

# CSE 4/560 Project Milestone 2

## 1. Project Detail

### 1.1 Project

#### EVENT MANAGEMENT SYSTEM

### 1.2 Team

#### ONE DIRECTION

### 1.3 Team Members

Name	UBIT name	UBID
Harshitha Bommakanti	hbommaka	50475216
Guru Kavya Sree G	gurukavy	50485454
JayaKalyani Kommanaboina	jayakaly	50455447

## 2. Problem Statement

### 2.1 Description

The proposed event management system database will provide a comprehensive solution for managing multiple events, venues, attendees, and expenses in a single platform. With the increasing trend of events and celebrations, there is a need for an efficient system to manage event-related information. It will allow event organizers to keep track of all the relevant information regarding events. This database contains the details of venue, manager, events, participants, registrations and their payment.

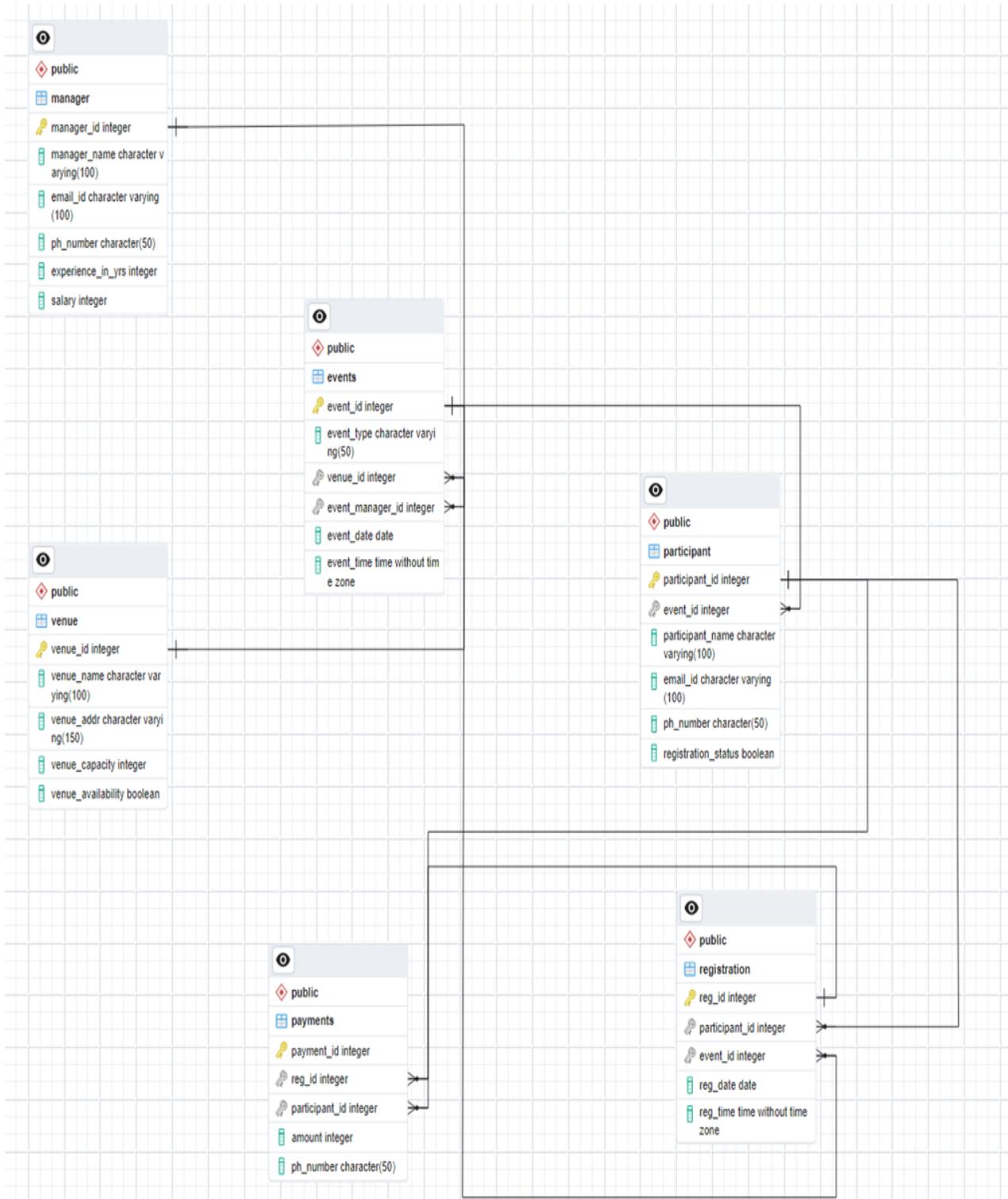
## 2.2 Comparison between DB systems and Excel files

An excel file can be a useful tool for organizing data, but it can quickly become cumbersome and difficult to manage as the amount of data and complexity of the project grows. A database provides a more efficient and scalable way to manage data, making it easier to retrieve, store, and analyze information. Additionally, a database provides greater data security and integrity than an excel file. This database will make the search for event-related information efficient and streamlined, as all the information will be stored in a structured manner in the database, making it easy to access and manage.

## 3. Target User

- The User
  - Event managers who are responsible to manage all aspects of an event, from initial planning and budgeting to attendee registration and post-event analysis.
- The administrator -
  - Owner of the event management system
- Real-life Scenario Description -
  - John is a manager looking to organize a corporate event. He registers as a user in the event management system by providing his details and creating a username and password. The login details are stored in the database. He can now create an event by specifying the venue, date, time, and other details. The event is stored in the database, along with the venue and manager details. Participants can register for the event by providing their details, and the registration details are stored in the database. Payments for the event can be made through the system, and the payment details are also stored in the database. This makes searching, organizing and managing the event much more efficient.

## 4. ER Diagram



## 5. Database Implementation

### 5.1 Data Schemas

- VENUE (venue\_id, venue\_name, venue\_addr, venue\_capacity, venue\_availability);
- MANAGER (manager\_id, manager\_name, email\_id, ph\_number, experience\_in\_yrs,salary);
- EVENT (event\_id, event\_type, venue\_id, event\_manager\_id, event\_date, event\_time)
- PARTICIPANT ( participant\_id,event\_id,participant\_name,email\_id, ph\_number, registration\_status);
- REGISTRATION (reg\_id, participant\_id, event\_id, reg\_date date, reg\_time time)
- PAYMENT (payment\_id, reg\_id integer NOT NULL, participant\_id, amount, ph\_number);

#### a) Venue table

```
Query   Query History
1  create table venue(
2      venue_id integer primary key NOT NULL,
3      venue_name varchar(100) NOT NULL,
4      venue_addr varchar(150) NOT NULL,
5      venue_capacity integer NOT NULL,
6      venue_availability boolean default 'TRUE',
7      );
8
9
```

#### b) Manager Table

```
Query   Query History
9
10 create table manager (
11     manager_id integer primary key NOT NULL,
12     manager_name varchar(100) NOT NULL,
13     email_id varchar(100) NOT NULL,
14     ph_number char(50) NOT NULL,
15     experience_in_yrs integer NOT NULL,
16     salary integer NOT NULL
17     );
18
19
```

c) Event Table

```
Query Query History
19
20 create table events (
21     event_id integer primary key NOT NULL,
22     event_type varchar(50) NOT NULL,
23     venue_id integer NOT NULL,
24     event_manager_id integer NOT NULL,
25     event_date date,
26     event_time time,
27     foreign key(venue_id) references venue(venue_id)
28         on update cascade on delete cascade,
29     foreign key(event_manager_id) references manager(manager_id)
30         on update cascade on delete cascade
31 );
32
33
```

d) Participant Table

```
Query Query History
32
33
34 create table participant (
35     participant_id integer primary key NOT NULL,
36     event_id integer NOT NULL,
37     participant_name varchar(100) NOT NULL,
38     email_id varchar(100) NOT NULL,
39     ph_number char(50) NOT NULL,
40     registration_status boolean default 'FALSE',
41     foreign key(event_id) references events(event_id)
42         on update cascade on delete cascade
43 );
44
45
```

e) Registration Table

```
Query Query History
45
46 create table registration (
47     reg_id integer primary key NOT NULL,
48     participant_id integer NOT NULL,
49     event_id integer NOT NULL,
50     reg_date date,
51     reg_time time,
52     foreign key(participant_id) references participant(participant_id)
53         on update cascade on delete cascade,
54     foreign key(event_id) references events(event_id)
55         on update cascade on delete cascade
56 );
57
```

### 5.1.2 Relationship between the tables :

The '**venue**' table will have a one-to-many relationship with the 'events' table, as one venue can host multiple events. The '**manager**' table will also have a one-to-many relationship with the '**events**' table, as one manager can manage multiple events. The '**participant**' table will have a one-to-many relationship with the 'events' table, as one event can have multiple participants. The '**registration**' table will have a one-to-many relationship with both the 'participant' and 'events' tables, as one participant can register for multiple events, and one event can have multiple participants registered. Finally, the '**payments**' table will have a one-to-one relationship with both the 'registration' and 'participant' tables, as one payment can be associated with one registration and one participant. These relationships are enforced through the use of foreign keys.

## 6. Primary key and Foreign Key

### 1. VENUE TABLE

Primary key - venue\_id : Uniquely identifies each venue in the table

Foreign key : None

### 2. MANAGER TABLE

Primary key - manager\_id : uniquely identifies each manager in the table

Foreign key : None

### 3. EVENT TABLE

Primary key - event\_id : Uniquely identifies each event in the table

Foreign key - venue\_id : on deletion of venue\_id in venue table automatically venue\_id in event table will get deleted

event\_manager\_id : on deletion event\_manager\_id in manager table automatically manager\_id will get deleted

### 4. PARTICIPANT TABLE

Primary key - participant-id : uniquely identifies each participant in the table

Foreign key - event-id : on deletion of event\_id in event table automatically event\_id gets deleted in Participant table

## 5. REGISTRATION TABLE

Primary key - reg\_id : uniquely identifies each id in the registration table

Foreign key - participant-id : on deletion of participant\_id in participant table automatically gets deleted in registration table too

Foreign key - event\_id : on deletion of event\_id in participant table automatically gets deleted in Registration table too

## 6. PAYMENTS TABLE

Primary key - payment\_id : uniquely identifies each id in the payments table (

Foreign key - reg\_id : on deletion of reg\_id in registration table automatically gets deleted in payments table too

participant-id : on deletion of participant\_id in participants table automatically gets deleted in payments table too

## 7. Records Insertion

A Python script was used to insert records into the respective tables of the database. The process began with creating a parent table, followed by randomly generating data for the required columns and inserting it into the table. The child tables were then populated by referencing the records from the parent table and generating other attributes randomly as per the requirements. This automation process allowed for quick and efficient insertion of large amounts of data into the database, reducing the need for manual input and saving time.

Using psycopg2, we connected to the event management database and thus generated fake datasets. We used various python modules i.e., psycopg2, datetime, random, and faker. Below are the screenshots of the records inserted in the table.

Venue

The screenshot shows the Spyder IDE interface. On the left, a code editor displays a Python script named VENUE.py. The script uses Faker to generate random data and insert it into a VENUE table. It connects to a PostgreSQL database, creates a cursor, and executes an INSERT INTO statement for 10,000 rows. On the right, a console window shows the command to run the file and the resulting output for 9999 tuples.

```

VENUE.py X MANAGER.py X EVENTS.py X PARTICIPANT.py X REGISTRATION.py X PAYMENTS.py X
# -*- coding: utf-8 -*-
"""
Created on Fri Mar  3 18:17:31 2023
@author: Jaya Kalyani
"""

from faker import Faker
import psycopg2

# Connect to the database
conn = psycopg2.connect(
    host="127.0.0.1",
    database="Event_Management",
    user="postgres",
    password="postgres"
)

# Create a cursor object
cur = conn.cursor()

# Use Faker to generate random data and insert into the VENUE table
fake = Faker()
for i in range(1,10000):

    venue_id = i
    venue_name = fake.company()
    venue_addr = fake.address()
    venue_capacity = fake.random_int(min=50, max=1000)
    venue_availability = fake.boolean(chance_of_getting_true=50)

    cur.execute("INSERT INTO VENUE (venue_id,venue_name,venue_addr,venue_capacity,venue_availability) VALUES (%s,%s,%s,%s,%s)", (venue_id,venue_name,venue_addr,venue_capacity,venue_availability))

# Commit the changes to the database
conn.commit()

# Close the cursor and database connection
cur.close()
conn.close()

```

In [9]: runfile('C:/Users/Jaya Kalyani/Desktop/DMQL\_PROJ/untitled8.py', wdir='C:/Users/Jaya Kalyani/Desktop/DMQL\_PROJ')

In [10]:

## Venue Details Table

**9999 tuples**

The screenshot shows the pgAdmin interface with a query editor and a data viewer. The query editor contains a simple SELECT statement. The data viewer displays the results of the query, which consists of 9999 rows of venue details. The columns are venue\_id, venue\_name, venue\_addr, venue\_capacity, and venue\_availability.

	venue_id [PK] integer	venue_name character varying (100)	venue_addr character varying (150)	venue_capacity integer	venue_availability boolean
1	1	Nielsen Ltd	30363 Rodriguez Summit Apt. 118 West Bradley, VI 65594	842	true
2	2	Davis LLC	62758 Bates Spur Josestad, RI 95760	356	false
3	3	Anderson, Davidson and White	USS Day FPO AE 07865	297	true
4	4	Diaz-Neal	3835 Washington Fields Apt. 353 New Robertburgh, NY 70004	119	false
5	5	Thompson, Martinez and Price	044 Carol Light Apt. 688 West Douglasborough, CA 49295	371	true
6	6	Hines-Bailey	USS Flores FPO AP 49252	124	true
7	7	Griffin, Ortiz and Lloyd	3116 Rodriguez Spring Potterfort, VA 78199	622	true
8	8	Cooper, Jordan and Johnson	21982 Combs Lake Suite 981 South Derek, MD 53759	884	false
9	9	Greer LLC	13603 Montgomery Manor West Karichester, DE 02211	697	true
10	10	Cunningham-Johnson	268 Gordon Expressway West Meganside, FM 65529	548	true
11	11	Malone-Conner	6196 Christopher Fall Suite 381 Lake Rebeccamouth, TN 42178	771	true
12	12	Knapp, Kirk and Taylor	6759 Johnson Gateway Lake Kennethburgh, IL 93547	227	true
13	13	Reeves-Smith	625 Brown Circle South Amanda, RI 58419	683	false
14	14	Reyes, Bowman and Rodriguez	USS Solis FPO AP 61668	417	true
15	15	Howe PLC	9795 Cortez Loaf West Bobbyberg, DC 57650	749	true
16	16	Evans, Hebert and Lee	230 Shelley Tunnel Apt. 999 Brianmouth, VA 35010	896	true
17	17	Solis-Davis	007 Moreno Track Suite 174 Port Jesusview, CO 40654	277	true
18	18	Allen-Flynn	5001 Sanders Greens Suite 624 Taylormouth, MI 11928	613	false

Total rows: 1000 of 9999    Query complete 00:00:00.323    Ln 1, Col 1

## Manager Table

The screenshot shows the Spyder IDE interface. On the left, the code editor displays a Python script named `VENUE.py`. The code uses the `Faker` library to generate random data for the `VENUE` table. It connects to a PostgreSQL database, creates a cursor, and inserts 4000 rows of data. The right side of the interface shows the `Usage` documentation and the IPython console output.

```

@uthor: Jaya Kalyani
"""

from faker import Faker
import psycopg2

# Connect to the database
conn = psycopg2.connect(
    host="127.0.0.1",
    database="Event_Management",
    user="postgres",
    password="postgres"
)

# Create a cursor object
cur = conn.cursor()

# Use Faker to generate random data and insert into the VENUE table
fake = Faker()
for i in range(1,4000):

    manager_id= i
    manager_name = fake.name()
    email_id=fake.email()
    ph_number=fake.phone_number()
    experience_in_yrs=fake.random_int(min=1, max=20)
    salary=fake.random_int(min=50000,max=100000)

    cur.execute("INSERT INTO MANAGER (manager_id,manager_name,email_id,ph_number,experience_in_yrs,salary) VALUES (%s,%s,%s,%s,%s,%s)",(manager_id,manager_name,email_id,ph_number,experience_in_yrs,salary))

# Commit the changes to the database
conn.commit()

# Close the cursor and database connection
cur.close()
conn.close()

```

**3999 tuples**

The screenshot shows the DBeaver interface with a query window open. The query is `SELECT * FROM MANAGER;`. The results are displayed in a table with 18 rows, showing columns for manager\_id, manager\_name, email\_id, ph\_number, experience\_in\_yrs, and salary.

	manager_id [PK] integer	manager_name character varying (100)	email_id character varying (100)	ph_number character (50)	experience_in_yrs integer	salary integer
1	1	Steven Warner	xjackson@example.com	(842)287-5977x9877	18	702193
2	2	Jordan Watson	joseph08@example.com	920.034.7823x75972	2	289608
3	3	Kimberly Powers	turnerrobert@example.com	+1-178-257-9539x15357	16	546255
4	4	Jennifer Hines	ysmith@example.net	6087576165	10	821654
5	5	Robert Miller	heatherortega@example.net	(408)246-8465x7868	13	337760
6	6	Jacob Johnson	jesse44@example.net	447.417.4913x34899	20	997396
7	7	Samantha Collins	timothylawson@example.net	+1-315-163-4703	19	648731
8	8	Dennis Mendez	mark09@example.net	469.663.9840x78704	12	173799
9	9	Allison Bean	ronaldwilliams@example.org	(695)222-8289	8	491245
10	10	Elaine Bond	qarias@example.com	305-170-9901x82629	17	81167
11	11	Carl Smith	gary76@example.net	+1-870-697-5334	3	85737
12	12	Johnathan Velez	russellwright@example.net	340-446-3224	15	738333
13	13	Dakota Pratt	cooperanthony@example.org	554-270-1154x415	3	705493
14	14	Tiffany Williams	jcruz@example.org	104.376.9118	4	617049
15	15	Nicholas White	morrowhenry@example.com	193-927-8974	4	159035
16	16	Denise Atkins	shannonjohnson@example.com	440.804.3313x2853	20	135767
17	17	Kathleen Anderson	rodney36@example.org	500-863-4381x13352	9	717227
18	18	Andrew Anderson	yfoster@example.net	276-831-3753	18	815477

Total rows: 1000 of 3999 Query complete 00:00:00.153

Ln 1, Col 1

## Events

The screenshot shows the Spyder IDE interface. On the left, several tabs are open: VENUE.py, MANAGER.py, EVENTS.py, PARTICIPANT.py, REGISTRATION.py, and PAYMENTS.py. The EVENTS.py tab contains the following Python code:

```

from faker import Faker
import psycopg2
import random

# Connect to the database
conn = psycopg2.connect(
    host="127.0.0.1",
    database="Event_Management",
    user="postgres",
    password="postgres"
)

# Create a cursor object
cur = conn.cursor()
cur.execute("SELECT venue_id FROM VENUE")
venue_ids = [result[0] for result in cur.fetchall()]
cur.execute("SELECT manager_id FROM MANAGER")
event_manager_ids = [result[0] for result in cur.fetchall()]

# Use Faker to generate random data and insert into the VENUE table
fake = Faker()
for i in range(1,10000):
    event_id = i
    type=["concert","wedding","racing","screening","birthday","anniversary","engagement","reception"]
    event_type = random.choice(type)
    venue_id = random.choice(venue_ids)
    event_manager_id = random.choice(event_manager_ids)
    event_date = fake.date_between(start_date='-5y',end_date='today')
    event_time = fake.time()

    cur.execute("INSERT INTO EVENTS (event_id,event_type,venue_id,event_manager_id,event_date, event_time)", (event_id, event_type, venue_id, event_manager_id, event_date, event_time))

conn.commit()
cur.close()
conn.close()

```

A help window titled "Usage" is open on the right, explaining how to get help for objects in the Editor or Console. It also includes a link to the Spyder tutorial.

**17999 tuples**

The screenshot shows the pgAdmin 4 interface. At the top, there are tabs for "Query" (selected), "Query History", and "Scratch Pad". Below the tabs, the query "1 SELECT\* FROM EVENTS;" is entered. The "Data Output" tab is selected, displaying the results of the query in a grid format. The columns are: event\_id [PK] integer, event\_type character varying (50), venue\_id integer, event\_manager\_id integer, event\_date date, and event\_time time without time zone. The data consists of 17999 rows, showing various event types like baby\_shower, dance, Auctions, etc., across different venues and managed by different event managers at specific dates and times.

	event_id [PK] integer	event_type character varying (50)	venue_id integer	event_manager_id integer	event_date date	event_time time without time zone
1		baby_shower	4178	974	2018-04-13	12:33:16
2		baby_shower	6676	7	2020-03-23	09:56:47
3		Auctions	7041	3269	2020-02-14	12:25:39
4		Product_launches	7535	850	2022-01-21	05:16:08
5		dance	8573	2099	2018-04-13	16:31:20
6		Orientations	5000	161	2022-12-12	16:26:40
7		balls	6855	505	2021-02-25	17:37:50
8		Reunions	8228	831	2018-04-01	00:46:43
9		Product_launches	3458	631	2022-03-17	02:40:31
10		Cultural_event	9415	3591	2022-11-26	21:07:25
11		house_warming	5036	1270	2020-03-16	23:53:12
12		Reunions	1263	85	2020-06-10	00:58:45
13		dance	5128	1407	2018-11-25	09:17:23
14		Art_shows	8149	3999	2021-12-06	01:20:02
15		Hackathons	6761	3348	2020-11-10	21:57:51
16		baby_shower	9540	2568	2019-01-11	04:59:17
17		Product_launches	8827	1953	2020-02-05	07:46:47
18		haldi	40	2437	2022-04-18	05:23:02

Total rows: 1000 of 17999 Query complete 00:00:00.202

Ln 1, Col 1

**PARTICIPANT:**

The screenshot shows the Spyder IDE interface. On the left, there is a code editor with several tabs open: VENUE.py, MANAGER.py, EVENTS.py, PARTICIPANT.py (which is currently selected), REGISTRATION.py, and PAYMENTS.py. The PARTICIPANT.py code uses the Faker library to generate random data and insert it into a database. It connects to a PostgreSQL database, creates a cursor, and then loops through 10,000 iterations to generate participants with names, emails, phone numbers, and registration statuses. Finally, it commits the changes and closes the connection. A help window titled 'Usage' is open, explaining how to get help for objects in the editor or console.

```

from faker import Faker
import psycopg2
import random

# Connect to the database
conn = psycopg2.connect(
    host="127.0.0.1",
    database="Event_Management",
    user="postgres",
    password="postgres"
)

# Create a cursor object
cur = conn.cursor()
cur.execute("SELECT event_id FROM EVENTS")
event_ids= [result[0] for result in cur.fetchall()]

# Use Faker to generate random data and insert into the VENUE table
fake = Faker()
for i in range(1,10000):
    participant_id = i
    event_id = random.choice(event_ids)
    participant_name = fake.name()
    email_id=fake.email()
    ph_number=fake.phone_number()
    registration_status = fake.boolean(chance_of_getting_true=60)

    cur.execute("INSERT INTO PARTICIPANT (participant_id,event_id, participant_name, email_id, ph_number, registration_status)",(participant_id, event_id, participant_name, email_id, ph_number, registration_status))

# Commit the changes to the database
conn.commit()

# Close the cursor and database connection
cur.close()
conn.close()

```

**39999 tuples**

The screenshot shows a database query tool with a 'Query History' tab containing the SQL command 'SELECT \* FROM PARTICIPANT;'. Below it is a 'Scratch Pad' tab. The main area displays the results of the query as a table. The table has 18 rows and 7 columns, corresponding to the fields in the PARTICIPANT table: participant\_id, event\_id, participant\_name, email\_id, ph\_number, and registration\_status. The data includes various names, email addresses, phone numbers, and registration statuses. At the bottom, it shows 'Total rows: 1000 of 39999' and 'Query complete 00:00:00.445'.

	participant_id [PK] integer	event_id integer	participant_name character varying (100)	email_id character varying (100)	ph_number character (50)	registration_status boolean
1		1	Krista Powell	kristiflynn@example.org	015-249-3583x9966	false
2		2	Brendan Hall	kathleen48@example.com	715.713.1040	true
3		3	Thomas Brown	michael41@example.com	990.131.0978	false
4		4	Beth Lopez	madisonbaxter@example.com	784-429-8020	false
5		5	14544	Beth Li	rharding@example.net	true
6		6	11668	Sarah Bowen	schultzpatrick@example.com	false
7		7	15267	Michelle Wright	(435)867-2247x62085	false
8		8	8327	Richard Chan DDS	274-274-4643x97326	false
9		9	3054	Michael Hayes	oking@example.net	true
10		10	2419	Guy Harrington	milleramy@example.com	true
11		11	1221	Sara Blankenship	smithvictoria@example.net	true
12		12	17289	Jason Oconnor	341-802-8657x45859	true
13		13	11046	Christina Duffy	jaydelgado@example.net	false
14		14	15849	Benjamin Meyer	anthony77@example.com	true
15		15	2380	Dr. Heather Moore DDS	699-460-9009	true
16		16	1184	Ronald Lambert	tyronefoster@example.net	true
17		17	10123	Ariana Case	joshua33@example.org	true
18		18	9377	Christopher Woods	(455)410-1297x82226	true

Total rows: 1000 of 39999 Query complete 00:00:00.445 Ln 1, Col 1

## REGISTRATION:

The screenshot shows the Spyder IDE interface. On the left, a code editor displays a Python script named `REGISTRATION.py`. The script uses the `Faker` library to generate random data and insert it into a PostgreSQL database. It connects to a database named `event_Management`, retrieves participant and event IDs, and then inserts 24079 tuples into the `REGISTRATION` table. A `Usage` help window is open on the right, providing information on how to get help for objects.

```

from faker import Faker
import psycopg2
import random

# Connect to the database
conn = psycopg2.connect(
    host="127.0.0.1",
    database="event_Management",
    user="postgres",
    password="postgres"
)

# Create a cursor object
cur = conn.cursor()
cur.execute("SELECT participant_id FROM PARTICIPANT")
participant_ids= [result[0] for result in cur.fetchall()]
cur.execute("SELECT event_id FROM EVENTS")
event_ids = [result[0] for result in cur.fetchall()]

# Use Faker to generate random data and insert into the VENUE table
fake = Faker()
for i in range(1,24079):
    reg_id= i
    participant_id = random.choice(participant_ids)
    event_id = random.choice(event_ids)
    reg_date = fake.date_between(start_date='-5y',end_date='today')
    reg_time = fake.time()

    cur.execute("INSERT INTO REGISTRATION (reg_id,participant_id,event_id,reg_date,reg_time) ")

# Commit the changes to the database
conn.commit()

# Close the cursor and database connection
cur.close()
conn.close()

```

**24079 tuples**

The screenshot shows the pgAdmin 4 interface. A query window is open with the following SQL command:

```
1 SELECT* FROM REGISTRATION;
```

The results are displayed in a table titled "Data Output". The table has 18 rows and 6 columns, with the following data:

	reg_id [PK] integer	participant_id integer	event_id integer	reg_date date	reg_time time without time zone
1	1	21051	15208	2018-04-16	15:06:46
2	2	26750	7361	2019-04-02	07:23:57
3	3	18912	5792	2018-08-02	11:08:55
4	4	6453	14767	2020-10-15	13:18:52
5	5	5286	6168	2021-10-24	10:09:26
6	6	3965	2976	2020-11-23	06:10:20
7	7	31731	7918	2018-06-06	08:49:09
8	8	15439	9489	2020-07-06	04:00:54
9	9	12013	5177	2020-04-15	05:19:17
10	10	37431	12343	2021-10-10	01:13:17
11	11	32291	16599	2020-07-18	23:40:32
12	12	31835	10004	2020-02-19	05:03:08
13	13	15914	5273	2020-12-31	12:36:57
14	14	31808	6149	2018-11-21	14:56:42
15	15	31977	15312	2018-05-30	21:19:53
16	16	17567	17223	2018-06-02	09:16:50
17	17	12074	2284	2019-12-26	16:24:22
18	18	37919	13145	2018-12-08	23:35:52

Total rows: 1000 of 24079 Query complete 00:00:00.264 Ln 1, Col 1

**PAYMENTS:**

The screenshot shows the Spyder IDE interface. On the left, there is a code editor with tabs for VENUE.py, MANAGER.py, EVENTS.py, PARTICIPANT.py, REGISTRATION.py, and PAYMENTS.py. The PAYMENTS.py tab is active, displaying the following Python code:

```

from faker import Faker
import psycopg2
import random

# Connect to the database
conn = psycopg2.connect(
    host="127.0.0.1",
    database="Event_Management",
    user="postgres",
    password="postgres"
)

# Create a cursor object
cur = conn.cursor()
cur.execute("SELECT participant_id FROM PARTICIPANT")
participant_ids= [result[0] for result in cur.fetchall()]
cur.execute("SELECT reg_id FROM REGISTRATION")
reg_ids = [result[0] for result in cur.fetchall()]

# Use Faker to generate random data and insert into the VENUE table
fake = Faker()
for i in range(1,24080):
    payment_id= i
    participant_id = random.choice(participant_ids)
    reg_id = random.choice(reg_ids)
    amount = fake.random_int(min=1000, max=10000)
    ph_number=fake.phone_number()

    cur.execute("INSERT INTO PAYMENTS (payment_id,participant_id,reg_id,amount,ph_number) VALUES (%s,%s,%s,%s,%s)",(payment_id,participant_id,reg_id,amount,ph_number))

# Commit the changes to the database
conn.commit()

# Close the cursor and database connection
cur.close()
conn.close()

```

To the right of the code editor is a "Usage" help panel and a "Console 7/A" tab where the command `In [9]: runfile('C:/Users/Jaya Kalyani/Desktop/DMQL_PROJ/untitled8.py', wdir='C:/Users/Jaya Kalyani/Desktop/DMQL_PROJ')` has been run.

## 24079 tuples

The screenshot shows the pgAdmin 4 interface. At the top, there are tabs for Query, Query History, and Scratch Pad. Below the tabs, a query window contains the SQL command:

```
1 SELECT* FROM PAYMENTS;
```

Underneath the query window is a Data Output tab showing the results of the query. The results are presented in a table with the following columns:

	payment_id [PK] integer	reg_id integer	participant_id integer	amount integer	ph_number character (50)
1		1	9785	35601	5705 001-573-231-1514x2725
2		2	1883	12777	8537 343-570-8249
3		3	14961	19717	8409 +1-956-752-5234
4		4	21644	10458	1360 +1-548-977-5565x335
5		5	4411	37935	8378 090.023.3041x060
6		6	1666	13879	6134 +1-665-016-9046x7417
7		7	23554	6531	6080 863-032-5999
8		8	12245	18626	1968 001-297-532-4673
9		9	17273	454	8635 633.909.4766x1954
10		10	2563	21807	6848 746.066.4076
11		11	21193	10710	8883 953.908.8581
12		12	21379	26041	3149 +1-294-669-7712x651
13		13	22735	16244	8202 428-609-2036
14		14	3738	12520	5515 593.859.7507
15		15	3713	37947	9685 933-940-5588x5738
16		16	16649	11922	1303 3142425053
17		17	14717	22014	8653 4972573622
18		18	1319	27719	9902 (689)193-4370x4606

At the bottom of the pgAdmin interface, it says "Total rows: 1000 of 24079" and "Query complete 00:00:00.170".

Ln 1, Col 1

## 8. Queries

### 1. Using the SELECT and WHERE clause

Get all the details of all the business events from the event table.

The screenshot shows a database interface with two tabs: "Query" and "Query History". The "Query" tab contains the SQL code:

```
1  SELECT* FROM events WHERE EVENT_TYPE = 'business_event';
```

The "Data Output" tab displays the results of the query. The table has the following columns:

	event_id [PK] integer	event_type character varying (50)	venue_id integer	event_manager_id integer	event_date date	event_time time without time zone
1	23	business_event	2009	1780	2019-02-28	10:31:05
2	36	business_event	5798	2047	2018-08-31	07:37:01
3	43	business_event	4470	2611	2018-10-31	08:40:01
4	65	business_event	3366	2976	2020-02-18	04:40:52
5	186	business_event	5520	1603	2018-08-16	02:44:24
6	188	business_event	2174	1448	2018-08-27	20:28:08
7	286	business_event	8714	1539	2020-10-22	04:58:19
8	289	business_event	3688	3820	2022-04-29	11:11:29
9	290	business_event	598	2945	2021-02-15	21:27:39
10	304	business_event	101	2962	2021-10-09	10:56:21
11	306	business_event	5663	1012	2020-06-30	07:43:09
12	352	business_event	5787	2424	2021-12-29	20:04:25

Total rows: 547 of 547    Query complete 00:00:00.215    Ln 1, Col 57

### 2. Using the SELECT and JOIN clause

Get all the event\_id, venue\_id, manager\_id and his name by joining events and manager table.

Query    Query History    Scratch Pad

```

1 SELECT e.event_id, e.venue_id, m.manager_id, m.manager_name
2 FROM events e
3 JOIN manager m
4 ON e.event_manager_id=m.manager_id;

```

Data Output    Messages    Notifications

	event_id	venue_id	manager_id	manager_name
1	1	4178	974	Michelle Solis
2	2	6676	7	Samantha Collins
3	3	7041	3269	Joshua Jones
4	4	7535	850	Melissa Lynn
5	5	8573	2099	Mark Santos
6	6	5000	161	Michael Terrell
7	7	6855	505	Mr. David Murphy
8	8	8228	831	Martha Carpenter
9	9	3458	631	Sierra Lee
10	10	9415	3591	Jeanette Black
11	11	5036	1270	Jennifer Mullen
12	12	1263	85	Douglas Watkins

Total rows: 1000 of 17999    Query complete 00:00:00.123    Ln 4, Col 36

### 3. Using the SELECT clause and WHERE clause

To retrieve the participants' details who are not in the registration table by using the subquery.

Query    Query History    Scratch Pad

```

1 SELECT participant_id, event_id, participant_name
2 FROM participant
3 WHERE participant_id
4 NOT IN (SELECT participant_id
5          FROM registration);

```

Data Output    Messages    Notifications

	participant_id	event_id	participant_name
1	1	7743	Krista Powell
2	2	127	Brendan Hall
3	3	9243	Thomas Brown
4	4	9468	Beth Lopez
5	9	3054	Michael Hayes
6	13	11046	Christina Duffy
7	15	2380	Dr. Heather Moore DDS
8	20	13290	Robert Lawson
9	22	13714	Jeffrey Roth
10	24	7452	Arthur Barnett
11	25	6157	Bethany Elliott
12	27	4316	Jeffrey Taylor

Total rows: 1000 of 21871    Query complete 00:00:00.136    Ln 1, Col 1

#### 4. Using the SELECT clause and GROUP BY and ORDER BY

To Get all the grouping the event types and counting the number of persons registered for each event and ordering them by event type.

The screenshot shows a database query interface with two tabs: 'Query' and 'Scratch Pad'. The 'Query' tab contains the following SQL code:

```

1 SELECT e.event_type, count(reg.event_id) as tot_mem_registered
2 FROM events e
3 JOIN registration reg
4 ON e.event_id = reg.event_id
5 GROUP BY e.event_type
6 ORDER BY event_type ASC;
7

```

The 'Data Output' tab displays the results of the query, which is a table with two columns: 'event\_type' and 'tot\_mem\_registered'. The data is as follows:

event_type	tot_mem_registered
anniversary	704
Art_shows	653
Auctions	725
Awards	787
baby_shower	690
balls	697
birthday	719
business_event	796
concert	749
Cultural_event	747
dance	706
Dinners	734
engagement	650
flash_mob	721

Below the table, the status bar shows 'Total rows: 33 of 33' and 'Query complete 00:00:00.070'. To the right, it says 'Ln 1, Col 1'.

#### 5. Update

This query can be used to update one or more attributes of a specific venue record.

The screenshot shows a database query interface with a single 'Query' tab. The code entered is:

```

1
2
3 UPDATE venue
4 SET venue_availability = FALSE
5 WHERE venue_id = 123;
6
7
8
9
10
11 UPDATE 1

```

Below the code, the message 'Query returned successfully in 50 msec.' is displayed.

## 6. Delete query

A "Delete participant" query is a database query used to remove one or more participant records from the "Participant" table. This query can be used to delete a specific participant record based on the participant ID, or to delete multiple participant records based on various conditions.

```
Query
8
9 DELETE FROM registration
10 WHERE participant_id = 456;
11
12 DELETE FROM participant
13 WHERE participant_id = 456;
14
15
16
17
18 DELETE 1

Query returned successfully in 125 msec.

Total rows: 1000 of 17999 Query complete 00:00:00.125
```

## 7. Left Outer Join

This query will return a list of all participants and their registration details (registration date and time), including those who have not yet registered for any events (i.e., their registration details will be NULL). The LEFT OUTER JOIN ensures that all records from the left table (participant) are included in the result set, even if there is no matching record in the right table (registration).

Query

```

1
2 SELECT participant.participant_id, participant.participant_name,
3 registration.reg_date, registration.reg_time
4 FROM participant
5 LEFT OUTER JOIN registration
6 ON participant.participant_id = registration.participant_id;
7
8
9
10
11
```

	participant_id integer	participant_name character varying (100)	reg_date date	reg_time time without time zone
1	21051	Angela Hamilton	2018-04-16	15:06:46
2	26750	Jacqueline Bailey	2019-04-02	07:23:57
3	18912	Sarah Daniels	2018-08-02	11:08:55
4	6453	Kaitlyn Johnson	2020-10-15	13:18:52
5	5286	Robert Davis	2021-10-24	10:09:26
6	3965	Michelle Kim	2020-11-23	06:10:20
7	31731	Michelle Simmons	2018-06-06	08:49:09
8	15439	Kim Brooks	2020-07-06	04:00:54
9	12013	Wesley Parker	2020-04-15	05:19:17
10	37431	Daniel Davies	2021-10-10	01:13:17
11	32291	Joshua Winters	2020-07-18	23:40:32
12	31835	Mary Jackson	2020-02-19	05:03:08
13	15914	Jennifer Davis	2020-12-31	12:36:57

Total rows: 1000 of 45948    Query complete 00:00:00.102

## 8. Function

A function named "check\_payment\_status" is created. It checks if a payment with the given payment\_id and reg\_id exists in the "payments" table. If the payment doesn't exist, the function inserts a new payment record with the given payment\_id, reg\_id, amount, phone number, and participant\_id. The function then returns the count of payments with the given payment\_id and reg\_id.

The screenshot shows a PostgreSQL query editor interface. The top bar has tabs for "Query" and "Query History" (selected), and "Scratch Pad". The main area contains the following SQL code:

```

1 CREATE OR REPLACE FUNCTION check_payment_status(pymnt_id int, rgst_id int, amnt int,
2 RETURNS INTEGER
3 AS
4 $$
5 DECLARE payment_cnt INTEGER;
6 BEGIN
7 SELECT COUNT(*)
8 INTO payment_cnt
9 FROM payments
10 WHERE payment_id = pymnt_id AND reg_id = rgst_id;
11
12 IF payment_cnt = 0 THEN
13     INSERT INTO payments VALUES (payment_id, reg_id, amount, ph_number, participant_i
14 END IF;
15 RETURN payment_cnt;
16 END;
17 $$ LANGUAGE plpgsql;

```

Below the code, the "Messages" tab is selected in a panel titled "Data Output Messages Notifications". The message log shows:

- CREATE FUNCTION
- Query returned successfully in 93 msec.

At the bottom, status bars show "Total rows: 0 of 0" and "Query complete 00:00:00.093". The cursor position is "Ln 17, Col 21".

The screenshot shows a PostgreSQL query editor interface. The top bar has tabs for "Query" and "Query History" (selected), and "Scratch Pad". The main area contains the following SQL code:

```

20 SELECT check_payment_status(1,9785,5705,'001-573-231-1514x2725',35601);
21
22
23
24
25
26
27
28
29
30
31
32

```

Below the code, the "Data Output" tab is selected in a panel titled "Data Output Messages Notifications". The data grid shows the result of the function call:

	check_payment_status	integer
1		1

## 9. Create View

This SQL query creates a view named "vw\_available\_venues". The view selects the venue\_id, venue\_name, venue\_addr, and venue\_capacity columns from the "venue" table where the venue\_availability column is 'true'. This view will show all available venues based on the value of the venue\_availability column, and it can be queried like a table to retrieve this information.

Query    Query History

```

22 CREATE OR REPLACE VIEW vw_available_venues
23 AS
24 SELECT venue_id, venue_name, venue_addr, venue_capacity
25 FROM venue
26 WHERE venue_availability = 'true';
27
28
29
30
31
32
33
34
35
36
37 Data Output    Messages    Notifications
38
39 CREATE VIEW
40
41 Query returned successfully in 145 msec.

```

28

```

29 SELECT * FROM vw_available_venues;
30
31 Data Output    Messages    Notifications
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51

```

	venue_id integer	venue_name character varying (100)	venue_addr character varying (150)	venue_capacity integer
1	1	Nielsen Ltd	30363 Rodriguez Summit Apt. 118 West Bradley, VI 65594	842
2	3	Anderson, Davidson and White	USS Day FPO AE 07865	297
3	5	Thompson, Martinez and Price	044 Carol Light Apt. 688 West Douglasborough, CA 49295	371
4	6	Hines-Bailey	USS Flores FPO AP 49252	124
5	7	Griffin, Ortiz and Lloyd	3116 Rodriguez Spring Potterfort, VA 78199	622
6	9	Greer LLC	13603 Montgomery Manor West Karichester, DE 02211	697
7	10	Cunningham-Johnson	268 Gordon Expressway West Meganside, FM 65529	548
8	11	Malone-Conner	6196 Christopher Fall Suite 381 Lake Rebeccamouth, TN 42178	771
9	12	Knapp, Kirk and Taylor	6759 Johnson Gateway Lake Kennethburgh, IL 93547	227
10	14	Reyes, Bowman and Rodriguez	USS Solis FPO AP 61668	417
11	15	Howe PLC	9795 Cortez Loaf West Bobbyberg, DC 57650	749
12	16	Evans, Hebert and Lee	230 Shelley Tunnel Apt. 999 Brianmouth, VA 35010	896
13	17	Salis Davis	007 Marlene Track Suite 17A Port. Incorruptible, CO 10651	277

Total rows: 1000 of 4927    Query complete 00:00:00.088    Ln 29, Col 1

## 9. Data Normalization

The following is a summary of the functional dependencies (FDs) and the satisfaction of the Boyce-Codd Normal Form (BCNF) in each of the six tables:

- a. VENUE TABLE: The FD  $\text{venue\_id} \rightarrow \text{venue\_name}, \text{venue\_addr}, \text{venue\_capacity}, \text{venue\_availability}$  holds, and it is nontrivial. The left-hand side of the FD is a super key, so the BCNF condition is satisfied.
- b. MANAGER TABLE: The FD  $\text{manager\_id} \rightarrow \text{manager\_name}, \text{email\_id}, \text{ph\_number}, \text{experience\_in\_yrs}, \text{salary}$  holds, and it is nontrivial. The left-hand side of the FD is a super key, so the BCNF condition is satisfied.
- c. EVENT TABLE: The FD  $\text{event\_id} \rightarrow \text{event\_type}, \text{venue\_id}, \text{event\_manager\_id}, \text{event\_date}, \text{event\_time}$  holds, and it is nontrivial. The left-hand side of the FD is a super key, so the BCNF condition is satisfied.
- d. PARTICIPANT TABLE: The FD  $\text{participant\_id} \rightarrow \text{event\_id}, \text{participant\_name}, \text{email\_id}, \text{ph\_number}, \text{registration\_status}$  holds, and it is nontrivial. The left-hand side of the FD is a super key, so the BCNF condition is satisfied.
- e. REGISTRATION TABLE: The FD  $\text{reg\_id} \rightarrow \text{participant\_id}, \text{event\_id}, \text{reg\_date}, \text{reg\_time}$  holds, and it is nontrivial. The left-hand side of the FD is a super key, so the BCNF condition is satisfied.
- f. PAYMENTS TABLE: The FD  $\text{payment\_id} \rightarrow \text{reg\_id}, \text{participant\_id}, \text{amount}, \text{ph\_number}$  holds, and it is nontrivial. The left-hand side of the FD is a super key, so the BCNF condition is satisfied.

## 10. Query Optimization

After indexing the queries, it is important to assess their performance, cost, and time. The optimization has resulted in very efficient performance and cost indicating that the

indexing has significantly improved the performance of the queries and has resulted in a more efficient use of system resources.

- The first query uses a JOIN operation to combine data from two tables, which can be more efficient than running separate queries and then merging the results. The second query creates an index on a frequently queried column, which can speed up data retrieval operations on that column.

```

31
32 SELECT e.event_id, e.venue_id, m.manager_id, m.manager_name
33 FROM events e
34 JOIN manager m
35 ON e.event_manager_id=manager_id;
36
37
38 Data Output Messages Notifications
39
40
41   event_id integer
42   venue_id integer
43   manager_id integer
44   manager_name character varying (100)
45
46
47
48
49
50
51
52

```

	event_id	venue_id	manager_id	manager_name
1	1	4178	974	Michelle Solis
2	2	6676	7	Samantha Collins
3	3	7041	3269	Joshua Jones
4	4	7535	850	Melissa Lynn
5	5	8573	2099	Mark Santos
6	6	5000	161	Michael Terrell
7	7	6855	505	Mr. David Murphy
8	8	8228	831	Martha Carpenter

Total rows: 1000 of 17999    Query complete 00:00:00.161    ✓ Successfully run. Total query runtime: 161 msec. 17999 rows affected. X    Ln 32, Col 1

```

37
38 CREATE UNIQUE INDEX index_venue_id
39 ON venue (venue_id);
40
41
42
43
44
45
46
47
48
49
50
51
52

```

	Data Output	Messages	Notifications
37		CREATE INDEX	
38			
39			
40			
41			
42			
43			
44			
45			
46			
47			
48			
49			
50			
51			
52			

CREATE INDEX  
Query returned successfully in 61 msec.

## b. By using Indexing

The query retrieves data from two tables, "events" and "registration", and summarizes the results by event type. It returns the count of total participants registered for each event type, sorted in ascending order by event type. The statistics can be observed in the below images.

```

57 ORDER BY event_type ASC,
58
59 EXPLAIN ANALYZE SELECT e.event_type, count(reg.event_id) as tot_mem_registered
60 FROM events e
61 JOIN registration reg
62 ON e.event_id = reg.event_id
63 GROUP BY e.event_type
64 ORDER BY event_type ASC;
65
66
67

```

Query History Data Output Messages Explain Notifications

QUERY PLAN

text
1 Sort (cost=1130.54..1130.63 rows=33 width=17) (actual time=28.524..28.526 rows=33 loops=1)
2 Sort Key: e.event_type
3 Sort Method: quicksort Memory: 27kB
4 -> HashAggregate (cost=1129.38..1129.71 rows=33 width=17) (actual time=28.469..28.474 rows=33 loops=1)
5 Group Key: e.event_type
6 -> Hash Join (cost=550.98..1008.99 rows=24079 width=13) (actual time=4.961..19.586 rows=24079 loops=1)
7 Hash Cond: (reg.event_id = e.event_id)
8 -> Seq Scan on registration reg (cost=0.00..394.79 rows=24079 width=4) (actual time=0.012..2.305 rows=24079 loops=1)
9 -> Hash (cost=325.99..325.99 rows=17999 width=13) (actual time=4.832..4.833 rows=17999 loops=1)
10 Buckets: 32768 Batches: 1 Memory Usage: 1091kB
11 -> Seq Scan on events e (cost=0.00..325.99 rows=17999 width=13) (actual time=0.009..2.041 rows=17999 loops=1)
12 Planning Time: 1.319 ms
13 Execution Time: 28.693 ms

Total rows: 13 of 13    Query complete 00:00:00.075    Ln 64, Col 25

After adding idx\_event\_id, idx\_reg\_event the performance has been improved and the execution time has reduced from 28.693 ms to 23.552 ms as observed in the outputs.

```

46
47 CREATE INDEX idx_event_id ON events (event_id);
48 CREATE INDEX idx_reg_event_id ON registration (event_id);
49
50
51
52
53
54
55
56
57
58

```

Query History Data Output Messages Explain Notifications

CREATE INDEX

Query returned successfully in 69 msec.

Query

```

56 GROUP BY e.event_type
57 ORDER BY event_type ASC;
58
59 EXPLAIN ANALYZE SELECT e.event_type, count(reg.event_id) as tot_mem_registered
60 FROM events e
61 JOIN registration reg
62 ON e.event_id = reg.event_id
63 GROUP BY e.event_type
64 ORDER BY event_type ASC;
65
66

```

QUERY PLAN  
text

1	Sort (cost=1130.54..1130.63 rows=33 width=17) (actual time=23.371..23.375 rows=33 loops=1)
2	Sort Key: e.event_type
3	Sort Method: quicksort Memory: 27kB
4	-> HashAggregate (cost=1129.38..1129.71 rows=33 width=17) (actual time=23.303..23.308 rows=33 loops=1)
5	Group Key: e.event_type
6	-> Hash Join (cost=550.98..1008.99 rows=24079 width=13) (actual time=5.240..16.760 rows=24079 loops=1)
7	Hash Cond: (reg.event_id = e.event_id)
8	-> Seq Scan on registration reg (cost=0.00..394.79 rows=24079 width=4) (actual time=0.012..1.801 rows=24079 loops=1)
9	-> Hash (cost=325.99..325.99 rows=17999 width=13) (actual time=5.113..5.113 rows=17999 loops=1)
10	Buckets: 32768 Batches: 1 Memory Usage: 1091kB
11	-> Seq Scan on events e (cost=0.00..325.99 rows=17999 width=13) (actual time=0.009..2.031 rows=17999 loops=1)
12	Planning Time: 2.575 ms
13	Execution Time: 23.552 ms

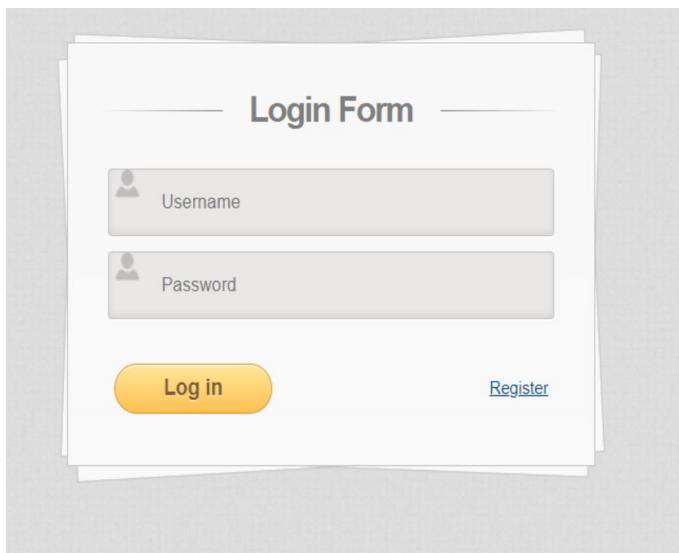
Total rows: 13 of 13    Query complete 00:00:00.063    Ln 65, Col 1

## Bonus:

### Project Application:

#### Login form:

The login form is a basic user interface that allows users to enter their login credentials, such as their username and password, to access the event management system.



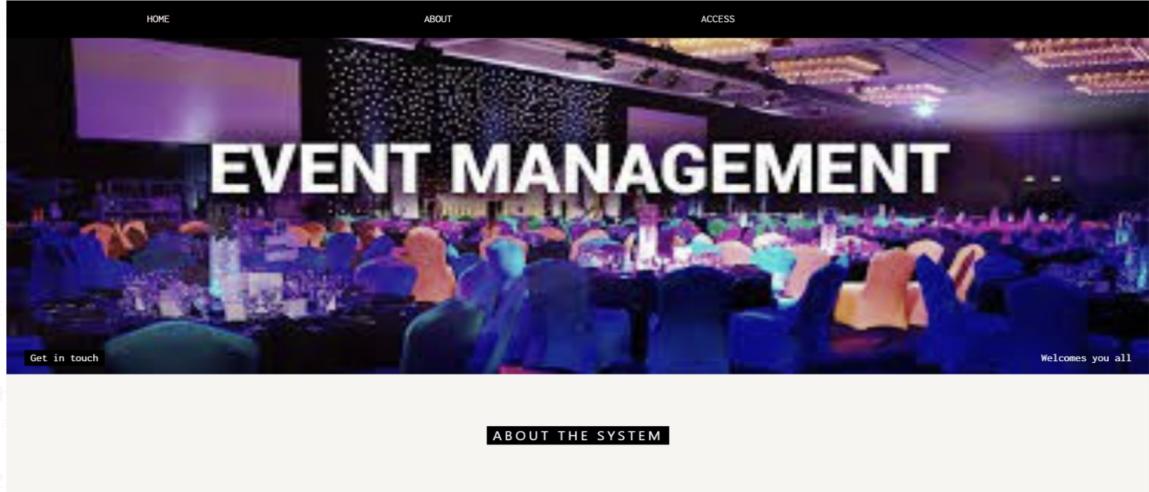
### Registration form:

A registration form allows users to create an account on a system or application. It consists of a series of input fields that collect information about the user, such as their name, email address, username, gender, age and password.

The image shows a wireframe of a registration form. At the top, it says "REGISTRATION FORM". Below that are five input fields, each with a small icon of a person and a label: "Username", "Password", "Email", "Gender", and "Age". Underneath these fields are two small blue links: "Back to home page" and "Sign in". At the bottom is a large yellow button with the word "Register" in white.

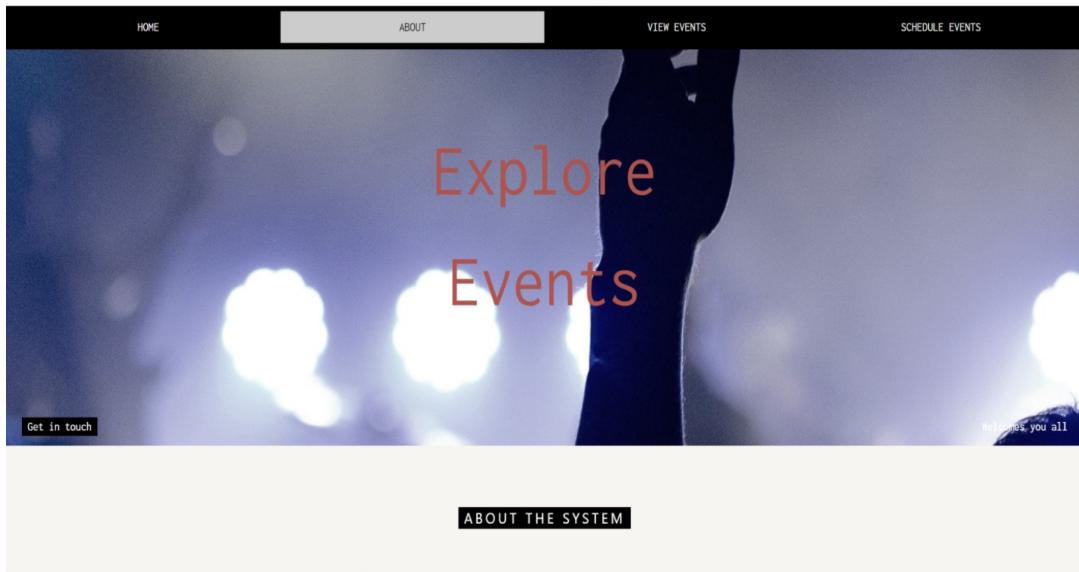
### Event management home page

An event management homepage serves as the main portal for an event management system to help them plan, organize, and manage events of various types and sizes.



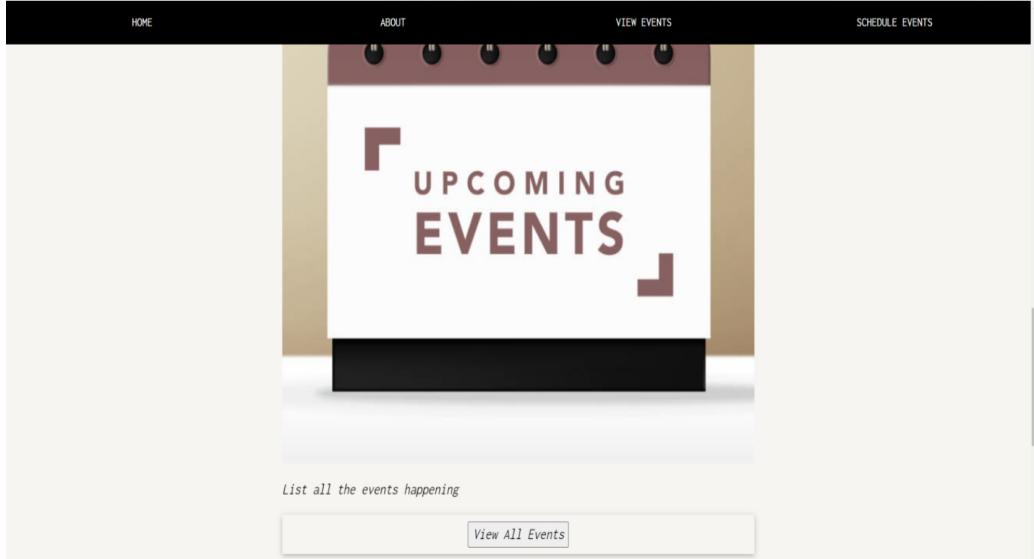
## User\_home\_page

The user home page displays information relevant to the user, such as their upcoming events, past events, and other important details.



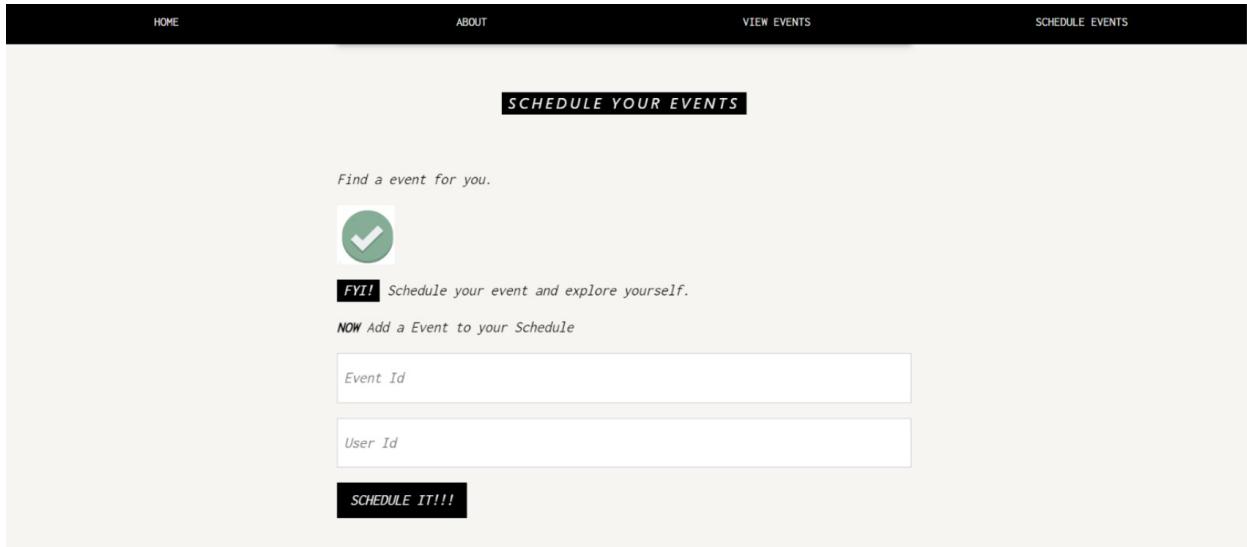
## View Events

This page displays a list of all upcoming events in the event management system.



## Schedule Events

This page allows users to schedule an event by entering the event id, and user id. Once the event has been scheduled, it will appear on the user's home page and in the system's event management dashboard.



## Add\_event

Once the user schedules an event they can add the event details such as duration, date location etc.

The screenshot shows a 'Add Event' form with the following fields:

- Event name
- Description
- Event date (mm/dd/yyyy)
- Event time
- Event duration
- Duration
- Event type (radio buttons for Event\_1 and Event\_2, with Event\_2 selected)
- Email
- Venue

At the bottom are buttons for 'Create' (highlighted in yellow), 'Delete Event', and 'Back to home Page'.

### **Delete\_event**

Whenever a user is willing to cancel an event they can do it through using the delete event option through their particular event id's.

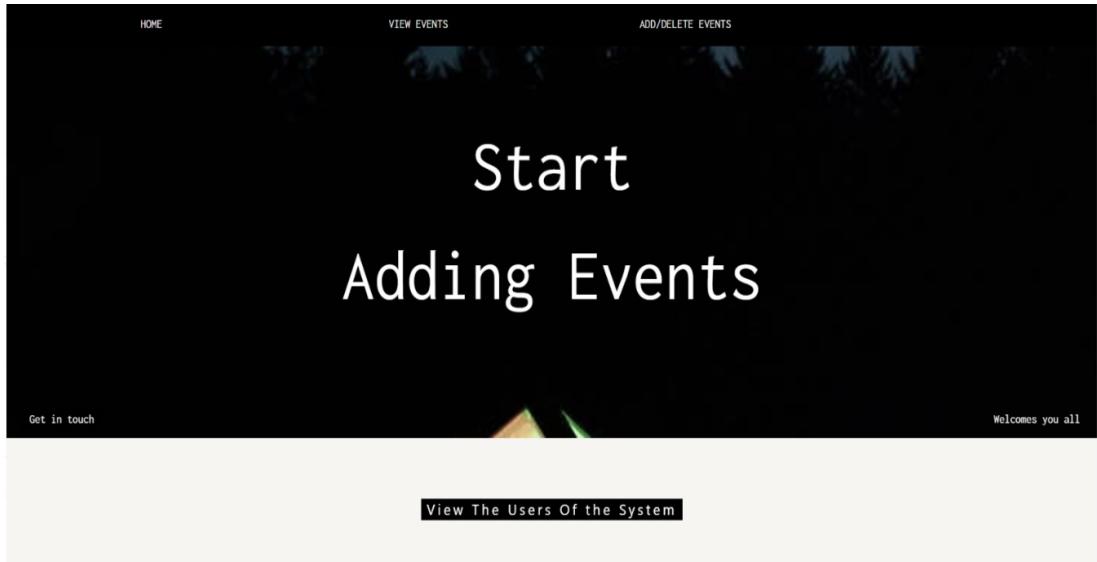
The screenshot shows a 'Delete Event' form with the following field:

- Event id

At the bottom are buttons for 'Delete Event' (highlighted in yellow) and 'Add Event'.

### **Admin\_homepage**

A landing page or dashboard within an event management system that is designed for administrators.



## 11. Conclusion

Based on the analysis of the event management system, it can be concluded that the system provides an effective way to manage various events and their related information. The system enables event managers to create, update and manage events, venues, participants and payments efficiently. The use of various database concepts such as normalization, indexes, and views helps in ensuring data integrity, performance and efficient data retrieval. However, the system can be further improved by adding user interfaces and dashboards to make it more user-friendly and easier to use for both event managers and participants. Overall, the event management system has the potential to streamline event planning and management processes and enhance the overall experience of organizing and participating in events.

## 12. References

1. <https://ijisrt.com/wp-content/uploads/2017/04/Study-on-Event-Management-Applications.pdf>
2. [https://www.researchgate.net/publication/259117550\\_From\\_Event\\_Management\\_to\\_Managing\\_Events](https://www.researchgate.net/publication/259117550_From_Event_Management_to_Managing_Events)
3. Theocharis, 2008, Special event management and event marketing: A case study of TKBL All Star 2011 in Turkey.
4. Gurung, Bikash,2013,Marketing in Event Management.





