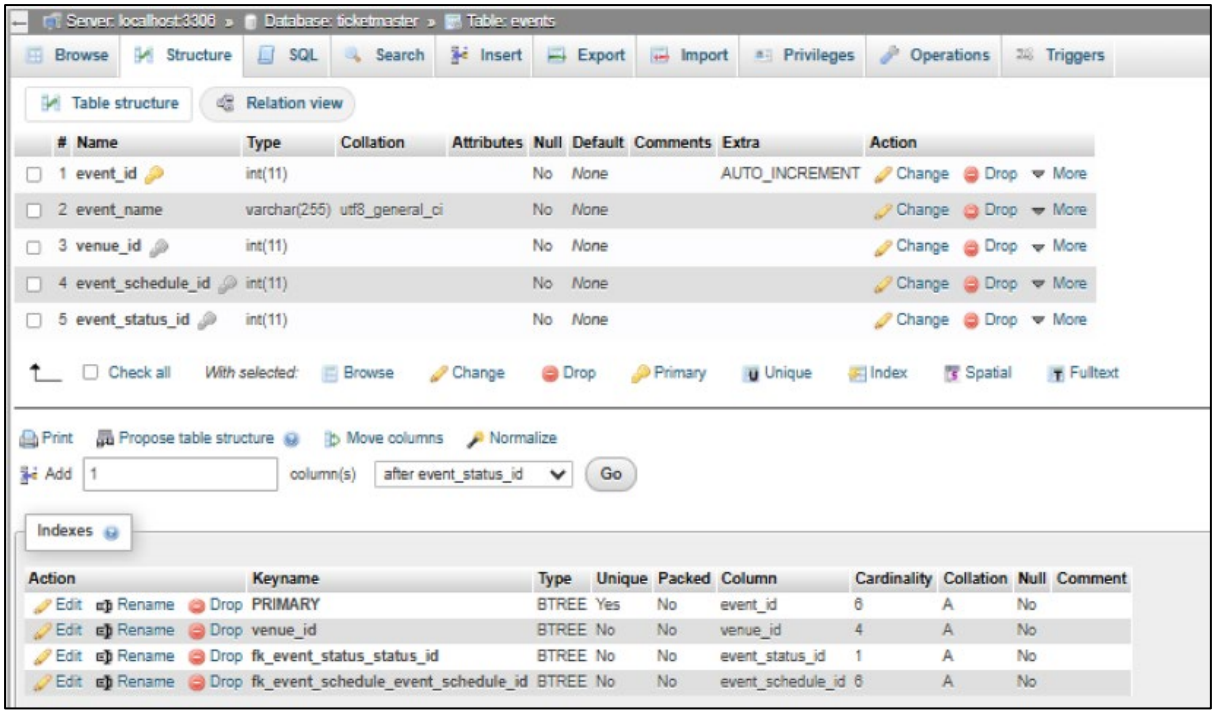

Figure 12 events structure

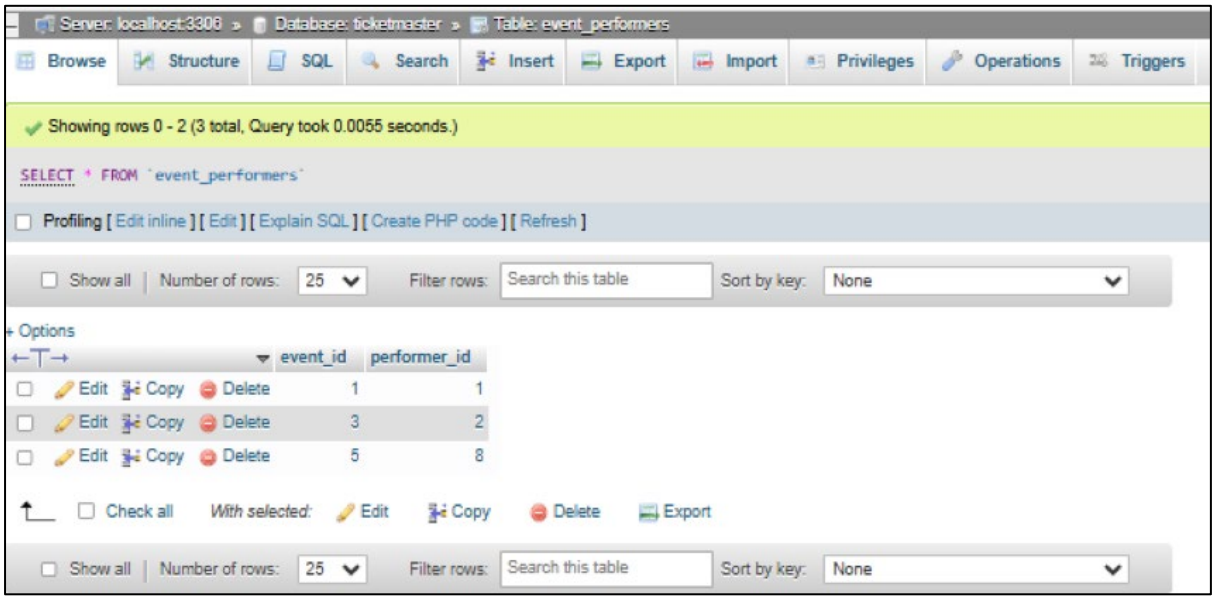

Figure 13 event_performers

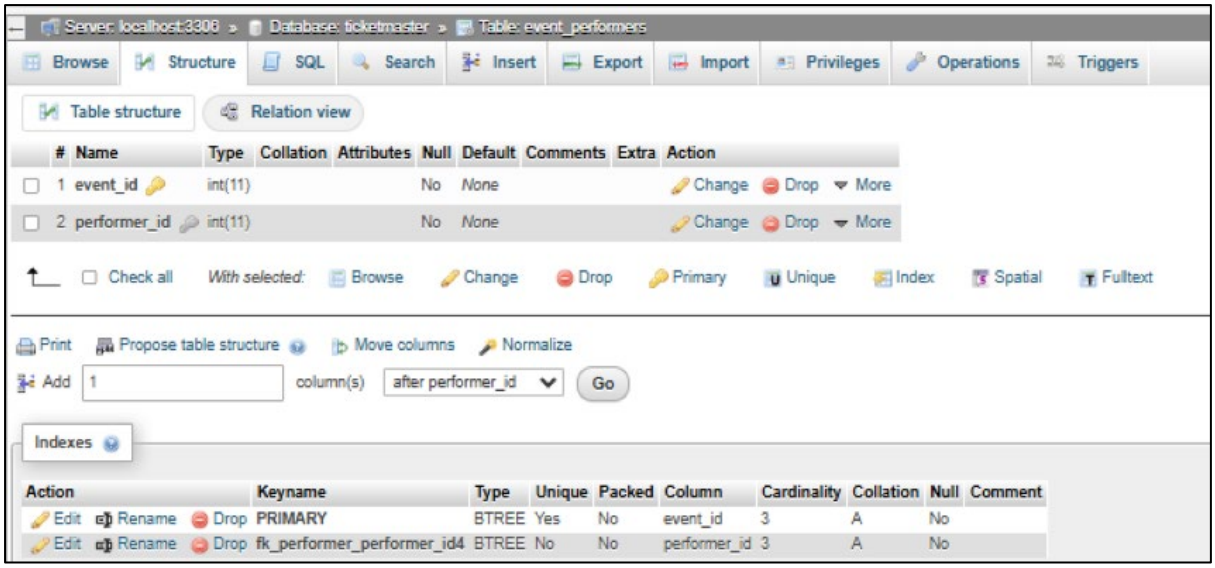

Figure 14 event_performers structure

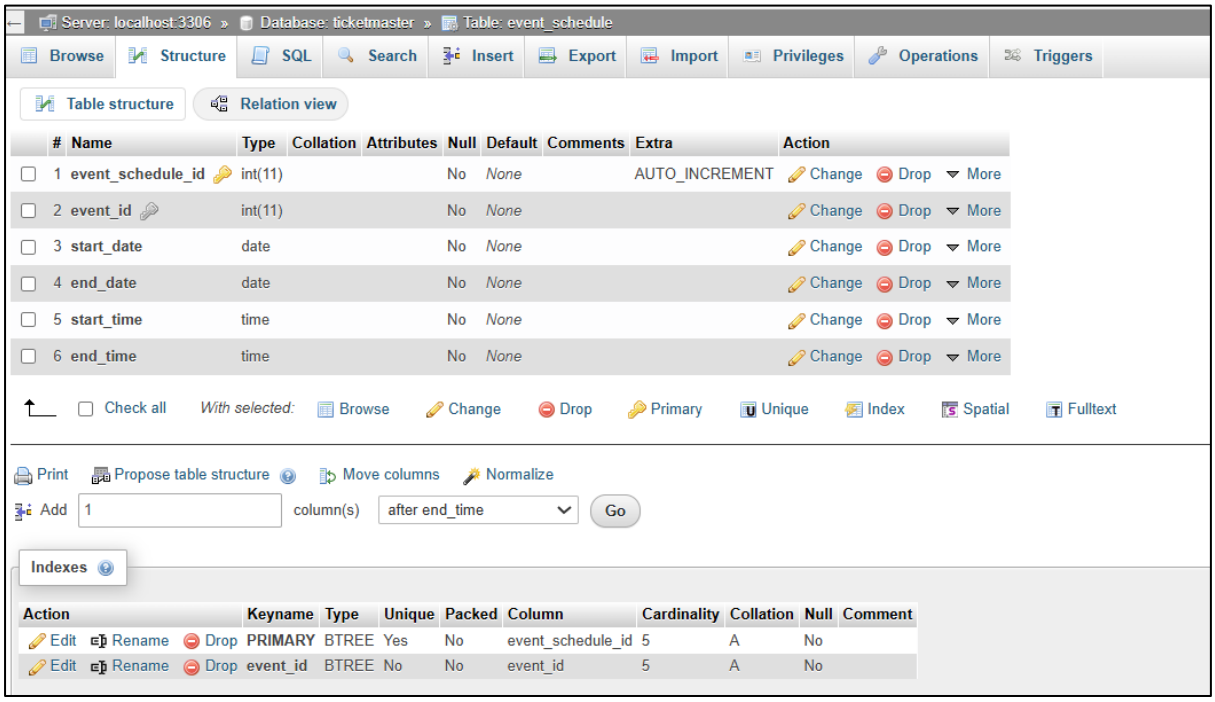

Figure 15 event_schedule

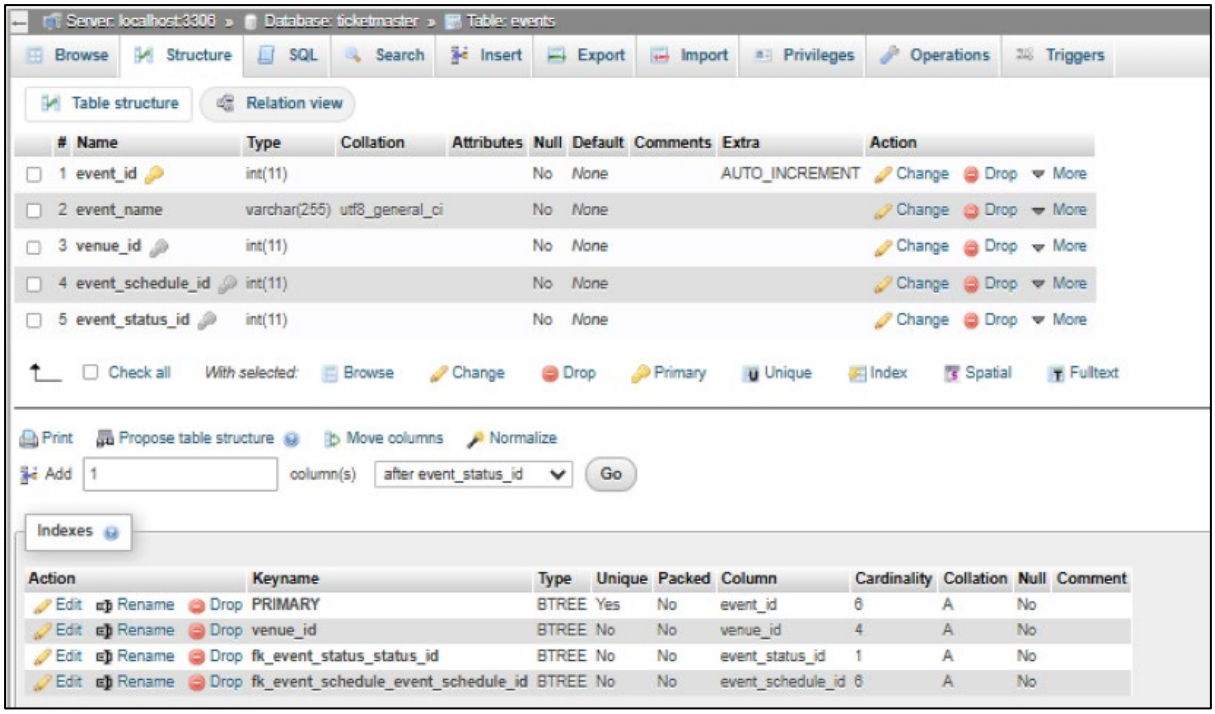

Figure 16 event_schedule structure

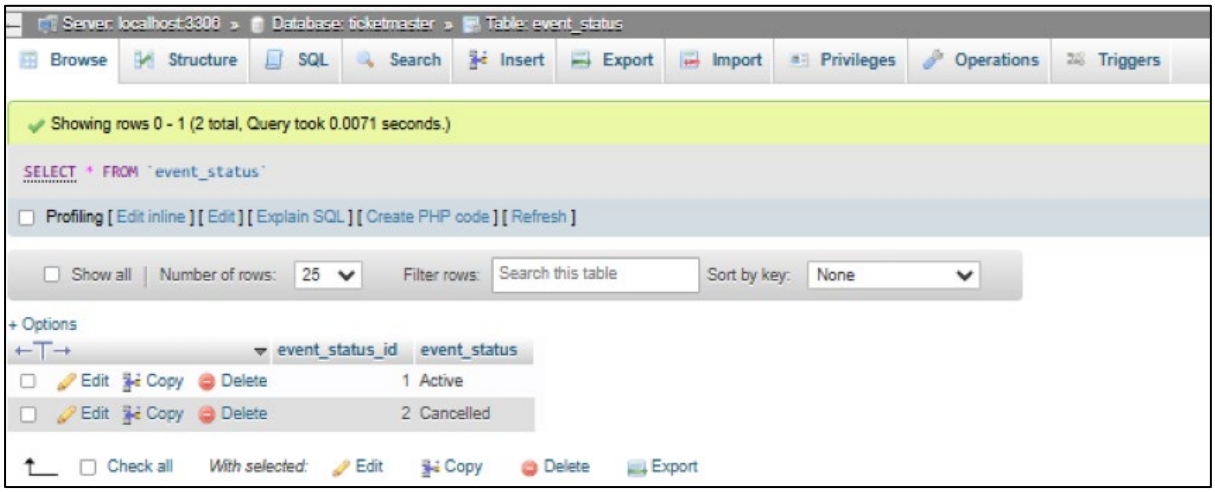

Figure 17 event__status

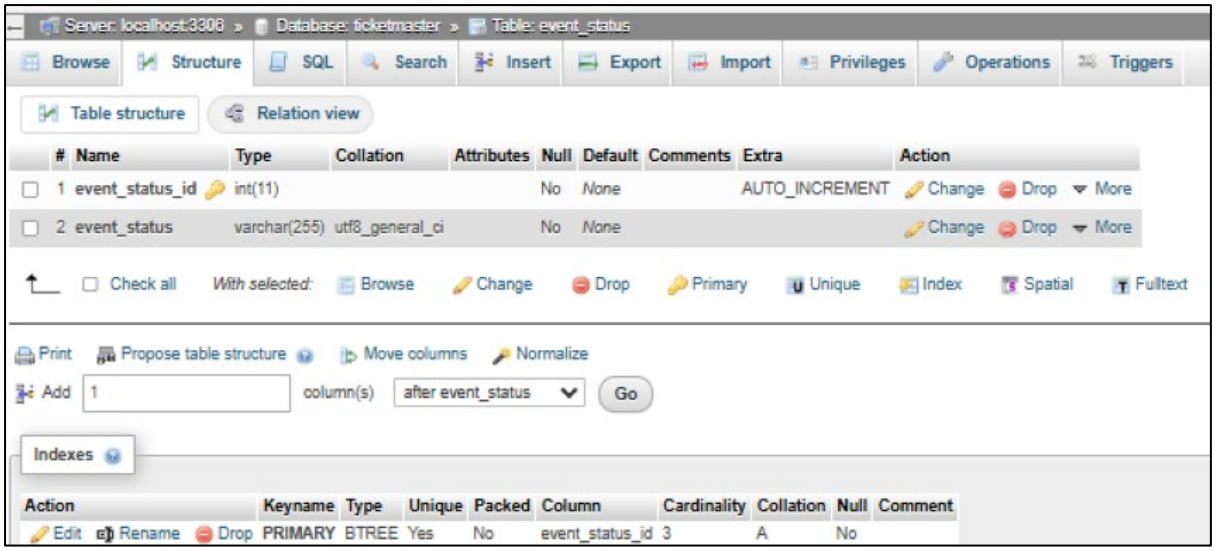

Figure 18 event_status structure

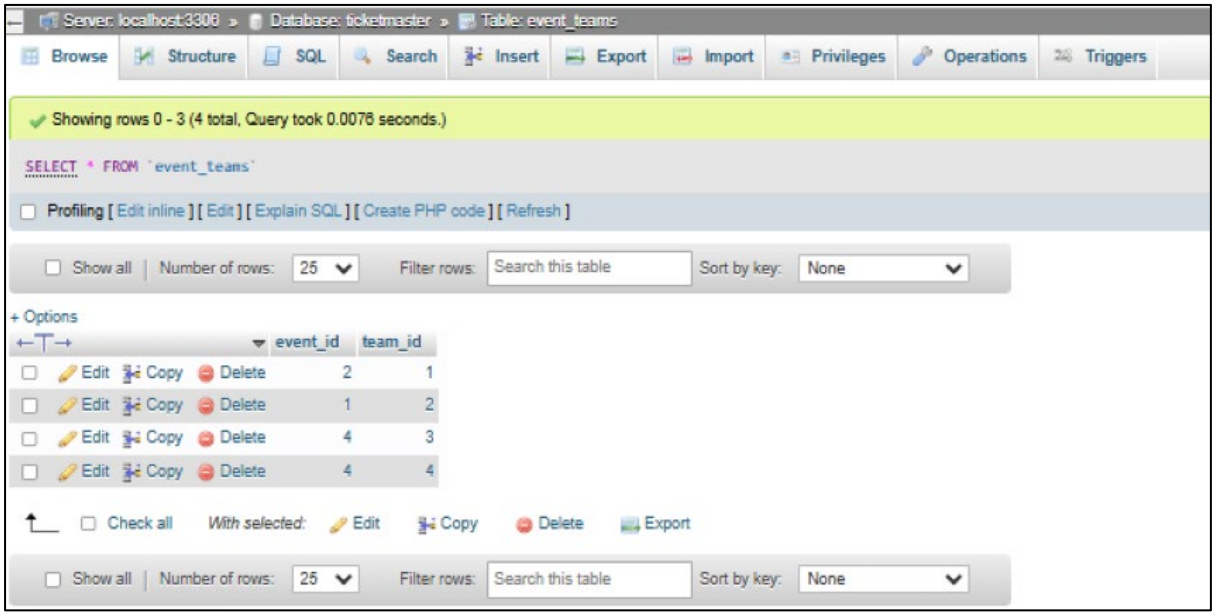

Figure 19 event_teams

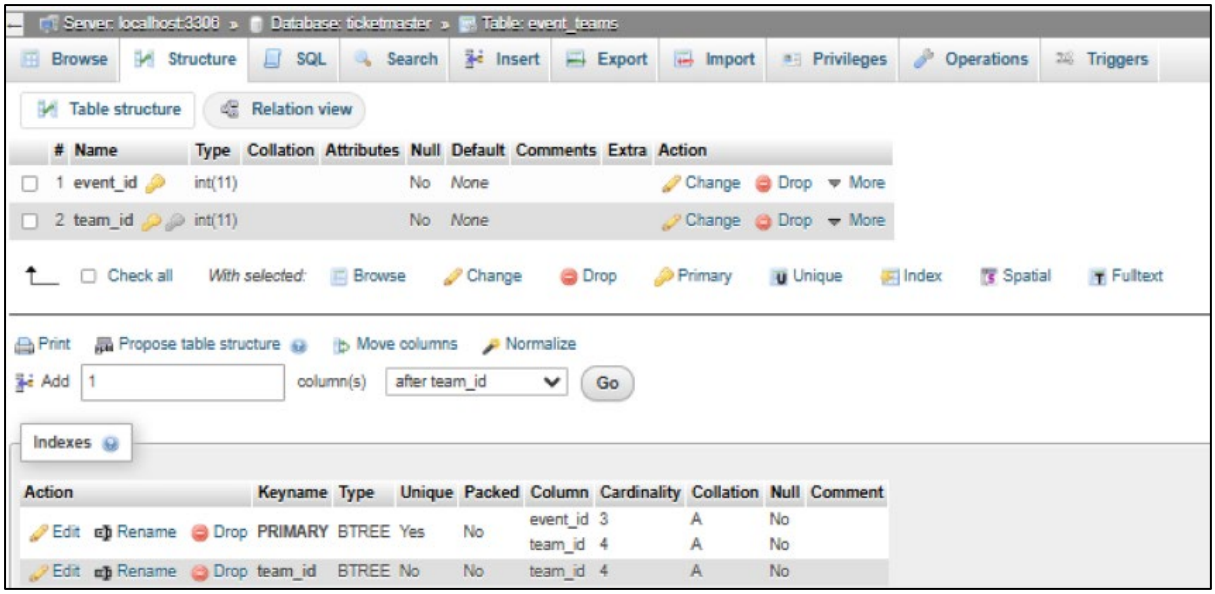

Figure 20 event_teams structure

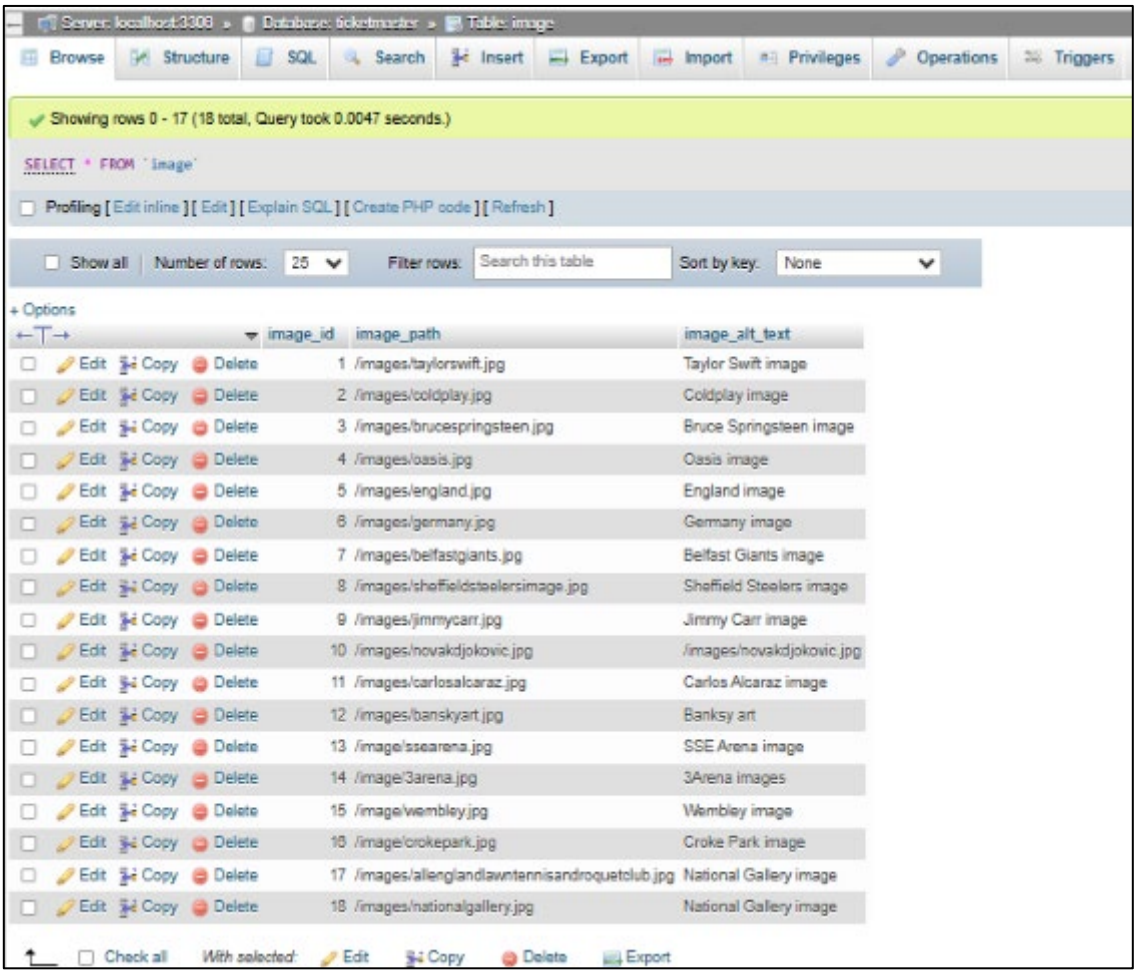

Figure 21 images

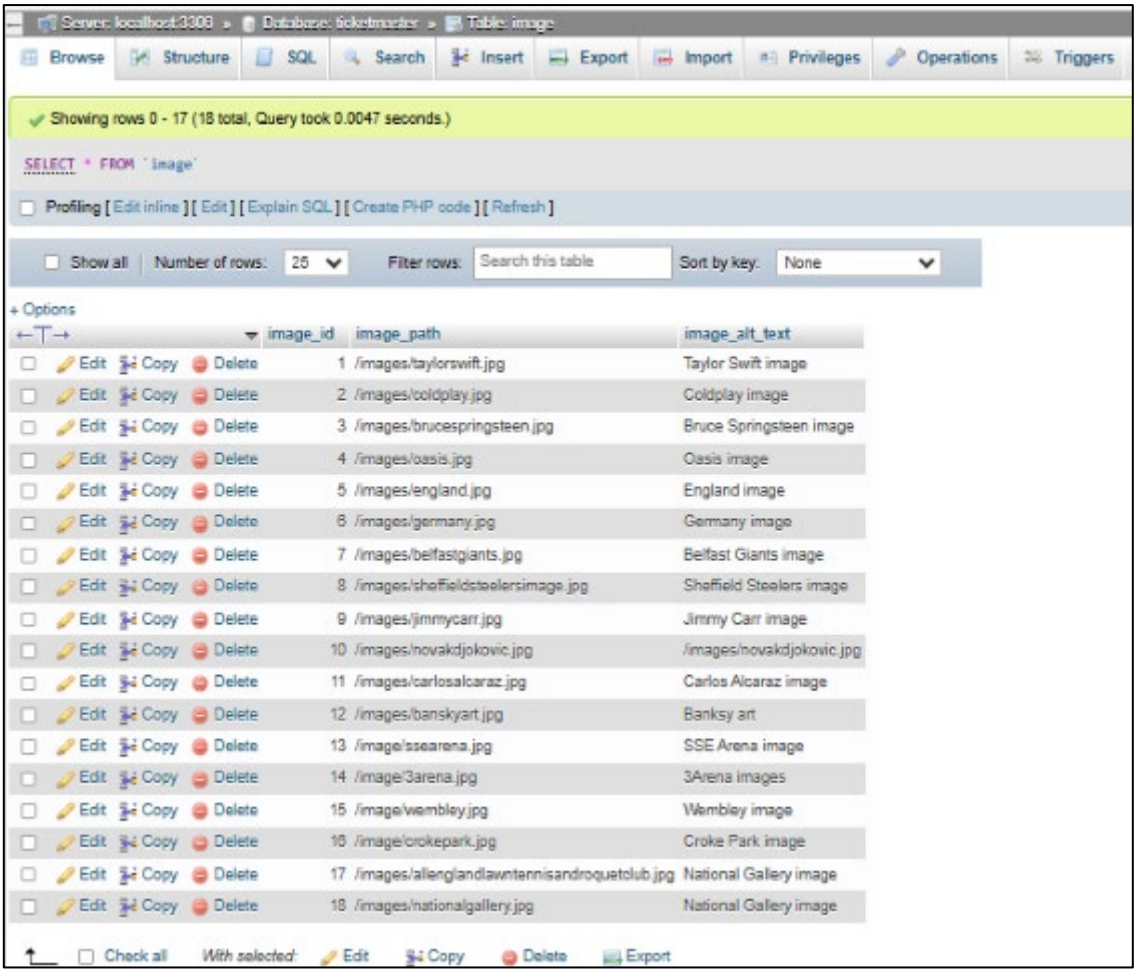

Figure 22 images

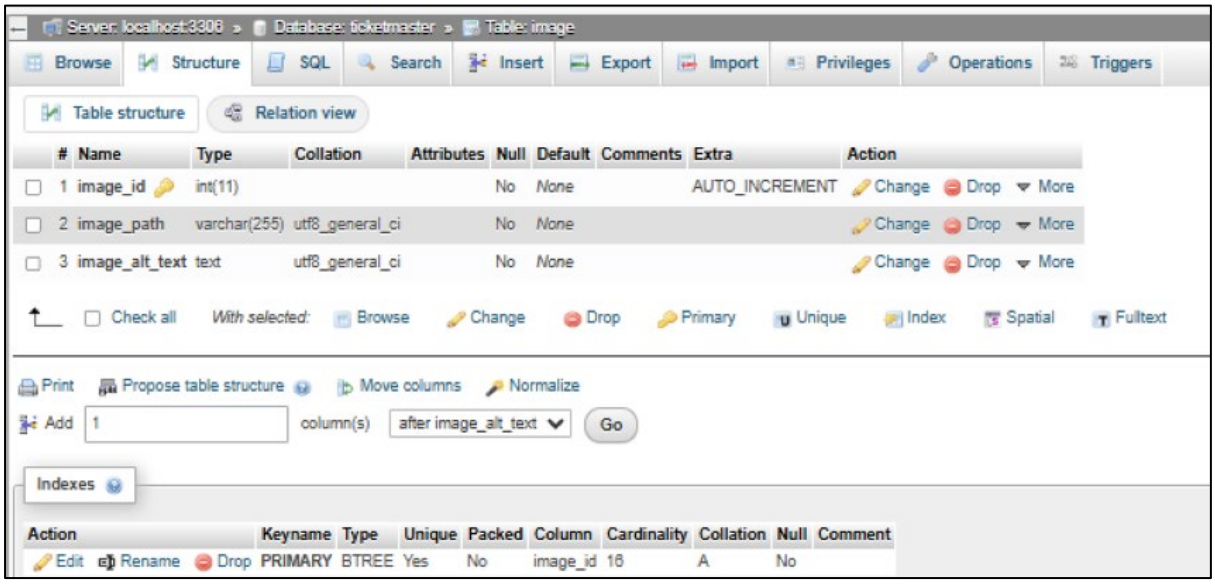

Figure 23 images structure

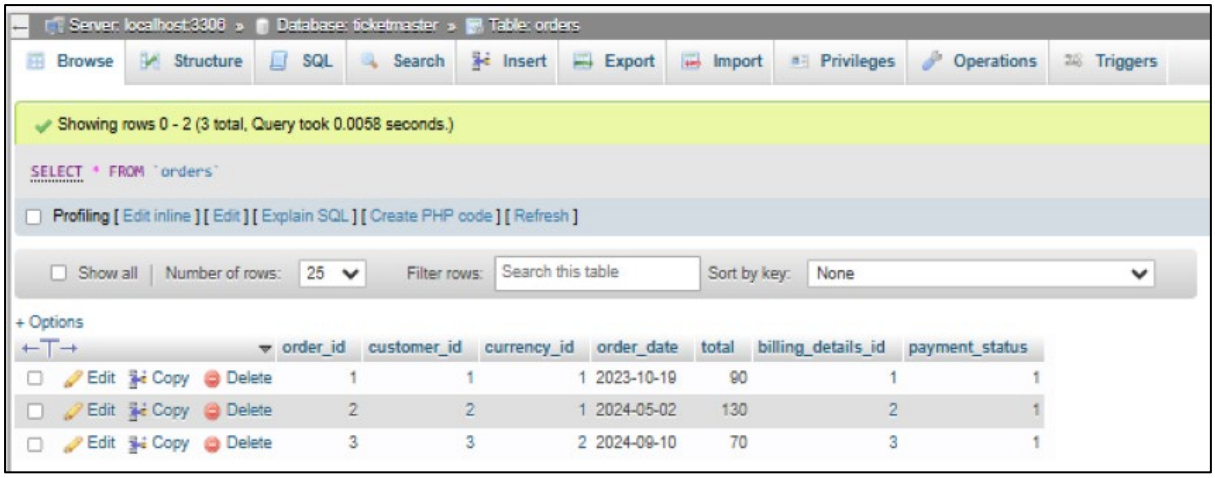

Figure 24 orders

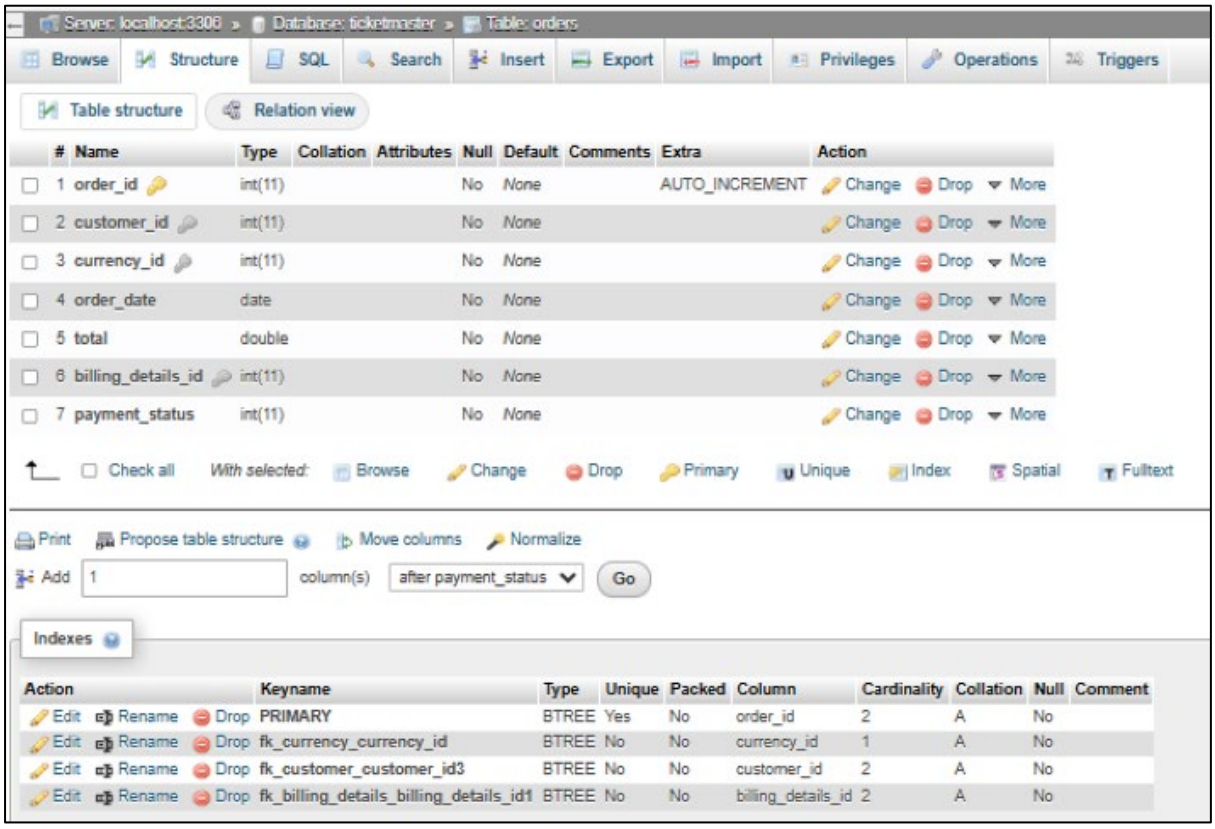

Figure 25 orders structure

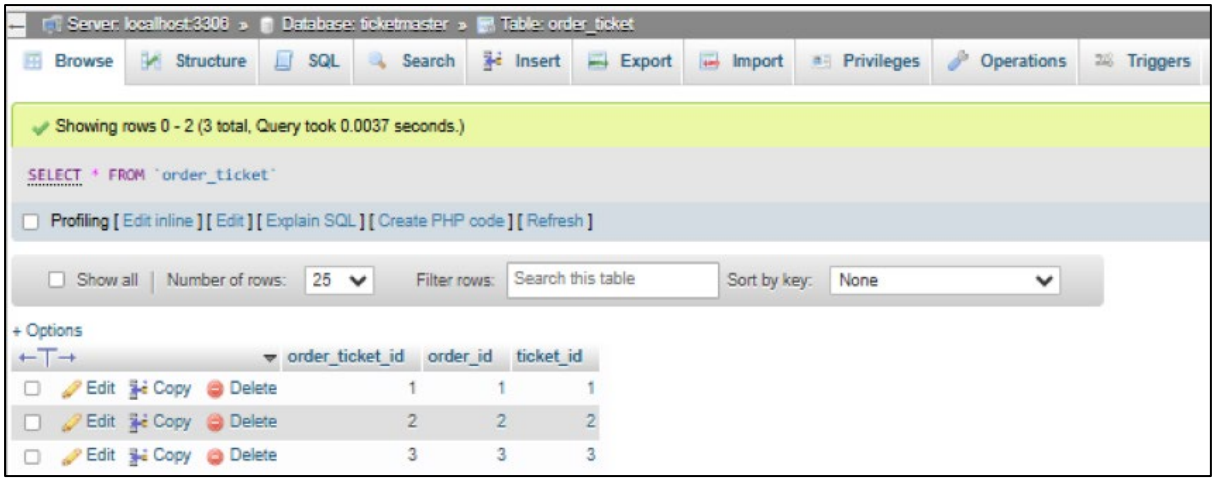

Figure 26 order_ticket

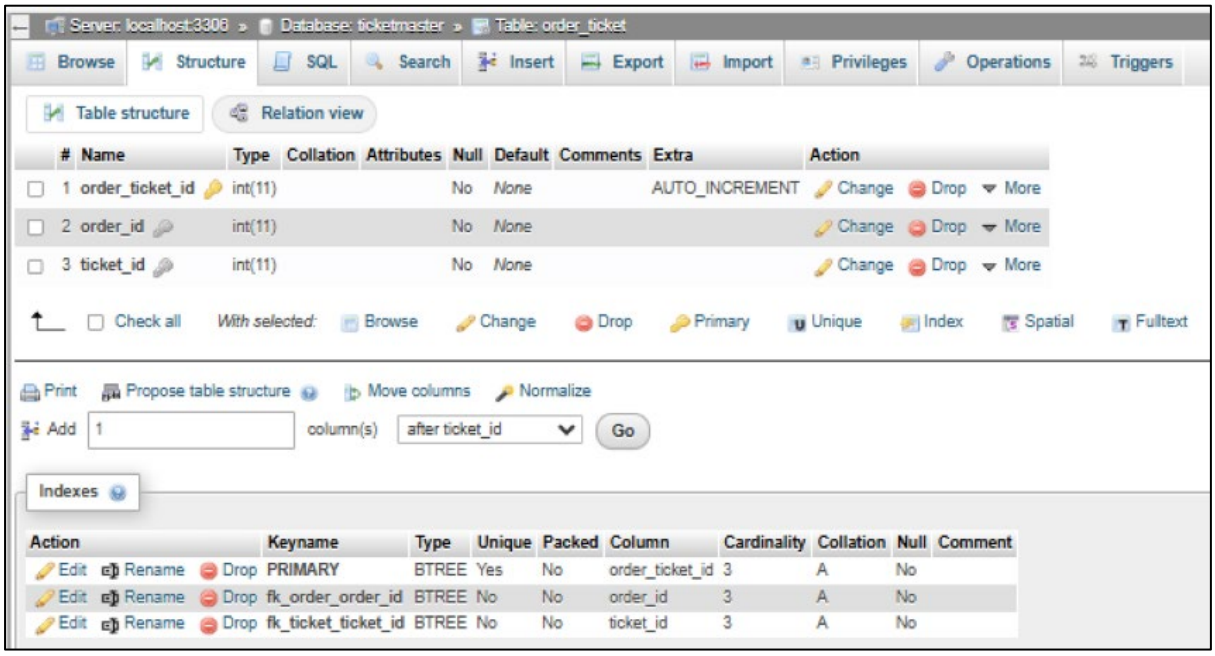

Figure 27 order_ticket structure

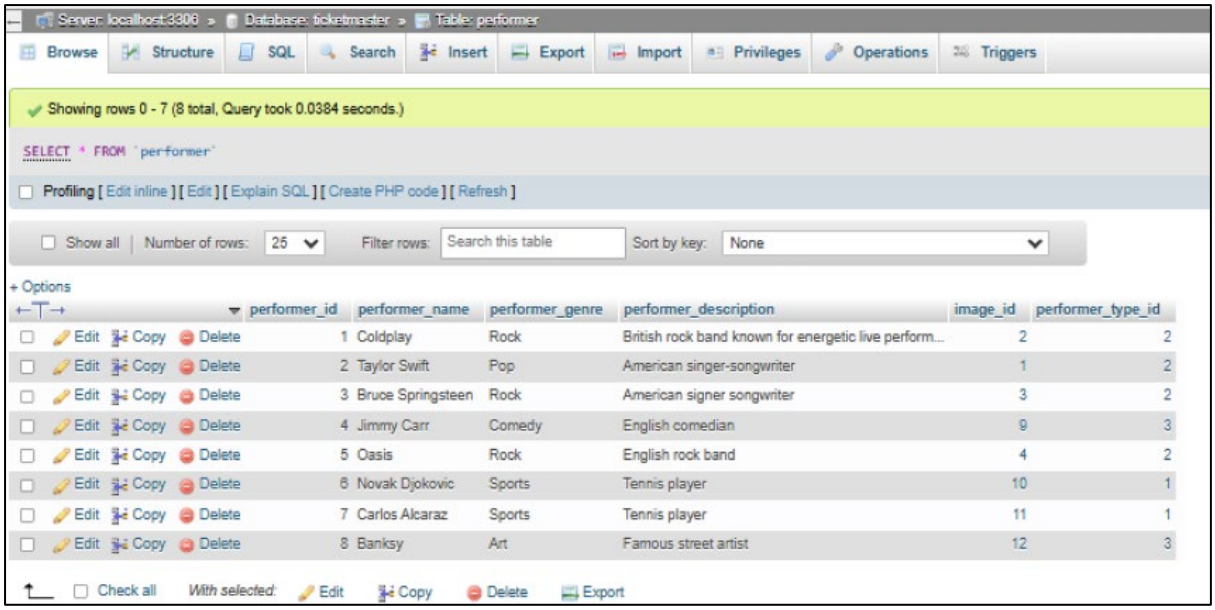

Figure 28 performer

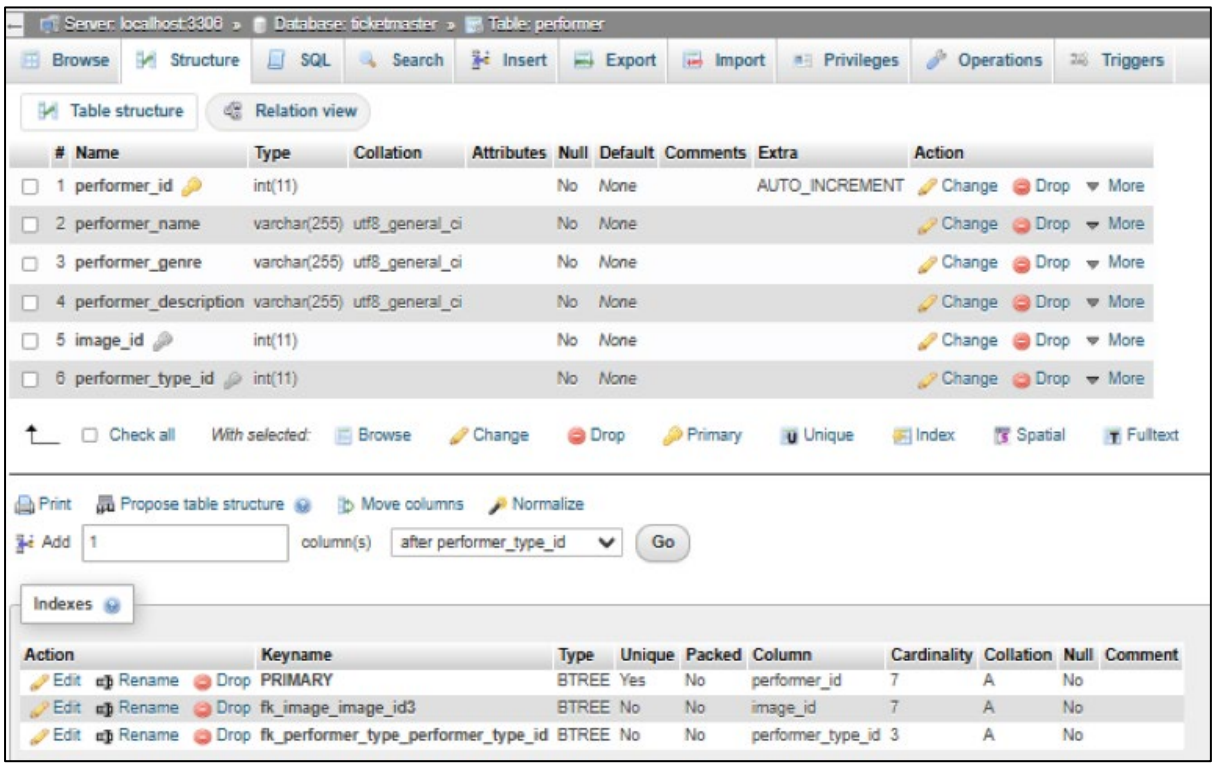

Figure 29 performer structure

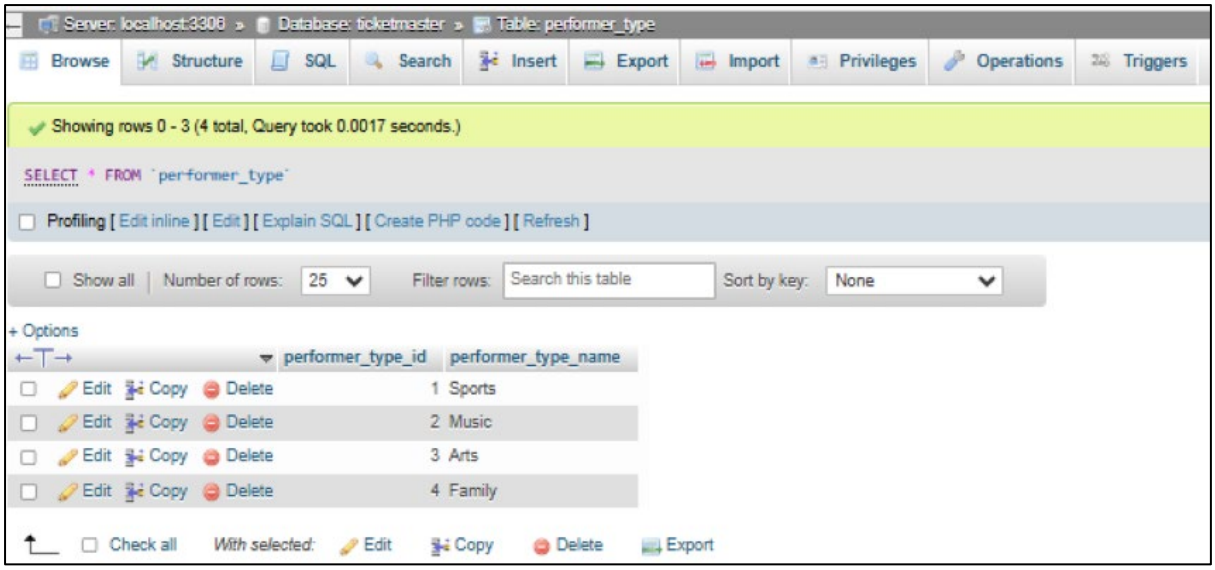

Figure 30 performer_type

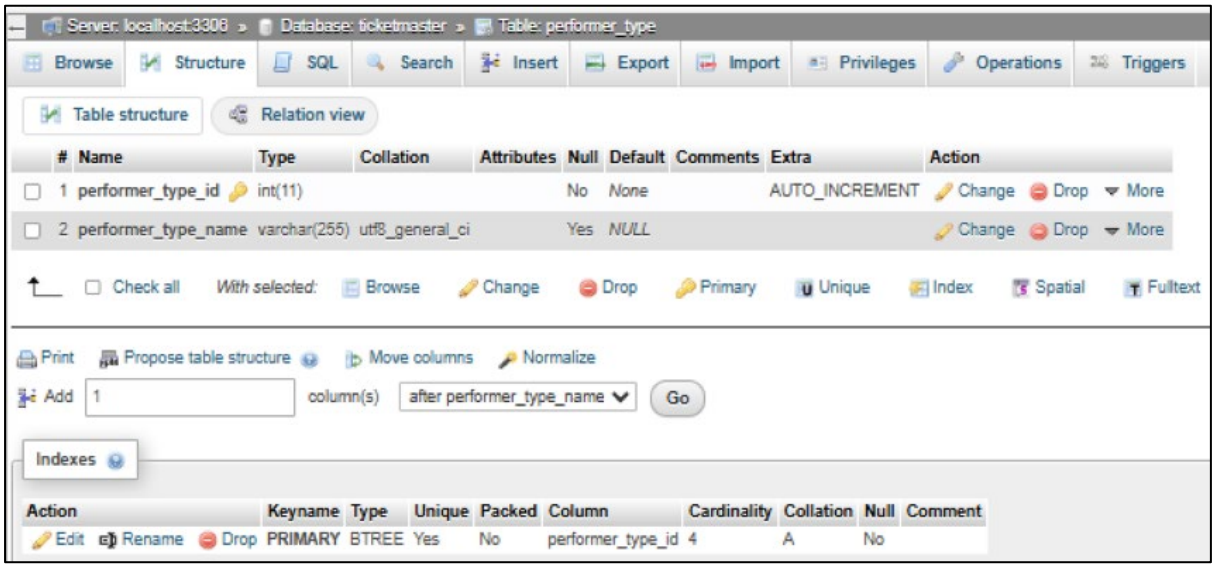

Figure 31 performer_type structure

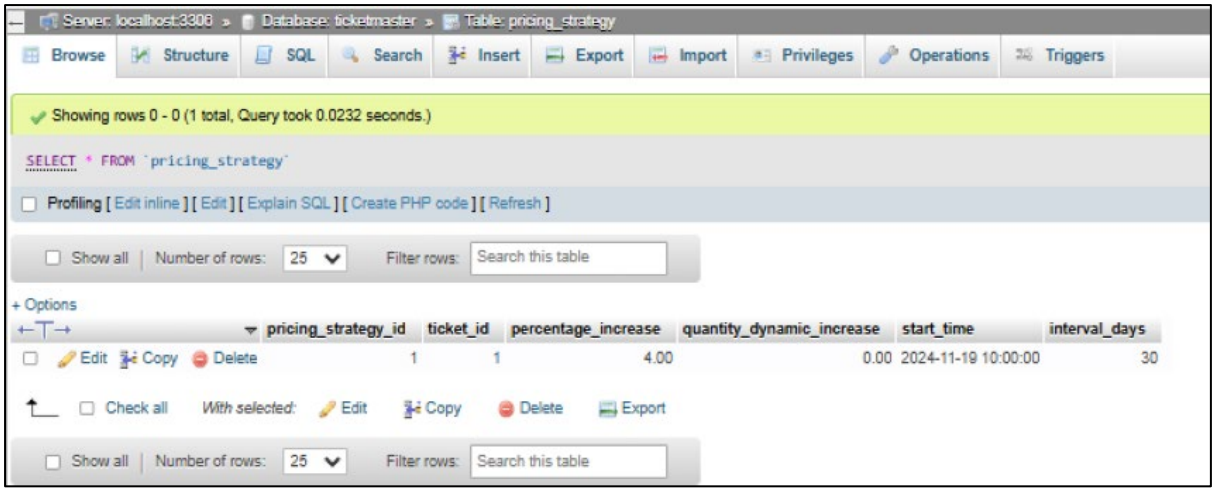

Figure 32 pricing_strategy

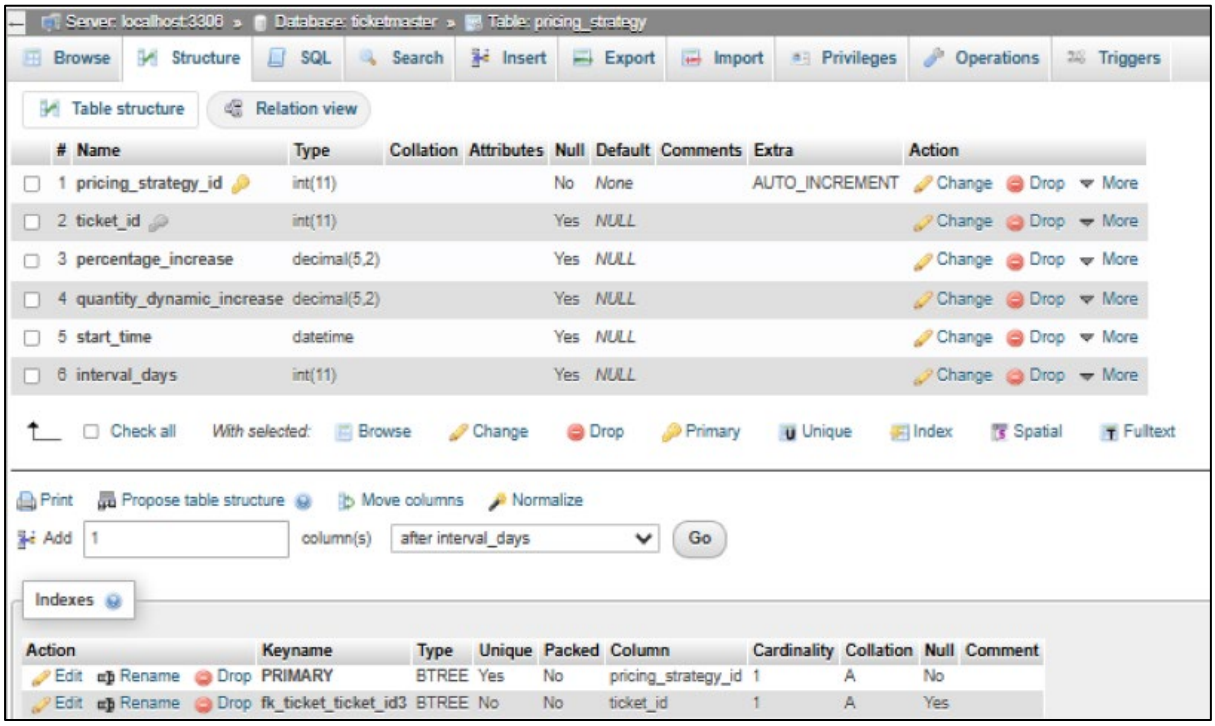

Figure 33 pricing_strategy structure

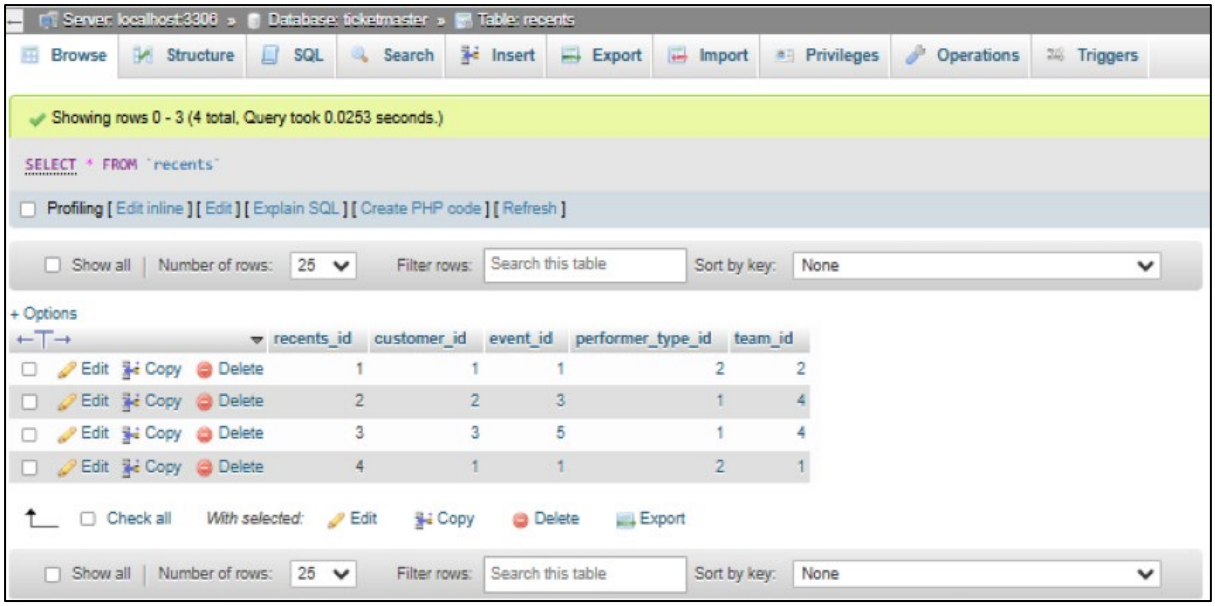

Figure 34 recents

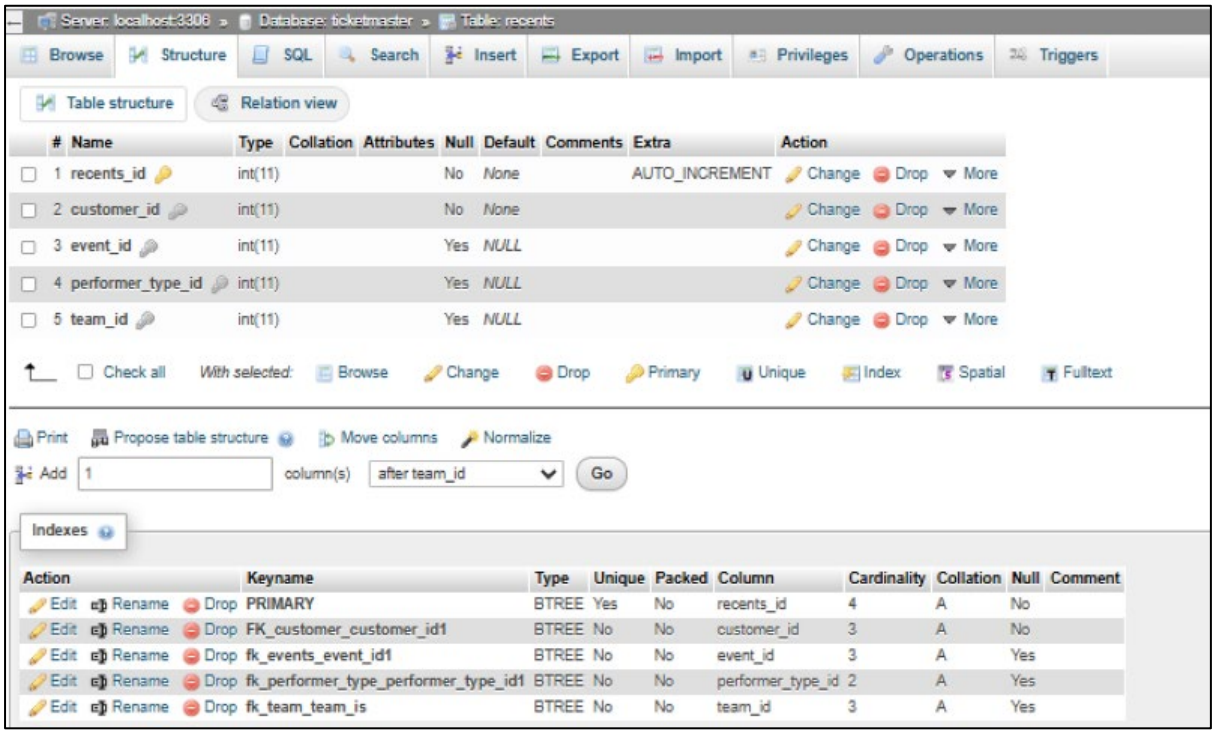

Figure 35 recents structure

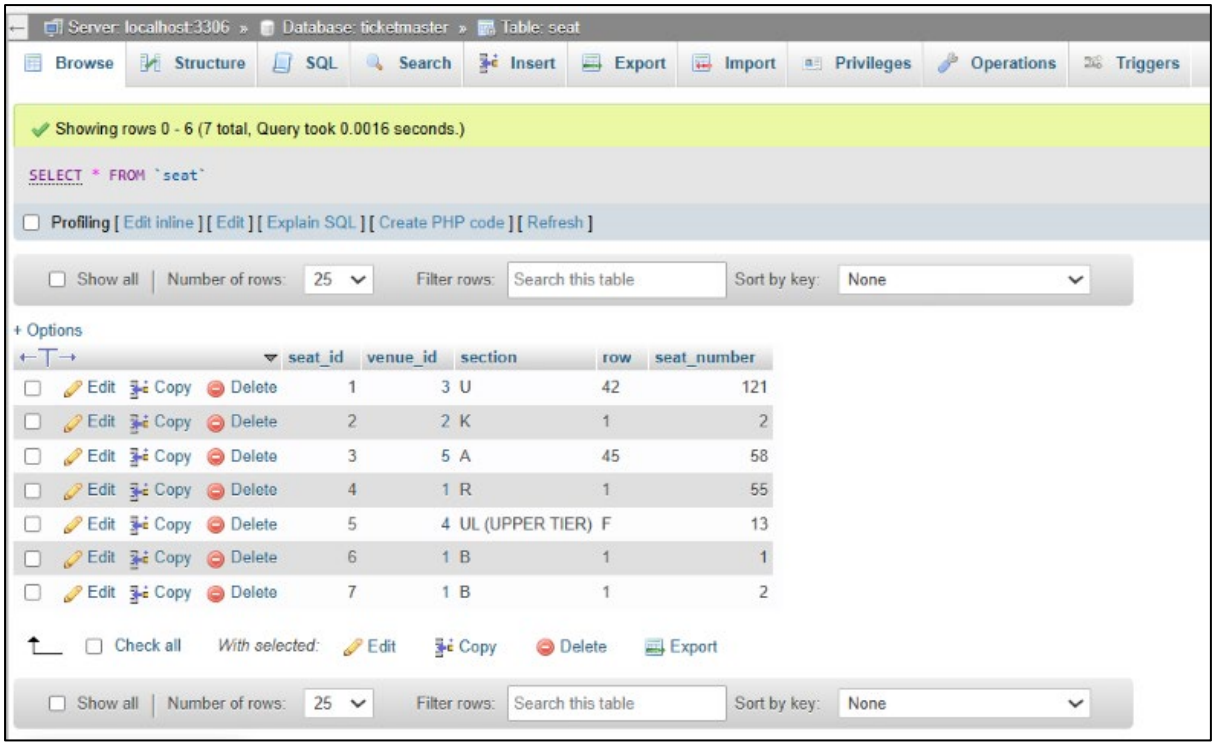

Figure 36 seat

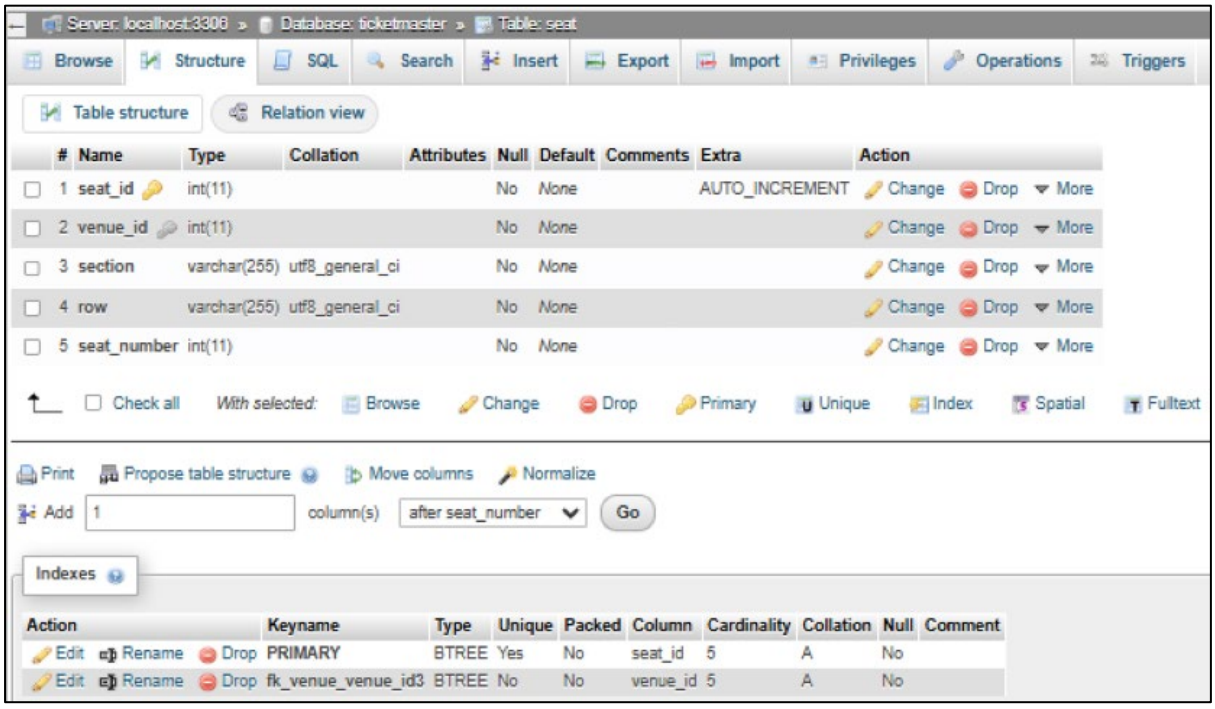

Figure 37 seat structure

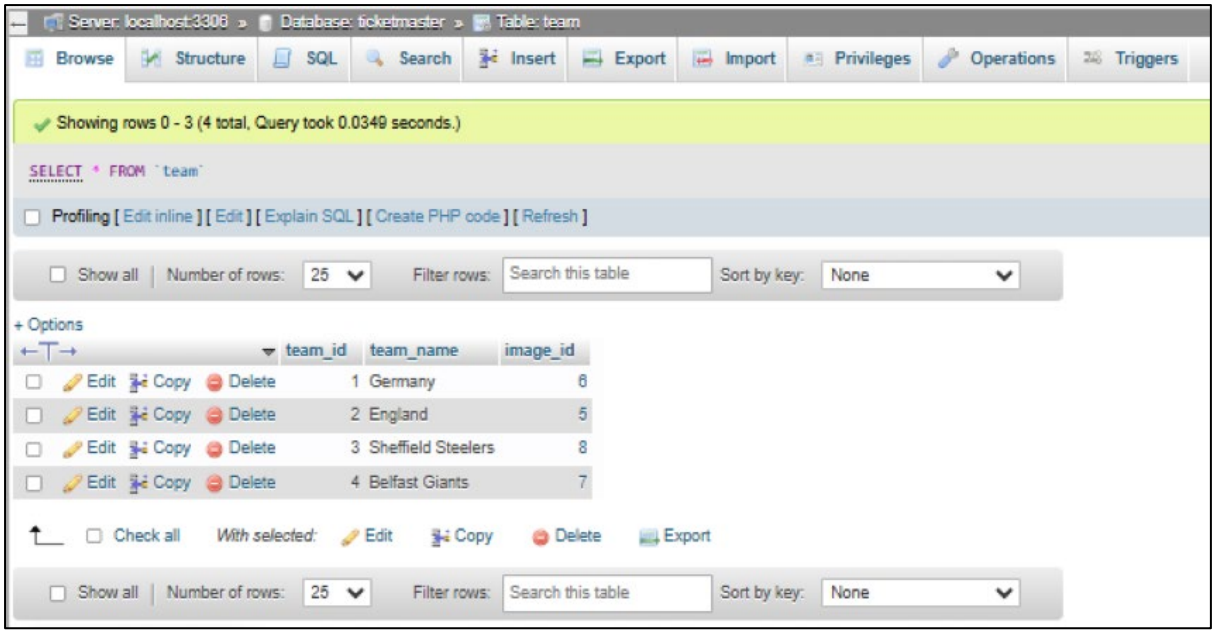

Figure 38 team

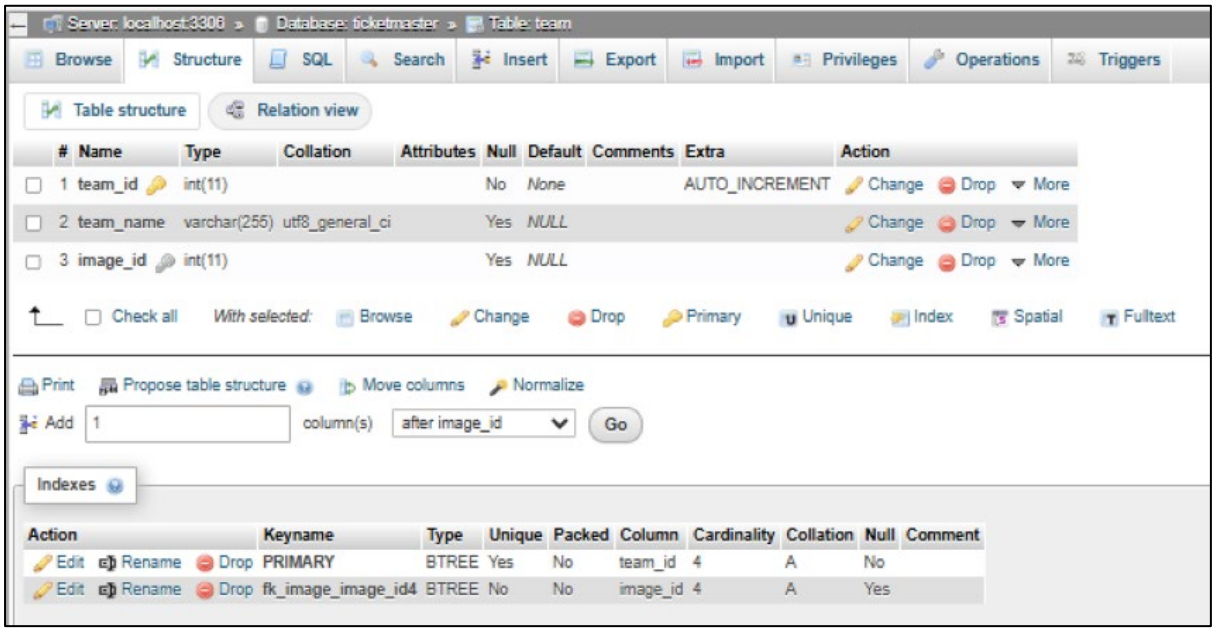

Figure 39 team structure

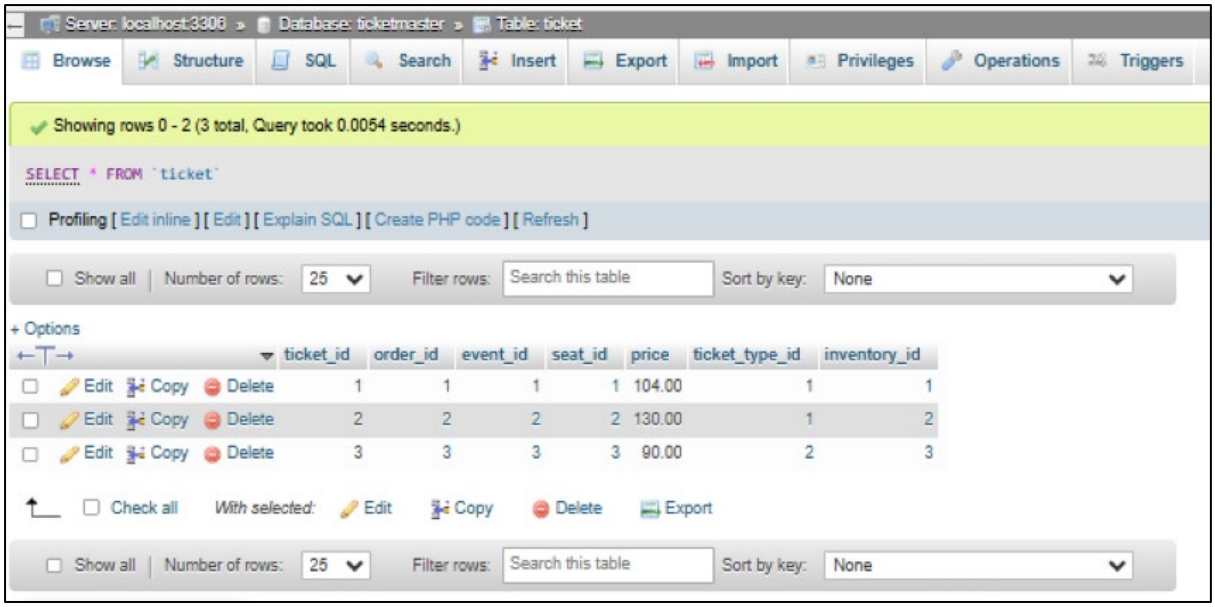

Figure 40 ticket

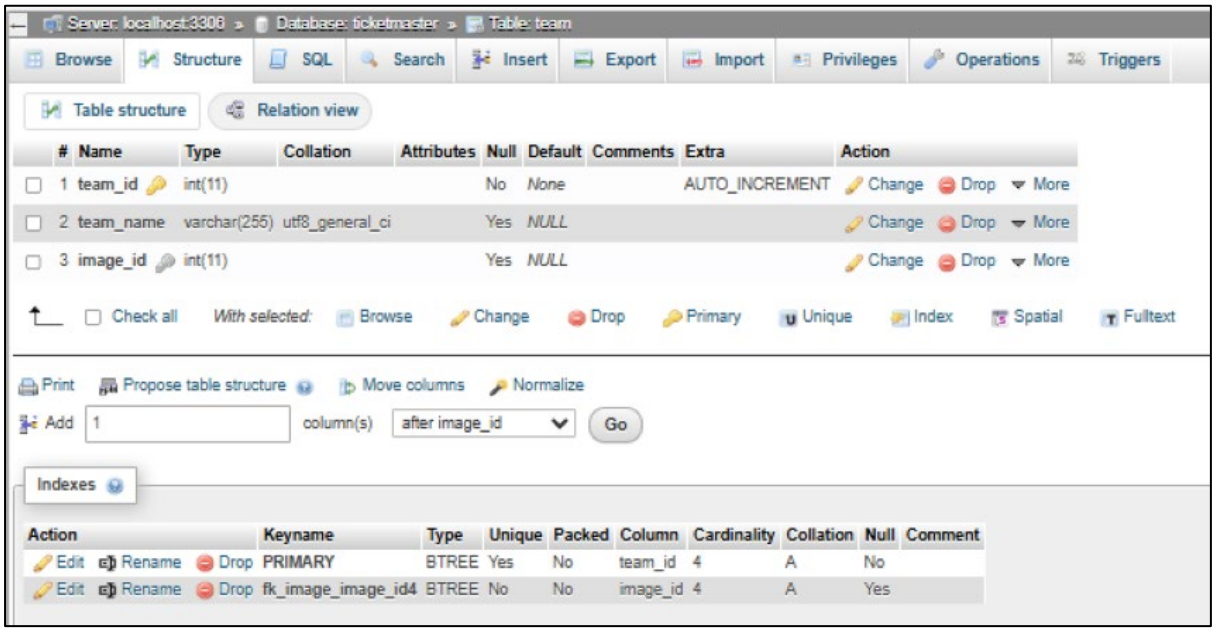

Figure 41 ticket structure

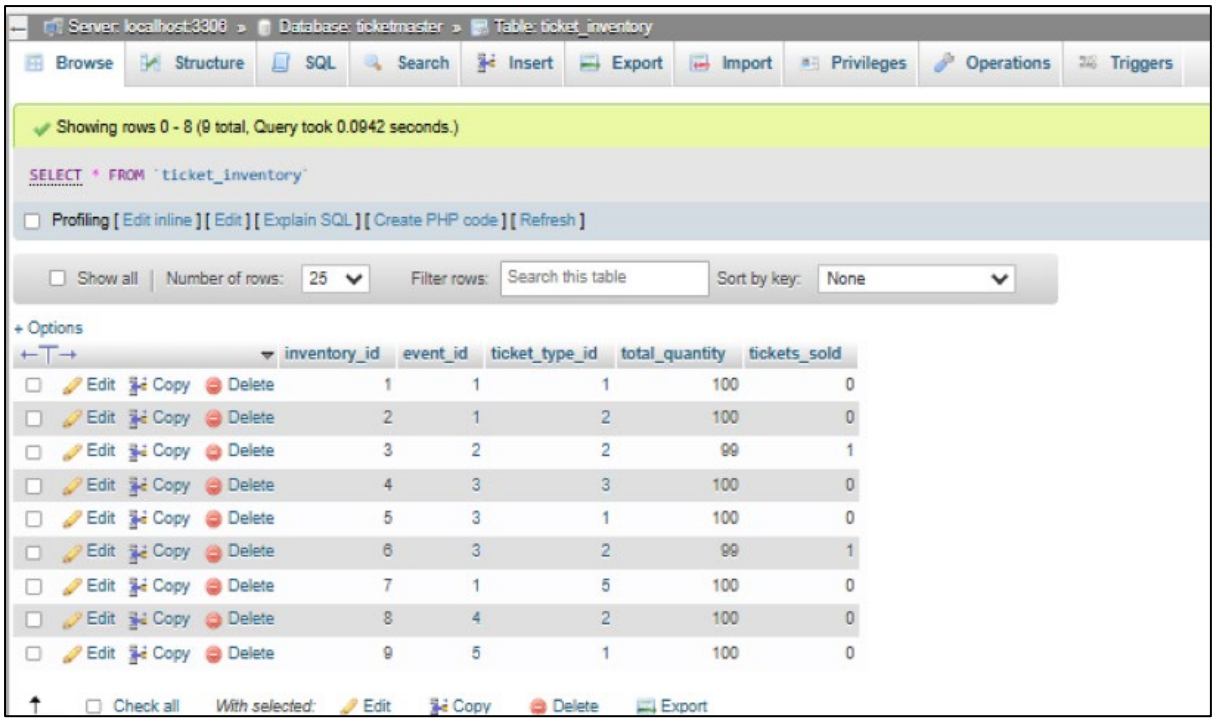

Figure 42 ticket_inventory

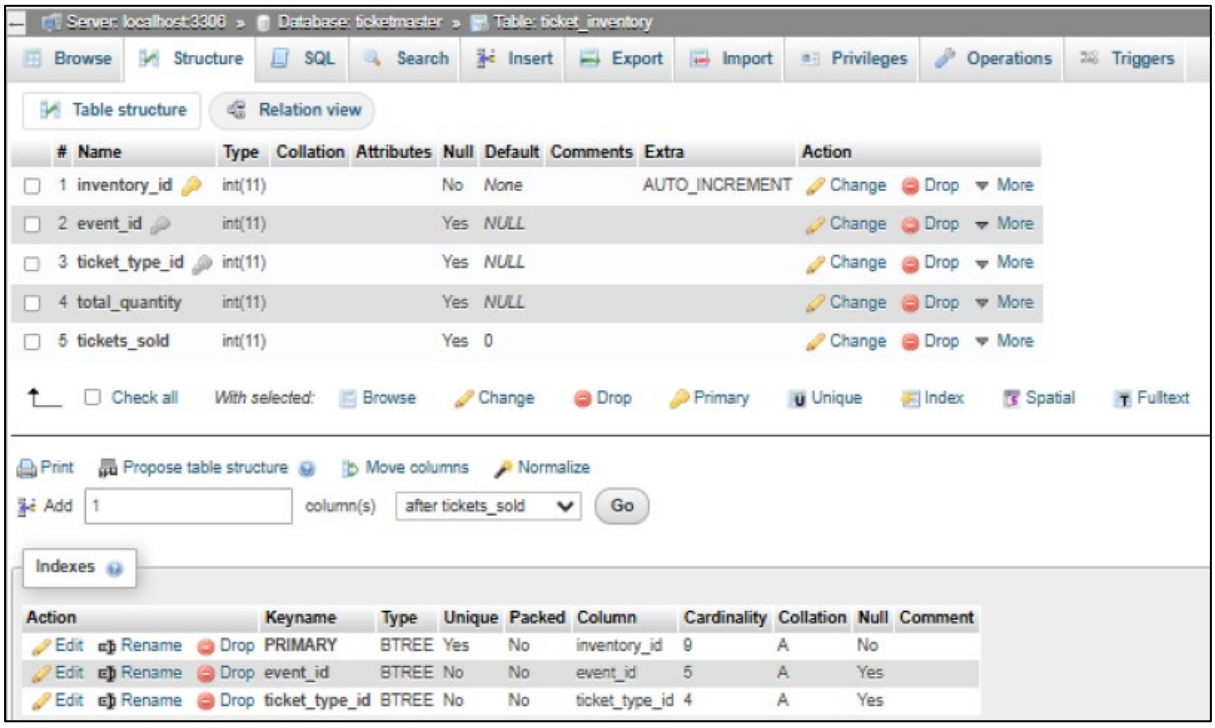

Figure 43 ticket_inventory structure

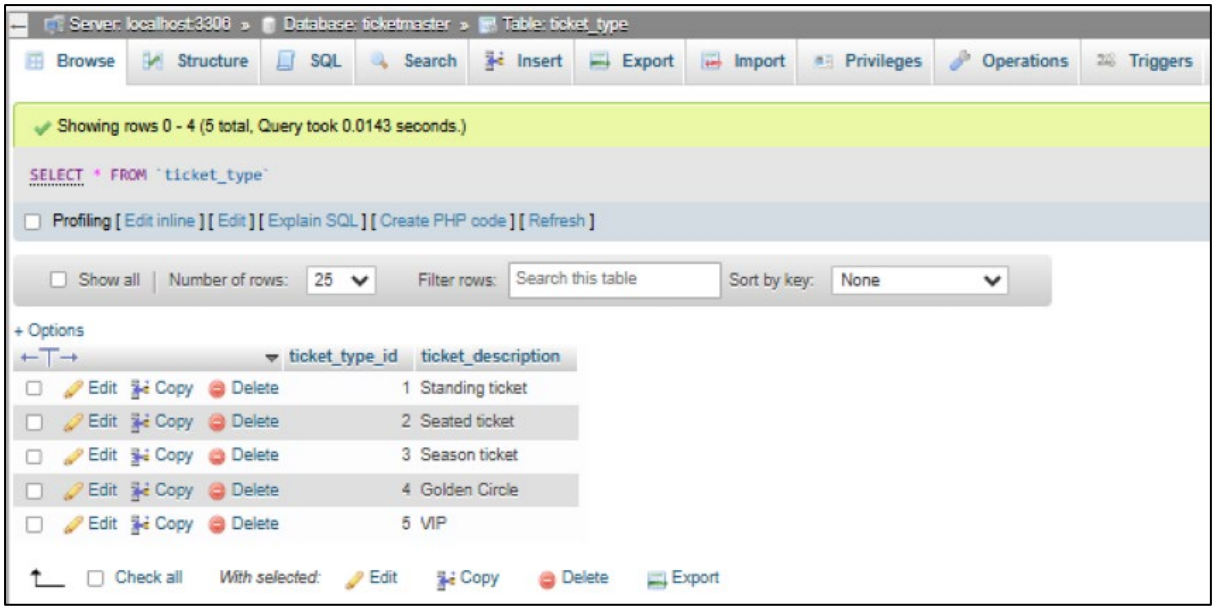

Figure 44 ticket_type

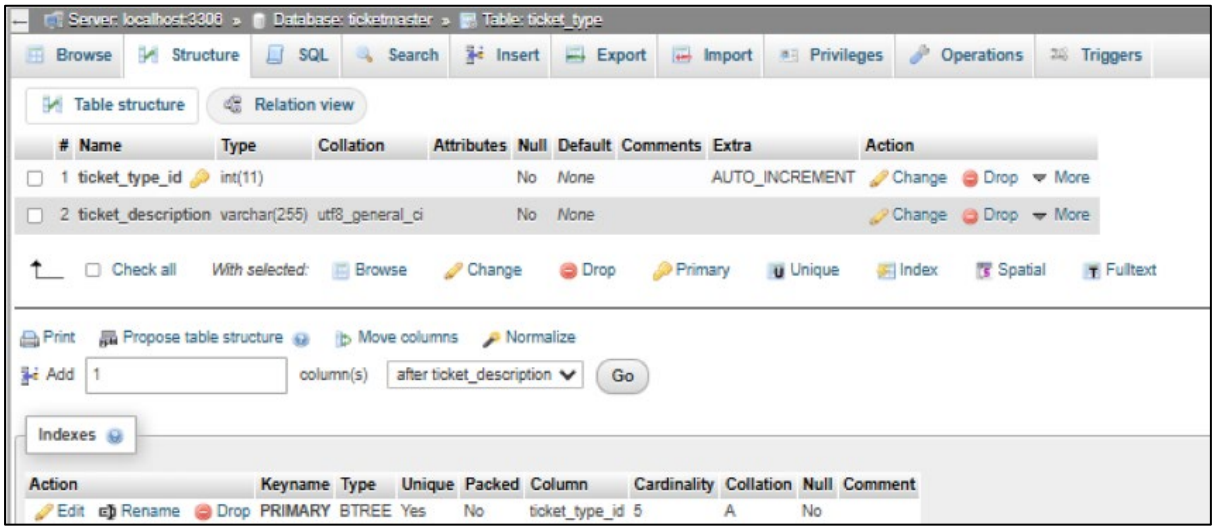

Figure 45 ticket_type structure

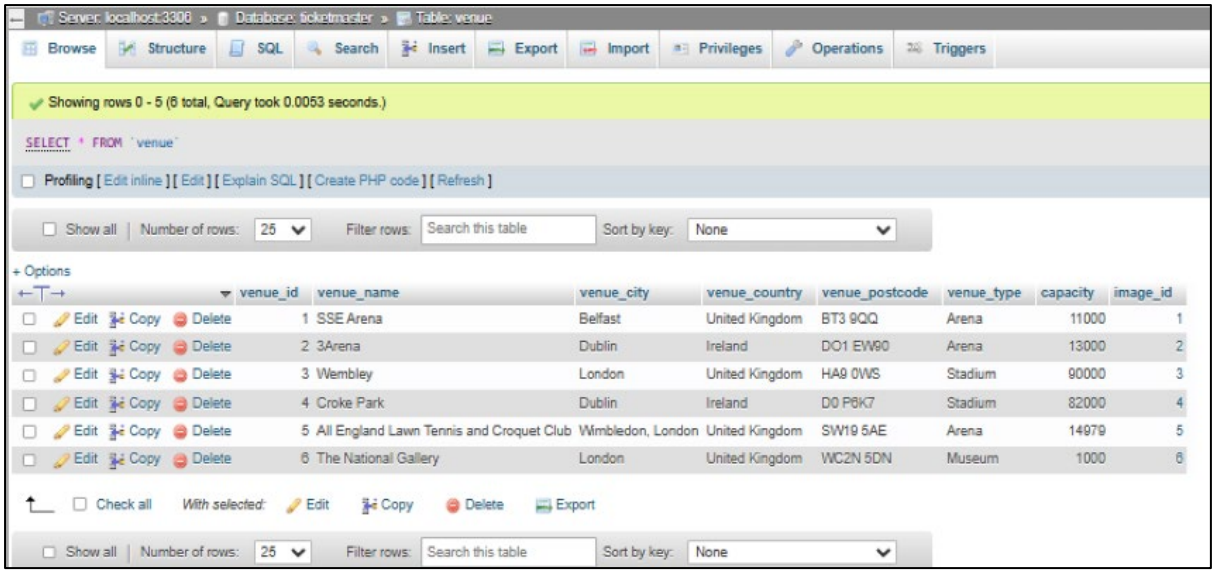

Figure 46 venue

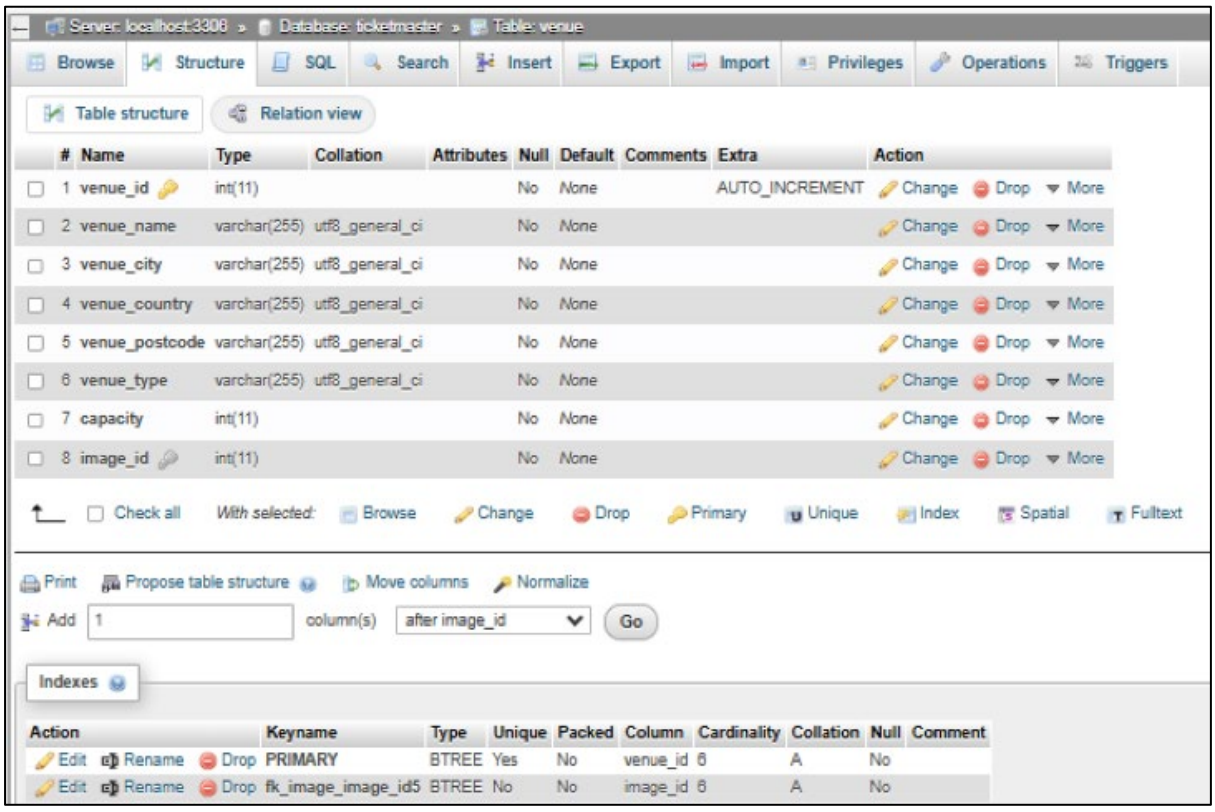

Figure 47 venue structure

SQL

The following SQL code has been assisted by the use of lecture notes and ChatGBT conversations.

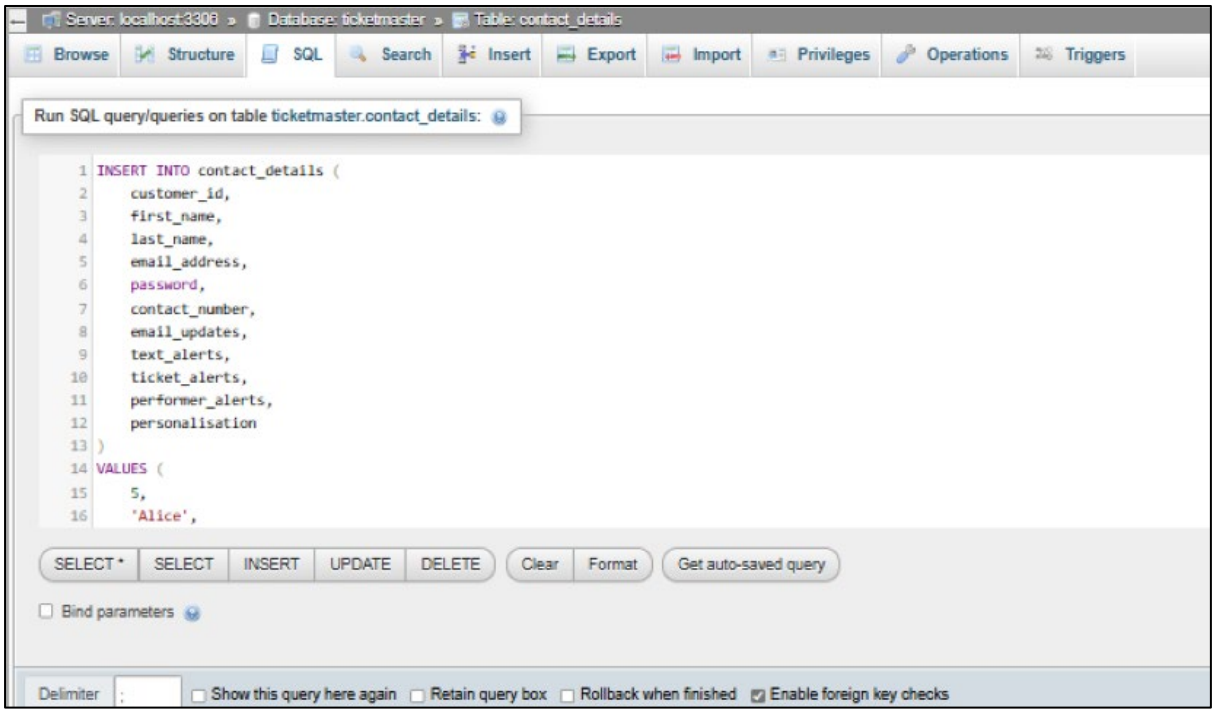

Figure 48 Creation of new contact details

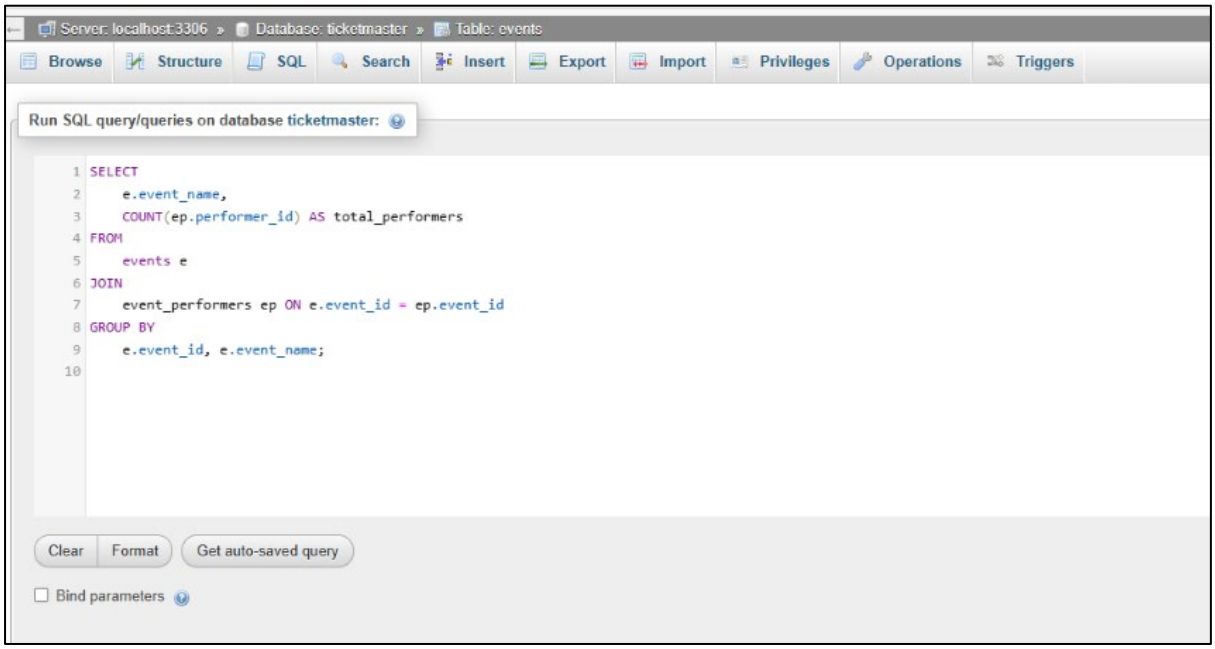

Figure 49 Count Performers Participating in Each Event

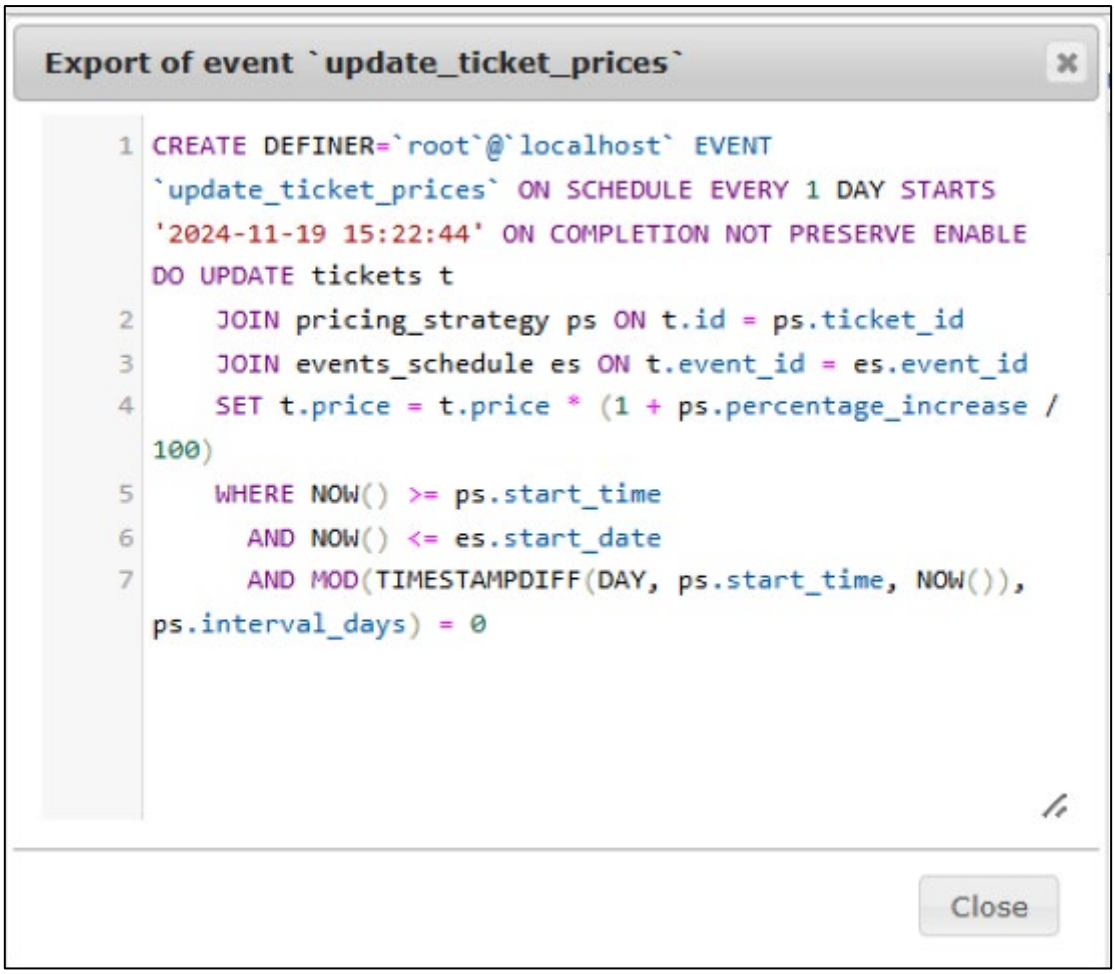

Figure 51 dynamic pricing logic within event procedure

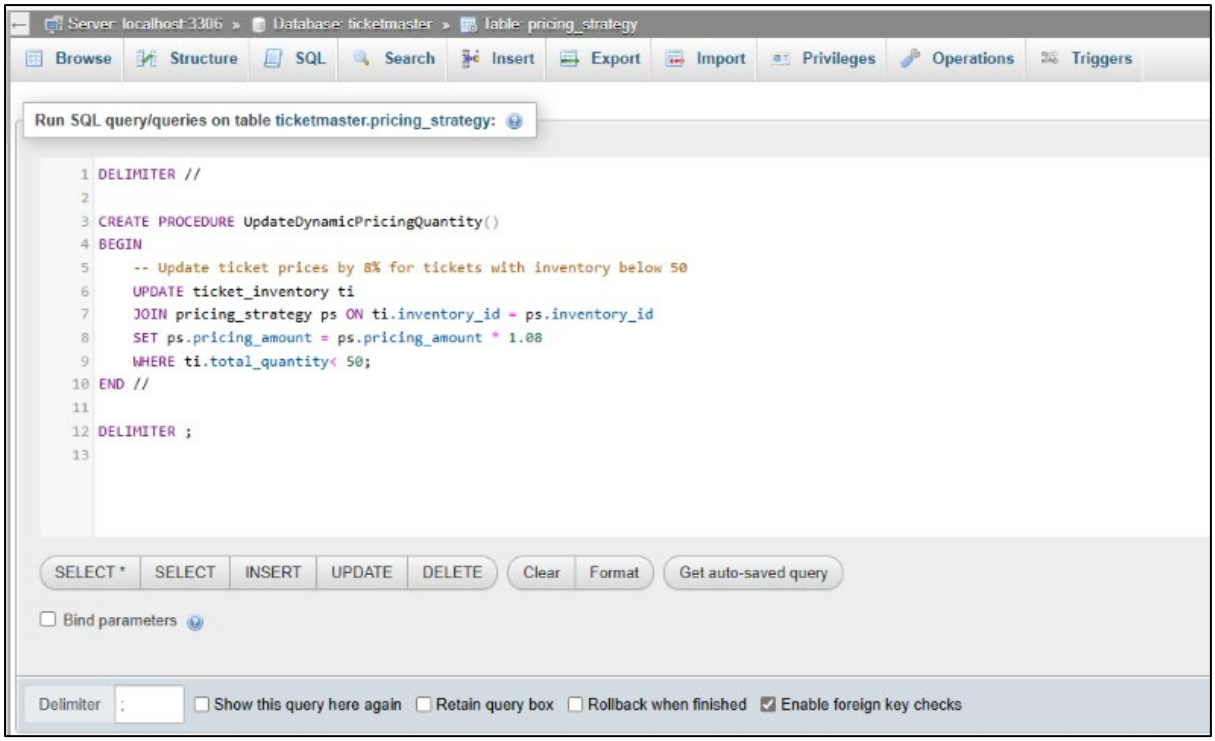

Figure 52 dynamic pricing for quantity of tickets

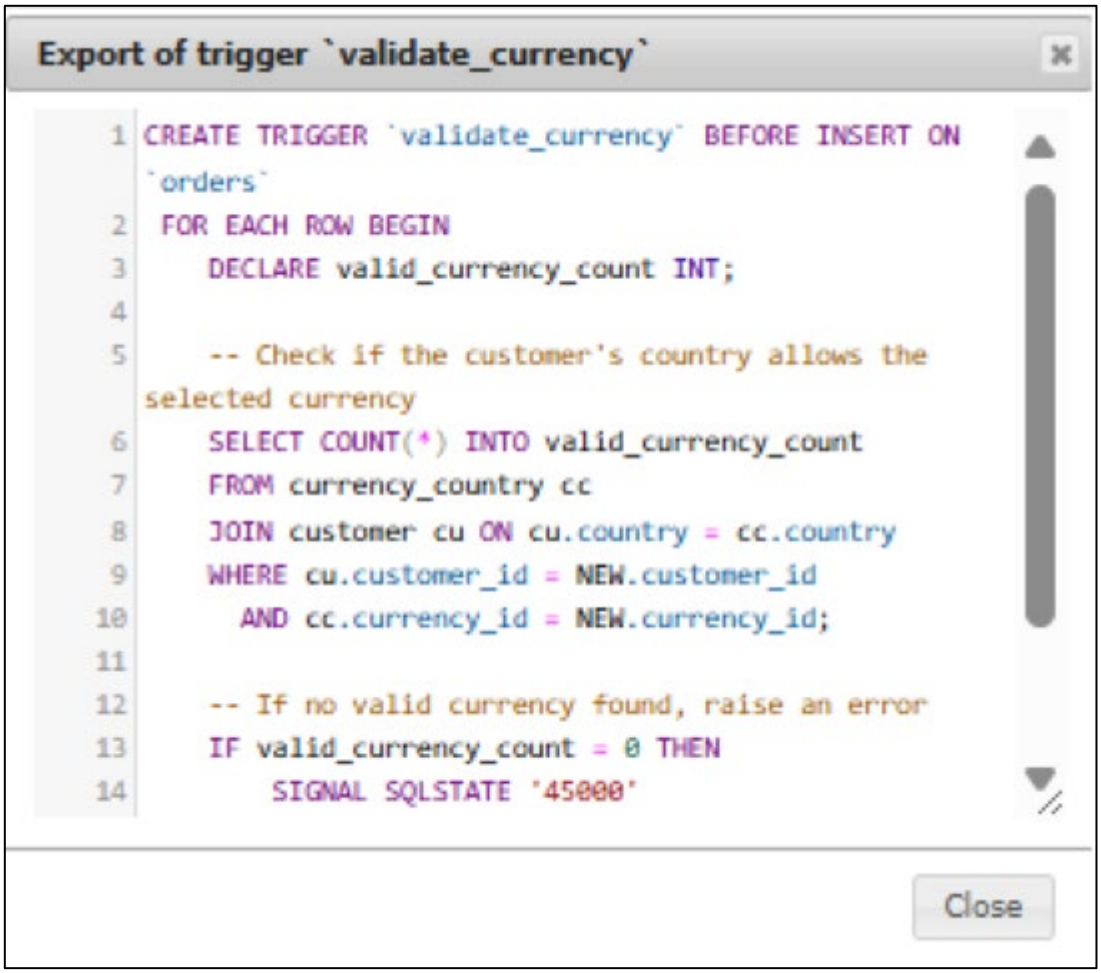

Figure 53 currency validation before order

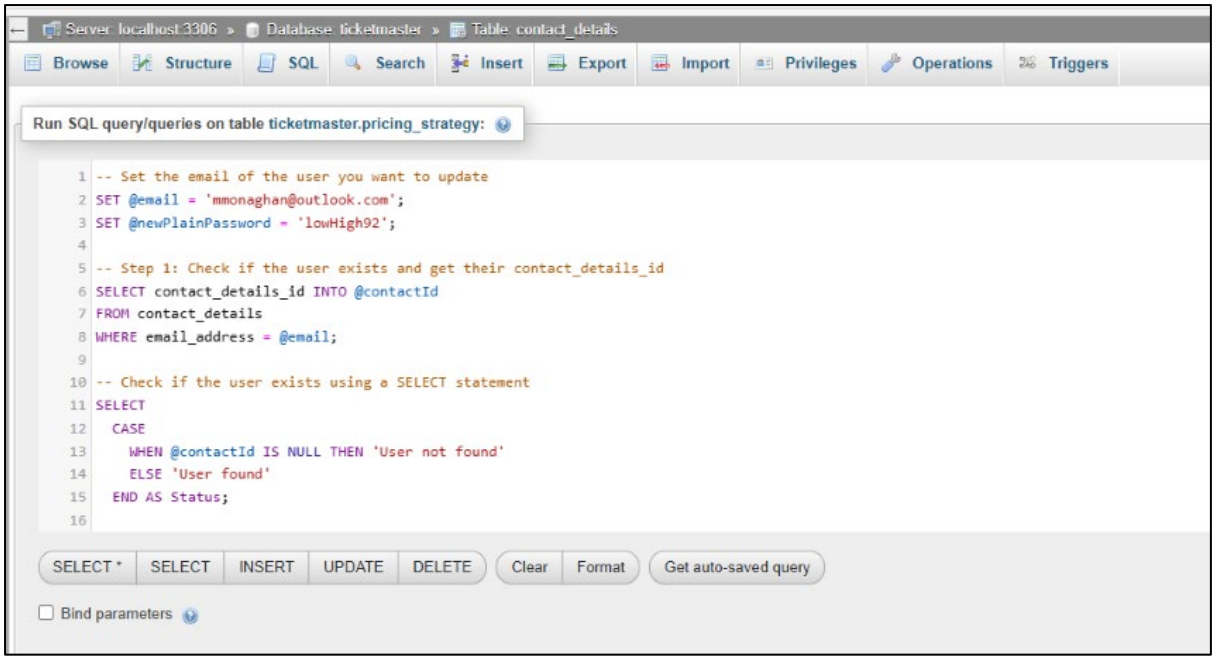

Figure 54 salting and hashing code