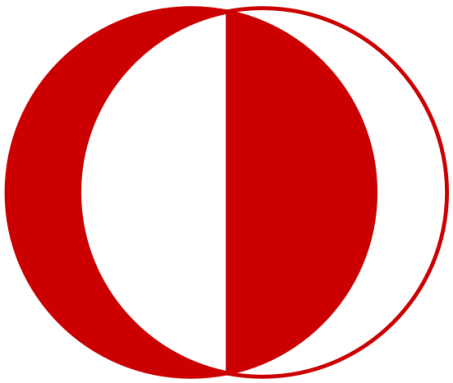

Middle East Technical University
Department of Statistics

STAT 311 – MODERN DATABASE SYSTEMS FINAL PROJECT

EVENTIFY

Jan 2026

Submitted to

Ms. Deniz Çelikel

Rabia Demircan	2623221
Timuçin Eke	2549244
Yusuf Özcan	2614782
Dilara Yıldırım	2614907
Kemal Can Yoloğlu	2614915

Table of Contents

1. Introduction & Project Overview

1.1 Introduction

The aim of this project is to design and implement a normalized relational database system for managing large-scale events and the operational processes associated with them. Most contemporary events need to deal with complex interactions among events, venues, sessions, participants, tickets, payments, sponsors, and organizers in a coordinated way. Quite often, handling such data without structure or proper connectivity leads to redundancy, inconsistencies, and eventually to problems with data integrity.

In this regard, the project will deliver a relationally correct database design that can accurately model any event management scenario in the real world with consistency, scalability, and efficient data retrieval.

1.2 Project Overview

The designed database system in this project, Eventify, handles events from planning to attending to gaining feedback. This designed system incorporates all major entities like events, venues, halls, sessions, speakers, participants, registrations, tickets, payments, sponsors, and organizers, as well as relationships between these entities.

A conceptual Entity-Relationship (ER) design is created to model the actual structure in the problem domain as a first step. The conceptual design is then converted to a relational database design by employing primary and foreign keys and integrity constraints to ensure conformance to normalization rules. The database will be implemented in PostgreSQL, and sample data will be created in SQL statements to reflect realistic data usage scenarios.

The database scheme generated is optimal as far as the analysis of data and user interface is concerned and can easily accommodate future additions in the form of graphical interfaces.

---

2. Project Aim & Objectives

The main aim of this project is developing a strong, normalized relational database system for handling events that reflects real-world conditions. Such a database system would capture intricate linkages between events, venues, sessions, participants, tickets, payments, sponsors, and organizers."

In order to fulfill this objective, the following specific goals have been set by the project:

- Enumerate the key entities involved in an event management system and assign attributes based on practical use cases.
- Build a full Entity–Relationship diagram illustrating the logical connections between entities, entailed one-to-many and many-to-many relationships.)
- Converting the conceptual design of ER model into a Relational Schema, ensuring Normalization techniques, designing appropriate Primary Key, Foreign Key, and Integrity constraints.
- To implement the relational schema by using PostgreSQL and maintain data consistency by using Constraints like UNIQUE, CHECK, and referential integrity constraints to generate mock data directly in SQL to model realistic system usage and facilitate execution of SQL commands on the data.

- Ensure database architecture scalability and logic consistency in preparation for further development of visualization and reporting of data using front-end techniques.

By satisfying this criterion, this project can prove the entire process of designing a database from conceptual to operational as per the up-to-date principles of designing a database system.

---

3. Data Model Design

3.1 ER Diagram Overview

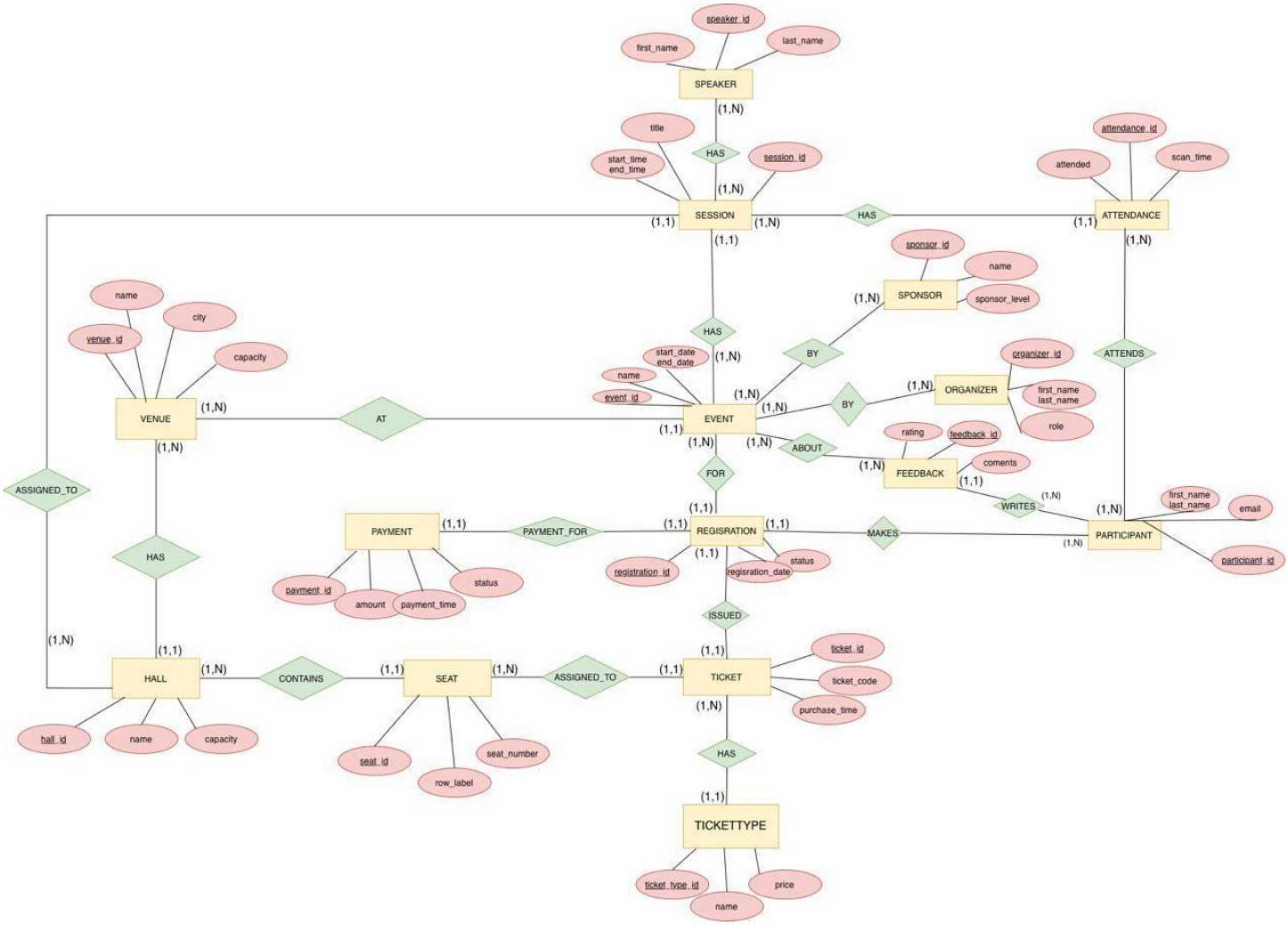

Figure 1. Conceptual Entity–Relationship (ER) diagram of the Eventify database system.

The conceptual ER diagram depicts the structure of the Eventify database, identifying the main entities involved in event management and their relationships. The rectangles represent entities such as Event, Participant, Session, Ticket, and Payment. The event is the central entity, because almost all other entities are directly or indirectly related to it. The ovals are attributes such as *name, start_date, status, and price*. The underlined attributes are primary keys such as event_id, participant_id, and registration_id. A primary key uniquely identifies each record in a table. The diamonds represent relationships between entities. For example, *"Participant MAKES Registration"* and *"Ticket HAS Tickettype"* are relationships. In addition, the diagram shows the cardinalities between entities such as (1,N) and (1,1).

3.2 Core Entities and Attribute Examples

The Eventify system includes several core entities that are associated with different roles and processes within an event management system . Key entities are **Event, Venue, Hall, Seat, Session, Speaker, Participant, Organizer, Sponsor, Registration, Ticket, TicketType, Payment, Feedback, and Attendance**.

For instance, some of entity attributes contain:

- **Participant**: `participant_id`, `first_name`, `last_name`, `email`, `phone`, `registered_at`, where email address is defined as unique, to prevent duplicate registration.
- **Event**: `event_id`, `name`, `event_type`, `start_date`, `end_date`, and `venue_id`, with date constraints to ensures that end date is not earlier than start date.
- **Session**: `session_id`, `event_id`, `hall_id`, `title`, `start_time`, and `end_time`, ensure that sessions are correctly linked to events and scheduled within valid time intervals.
- **Seat**: `seat_id`, `hall_id`, `row_label`, and `seat_number`, uniquely defined in each hall to prevent duplication.

3.3 Relationship Design and Cardinalities

The ER model contains different relationship types such as one-to-one, one-to-many, many-to-many.
For instance:

- **Venue–Hall (1–N)**: A venue may contain multiple halls, but each hall belongs to only one venue.
- **Hall–Seat (1–N)**: Each hall has multiple seats, and each seat belongs to a single hall.
- **Event–Venue (N–1)**: Each event is located at one venue, but venues may host multiple events.
- **Event–Session (1–N)**: An event can have one or many sessions, and each session can be related to exactly one event.
- **Session–Speaker (N–N)**: Sessions may have multiple speakers, and speakers may participate in multiple sessions.
- **Event–Sponsor (N–N)**: Events can have multiple sponsors, and sponsors may support multiple events.
- **Participant–Registration (1–N)**: A participant may register for more than one event, but each registration belongs to one participant ; this is applied by a uniqueness constraint.
- **Registration–Ticket (1–1)**: Each registration has one ticket and each ticket belongs to exactly one registration .
- **TicketType–Ticket (1–N)**: A ticket type may correspond to many tickets, while each ticket belongs to a single ticket type.

3.4 Relational Schema (ERD) and Implementation Mapping

Figure 2. Relational schema (ERD) generated from the ER model and implemented in PostgreSQL.

The conceptual ER model is transformed into a relational schema by mapping entities to tables and relationships to foreign keys or associative tables. Primary keys uniquely identify each record, while foreign keys enforce referential integrity across related tables. Many-to-many relationships, such as **Session–Speaker** and **Event–Sponsor**, are implemented using junction tables with composite primary keys.

Additional integrity constraints, including `UNIQUE`, `CHECK`, and cascading rules, are applied to preserve consistency and prevent invalid data states. This relational design ensures that the logical schema remains faithful to the conceptual model while being fully compatible with SQL-based implementation.

---

4. Data Generation and Implementation

4.1 Implementation Approach

The database is built by using PostgreSQL. The setup process is handled by a SQL script which enables repeatable execution. In necessary cases existing tables are deleted then table creation statements that define attributes, keys and constraints are executed.

4.2 Mock Data Generation

The mock data is generated in PostgreSQL by using SQL INSERT statements and built-in functions such as generate_series. This approach provides a convenient dataset.

Selected SQL codes are shown below in order to exemplify the data generation process.

- Venues, events and halls are inserted to indicate different types of events and locations.

```sql
INSERT INTO halls (venue_id, name, capacity) VALUES
(1, 'Main Hall',         800),
(1, 'Workshop Room 1',   150),
(2, 'Concert Hall',      1000),
(2, 'Black Box Room',    200);
```

- Seats are generated for each hall to create structured seating orders.

```sql
INSERT INTO seats (hall_id, row_label, seat_number)
SELECT h.hall_id,
       chr(64 + r) AS row_label,    -- 65='A'
       s AS seat_number
FROM halls h
CROSS JOIN generate_series(1,5) AS r
CROSS JOIN generate_series(1,10) AS s;
```

- Participants are collectively created with unique personal information.

```sql
INSERT INTO participants (participant_id, first_name, last_name, email, phone)
SELECT
    g AS participant_id,
    'Participant_' || g AS first_name,
    'User_' || g AS last_name,
    'user' || g || '@example.com' AS email,
    '555-' || LPAD(g::text, 4, '0') AS phone
FROM generate_series(1, 200) AS g;
```

- Registrations, tickets, and payments are inserted to simulate the entire process of attending an event.

```sql
INSERT INTO payments (registration_id, amount, payment_date, payment_method, status)
SELECT
    r.registration_id,
    tt.price,
    r.registration_date + INTERVAL '2 hours' AS payment_date,
    CASE WHEN r.registration_id % 5 = 0 THEN 'bank_transfer' ELSE 'credit_card' END AS payment_method,
    CASE WHEN r.registration_id % 7 = 0 THEN 'failed' ELSE 'paid' END AS status
FROM registrations r
JOIN tickets t   ON t.registration_id = r.registration_id
JOIN ticket_types tt ON tt.ticket_type_id = t.ticket_type_id
WHERE r.registration_id % 7 <> 0;
```

- Attendance and feedback are inserted to picture the interactions after an event.

```sql
INSERT INTO attendance (session_id, participant_id, attended, scan_time)
SELECT
    s.session_id,
    r.participant_id,
    (r.status = 'attended') AS attended,
    CASE
        WHEN r.status = 'attended'
        THEN s.start_time + INTERVAL '5 minutes'
        ELSE NULL
    END AS scan_time
FROM registrations r
JOIN sessions s
  ON s.event_id = r.event_id
WHERE s.session_id IN (
    -- Her event için ilk 2 session
    SELECT s2.session_id
    FROM sessions s2
    WHERE s2.event_id = s.event_id
    ORDER BY s2.start_time
    LIMIT 2
);
```

4.3 Integrity Enforcement Through Constraints

Data integrity is ensured by using schema-level constraints, such as:

- Non-negative capacity constraints for venues and halls.
- Time constraints providing valid session plans.
- Uniqueness constraints that prevent duplicate registrations for the same event.
- One-to-one enforcement between registrations and tickets.

These constraints assure that all generated data reflect real life systems and there are no irrational operations.

---

5. Example Queries and Outputs

Aim of this section is providing sample SQL queries and their outputs to represent different queries and possible usage scenarios.
These examples contain simple return queries, multi-table joins, some grouping and aggregation operations, conditional filtering, common table expressions and applicability focused queries.

5.1 Event and Venue Information Retrieval

```sql
SELECT
  e.event_id,
  e.name AS event_name,
  e.event_type,
  e.start_date,
  e.end_date,
  v.name AS venue_name,
  v.city
FROM events e
JOIN venues v ON v.venue_id = e.venue_id
ORDER BY e.start_date, e.event_id;
```

This query returns event information with its venue details such as its name and the city. This query indicates a basic join operation and shows how event data is supported by attributes of the venue.

	event_id integer	event_name character varying (150)	event_type character varying (50)	start_date date	end_date date	venue_name character varying (100)	city character varying (100)
1	1	Data Science Summit 20…	Conference	2025-04-10	2025-04-11	Ankara Convention Cen…	Ankara
2	2	Indie Music Night	Concert	2025-05-05	2025-05-05	Istanbul Music Hall	Istanbul
3	3	Hands-on Workshop Seri…	Workshop	2025-06-15	2025-06-16	Ankara Convention Cen…	Ankara

Output:

The output displays the list of the events in an order depending on the start date and each row includes the name of the event, its type, start and end date of the event, venue name and the city.

5.2 Session Schedule per Event

```sql
SELECT
  e.event_id,
  e.name AS event_name,
  s.session_id,
  s.title AS session_title,
  s.start_time,
  s.end_time,
  h.name AS hall_name,
  v.name AS venue_name,
  v.city
FROM events e
JOIN sessions s ON s.event_id = e.event_id
JOIN halls h    ON h.hall_id = s.hall_id
JOIN venues v   ON v.venue_id = e.venue_id
ORDER BY e.event_id, s.start_time;
```

This query exemplifies a multi-table join operation across events, sessions, halls and venues. This query indicates the connection between session, events and locations.

	event_id integer	event_name character varying (150)	session_id integer	session_title character varying (150)	start_time timestamp without time zone	end_time timestamp without time zone	hall_name character varying (100)	venue_name character varying (100)	city character va
1	1	Data Science Summit 20...	1	Opening Keynote: Future of...	2025-04-10 09:00:00	2025-04-10 10:00:00	Main Hall	Ankara Convention Cen...	Ankara
2	1	Data Science Summit 20...	2	Workshop: Intro to R for St...	2025-04-10 11:00:00	2025-04-10 13:00:00	Workshop Room 1	Ankara Convention Cen...	Ankara
3	1	Data Science Summit 20...	3	Panel: Ethics in Data Scien...	2025-04-11 14:00:00	2025-04-11 15:30:00	Main Hall	Ankara Convention Cen...	Ankara
4	2	Indie Music Night	4	Opening Act	2025-05-05 19:00:00	2025-05-05 19:45:00	Concert Hall	Istanbul Music Hall	Istanbul
5	2	Indie Music Night	5	Main Performance	2025-05-05 20:00:00	2025-05-05 21:30:00	Concert Hall	Istanbul Music Hall	Istanbul
6	2	Indie Music Night	6	Afterparty DJ Set	2025-05-05 21:45:00	2025-05-05 23:00:00	Black Box Room	Istanbul Music Hall	Istanbul
7	3	Hands-on Workshop Seri...	7	Cloud Essentials	2025-06-15 10:00:00	2025-06-15 12:00:00	Workshop Room 1	Ankara Convention Cen...	Ankara
8	3	Hands-on Workshop Seri...	8	DevOps in Practice	2025-06-15 13:00:00	2025-06-15 15:00:00	Workshop Room 1	Ankara Convention Cen...	Ankara
9	3	Hands-on Workshop Seri...	9	Microservices Design	2025-06-16 09:30:00	2025-06-16 11:30:00	Main Hall	Ankara Convention Cen...	Ankara

Output:
The output shows the title of the sessions, their corresponding events, start and end times and the location of each session.

5.3 Revenue Analysis per Event

Query | Query History

```sql
SELECT
  e.event_id,
  e.name AS event_name,
  COUNT(p.payment_id) AS num_payments,
  SUM(p.amount)       AS total_revenue
FROM events e
JOIN registrations r ON r.event_id = e.event_id
JOIN payments p      ON p.registration_id = r.registration_id
WHERE p.status = 'paid'
GROUP BY e.event_id, e.name
ORDER BY total_revenue DESC;

```

This query aggregates payment amount and counts the number of payments for each event in order to make a financial analysis. It uses aggregation functions such as COUNT and SUM for each event.

	event_id [PK] integer	event_name character varying (150)	num_payments bigint	total_revenue numeric
1	1	Data Science Summit 20...	69	6900.00
2	3	Hands-on Workshop Seri...	43	5160.00
3	2	Indie Music Night	60	4800.00

Output:
The output indicates a row for every single event which reports the number of payments and total revenue gained from the sales.

5.4 Ticket Type Utilization and Capacity Tracking

Query | Query History

```sql
SELECT
  tt.event_id,
  tt.ticket_type_id,
  tt.name AS ticket_type_name,
  tt.capacity,
  COUNT(t.ticket_id) AS sold_tickets,
  (tt.capacity - COUNT(t.ticket_id)) AS remaining_capacity
FROM ticket_types tt
LEFT JOIN tickets t ON t.ticket_type_id = tt.ticket_type_id
GROUP BY tt.event_id, tt.ticket_type_id, tt.name, tt.capacity
HAVING COUNT(t.ticket_id) > 0
ORDER BY tt.event_id, sold_tickets DESC;
```

This query analyzes ticket types by comparing remaining capacities and the number of tickets which are sold for each type. It indicates grouping operations and conditional filtering to evaluate ticket sales.

	event_id integer	ticket_type_id [PK] integer	ticket_type_name character varying (50)	capacity integer	sold_tickets bigint	remaining_capacity bigint
1	1	1	Standard	150	80	70
2	2	3	Regular	600	70	530
3	3	5	Workshop Pass	120	50	70

Output:

The output shows types of tickets, capacities, number of tickets which are sold and remaining capacities.

5.5 Session Attendance Analysis

Query | Query History

```sql
WITH session_attendance AS (
  SELECT
    a.session_id,
    COUNT(*) AS total_scans,
    SUM(CASE WHEN a.attended THEN 1 ELSE 0 END) AS attended_count
  FROM attendance a
  GROUP BY a.session_id
)
SELECT
  s.session_id,
  s.title AS session_title,
  sa.total_scans,
  sa.attended_count,
  ROUND(100.0 * sa.attended_count / NULLIF(sa.total_scans, 0), 2) AS attendance_rate_percent
FROM session_attendance sa
JOIN sessions s ON s.session_id = sa.session_id
ORDER BY attendance_rate_percent DESC, sa.total_scans DESC;
```

This query measures the participation rates for each session by using attendance records. A common table expression is used to calculate attendance information.

	session_id [PK] integer	session_title character varying (150)	total_scans bigint	attended_count bigint	attendance_rate_percent numeric
1	4	Opening Act	70	24	34.29
2	5	Main Performance	70	24	34.29
3	1	Opening Keynote: Future of...	80	26	32.50
4	2	Workshop: Intro to R for St...	80	26	32.50
5	7	Cloud Essentials	50	16	32.00
6	8	DevOps in Practice	50	16	32.00

Output:
The output displays sessions with the number of total scans, number of attended participants and the percentage of attendance.

5.6 Event Feedback Summary

Query | Query History

```
SELECT
  e.event_id,
  e.name AS event_name,
  AVG(f.rating) AS avg_rating,
  COUNT(f.feedback_id) AS num_feedbacks
FROM events e
LEFT JOIN feedback f ON f.event_id = e.event_id
GROUP BY e.event_id, e.name
ORDER BY avg_rating DESC NULLS LAST, num_feedbacks DESC;
```

This query analyzes the feedback of participants by calculating average ratings and number of feedbacks per event.

	event_id [PK] integer	event_name character varying (150)	avg_rating numeric	num_feedbacks bigint
1	1	Data Science Summit 20...	3.0000000000000000	80
2	2	Indie Music Night	3.0000000000000000	70
3	3	Hands-on Workshop Seri...	[null]	0

Output:
The output shows the total number of feedback and average rating for each event, providing understanding of participant satisfaction.

5.7 Sponsor Contribution Overview

Query | Query History

```sql
SELECT
  e.event_id,
  e.name AS event_name,
  s.sponsor_level,
  COUNT(es.sponsor_id) AS sponsor_count,
  SUM(es.contribution_amount) AS total_sponsorship
FROM events e
JOIN event_sponsors es ON es.event_id = e.event_id
JOIN sponsors s        ON s.sponsor_id = es.sponsor_id
GROUP BY e.event_id, e.name, s.sponsor_level
ORDER BY total_sponsorship DESC;
```

By aggregating contribution amounts of sponsors for each event this query analyzes sponsorship data . It highlights how many-to-many relationships are handled with relational tables.

	event_id integer	event_name character varying (150)	sponsor_level character varying (50)	sponsor_count bigint	total_sponsorship numeric
1	1	Data Science Summit 20...	Platinum	1	50000.00
2	1	Data Science Summit 20...	Gold	1	25000.00
3	3	Hands-on Workshop Seri...	Platinum	1	20000.00
4	2	Indie Music Night	Silver	1	15000.00

Output:

The output indicates the total amount of contribution for events grouped by sponsor level. It allows the comparison of sponsor contributions.

5.8 Schedule Conflict Detection

Query | Query History

```sql
SELECT
  s1.hall_id,
  s1.session_id AS session_a,
  s1.title      AS title_a,
  s1.start_time AS start_a,
  s1.end_time   AS end_a,
  s2.session_id AS session_b,
  s2.title      AS title_b,
  s2.start_time AS start_b,
  s2.end_time   AS end_b
FROM sessions s1
JOIN sessions s2
  ON s1.hall_id = s2.hall_id
 AND s1.session_id < s2.session_id
 AND s1.start_time < s2.end_time
 AND s2.start_time < s1.end_time
ORDER BY s1.hall_id, s1.start_time;
```

As an example which focuses on integrity, this query checks for conflicting session schedules within the same hall. While basic time constraints are applied at the schema level, this query provides an additional analytical validation of scheduling applicability.

hall_id integer	session_a integer	title_a character varying (150)	start_a timestamp without time zone	end_a timestamp without time zone	session_b integer	title_b character varying (150)	start_b timestamp without time zone	end

Output:

The output returned no rows for our current dataset, which means that there are no conflicting sessions for the

same hall. This result approves that there is no hall which is booked at the same time so the current schedule is consistent and applicable.

6. Frontend Visualization and User Interface

A live version of the frontend application is available at:
https://kemalcanyologlu.github.io/eventify-frontend/

A user-side frontend that works in an interactive and user-friendly way was developed to support the relational database design and SQL based data processing.

Instead of using a backend server, a frontend that directly uses the CSV files which are exported from the PostgreSQL database is utilized.

The data flow of the system follows the pipeline which is shown below:

PostgreSQL → SQL Queries → CSV Files → JavaScript Objects → User Interface

This approach preserves database integrity and query logic while making efficient analysis of relational data possible.

6.1 Main Frontend Interface

The main frontend interface displays events, sessions, summary of the tickets and participants. In this way, it provides an integrated view of the system.

Figure 6.1 shows the complete frontend layout, where different database entities are displayed as structured tables and cards. The interface updates its content based on user interactions and selected filters.

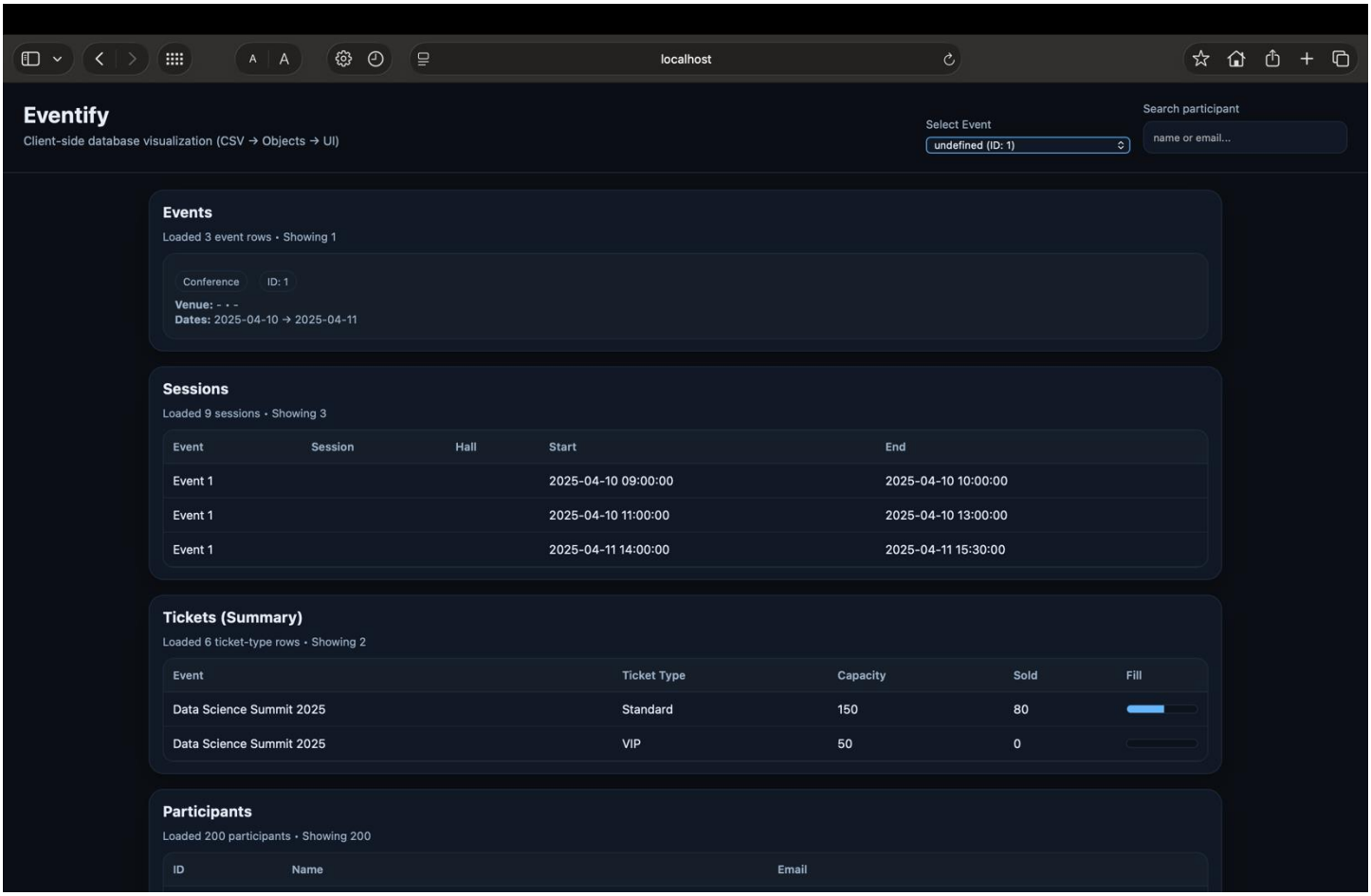

Figure 6.1: Main frontend interface visualizing events, sessions, ticket summaries, and participants.

6.2 Event Selection and Filtering

The interface contains an event selection feature which allows users to filter displayed data by event. When an event is filtered, only the information about that specific event is displayed.
This feature mirrors the use of foreign key relationships in the underlying relational schema, especially the association between events and sessions.

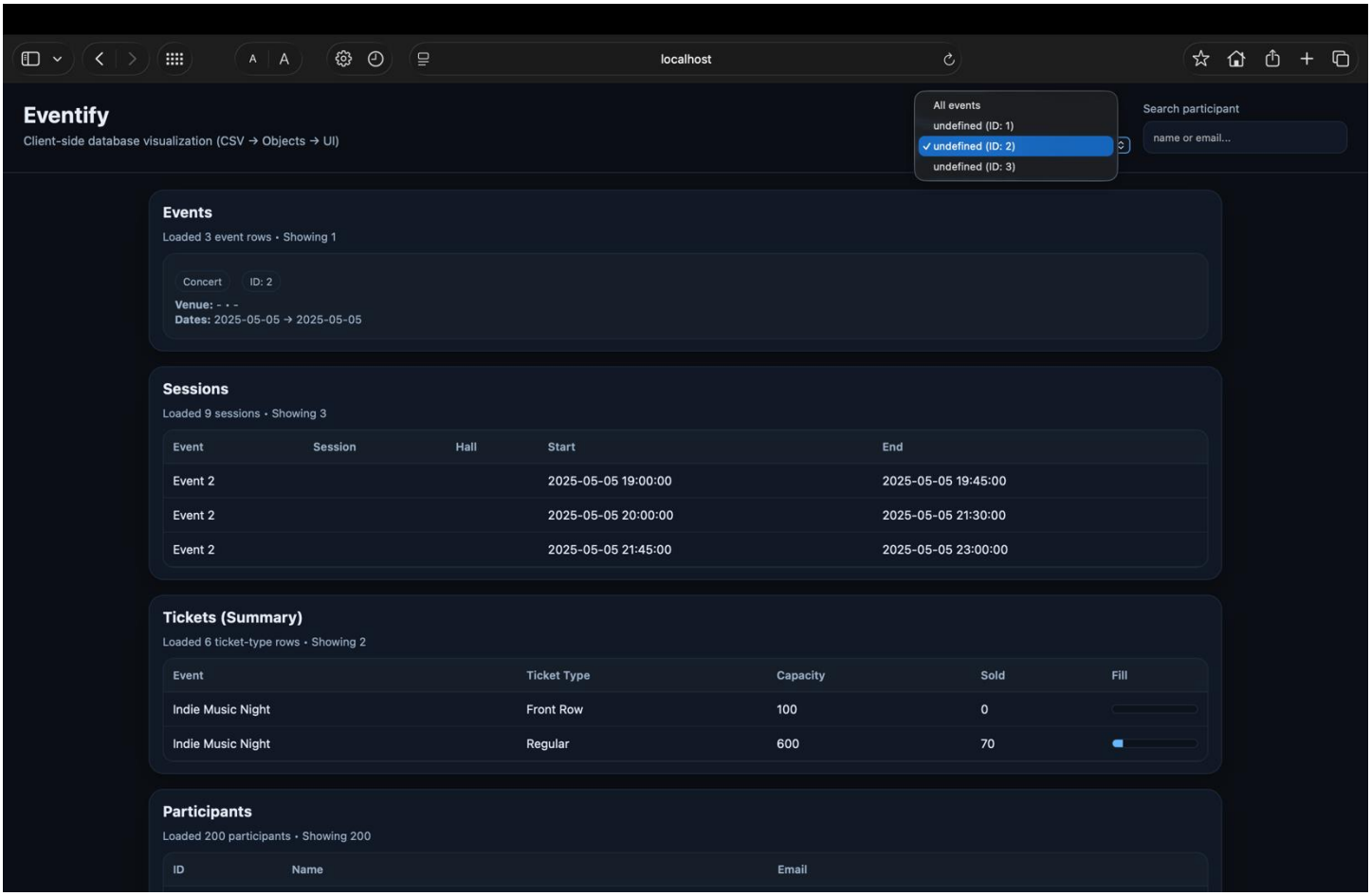

Figure 6.2: Event selection feature is used to filter frontend data by event.

6.3 Ticket Summary Visualization

Ticket data is visualized by using aggregated data which is provided from SQL `GROUP BY` queries . The frontend displays a summarized view of tickets, which shows the type of the ticket, ticket capacity, number of tickets that are sold and fill ratios.

A progress bar is used to display the percentage of the tickets sold; in this way it is easier to interpret the ticket data.

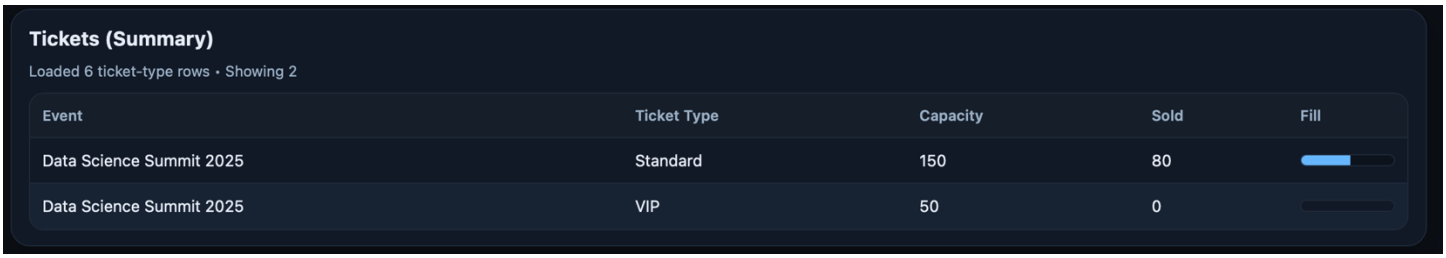

Figure 6.3: Ticket summary showing capacity, sold tickets, and fill ratio by progress bars.

6.4 Participant Search and Interaction

The participants section has a search table which makes filtering participants by name or their email address possible. This functionality indicates user-side data manipulation using JavaScript, where CSV data is separated into objects and dynamically filtered without additional database queries.

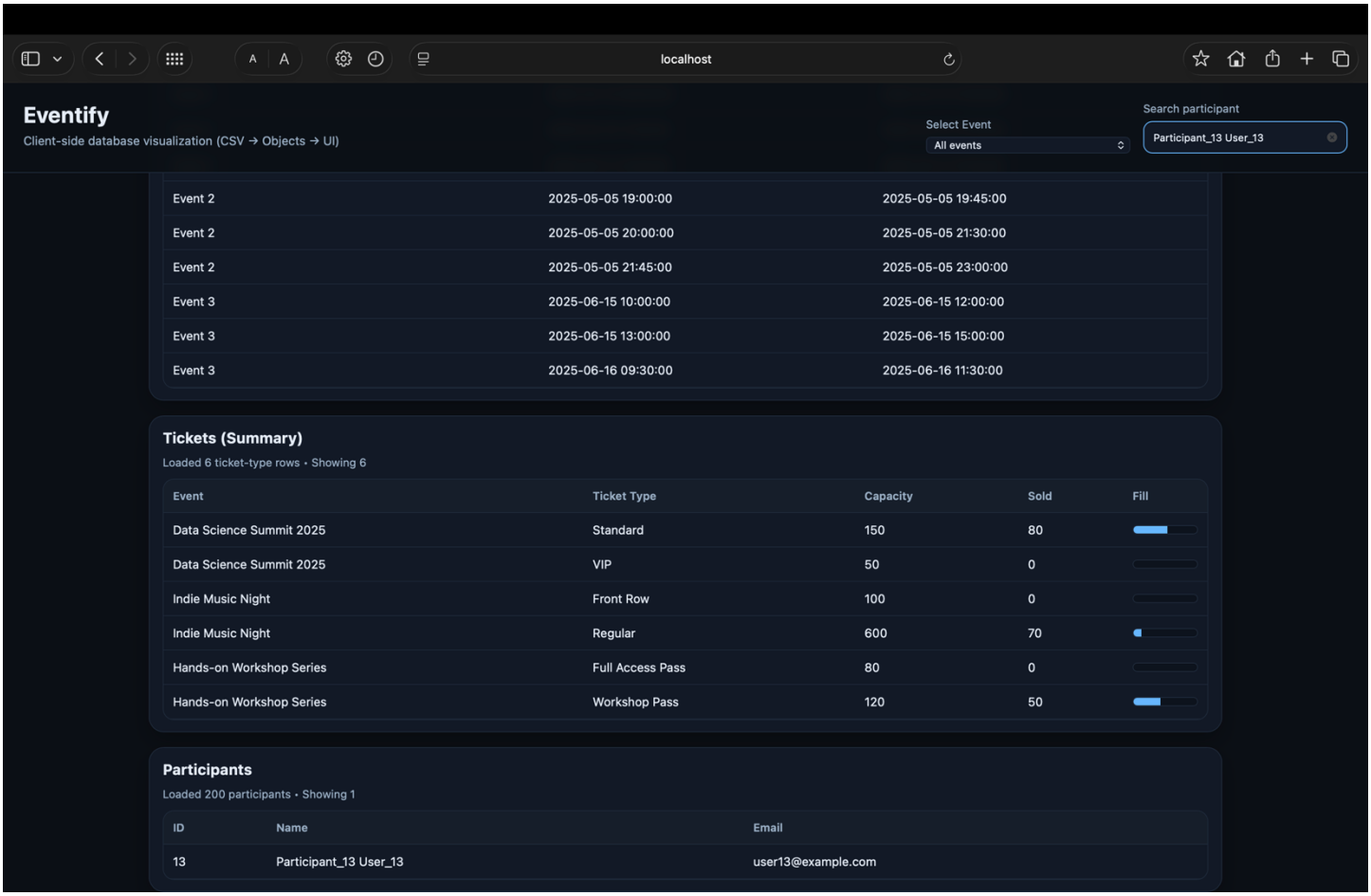

Figure 6.4: Participant table with user-side search functionality.

6.5 Future Work(s)

The current frontend design shows how relational database concepts like entity relationships, aggregation, and filtering can be visualized in a user-side environment using CSV outputs which are provided by using SQL queries. This approach links database design to user interaction, however some improvements can be taken into consideration for future development.

In future studies, the system can be further developed by integrating a backend service to provide real-time data updates and direct database access. Additional features such as advanced filtering, sorting, paging, and role-based access control can improve usability and scalability. Furthermore, the system can be developed in a way that it becomes a more interactive platform for managing events by updating its frontend structure to support different data sources or larger datasets.

7. Author Contributions

Everyone played an equal role in this project.

Kemal Can Yoloğlu, Yusuf Özcan, Dilara Yıldırım, Rabia Demircan, and Timuçin Eke worked together to create the relational database schema, write SQL queries, and organize the data extraction process. The group took part in data modeling, checking queries, and designing integrity constraints as a team. The team put together the frontend visualization, which included processing data creating the user interface, and planning how users would interact with it. Everyone contributed to analyze and interpret the results, make figures, and write the final project report

8. Conclusion

This project showcased the creation and setup of a relational event management database with PostgreSQL. It relied on well-structured SQL queries and rules to keep data consistent. The database aimed to represent events, sessions, tickets, and participants while maintaining proper data organization and relationships.

The project exported query results as CSV files and displayed them through a frontend app on the client side. This showed how relational database ideas can work with modern user interfaces. This method keeps data storage separate from how it's shown, which makes the system easier to move, grow, and understand. In the end, the project put database theory into practice creating a useful and expandable system.