The design of a robust and efficient database is a meticulous process that combines theoretical rigour with pragmatic decision-making, particularly in the context of reverse engineering an existing, global and cutting-edge booking system such as Ticketmaster. The aspiration of this report is to chronicle the design process from initial assignment release to the finished database schema and to explain and justify the design decisions made during this journey before exploring the future improvements that could be made.

Overview of the Database – Initial to Final Schema Diagram

Batini et al [1] assert that *'decisions made in the early stage of database design will have lasting implications, affecting everything from storage efficiency to query performance'* and as a result it was clear that effective design choices, such as schema normalisation, indexing strategies, and referential integrity constraints, must form the backbone of the database system being created.

The first logical step in developing the database, therefore, was to create a strong conceptual model for Ticketmaster. Undoubtedly, Ticketmaster's mission is to connect customers with a reliable means of purchasing tickets and associated additional extras, for a particular event, in a particular venue, for money, while collecting information to support the company's business intelligence and marketing needs.

The analysis of this conceptual model led to the conclusion that *'trade-offs are integral to database design [2]'* and *'that a well-designed database requires careful analysis of conflicting requirements to meet both user needs and system constraints, understanding that no single design can maximise all factors simultaneously [3]'.* Resultantly, the efficiency of connecting customers, to tickets, for money became the main priority of the system, with it being decided that additional Ticketmaster features would only be implemented when the ticket booking process was optimised and on the condition that these features would not reduce this efficiency.

Entity discovery – *'the process of identifying distinct objects or concepts within a domain that have unique properties and relationships, forming the foundation for a structured and meaningful data model [4]'* – and connecting associated attributes led to the production of the first schema diagram, Figure 1, shown below.

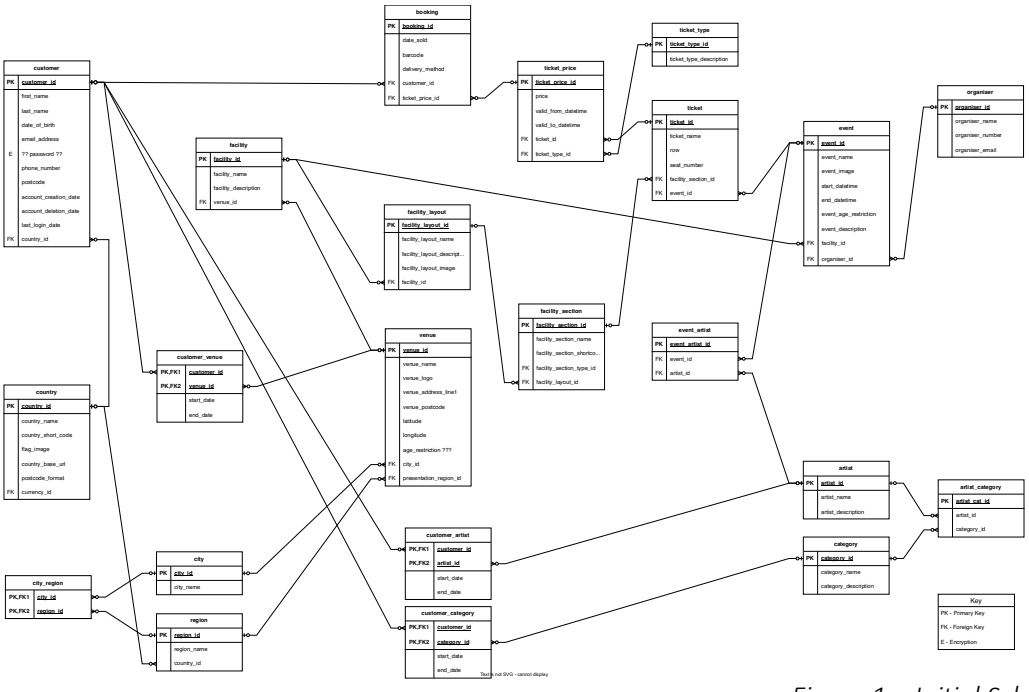

Figure 1 – Initial Schema Design

Entity discovery happened over several sessions, in both group and personal settings, with each session providing more entity tables, further attributes and further relationships between these entities. It became clear

that this process, conducted *'in an inherently uncertain environment [5]'*, would involve an iterative approach. Unlike traditional linear approaches, this dynamic methodology emphasises continuous refinement and *'acknowledges that initial assumptions may need adjustment as development progresses [6]'*.

This iterative process, which occurred over several weeks, led to discussions, debates and the addition of further features and functionality within the database. With significant module testing during this period, confidence grew that the database being created followed Ted Codd's principles for the *'relational model's structure […] and thus would allow for powerful and efficient data retrieval [7]'*, adhered to Chen's research ensuring that the relationships being defined within the database *'reflected real-world semantics rather than forcing the real world into a predefined structure [8]'* while for the most part also meeting the principles of normalisation, with the instances where these requirements were not adhered to discussed further in the assignment.

Within this period, and as a result of implementing the overarching principles led out above, the Ticketmaster database transformed from a naïve, overly simplistic design to one which is capable of handling some of Ticketmaster's most complex features, including multi-country sign up, single event ticket sales, season pass ticket sales, the purchasing of event specific additional extras, reserving basketed tickets, processing refunds and rereleasing tickets to business intelligence data production. The result can be seen in Figure 2, shown below.

Contrary to programming languages, *'where coding standards are well-established, database design lacks universally accepted style guides, leaving many syntactic and structural decisions to individual discretion [9]'*. As a result, the in-house style guide conventions devised by Dr. Neil Anderson were followed, namely, that database, table and attribute names would be written in the singular, in lowercase, with underscores when needed (no spaces) and that attribute names would begin with the table name if feasible and all names would be semantically meaningful.

Foreign key (FK) names would mirror the Primary Key (PK) name except where a renamed FK would be more semantically meaningful as seen in ***country_id*** (PK) in the country table mapping to ***customer_home_country_id*** (FK) in the customer table as this table requires the customer's home country.

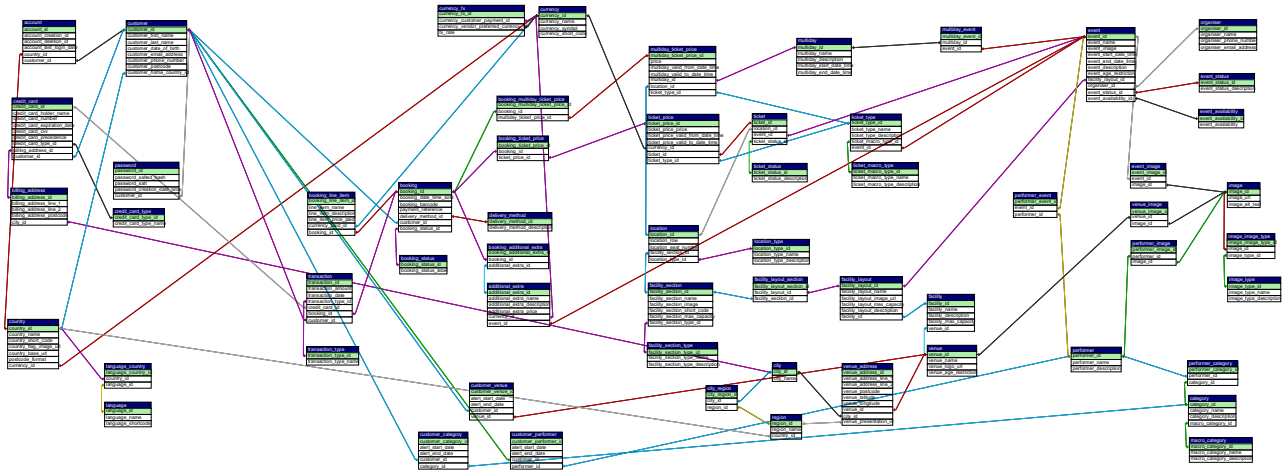

Figure 2 – Final Schema Design

Although the above schema clearly shows the stored attributes, it must also be recognised that many of these attributes were specifically chosen because they could be used to derive other attributes which would be semantically useful to Ticketmaster. These *'derived attributes represent values that can be obtained from related stored data or calculated based on other values, [10]'* but are not physically stored within the data schema.

Some examples in the database presented, include:

- ***customer_date_of_birth*** will be used to derive the customer age and therefore if they are allowed to buy tickets for an event (***event_age_restriction***) or venue (***venue_ age_restriction***).
- ***account_last_login_date*** will allow time since last login to be derived allowing Ticketmaster to adhere to their 'Privacy Policy' which states that all customer details must be deleted after the *'account has 7 years' of inactivity [11]'.*

Although these derived attributes will take some business logic at the application layer, their inclusion is justified, as derived attributes are vital for decreasing database maintenance, thus instrumental in maintaining data integrity and *'reducing data redundancy [12]'*. The inclusion of ***account_time_since_last_login*** or ***customer_age***, for example, would have enforced a requirement for daily database maintenance.

It must be noted that some attributes, such as ***alert_end_date*** for customer, venue and category alerts, were set as nullable allowing for future functionality. Not only would the ***alert_end_date*** trigger the stopping of alerts to the customer programmatically, but it would also allow for invaluable data analytics to be derived from this data – changing global trends, fan retention rates, average lifetime value for a particular performer, venue or category as well as the success of targeted marketing campaigns through the alert system.

These nuanced decisions are integral in creating a database that has *'a strong focus on the end-user experience, ensures that data organization and retrieval processes directly support customer requirements, streamlines service delivery, and enhances satisfaction [13]'*.

Design Assumptions

The following assumptions were made when reverse engineering the Ticketmaster database:

1. Ticketmaster is a multi-national company employing 6,500 people with dozens of database administrators, data engineers, and data architects with countless years of experience while this is an assignment for a university module. Although functionality, such as gift cards, customer support, help guides, and countless other features certainly do have database components, my scope will not be exhaustive and will be limited to the core functionality outlined above in the overview.
2. It is assumed that this database will have an application layer built on top of it and that business logic and validation will be handled by the application developer. Although there are some rudimentary checks, for example to mark a ticket as available, temporarily reserved or sold (***ticket_id***), there are no validation procedures in place to ensure that this ticket is not sold again or that duplicate ***events***, ***facility_sections*** or ***tickets*** cannot be created. This would be of paramount importance to ensure the smooth running of events.
3. Although payment has been coarsely modelled for demonstration purposes, with *'the frequency and severity of data breaches illustrating a systemic challenge in securing sensitive information [14]'* it is likely that Ticketmaster, as demonstrated by their 'PayPal – Preferred Payment Status', allows these to be handled by external payment providers. Although basic two-way card encryption is demonstrated, the reasons for its inadequacy as a security measure will be discussed in a subsequent section.
4. In a similar vein to payment, as *'traditional password-based authentication systems are becoming particularly vulnerable to attacks, such as brute force or phishing [15]'*, it is assumed that an external authentication provider would be used in the production application to ensure user and system security. Although, one-way card encryption has been demonstrated, its inadequacy for production software will be explained in a subsequent section.
5. A natural progression from payments is the use of currency within the database. Although Ticketmaster allows a customer to log on to another country's site and purchase tickets in their currency, this requires complex conversions based on changing exchange rates. Although crude provisions have been made for

this in the ***currency_fx*** table, it is expected that on the production site that an API, such as Fixer, would facilitate this.

6. Finally, although a cancellation process has been modelled to allow the rerelease of tickets for sale again and a refund of payment in the ***transactions*** table, this does not consider the complexities nor account for the policies that govern refunds within Ticketmaster.

Primary Keys, Foreign Key Constraints between Tables and Data Types

This section will discuss the considerations I made when choosing PKs, why I decided to use auto-increment at all cardinality levels, the importance and function of FKs, before giving examples from the database project that highlight and justify these key considerations. Insight into the data types used and the justification regarding such will also be provided.

Primary Keys

A PK within a relational database is a unique attribute or set of attributes (composite key) used to identify each row within a table. As a result, a PK never accepts a null value. This uniqueness is essential for enforcing data integrity, which prevents duplicate rows and allows each entry to be distinctly recognised. Therefore, the PKs are fundamental to the structure of the designed database, as they *'support both data integrity and efficient data retrieve [16]'.*

The PKs in the created database also play a key role in establishing relationships between tables through FKs, which reference the PK in a related table. Within the created database the PKs create a stable foundation for *'relational links across tables, ensuring each entity can be traced and referenced reliably [17]'*. Upon evaluating the importance of PKs for system functionality, it was decided that all PKs would be enforced using the auto-increment (AI) function.

Why Auto-Increment?

The use of a surrogate primary key, a key that replaces another key that could naturally serve as a PK, in the form of AI, which automatically generates a unique, sequential record for each record was vehemently debated at our group sessions.

It was posited that in the cases of a one-to-many relationship that other unique identifiers could be used. For example, under the ISO 3166-1 alpha-2 system all countries are given a unique 2 letter code, and this could be used as the PK in the ***country*** table or ***customer_email_address*** could be used as the PK in the ***customer*** table. Although, seemingly good examples, edge cases leave the database vulnerable to inconsistencies. For example, in some services unused email addresses get recirculated into the system and our system would encounter issues if the ISO 3166-1 alpha-2 system were to make any changes.

Although fully aware of the drawbacks of AI, namely scalability issues in high volume transactions, data migration challenges and the limited meaningfulness of AI, the benefits of simplicity, ensured uniqueness and efficiency justified the use of AI in the created database.

The use of AI was also hotly contested as regards many-to-many relationships. It was contended that the two FKs could be used as a composite primary key. Although this approach has the benefits of minimal redundancy, *'clear semantics [18]'* and reduced storage, it ultimately increases indexing complexity, creates inflexibility for future changes and *'creates complexity in referencing [19]'*. The design justification for using AI, in this case, was that it allows simplified referencing, flexibility and indexing efficiency,

Foreign Key Constraints

An FK is an attribute or set of attributes in a relational database that establishes a link between data in two tables. Specifically, it refers to the PK in another table, creating a relationship that allows for the integration of related data. They ensure referential integrity, which prevents orphaned records and maintains data accuracy. Within the created database FK Constraints enforce rules, such as restricting deletion or updates of referenced records, thereby safeguarding data integrity.

Database Examples

The utility of these links, along with the design justifications will be highlighted in the following examples. The construction of our overall venue set up allowed for the use of FKs to set up several cardinality types.

Venue - The ***venue*** table provides an example of how a one-to-many relationship has been used to create flexibility in the system and to mirror the real world situation that we are modelling. Within this set up, a venue can have multiple facilities. This allows for our database to manage venues with more than one facility. An SQL query on the facility table confirms this, showing that The RDS has 5 facilities, namely Shelbourne Hall, Simmonscourt Hall, Concert Hall, The Dodder Suite and the RDS Rugby Arena. As a result of this connection these facilities can all be used independently from the overarching venue.

Facility Layout Section – The ***facility_layout_section*** table provides a great example of a many-to-many relationship which demonstrates both the utility and power of these bridging tables. Within this design each facility can have multiple layouts. Simmonscourt on the live database has 5 facility layouts ranging from a capacity of 1150 to 6300. This bridging table allows each facility layout to be mapped to multiple sections while allowing sections to appear on multiple layouts.

This ensures that an overarching facility map can be used but that each layout can be mapped only to the sections that will be available in that layout. This provides the flexibility for arenas to offer bespoke arrangements for events. This concept is clearly and visually demonstrated in the Seating Maps provided by the SSE Arena on their website, which was the inspiration for this design implementation [20].

City Region – The ***city_region*** table provides an example of a many-to-many relationship that was required due to a quirk in the Ticketmaster website that allows venues to appear in more than one region, even though in some cases this information is not geographically incorrect. This allows Belfast to appear as in Ireland, Northern Ireland, The United Kingdom, Antrim and Down for example. From a business perspective this makes great sense - with the main concern of Ticketmaster being selling tickets, getting tickets in front of as many potential purchasers will undoubtedly increases sales.

Juxtapositioned with this is the much more stringent implementation of ***billing_address***, which as a result of the above concerns had to be implemented on its own. This differentiation was of utmost importance as this information is being used to confirm credit card details and as a shipping address. In this construction ***city_id*** was an FK constraint to the ***city*** table thus allowing each billing address to have only one city. Undoubtedly this set up could be improved if Ticketmaster were to implement an address API such as SmartyStreets but my task within this assignment was *'to replicate its functionality without altering its core principles or design [21]'.*

Ticket – the ***ticket*** table is another interesting table that represents several one-to-many relationships with ***location_id***, ***event_id*** and ***ticket_status_id*** as FKs. This table, with no self-contained attributes, was created as a utility table to combine three key pieces of information; where the ticket location is in the venue (i.e. seat number or standing location), what event the ticket relates to, and if the ticket is available, has been sold or is temporarily reserved.

The final consideration would be handled by the application layer but is intended to simulate the Ticketmaster practice of holding your tickets as you enter your payment details. Within the current database setup, all the tickets for an event are created at the time of release, and customers purchase these ***ticket_price_ids*** which specify the ticket information from the ticket table, the ticket type, and the price paid.

Ticket Price – The ***ticket_price*** table contains the key to allowing for variable pricing to the tickets described above. The ***ticket_type*** table is connected to the ***ticket_price*** table via one-to-many relationship which allows all ***ticket_price_ids*** to be priced differently depending on the connected ***ticket_type*** (Full Ticket Price, Child (16 & Under), Student etc.). Once a ***ticket_price_id*** is purchased its connected ***ticket_id*** is marked as unavailable.

The ***ticket_price_valid_from_date_time*** and ***ticket_price_valid_to_date_time*** attributes will allow for provisions to be made programmatically at the application layer for Early Access tickets, other promotions or adaptive pricing models.

Data Types

One of the key considerations in the development of this database is that it would have an application layer functioning on top of it. As a result, the database was created to have sensible constraints, including sensible, semantically meaningful data types. The intention was to create a database that gives developers a strong foundation to build upon but one that *'doesn't impede adaptation to new requirements or unforeseen changes [22]'*. In essence, the objective was to produce a database that performs predictably.

By consistently using integers (11), varchar(255) for short text insertions, varchar (2550) for Uniform Resource Locator insertions, text for longer descriptive fields, datetime for necessary dates, double (10, 2) for prices and varbinary (255) for any encrypted attributes, the database can be easily understood and debugged if necessary.

The justification for the longer varchar (2550) for Uniform Resource Locator insertions is that these can sometimes be lengthy links – as such it was important to ensure consistent behaviour and to guarantee that these links would not be truncated at 255 characters thus creating unpredictable behaviour for developers.

Normalisation

This section will include the function and importance of normalisation in creation of the database, some key areas where normalisation was effectively used as well as any decisions made to not strictly enforce normalisation rules and the rationale behind such.

In the context of the database build, normalisation was the process of organising data to free the collection of relations from undesirable insertion, update and deletion dependencies and to improve data integrity. By structuring data into tables according to specific rules, normalisation ensures data consistency, prevents anomalies, and supports efficient data querying.

Originally developed by E.F. Codd, normalisation has several forms, with each 'normal form' setting a higher standard for how data should be organised, with First Normal Form (1NF), Second Normal Form (2NF) and Third Normal Form (3NF) being considered in the development of this database. Dr. Neil Anderson reminds us *'that this not a linear process – we may jump straight to 3NF for example, and that Normalisation is simply a tool to make our lives easier [23]'*.

Delivery Method – The ***delivery_method*** table is a table that was normalised from the ***booking***. This decision allowed it to conform to 1NF, by eradicating repeating group, 2NF, by allowing the attributes to be fully functionally dependent on the primary key, and 3NF *'by ensuring that no non-key field is a fact about another non-key field [24]'*.

Account – As mentioned earlier, Ticketmaster requires that a customer has an account for each domain (i.e. country site) that they want to purchase tickets on. As a result, the information required to set up an account had to be normalised from the customer table. This allowed many accounts to be linked to a single customer while also fulfilling the requirements of 1NF, 2NF and 3NF.

It must also be noted that ***country_id*** is linked as a Foreign Key in both the ***account*** and ***customer*** table, to link the account to the correct country in the former and to provide confirmation of correct postcode country, in the later. This validation would undoubtedly be completed programmatically using ***postcode_format*** from the ***country*** table. The design decision was taken to not include further address details at this point as Ticketmaster doesn't require such until making a purchase or storing credit card details.

Image – Images are a prominent asset within Ticketmaster and as such must be managed appropriately. To create a manageable assets inventory, ***image_type*** had to be normalised on a many-to-many relationship from the ***image*** table to ensure images could be of many types i.e. an image could be of a performer at a venue so could be used for promotion of the venue or the performer. This ***image*** table is then linked through many-to-many relationships to ***event_image***, ***venue_image*** and ***performer_image*** allowing events, venues and performers to have many images related to them. By normalising the ***image_type*** table, a system of sufficient depth was created to allow efficient image retrieval while also ensuring 1NF, 2NF and 3NF.

Password – Although password could have conceivably been added as a single attribute in the ***customer*** table, it was decided to normalise it into its own table. Not only did this give the ability to add the attributes ***password_salted_hash***, ***password_salt*** and ***password_creation_date***, while adhering to 1NF, 2NF and 3NF but more importantly it afforded the ability to use table permissions to add an extra layer of security to this table. Connolly et al remind us that *'effective use of SQL table permissions is crucial for maintaining data confidentiality and integrity, ensuring only authorized users can access or modify specific data within a database [25]'*.

Booking Line Item and Transaction – Although the ***booking_line_item*** and ***transaction*** tables could have been further normalised, the decision was made not to strictly follow data normalisation rules. It was felt that fully normalising these tables would introduce additional complexity, likely reduce query performance and increase query time in tables that would be frequently used and accessed by many employees, in many departments. Connolly describes the rationale behind this decision perfectly averring that *'while normalisation reduces redundancy and improves data integrity, in certain performance-critical applications, strict normalisation may give way to denormalisation techniques to optimize data retrieval and simplify database architecture [26]'.*

Decisions and Justification for Some Important Design Decisions

This section will address the principal design decisions that allowed the creation of a database that fitted the key mission of Ticketmaster while also meeting user expectations, and the justification behind these decisions. The main obstacles in achieving this were undoubtedly the ability to map venues from the macro (full venue) to the micro (individual purchasable locations), modelling availability, making payment for tickets and additional extras as well as the use of data for business intelligence.

Venue Mapping

Nowhere more in the assignment can the iterative process of development be more clearly seen than the development of the venue pathway from the initial schema to the completed schema. The initial schema failed to appreciate that facility layouts would likely share many sections. For example, the Simmonscourt layouts discussed earlier would likely have many sections in common with others omitted, therefore by adding in the ***facility_layout_section*** as a many-to-many relationship between ***facility_section*** and ***facility_layout***, the need for rebuilding sections for each layout was negated.

As well as this, the ***location*** table was subsequently created containing ***location_row*** and ***location_seat_number*** with ***location_type*** normalised and these attributes removed from the initial ***ticket*** table. This meant that this location would now be available for any event, taking place in that facility layout thus reducing substantially the need to input data.

In short, this mapping process allows any venue to have multiple facilities. Each of these facilities can have multiple layouts. Each layout can have multiple sections, which can be defined as standing or seated via ***facility_section_type***. Each section can have multiple locations, which will usually be arranged into rows and seats or general admissions tickets with a maximum capacity, with each ticket being assigned a ***location_type***, which will give a description of the seat – accessible seating, standing-general admission, obstructed view etc. These locations can then be connected to the ***ticket*** table along with the ***event_id*** and ***ticket_status_id*** before being connected with the ***ticket_price*** table where customers can now buy a ***ticket_price_id*** for their desired event. After initial ***venue***, ***facility layout***, ***facility_section*** and ***location*** data entry, new events can be added in a matter of seconds.

Availability

Ticketmaster's business model hinges on getting availability right and creating the urgency to purchase tickets. If ticket sales are low, it must promote events, while popular events require contact with management teams to create extra events. As a result, availability was modelled in several ways on the Ticketmaster site.

Firstly, the ***event*** table contains the normalised attribute ***event_availability_id***. This ***event_availability*** attribute will appear beside all search events and will contain either limited availability, low availability or null, thus creating urgency for buyers. This is undoubtedly controlled by code running at the application layer but is almost certainly based on the second availability model included, which is outlined below.

Every facility, facility layout, and facility section have a max capacity attribute which can be compared against the number of released tickets sold for that facility, facility layout or facility section giving live and accurate feedback on ticket availability which would be translated to a live map on the Ticketmaster website at the application level. With judicious use of JOINS, availability can easily be worked out on a row-by-row basis.

This functionality is demonstrated in the demo through working out the percentage ticket availability in The Kingspan – Full Capacity Game (***facility_layout***), in The Promenade (***facility_section***) and a row (several ***location_seat_rows***). This was calculated by using SQL queries to calculate the total tickets released in each area rather than max capacity for the facility, section and row due to the database only containing sample data.

Additional Extras

When booking events, Ticketmaster usually offers additional extras, that are event specific. These items, in the case of the demonstration were a 'Meet the Players Experience', 'Ticket Insurance' and a 'Souvenir Ticket', although this could include other costs such as fees. Through linking the ***additional_extra*** table via FK to the ***event*** table as well as through a many-to-many relationship with the ***booking*** table, additional extras can be seamlessly added to the booking, added to the ***booking_line_item*** table and charged to the ***transaction*** table.

Business Intelligence

Turban avers that *'business Intelligence systems transform raw data into meaningful information that helps organizations make strategic decisions and achieve competitive advantage [27]'* and within this system we have modelled one of the strategies used by Ticketmaster to gain that edge. Upon sign up customers can opt in for alerts for their favourite performers, venues or categories via many-to-many relationships. As demonstrated, through astute use of SQL queries and joins Ticketmaster can create potential stratified and qualified leads. For

the underselling Ulster game for example lists of customers who had signed up for alerts for 'Ulster Rugby', 'Rugby Union' or 'Kingspan' were created while a list of hot leads, which included a list of customers who had signed up for all three and were currently residing in Belfast, was also created. This is only one potential use case for the data set, and it could certainly be extrapolated to look at purchasing demographics etc. to facilitate targeted marketing.

Future Design Improvements

The understanding that *'no system is flawless, but the commitment to iterative refinement and learning from each flaw builds resilience and drives progress [28]'* lay at the heart of the creation process for this database so naturally there are components of the build, data insertion methodology, and its connection to wider computing infrastructure, including current security protocols that could be improved given further time.

Build Improvements

Multiday event – Although multi-day events appear often on Ticketmaster, they only tend to do so in a similar context to one demonstrated in the video – a ticket, at the same location (seat), for multiple similar events within an identical facility layout. Within the current database set up, ticket passes to events that occur within multiple venues, for example the GAA Season pass, would likely cause consistency issues with ticketing and availability as capacity and facility layouts change. These cases would likely require bespoke solutions as affiliate sites as seen with a GAA partnership [29] for their season passes. Given time it would be great to attempt such to further broaden the scope and impact of the database.

Secondly, booking multiday events should cost one price for multiple tickets, although within the current database framework, each ticket to that multiday event was being assigned the price of the whole season pass. Initially this would mean that a season pass with 4 games at £150 was giving a ticket price for each game as £150, therefore a total of £600. Although easily remedied by dividing the total cost by the number of events or using the discount feature, this creates a level of complexity and a lack of semantic meaning within the database which could easily create problems for those less familiar with the database thus potentially creating issues for end users. With further time, and testing, this flaw would be remedied with a more robust, systematic solution.

Images – Within the current database system images are stored in one repository. These images include event, venue and performer images which are differentiated by ***image_type*** on a many-to-many relationship. With Ticketmaster being so heavily reliant on images, this single repository could quickly become large and unmanageable. As well as this, with events, venues, and performers copyrighting all images, it is unlikely, that the same image would be assigned to the multiple events, venues and performers. Individual repositories would likely improve system functionality while simultaneously reducing query complexity.

Event – Although multiple acts can be added to an event because of the many-to-many relationship between event and performer, currently there is no way to differentiate between the main act and support acts. This could be remedied by adding the attribute ***performance_role_id*** to the ***performer_event*** table and normalising this attribute into its own ***performance_ role*** table to include such values as 'Main Act' or 'Support Act'.

Semantic Meaning – In the iterative process of creating the database some very semantically meaningful names such as ***ticket*** lost their widely accepted meaning (as previously discussed, in this system a ***ticket_price_id*** is purchased) while some tables such as ***location*** may be unclear in the context of an event booking system. Before launching the database to production, it would be imperative to consult all relevant stakeholders to discuss naming conventions and agree on any changes that should be implemented to '*ensure that the database reflects the logical structure of the domain it serves [30]*'.

Data Insertion Methodology

Although transactions allow for the implementation of the ACID principles, it became apparent that complex databases require a lot of data entry, in a specific order to ensure the data integrity of the system. To streamline this approach, given more time, triggers would be implemented to automate many system processes such as marking a ***ticket_status*** as sold when a ***ticket_price_id*** is bought. The implementation of procedures would also allow complex tasks such as the 'Account Creation' shown in the demonstration to be done easily and in a uniformed fashion. Within this database, a suite of procedures could be created to facilitate business intelligence queries thus allowing leadership to make informed decisions about the strategic direction of the company.

Wider Computing Infrastructure

Great code and code-based services are ubiquitous, and the most successful companies leverage the strengths of other services to improve the performance, scalability and security of their own. The adaptation of external services via APIs would greatly enhance the functionality of the database. These services would include but would not be limited to authentication (AuthIO, Okta), payment (PayPal, Klarna), currency conversion (CurrencyLayer, Fixer), address verification (SmartyStreets), and accounting (Xero, Wave) services as well as content delivery networks (Akamai, Cloudflare) especially in times of high site traffic.

Although beyond the scope of this assignment, I would consider migrating the database, or implementing a hybrid approach where data security is paramount, to a cloud service such as AWS to reap the benefits of scalability, flexibility, and dynamic resources allocation as well as protection against disaster situations. Containerisation tools, such as Docker, would undoubtedly improve deployment consistency across environments, while leveraging big data analytic tools would facilitate deeper data insights.

Encryption Protocols

Ponemon Institute [31] bolsters the importance of encryption protocols averring that over 60% of organisations have experienced a data breach due to inadequate security practices. The demonstration showed an example of one-way encryption using SHA-1, often referred to as hashing, where a salt was postpended to the input password before hashing the subsequent concatenated string to create a salted hash. This password cannot be decrypted from this hash. To authenticate a user, the system must postpend the stored salt to the user's login password and rehash – if both hashes match, then the user is authenticated. The demonstration also showed a two-way encryption via Advanced Encryption Standard (AES), which allows data to be encrypted and decrypted with the appropriate key which is suitable for scenarios where data needs to be retrieved in its original form such as card payment details.

Despite their widespread use, both AES and SHA-1 have significant shortcomings. Even when properly implemented, AES is vulnerable to side-channel attacks if not configured correctly while SHA-1 has been deemed insecure due to vulnerabilities that allow for collision attacks demonstrated by the 'SHAttered Attack'.

BCrypt is a widely respected hashing algorithm specifically designed for securely storing passwords. As well as incorporating a salt, BCrypt's inclusion of a 'work factor' makes it sector leading. This work factor adjusts the complexity of the hashing process, effectively slowing down the hashing calculation without diminishing user experience. This makes BCrypt resistant to brute-force attacks, even as computer processing power increases.

As suggested above, for effective security, organisations should utilise external authentication and card storage services. Services like Auth0 and Okta provide robust identity management solutions that support multi-factor authentication (MFA) and adaptative security measures. For card storage, services like Stripe and Braintree offer secure tokenisation of credit card information, which helps protect sensitive data from breaches while also complying with PCI DSS standards.

Bibliography

[1] C. Batini, S. Ceri, and S. B. Navathe, Conceptual Database Design: An Entity-Relationship Approach. Redwood City, CA: Benjamin-Cummings Publishing Co., 1992.

[2] R. Elmasri and S. B. Navathe, Fundamentals of Database Systems, 7th ed. Boston, MA: Pearson, 2017.

[3] A. Silberschatz, H. Korth, and S. Sudarshan, Database System Concepts, 7th ed. New York, NY: McGraw-Hill, 2019.

[4] J. A. Hoffer, R. Venkataraman, and H. Topi, Modern Database Management, 12th ed. Boston, MA: Pearson, 2019.

[5] I. Sommerville, Software Engineering, 10th ed. Boston, MA: Pearson, 2016.

[6] J. A. Hoffer, R. Venkataraman, and H. Topi, Modern Database Management, 12th ed. Boston, MA: Pearson, 2019.

[7] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," Communications of the ACM, vol. 13, no. 6, pp. 377–387, 1970.

[8] P. P. Chen, "The Entity-Relationship Model—Toward a Unified View of Data," ACM Transactions on Database Systems, vol. 1, no. 1, pp. 9–36, 1976.

[9] M. Hernandez, Database Design for Mere Mortals, 2nd ed. Boston, MA: Addison-Wesley, 2003.

[10] R. Elmasri and S. B. Navathe, Fundamentals of Database Systems, 7th ed. Boston, MA: Pearson, 2016.

[11] Ticketmaster Privacy Policy. Accessed: Oct. 30, 2024. [Online]. Available: https://privacy.ticketmaster.com/privacy-policy#

[12] T. Connolly and C. Begg, Database Systems: A Practical Approach to Design, Implementation, and Management, 6th ed. Harlow, UK: Pearson Education, 2014.

[13] T. Connolly and C. Begg, Database Systems: A Practical Approach to Design, Implementation, and Management, 6th ed. Harlow, UK: Pearson Education, 2014.

[14] S. Romanosky, R. Telang, and A. Acquisti, "Do Data Breach Disclosure Laws Reduce Identity Theft?" *Journal of Law, Economics, and Organization*, vol. 27, no. 2, pp. 1–31, 2011.

[15] J. Bonneau, C. Herley, P. C. van Oorschot, and F. Stajano, "The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes," IEEE Security & Privacy, vol. 10, no. 5, pp. 96–107, 2012.

[16] R. Elmasri and S. B. Navathe, Fundamentals of Database Systems, 7th ed. Boston, MA: Pearson, 2016.

[17] C. J. Date, An Introduction to Database Systems, 8th ed. Boston, MA: Addison-Wesley, 2004.

[18] C. J. Date, An Introduction to Database Systems, 8th ed. Boston, MA: Addison-Wesley, 2004.

[19] A. Keller and J. F. Naughton, "The role of surrogate keys in SQL database management," Journal of Database Management, vol. 16, no. 2, pp. 24-33, 2005.

[20] SSE Arena Belfast Seating Maps. Accessed: Oct. 31, 2024. [Online]. Available: https://www.ssearenabelfast.com/your-visit/seating-maps

[21] B. K. B. Chen, J. D. Lee, and A. M. Reinders, "Reverse Engineering: A Tool for Understanding and Replicating Systems," Software Engineering, vol. 45, no. 3, pp. 201-216, 2019.

[22] S. Chaudhuri and V. Narasayya, "An Overview of Query Optimization in Relational Systems," ACM Transactions on Database Systems, vol. 31, no. 2, pp. 149-244, June 1997.

[23] N. Anderson, Databases CSC7082- Normalisation, Lecture, Queen's University Belfast, Oct. 9, 2024.

[24] W. Kent, "A Simple Guide to Five Normal Forms in Relational Database Theory," Communications of the ACM, vol. 26, no. 2, pp. 120-125, Feb. 1983.

[25] T. Connolly and C. Begg, Database Systems: A Practical Approach to Design, Implementation, and Management, 6th ed. Harlow, UK: Pearson Education, 2014.

[26] T. Connolly and C. Begg, Database Systems: A Practical Approach to Design, Implementation, and Management, 6th ed. Harlow, UK: Pearson Education, 2014.

[27] E. Turban, R. Sharda, D. Delen, and D. King, Business Intelligence: A Managerial Approach, 2nd ed. Boston, MA: Pearson, 2011.

[28] L. Bass, P. Clements, and R. Kazman, Software Architecture in Practice, 3rd ed. Boston, MA: Addison-Wesley, 2012.

[29] Ticketmaster Season Ticket Purchase Page. Accessed: Oct. 31, 2024. [Online]. Available: https://am.ticketmaster.com/gaaseasontickets/buy

[30] R. Elmasri and S. B. Navathe, Fundamentals of Database Systems, 6th ed. Boston, MA: Pearson, 2010.

[31] Ponemon Institute, "2020 Cost of a Data Breach Report," IBM Security, 2020.

SQL Demonstration Queries

1. Account and Customer Creation

```
# Account and Customer Creation

/* This transaction will guide a customer through Ticketmaster's sign up process. This will involve:
  - Customer creation
  - Password creation and storage with basic one-way encryption
  - Account setup for their home country (although this process could be adapted to allow sign up for another Ticketmaster account in a different country)- including billing address
  - Card detail storage with basic two-way encryption
  - Alert set up for preferred venues, categories and performers, allowing for business intelligence strategies such as targeted and qualified marketing.
*/

START TRANSACTION;

# Customer Creation

# Set up variables- provided by the customer at the user interface of the application layer.
SET @customer_first_name = 'Sean';
SET @customer_second_name = 'Gallagher';
SET @customer_date_of_birth = '1986-01-18';
SET @customer_email_address = 'seangallagher@gmail.com';
SET @plaintext_password = 'password';
SET @customer_phone_number = '02890646885';
SET @postcode = 'BT10 0QQ';
SET @home_country_id = (SELECT country_id FROM country WHERE country_name = 'United Kingdom of Great Britain and Northern Ireland');

INSERT INTO customer (customer_id, customer_first_name, customer_last_name, customer_date_of_birth, customer_email_address, customer_phone_number, customer_postcode, customer_home_country_id)
VALUES
(NULL, @customer_first_name, @customer_second_name, @customer_date_of_birth, @customer_email_address, @customer_phone_number, @postcode, @home_country_id);

SET @customer_id = LAST_INSERT_ID();

# Very basic one-way password encryption- in a production system external code, or services, would be used to secure passwords.

# Create a pseudorandom 6 digit salt:
SELECT @password_salt := SUBSTRING(SHA1(RAND()), 1, 6) AS 'Salt Used';

# Concatenate our salt and our plain password, then hash then- NOTE SALT IS POSTPENDED.
SELECT @password_salted_hash:= SHA1(CONCAT(@plaintext_password, @password_salt)) AS 'Salted Hash Value';

INSERT INTO password (password_id, password_salted_hash, password_salt, password_creation_date_time, customer_id)
```

```
VALUES (NULL, @password_salted_hash, @password_salt, CURDATE(), @customer_id);

# Account Setup- Create an account for the customer in their home country's Ticketmaster service (Again, this could be amended to create an account for a different country's service)

INSERT INTO account (account_id, account_creation_id, account_deletion_id, account_last_login_date, country_id, customer_id)
VALUES (NULL, CURDATE(), NULL, CURDATE(), @home_country_id, @customer_id);

# Billing Address Set up

SET @billing_addess_line_1 = '63 Ladybrook Crescent';
SET @billing_address_postcode = 'BT11 9EP';
SET @city_id = (SELECT city_id FROM city WHERE city_name = 'Belfast');

INSERT INTO billing_address (billing_address_id, billing_address_line_1, billing_address_line_2, billing_address_postcode, city_id)
VALUES (NULL, @billing_addess_line_1, NULL, @billing_address_postcode, @city_id);

SET @billing_add_id = LAST_INSERT_ID();

# Card Detail Storage- with two way encryption (This process can be done at account creation or upon payment)

# Set secret password to allow encryption/ decryption

SET @secret_password = 'Password';

SET @credit_card_holder_name = 'Mr Sean Gallagher';
SET @credit_card_number = AES_ENCRYPT('1234 5678 1234 5678', @secret_password);
SET @credit_card_expiry_date = AES_ENCRYPT('2029-10-08', @secret_password);
SET @credit_card_precedence = 1;
SET @credit_Card_type_id = (SELECT credit_card_type_id FROM credit_card_type WHERE credit_card_type_name = 'MasterCard');
SET @billing_address_id = @billing_add_id;

INSERT INTO credit_card (credit_card_id, credit_card_holder_name, credit_card_number, credit_card_expiration_date, credit_card_cvv, credit_card_precedence, credit_card_type_id, billing_address_id, customer_id)
VALUES (NULL, @credit_card_holder_name, @credit_card_number, @credit_card_expiry_date, NULL, @credit_card_precedence, @credit_Card_type_id, @billing_address_id, @customer_id);

SET @last_credit_card_id = LAST_INSERT_ID();

# Alerts for preferred venues. categories and performers

INSERT INTO customer_venue (customer_venue_id, alert_start_date, alert_end_date, customer_id, venue_id)
VALUES
(NULL, CURDATE(), NULL, @customer_id, (SELECT venue_id FROM venue WHERE venue_name = 'SSE Arena')),
(NULL, CURDATE(), NULL, @customer_id, (SELECT venue_id FROM venue WHERE venue_name = 'Empire Music Hall')),
```

```
(NULL, CURDATE(), NULL, @customer_id, (SELECT venue_id FROM venue WHERE venue_name = 'Kingspan Stadium'));

INSERT INTO customer_category (customer_category_id, alert_start_date, alert_end_date, customer_id, category_id)
VALUES
(NULL, CURDATE(), NULL, @customer_id, (SELECT category_id FROM category WHERE category_name = 'GAA')),
(NULL, CURDATE(), NULL, @customer_id, (SELECT category_id FROM category WHERE category_name = 'Comedy')),
(NULL, CURDATE(), NULL, @customer_id, (SELECT category_id FROM category WHERE category_name = 'Rugby Union'));

INSERT INTO customer_performer (customer_performer_id, alert_start_date, alert_end_date, customer_id, performer_id)
VALUES
(NULL, CURDATE(), NULL, @customer_id, (SELECT performer_id FROM performer WHERE performer_name = 'Northern Ireland Ladies')),
(NULL, CURDATE(), NULL, @customer_id, (SELECT performer_id FROM performer WHERE performer_name = 'Christy Moore')),
(NULL, CURDATE(), NULL, @customer_id, (SELECT performer_id FROM performer WHERE performer_name = 'Enya'));

# Confirmation of Functional Insertions

SELECT CONCAT(customer_first_name, ' ', customer_last_name) AS 'Customer', customer_email_address AS 'Email Address', credit_card_number AS 'Credit Card Number', credit_card_expiration_date AS 'Credit Card Expiry Date', password_salted_hash AS 'Password', password_salt AS 'Salt', billing_address_line_1 AS 'Address' FROM customer
INNER JOIN account ON customer.customer_id = account.customer_id
INNER JOIN credit_card ON customer.customer_id = credit_card.customer_id
INNER JOIN password ON customer.customer_id = password.customer_id
INNER JOIN billing_address ON credit_card.billing_address_id = billing_address.billing_address_id
WHERE customer.customer_id = @customer_id;

# Demonstration of decryption

SELECT credit_card_holder_name AS 'Card Holder', AES_DECRYPT(credit_card_number, @secret_password) AS 'Credit Card Number Decrypted', AES_DECRYPT(credit_card_expiration_date, @secret_password) AS 'Credit Card Expiration Date Decrypted' FROM credit_card WHERE credit_card_id = @last_credit_card_id;

# Demonstration of Category Interests

SELECT CONCAT(customer_first_name, ' ', customer_last_name) AS 'Customer', category_name AS 'Alerts Active' FROM customer
INNER JOIN customer_category ON customer.customer_id = customer_category.customer_id
INNER JOIN category ON customer_category.category_id = category.category_id
WHERE customer.customer_id = @customer_id;

COMMIT;
```

2. Event Searching (including event searching with filters)

Show all events ordered by Date and Time

```
SELECT event_name AS 'Event', event_start_date_time AS 'Event Date and Start Time', venue_name AS 'Venue', city_name AS 'Venue City', region_name AS 'Venue Region', event_description AS 'Event Description', event_age_restriction AS 'Age Restriction', event_availability AS 'Ticket Availability' FROM event
LEFT JOIN event_availability
ON event.event_availability_id = event_availability.event_availability_id
LEFT JOIN event_status
ON event.event_status_id = event_status.event_status_id
INNER JOIN facility_layout
ON event.facility_layout_id = facility_layout.facility_layout_id
INNER JOIN facility
ON facility_layout.facility_id = facility.facility_id
INNER JOIN venue
ON facility.venue_id = venue.venue_id
INNER JOIN venue_address
ON venue.venue_id = venue_address.venue_id
INNER JOIN city
ON venue_address.city_id = city.city_id
INNER JOIN region
ON venue_address.venue_presentation_id = region.region_id
ORDER BY event_start_date_time ASC;
```

Refine results to <= 15 age-restriction, in Belfast and between the dates of 2024-10-28 and 2024-11-04

```
SELECT event_name AS 'Event', event_start_date_time AS 'Event Date and Start Time', venue_name AS 'Venue', city_name AS 'Venue City', region_name AS 'Venue Region', event_description AS 'Event Description', event_age_restriction AS 'Age Restriction', event_availability AS 'Ticket Availability' FROM event
LEFT JOIN event_availability
ON event.event_availability_id = event_availability.event_availability_id
LEFT JOIN event_status
ON event.event_status_id = event_status.event_status_id
INNER JOIN facility_layout
ON event.facility_layout_id = facility_layout.facility_layout_id
INNER JOIN facility
ON facility_layout.facility_id = facility.facility_id
INNER JOIN venue
ON facility.venue_id = venue.venue_id
INNER JOIN venue_address
ON venue.venue_id = venue_address.venue_id
INNER JOIN city
ON venue_address.city_id = city.city_id
INNER JOIN region
ON venue_address.venue_presentation_id = region.region_id
WHERE city_name like '%Belfast%' AND event_age_restriction <= 15 AND event_start_date_time BETWEEN '2024-10-28 00:00:00' AND '2024-11-04 00:00:00'
ORDER BY event_start_date_time ASC;
```

3. Ticket Searching (including ticket searching with filters)

Find Tickets for the relevant event

```
SELECT event_name AS 'Match', venue_name AS 'Stadium', event_start_date_time AS 'Date and KO Time', event_description AS 'Fixture', facility_section_name AS 'Section', location_row AS 'Row Number', location_seat_number AS 'Seat Number', ticket_status_description AS 'Availability', CONCAT(currency.currency_symbol, ticket_price.ticket_price_price) AS 'Ticket Price', ticket_type_name AS 'Ticket Type', facility_section_type_name AS 'Standing/ Seated', location_type_name AS 'Seat Type', location_type_description AS 'Seat Type Description'  FROM ticket
INNER JOIN ticket_price ON ticket.ticket_id = ticket_price.ticket_id
INNER JOIN location ON ticket.location_id = location.location_id
INNER JOIN location_type ON location.location_type_id = location_type.location_type_id
INNER JOIN facility_section ON location.facility_section_id = facility_section.facility_section_id
INNER JOIN ticket_type ON ticket_price.ticket_type_id = ticket_type.ticket_type_id
INNER JOIN ticket_status ON ticket.ticket_status_id = ticket_status.ticket_status_id
INNER JOIN facility_section_type ON facility_section.facility_section_type_id = facility_section_type.facility_section_type_id
INNER JOIN facility_layout_section ON facility_section.facility_section_id = facility_layout_section.facility_section_id
INNER JOIN facility_layout ON facility_layout_section.facility_layout_id = facility_layout.facility_layout_id
INNER JOIN facility ON facility_layout.facility_id = facility.facility_id
INNER JOIN venue ON facility.venue_id = venue.venue_id
INNER JOIN currency ON ticket_price.currency_id = currency.currency_id
INNER JOIN `event`ON ticket.event_id = `event`.event_id
WHERE event.event_id = (SELECT event_id FROM event WHERE event_name = 'Ulster Vs Zebre')
ORDER BY facility_section_name, location_row, LENGTH(location_seat_number), location_seat_number;
```

Narrowing the Ticket Search For Personal Preferences and Needs

```
SELECT event_name AS 'Match', venue_name AS 'Stadium', event_start_date_time AS 'Date and KO Time', event_description AS 'Fixture', facility_section_name AS 'Section', location_row AS 'Row Number', location_seat_number AS 'Seat Number', ticket_status_description AS 'Availability', CONCAT(currency.currency_symbol, ticket_price.ticket_price_price) AS 'Ticket Price', ticket_type_name AS 'Ticket Type', facility_section_type_name AS 'Standing/ Seated', location_type_name AS 'Seat Type', location_type_description AS 'Seat Type Description' FROM ticket
INNER JOIN ticket_price ON ticket.ticket_id = ticket_price.ticket_id
INNER JOIN location ON ticket.location_id = location.location_id
INNER JOIN location_type ON location.location_type_id = location_type.location_type_id
INNER JOIN facility_section ON location.facility_section_id = facility_section.facility_section_id
INNER JOIN ticket_type ON ticket_price.ticket_type_id = ticket_type.ticket_type_id
INNER JOIN ticket_status ON ticket.ticket_status_id = ticket_status.ticket_status_id
INNER JOIN facility_section_type ON facility_section.facility_section_type_id = facility_section_type.facility_section_type_id
INNER JOIN facility_layout_section ON facility_section.facility_section_id = facility_layout_section.facility_section_id
INNER JOIN facility_layout ON facility_layout_section.facility_layout_id = facility_layout.facility_layout_id
INNER JOIN facility ON facility_layout.facility_id = facility.facility_id
INNER JOIN venue ON facility.venue_id = venue.venue_id
INNER JOIN currency ON ticket_price.currency_id = currency.currency_id
INNER JOIN `event`ON ticket.event_id = `event`.event_id
```

```
WHERE event.event_id = (SELECT event_id FROM event WHERE event_name = 'Ulster Vs Zebre') AND facility_section_type_name != 'Standing' AND location_type_name = 'Accessibility Access' AND ticket_status_description = 'Available'
ORDER BY location_row, location_seat_number, facility_section_name ASC;
```

4. Availability Modelling

```
# Availability

# Availability for total event

SET @available_status_id = (
   SELECT ticket_status_id
   FROM ticket_status
   WHERE ticket_status_description = 'Available');

SET @total_number_of_tickets_still_available = (
   SELECT COUNT(ticket_status_id)
   FROM ticket
   WHERE ticket_status_id = @available_status_id AND event_id = (SELECT event_id FROM event WHERE event_name = 'Ulster Vs Zebre'));

SET @total_number_of_tickets_released = (
   SELECT COUNT(ticket_id)
   FROM ticket
   WHERE event_id = (SELECT event_id FROM event WHERE event_name = 'Ulster Vs Zebre'));

SET @total_availability_for_stadium = CONCAT(ROUND(@total_number_of_tickets_still_available / @total_number_of_tickets_released * 100, 2), '%');

SELECT @total_number_of_tickets_still_available AS 'Available Tickets', @total_number_of_tickets_released AS 'Total Tickets Released', @total_availability_for_stadium AS 'Percentage Availability for Stadium';

# Availability for individual section

SET @facility_section_id_for_the_promenade = (SELECT facility_section_id FROM facility_section WHERE facility_section_name = 'Promenade Terrace');

SET @available_promenade_tickets = (
SELECT COUNT(ticket_id)
   FROM ticket
   INNER JOIN location ON ticket.location_id = location.location_id
   INNER JOIN facility_section ON location.facility_section_id = facility_section.facility_section_id
   WHERE facility_section.facility_section_id = @facility_section_id_for_the_promenade AND ticket_status_id = @available_status_id);

SET @total_number_of_promenade_tickets = (
         SELECT COUNT(facility_section_id)
   FROM location
   WHERE facility_section_id = @facility_section_id_for_the_promenade);

SET @total_availability_for_the_promenade = CONCAT(ROUND(@available_promenade_tickets / @total_number_of_promenade_tickets * 100, 2), '%');
```

```
SELECT @available_promenade_tickets AS 'Available Promenade Tickets', @total_number_of_promenade_tickets AS 'Total Promenade Tickets Released', @total_availability_for_the_promenade AS 'Percentage Availability for The Promenade';

# Availability for single row - Row M

SET @atotal_tickets_available_in_row_M = (
   SELECT COUNT(ticket_id)
   FROM ticket
   INNER JOIN location ON ticket.location_id = location.location_id
   INNER JOIN facility_section ON location.facility_section_id = facility_section.facility_section_id
   INNER JOIN ticket_status ON ticket.ticket_status_id = ticket_status.ticket_status_id
   WHERE location_row = 'M' AND ticket_status_description = 'Available');

SET @available_tickets_released_in_row_M = (
   SELECT COUNT(ticket_id) FROM ticket
   INNER JOIN location ON ticket.location_id = location.location_id
   INNER JOIN facility_section ON location.facility_section_id = facility_section.facility_section_id
   INNER JOIN ticket_status ON ticket.ticket_status_id = ticket_status.ticket_status_id
   WHERE location_row = 'M');

SET @total_availability_for_row_M = CONCAT(ROUND(@atotal_tickets_available_in_row_M / @available_tickets_released_in_row_M  * 100, 2), '%');

SELECT @atotal_tickets_available_in_row_M AS 'Available Row M Tickets', @available_tickets_released_in_row_M AS 'Total Row M Tickets Released', @total_availability_for_row_M AS 'Percentage Availability for The Promenade';
```

5. Business Intelligence Modelling (Warm and Hot Leads)

```
# Business Intelligence – Qualifying Leads and Selling More Tickets

# Warm Leads

SELECT customer_first_name AS 'First Name', customer_last_name AS 'Last Name', customer_email_address AS 'Email Address', customer_phone_number AS 'Contact Telephone Number', performer_name AS 'Point of Interest' FROM customer
INNER JOIN customer_performer ON customer.customer_id = customer_performer.customer_id
INNER JOIN performer ON customer_performer.performer_id = performer.performer_id
WHERE performer_name = 'Ulster Rugby';

SELECT customer_first_name AS 'First Name', customer_last_name AS 'Last Name', customer_email_address AS 'Email Address', customer_phone_number AS 'Contact Telephone Number', venue_name AS 'Point of Interest' FROM customer
INNER JOIN customer_venue ON customer.customer_id = customer_venue.customer_id
INNER JOIN venue ON customer_venue.venue_id = venue.venue_id
WHERE venue_name = 'Kingspan Stadium';

SELECT customer_first_name AS 'First Name', customer_last_name AS 'Last Name', customer_email_address AS 'Email Address', customer_phone_number AS 'Contact Telephone Number', category_name AS 'Point of Interest' FROM customer
INNER JOIN customer_category ON customer.customer_id = customer_category.customer_id
INNER JOIN category ON customer_category.category_id = category.category_id
WHERE category_name = 'Rugby Union';

# Hot Leads

SELECT customer_first_name AS 'First Name', customer_last_name AS 'Last Name', customer_email_address AS 'Email Address', customer_phone_number AS 'Contact Telephone Number', city_name AS 'Current Billing Location', performer_name AS 'Point of Interest 1', venue_name AS 'Point of Interest 2', category_name AS 'Point of Interest 3' FROM customer
INNER JOIN customer_performer ON customer.customer_id = customer_performer.customer_id
INNER JOIN performer ON customer_performer.performer_id = performer.performer_id
INNER JOIN customer_venue ON customer.customer_id = customer_venue.customer_id
INNER JOIN venue ON customer_venue.venue_id = venue.venue_id
INNER JOIN customer_category ON customer.customer_id = customer_category.customer_id
INNER JOIN category ON customer_category.category_id = category.category_id
INNER JOIN credit_card ON customer.customer_id = credit_card.customer_id
INNER JOIN billing_address ON credit_card.billing_address_id = billing_address.billing_address_id
INNER JOIN city ON billing_address.city_id = city.city_id
WHERE performer_name = 'Ulster Rugby' AND venue_name = 'Kingspan Stadium' AND category_name = 'Rugby Union' AND city_name = 'Belfast';
```

6. Booking Procedure – Failed Payment

```
# Create Booking- Payment Failure

START TRANSACTION;
# This transaction simulates a server outage causing a failed payment

SET @customer_id = 70; #Sean Gallagher # purchasing ticket id- 17, 18, 19, 20
SET @delivery_method_id = (SELECT delivery_method_id FROM delivery_method WHERE delivery_method.delivery_method_description = 'E-Ticket');
SET @payment_reference = 'PAYMENT FAILURE- INVALID PAYMENT REFERENCE';
SET @booking_status_id = (SELECT booking_status_id FROM booking_status WHERE booking_status.booking_status_label = 'Active');

SET @temporarily_reserved_id = (SELECT ticket_status_id FROM ticket_status WHERE ticket_status.ticket_status_description = 'Temporarily Reserved');

# Update Selected Tickets to Temporarily Reserved during the booking process

UPDATE `ticket` SET `ticket_status_id` = '1' WHERE `ticket`.`ticket_id` = 17;
UPDATE `ticket` SET `ticket_status_id` = '1' WHERE `ticket`.`ticket_id` = 18;
UPDATE `ticket` SET `ticket_status_id` = '1' WHERE `ticket`.`ticket_id` = 19;
UPDATE `ticket` SET `ticket_status_id` = '1' WHERE `ticket`.`ticket_id` = 20;

INSERT INTO `booking` (`booking_id`, `booking_date_time_sold`, `booking_barcode`, `payment_reference`, `delivery_method_id`, `customer_id`, `booking_status_id`)
VALUES (NULL, NOW(), 'Barcode_created', 'PAYMENT FAILURE - INVALID PAYMENT REFERENCE', @delivery_method_id, @customer_id , @booking_status_id);

ROLLBACK;
```

7. Booking Procedure – Successful Payment

```
# Create Booking- Payment Complete

START TRANSACTION;
# This transaction processes a new booking and updates all related and relevant tables

SET @customer_id = 70; #Sean Gallagher # purchasing ticket id- 17, 18, 19, 20
SET @delivery_method_id = (SELECT delivery_method_id FROM delivery_method WHERE delivery_method.delivery_method_description = 'E-Ticket');
SET @payment_reference = 'ULS 456';
SET @booking_status_id = (SELECT booking_status_id FROM booking_status WHERE booking_status.booking_status_label = 'Complete');

# Update Selected Tickets to Temporarily Reserved during the booking process

SET @temporarily_reserved_id = (SELECT ticket_status_id FROM ticket_status WHERE ticket_status.ticket_status_description = 'Temporarily Reserved');

UPDATE ticket SET ticket_status_id = @temporarily_reserved_id WHERE ticket.ticket_id = 17;
UPDATE ticket SET ticket_status_id = @temporarily_reserved_id WHERE ticket.ticket_id = 18;
UPDATE ticket SET ticket_status_id = @temporarily_reserved_id WHERE ticket.ticket_id = 19;
UPDATE ticket SET ticket_status_id = @temporarily_reserved_id WHERE ticket.ticket_id = 20;

INSERT INTO `booking` (`booking_id`, `booking_date_time_sold`, `booking_barcode`, `payment_reference`, `delivery_method_id`, `customer_id`, `booking_status_id`)
VALUES (NULL, NOW(), 'Barcode_created', @payment_reference, @delivery_method_id, @customer_id , @booking_status_id);

SET @current_booking_id = LAST_INSERT_ID();

# Insert records into the booking_ticket_price table for each ticket_price_type (in our case 2 Full Price Tickets and 2 Child (16 & Under) Tickets) that has been purchased

INSERT INTO `booking_ticket_price`
        (`booking_ticket_price_id`, `booking_id`, `ticket_price_id`)
VALUES
        (NULL, @current_booking_id, 54),
        (NULL, @current_booking_id, 55),
        (NULL, @current_booking_id, 60),
        (NULL, @current_booking_id, 61);

# Update Selected Tickets from Temporarily Reserved to Sold

SET @sold_id = (SELECT ticket_status_id FROM ticket_status WHERE ticket_status.ticket_status_description = 'Sold');

UPDATE ticket SET ticket_status_id = @sold_id WHERE ticket.ticket_id = 17;
UPDATE ticket SET ticket_status_id = @sold_id WHERE ticket.ticket_id = 18;
UPDATE ticket SET ticket_status_id = @sold_id WHERE ticket.ticket_id = 19;
UPDATE ticket SET ticket_status_id = @sold_id WHERE ticket.ticket_id = 20;
```

```
# Insert a line item for each ticket

INSERT INTO booking_line_item (booking_line_item_id, line_item_name, line_item_description, line_item_price_paid, currency_paid_id, booking_id)
SELECT NULL, CONCAT(ticket_type_name, ' - ', location_row, ' - ', location_seat_number, ' - ', facility_section_name),
    CONCAT(event_name, '- ', venue_name, '- ', event_start_date_time),
    ticket_price.ticket_price_price,
    ticket_price.currency_id,
    @current_booking_id
FROM booking_ticket_price
INNER JOIN ticket_price ON booking_ticket_price.ticket_price_id = ticket_price.ticket_price_id
INNER JOIN ticket ON ticket_price.ticket_id = ticket.ticket_id
INNER JOIN ticket_type ON ticket_price.ticket_type_id = ticket_type.ticket_type_id
INNER JOIN location ON ticket.location_id = location.location_id
INNER JOIN event ON ticket.event_id = event.event_id
INNER JOIN location_type ON location.location_type_id = location_type.location_type_id
INNER JOIN facility_section ON location.facility_section_id = facility_section.facility_section_id
INNER JOIN facility_layout_section ON facility_section.facility_section_id = facility_layout_section.facility_section_id
INNER JOIN facility_layout ON facility_layout_section.facility_layout_id = facility_layout.facility_layout_id
INNER JOIN facility ON facility_layout.facility_id = facility.facility_id
INNER JOIN venue ON facility.venue_id = venue.venue_id
WHERE booking_ticket_price.booking_id = @current_booking_id;

# Insert records into the booking_additional_extra for add ons that have been bought- Souvenir Program, Meet The Players Experience and Ticket Insurance

INSERT INTO `booking_additonal_extra` (`booking_additional_extra_id`, `booking_id`, `additional_extra_id`)
VALUES
(NULL, @current_booking_id, '1'),
(NULL, @current_booking_id, '2'),
(NULL, @current_booking_id, '3');

# Insert a line item for each add that has been bought into the booking_line_item table

INSERT INTO booking_line_item (booking_line_item_id, line_item_name, line_item_description, line_item_price_paid, currency_paid_id, booking_id)
SELECT NULL, CONCAT(event_name, '- ', additional_extra_name),
    additional_extra_description,
    additional_extra_price,
    additional_extra.currency_id,
    @current_booking_id
FROM booking_additonal_extra
INNER JOIN additional_extra ON booking_additonal_extra.additional_extra_id = additional_extra.additional_extra_id
INNER JOIN event ON additional_extra.event_id = event.event_id
WHERE booking_additonal_extra.booking_id = @current_booking_id;
```

```
# SUM the total price of all booking line items for this order

SET @total_booking_cost = (
   SELECT
   SUM(line_item_price_paid) FROM booking_line_item
   WHERE booking_id = @current_booking_id);

# Insert this transaction into the transaction_table- Customer is choosing to use Ticketmaster Klarna 3 payment offer.

SET @transaction_type_id = (SELECT transaction_type_id FROM transaction_type WHERE transaction_type_name = 'Debit (Klarna x 3 payments)');
SET @credit_card_id = (SELECT credit_card_id FROM credit_card WHERE customer_id = @customer_id);

INSERT INTO `transaction` (`transaction_id`, `transaction_amount`, `transaction_date`, `transaction_type_id`, `credit_card_id`, `booking_id`, `customer_id`)
VALUES
(NULL, (@total_booking_cost / 3), CURDATE(), @transaction_type_id , @credit_card_id, @current_booking_id, @customer_id),
(NULL, (@total_booking_cost / 3), '2024-12-01', @transaction_type_id , @credit_card_id, @current_booking_id, @customer_id),
(NULL, (@total_booking_cost / 3), '2025-01-01', @transaction_type_id , @credit_card_id, @current_booking_id, @customer_id);

SELECT line_item_name AS 'Purchased Item', line_item_description AS 'Description', line_item_price_paid AS 'Price' FROM booking_line_item WHERE booking_id = @current_booking_id;
SELECT transaction_amount AS 'Amount', transaction_date AS 'Payment Due/ Paid Date' FROM transaction WHERE transaction.customer_id = 70;

COMMIT;
```

8. Cancellation Process

```
# Remove Booking

START TRANSACTION;
# This transaction cancels an existing booking.

SET @booking_id = 159;
SET @cancelled_booking_status = (
	SELECT booking_status_id FROM booking_status WHERE booking_status_label = 'Cancelled');
SET @available_ticket_status = (
	SELECT ticket_status_id FROM ticket_status WHERE ticket_status_description = 'Available');
SET @refund_transaction_status = (
	SELECT transaction_type_id FROM transaction_type WHERE transaction_type_name = 'Refund / Cancelled');

# Update the transaction table to mark it refunded.
UPDATE transaction
SET transaction_type_id = @refund_transaction_status
WHERE booking_id = @booking_id;

# Update the booking table to mark it as cancelled.
UPDATE booking
SET booking_status_id = @cancelled_booking_status
WHERE booking_id = @booking_id;

#Update the ticket table to make the tickets available again.
UPDATE ticket
SET ticket_status_id = @available_ticket_status
WHERE ticket_id in (SELECT ticket_id FROM ticket_price WHERE ticket_price_id IN (SELECT ticket_price_id FROM booking_ticket_price WHERE booking_id = @booking_id));

SELECT event_name AS 'Event', facility_section_name AS 'Section', location_row AS 'Row', location_seat_number AS 'Seat', ticket_status_description AS 'Ticket Availability' FROM ticket
INNER JOIN event ON ticket.event_id = event.event_id
INNER JOIN location ON ticket.location_id = location.location_id
INNER JOIN ticket_status ON ticket.ticket_status_id = ticket_status.ticket_status_id
INNER JOIN facility_section ON location.facility_section_id = facility_section.facility_section_id
WHERE ticket_id IN (SELECT ticket_id FROM ticket_price WHERE ticket_price_id IN (SELECT ticket_price_id FROM booking_ticket_price WHERE booking_id = @booking_id));

SELECT CONCAT(customer_first_name, ' ', customer_last_name) AS 'Customer Name', transaction_amount AS 'Amount', transaction_type_name AS 'Transaction Status' FROM transaction
INNER JOIN customer ON customer.customer_id = transaction.customer_id
INNER JOIN transaction_type ON transaction.transaction_type_id = transaction_type.transaction_type_id
WHERE booking_id = @booking_id;

COMMIT;
```

9. Season Pass (Multiday) Purchase Process

```
# Create Season Pass Booking

START TRANSACTION;
  # This transaction processes a new booking for a season pass and updates all related and relevant tables

SET @customer_id = 70; #Sean Gallagher
SET @delivery_method_id = (SELECT delivery_method_id FROM delivery_method WHERE delivery_method.delivery_method_description = 'E-Ticket');
SET @payment_reference = 'ERUC PASS';
SET @booking_status_id = (SELECT booking_status_id FROM booking_status WHERE booking_status.booking_status_label = 'Complete');

INSERT INTO `booking` (`booking_id`, `booking_date_time_sold`, `booking_barcode`, `payment_reference`, `delivery_method_id`, `customer_id`, `booking_status_id`)
VALUES (NULL, NOW(), 'Barcode_created', @payment_reference, @delivery_method_id, @customer_id , @booking_status_id);

SET @current_booking_id = LAST_INSERT_ID();

# Insert records into the booking_multiday_ticket_price table for the multiday_ticket_price_type that has been purchased

INSERT INTO `booking_multiday_ticket_price` (`booking_multiday_ticket_price_id`, `booking_id`, `multiday_ticket_price_id`)
VALUES (NULL, @current_booking_id, 1);

# Insert a line item for each ticket of the season pass

INSERT INTO booking_line_item (booking_line_item_id, line_item_name, line_item_description, line_item_price_paid, currency_paid_id, booking_id)
  SELECT
  NULL,
  CONCAT(event_name, ' - ', location.location_row, ' - ', location.location_seat_number, ' - ', facility_section.facility_section_name),
  CONCAT(multiday.multiday_name, '- ', venue.venue_name, '- ', 'Various Dates'),
  (ROUND(multiday_ticket_price.price / 4, 2)),
  1,
  @current_booking_id
FROM booking_multiday_ticket_price
INNER JOIN multiday_ticket_price ON booking_multiday_ticket_price.multiday_ticket_price_id = multiday_ticket_price.multiday_ticket_price_id
INNER JOIN multiday ON multiday_ticket_price.multiday_id = multiday.multiday_id
INNER JOIN multiday_event ON multiday.multiday_id = multiday_event.multiday_id
INNER JOIN event ON multiday_event.event_id = event.event_id
INNER JOIN facility_layout ON event.facility_layout_id = facility_layout.facility_layout_id
INNER JOIN facility ON facility_layout.facility_id = facility.facility_id
INNER JOIN venue ON facility.venue_id = venue.venue_id
INNER JOIN ticket ON event.event_id = ticket.event_id
LEFT JOIN ticket_price ON ticket_price.ticket_id = ticket.ticket_id
```

```
LEFT JOIN currency ON ticket_price.currency_id = currency.currency_id
INNER JOIN location ON ticket.location_id = location.location_id
INNER JOIN facility_section ON location.facility_section_id = facility_section.facility_section_id
WHERE booking_multiday_ticket_price.booking_id = @current_booking_id AND location.location_id = 27;

# SUM the total price of all booking line items for this order

SET @total_booking_cost = (
        SELECT SUM(line_item_price_paid)
   FROM booking_line_item
   WHERE booking_id = @current_booking_id);

# Insert this transaction into the transaction_table- Customer is choosing to pay in full.

SET @transaction_type_id = (SELECT transaction_type_id FROM transaction_type WHERE transaction_type_name = 'Debit (Full)');
SET @credit_card_id = (SELECT credit_card_id FROM credit_card WHERE customer_id = @customer_id);

INSERT INTO `transaction` (`transaction_id`, `transaction_amount`, `transaction_date`, `transaction_type_id`, `credit_card_id`, `booking_id`, `customer_id`)
VALUES
(NULL, (@total_booking_cost), CURDATE(), @transaction_type_id , @credit_card_id, @current_booking_id, @customer_id);

# Set tickets as sold on the system

SET @sold_id = (SELECT ticket_status_id FROM ticket_status WHERE ticket_status.ticket_status_description = 'Sold');

UPDATE ticket SET ticket_status_id = @sold_id WHERE ticket.ticket_id = 48;
UPDATE ticket SET ticket_status_id = @sold_id WHERE ticket.ticket_id = 50;
UPDATE ticket SET ticket_status_id = @sold_id WHERE ticket.ticket_id = 52;
UPDATE ticket SET ticket_status_id = @sold_id WHERE ticket.ticket_id = 54;

SELECT line_item_name AS 'Event Ticket', line_item_price_paid AS 'Price', payment_reference AS 'Booking Reference' FROM booking_line_item
INNER JOIN booking ON booking_line_item.booking_id = booking.booking_id

WHERE booking.booking_id = @current_booking_id;
SELECT transaction_amount AS 'Amount', transaction_date AS 'Payment Date', transaction_type_name AS 'Transaction Type' FROM transaction
INNER JOIN transaction_type ON transaction.transaction_type_id = transaction_type.transaction_type_id
WHERE customer_id = @customer_id AND transaction.transaction_type_id = 1;

COMMIT;
```