**Due Date: 7/3/2023**

**Name: Abdullahi Farah**

**Query Scenario 1: List all events along with their organizers' names and contact details.**

SELECT

e.event\_name,

e.event\_date,

e.event\_location,

o.organizer\_name,

o.organizer\_contact

FROM

Events e

INNER JOIN

Organizers o ON e.event\_id = o.organizer\_id;

In this query scenario, we want to retrieve a list of all events along with their respective organizers' names and contact details. To achieve this, we use an INNER JOIN between the 'Events' and 'Organizers' tables based on the event\_id and organizer\_id.

1. **SELECT**: We specify the columns we want to retrieve in the SELECT clause:
   * **e.event\_name**: The name of the event.
   * **e.event\_date**: The date of the event.
   * **e.event\_location**: The location of the event.
   * **o.organizer\_name**: The name of the organizer.
   * **o.organizer\_contact**: The contact details of the organizer.
2. **FROM**: We specify two tables, 'Events' and 'Organizers', that we are querying from. We give them aliases 'e' and 'o', respectively.
3. **INNER JOIN**: We use an INNER JOIN to combine records from the 'Events' and 'Organizers' tables based on the event\_id in the 'Events' table (e.event\_id) and the organizer\_id in the 'Organizers' table (o.organizer\_id). This ensures that we only include events that have corresponding organizers.
4. **ON**: We specify the condition for the JOIN, which is that the event\_id in the 'Events' table should match the organizer\_id in the 'Organizers' table.

The query will execute and produce a result set with columns 'event\_name', 'event\_date', 'event\_location', 'organizer\_name', and 'organizer\_contact', showing each event along with its respective organizer's name and contact details.

**Query Scenario 2: Count the number of attendees registered for each event.**

SELECT

e.event\_name,

COUNT(a.attendee\_id) AS num\_attendees

FROM

Events e

LEFT JOIN

Attendees a ON e.event\_id = a.event\_id

GROUP BY

e.event\_name;

In this query scenario, we want to count the number of attendees registered for each event. To achieve this, we use a LEFT JOIN between the 'Events' and 'Attendees' tables and then apply the COUNT function to count the number of attendees for each event.

1. **SELECT**: We specify the columns we want to retrieve in the SELECT clause:
   * **e.event\_name**: The name of the event.
   * **COUNT(a.attendee\_id) AS num\_attendees**: The number of attendees for each event. We use the COUNT function to count the number of attendee IDs (attendee\_id) in the 'Attendees' table. The alias 'num\_attendees' is used for clarity.
2. **FROM**: We specify two tables, 'Events' and 'Attendees', that we are querying from. We give them aliases 'e' and 'a', respectively.
3. **LEFT JOIN**: We use a LEFT JOIN to include all events from the 'Events' table, regardless of whether they have corresponding records in the 'Attendees' table.
4. **ON**: We specify the condition for the JOIN, which is that the event\_id in the 'Events' table (e.event\_id) should match the event\_id in the 'Attendees' table (a.event\_id).
5. **GROUP BY**: We use the GROUP BY clause to group the results by event\_name. This groups the count of attendees for each event.

The query will execute and produce a result set with columns 'event\_name' and 'num\_attendees', showing each event along with the number of attendees registered for that event.

**Query Scenario 3: Find the event with the highest number of registrations.**

SELECT

e.event\_name,

COUNT(r.registration\_id) AS num\_registrations

FROM

Events e

LEFT JOIN

Attendees a ON e.event\_id = a.event\_id

LEFT JOIN

Registrations r ON a.attendee\_id = r.attendee\_id

GROUP BY

e.event\_name

ORDER BY

num\_registrations DESC

LIMIT 1;

In this query scenario, we want to find the event with the highest number of registrations among all events. To achieve this, we use LEFT JOINs between the 'Events', 'Attendees', and 'Registrations' tables. We then apply the COUNT function to count the number of registrations for each event and sort the results in descending order based on the number of registrations. Finally, we use the LIMIT 1 clause to fetch only the top event with the highest number of registrations.

1. **SELECT:** We specify the columns we want to retrieve in the SELECT clause:
   * **e.event\_name:** The name of the event.
   * **COUNT(r.registration\_id) AS num\_registrations:** The number of registrations for each event. We use the COUNT function to count the number of registration IDs **(registration\_id) in the 'Registrations' table. The alias 'num\_registrations' is used for clarity.**
2. **FROM:** We specify three tables, 'Events', 'Attendees', and 'Registrations', that we are querying from. We give them aliases 'e', 'a', and 'r', respectively.
3. **LEFT JOINs:** We use LEFT JOINs to combine records from the 'Events', 'Attendees', and 'Registrations' tables based on event\_id and attendee\_id.
4. **ON:** We specify the conditions for the JOINs. The event\_id in the 'Events' table (e.event\_id) should match the event\_id in the 'Attendees' table (a.event\_id), and the attendee\_id in the 'Attendees' table (a.attendee\_id) should match the attendee\_id in the 'Registrations' table (r.attendee\_id).
5. **GROUP BY:** We use the GROUP BY clause to group the results by event\_name. This groups the count of registrations for each event.
6. **ORDER BY:** We use the ORDER BY clause to sort the results in descending order based on the number of registrations (num\_registrations).
7. **LIMIT:** We use the LIMIT 1 clause to fetch only the top event with the highest number of registrations.

The query will execute and produce a result set with columns 'event\_name' and 'num\_registrations', showing the event with the highest number of registrations at the top. If there are multiple events with the same highest number of registrations, only one event will be returned due to the LIMIT 1 clause.

**Query Scenario 4: List all events with their available resources and their capacity.**

SELECT

e.event\_name,

r.resource\_name,

r.resource\_capacity

FROM

Events e

CROSS JOIN

Resources r;

In this query scenario, we want to retrieve a list of all events along with the available resources and their respective capacities. To achieve this, we use a CROSS JOIN between the 'Events' and 'Resources' tables.

1. **SELECT**: We specify the columns we want to retrieve in the SELECT clause:
   * **e.event\_name**: The name of the event.
   * **r.resource\_name**: The name of the resource.
   * **r.resource\_capacity**: The capacity of the resource.
2. **FROM**: We specify two tables, 'Events' and 'Resources', that we are querying from. We give them aliases 'e' and 'r', respectively.
3. **CROSS JOIN**: We use a CROSS JOIN to combine all records from the 'Events' table with all records from the 'Resources' table. This means that every event will be paired with every resource, creating all possible combinations.

The query will execute and produce a result set with columns 'event\_name', 'resource\_name', and 'resource\_capacity', showing each event along with all available resources and their respective capacities.

**Query Scenario 5: Retrieve all events that have a 'Confirmed' registration status.**

SELECT

e.event\_name,

r.registration\_date,

r.registration\_status

FROM

Events e

INNER JOIN

Attendees a ON e.event\_id = a.event\_id

INNER JOIN

Registrations r ON a.attendee\_id = r.attendee\_id

WHERE

r.registration\_status = 'Confirmed';

In this query scenario, we want to retrieve all events that have registrations with a 'Confirmed' status. To achieve this, we use INNER JOINs between the 'Events', 'Attendees', and 'Registrations' tables, and then apply a WHERE clause to filter events with a 'Confirmed' registration status.

1. **SELECT**: We specify the columns we want to retrieve in the SELECT clause:
   * **e.event\_name**: The name of the event.
   * **r.registration\_date**: The registration date for the event.
   * **r.registration\_status**: The registration status.
2. **FROM**: We specify three tables, 'Events', 'Attendees', and 'Registrations', that we are querying from. We give them aliases 'e', 'a', and 'r', respectively.
3. **INNER JOINs**: We use INNER JOINs to combine records from the 'Events', 'Attendees', and 'Registrations' tables based on event\_id and attendee\_id.
4. **ON**: We specify the conditions for the JOINs. The event\_id in the 'Events' table (e.event\_id) should match the event\_id in the 'Attendees' table (a.event\_id), and the attendee\_id in the 'Attendees' table (a.attendee\_id) should match the attendee\_id in the 'Registrations' table (r.attendee\_id).
5. **WHERE**: We use the WHERE clause to filter the results and include only those events where the registration\_status in the 'Registrations' table is 'Confirmed'.

The query will execute and produce a result set with columns 'event\_name', 'registration\_date', and 'registration\_status', showing all events that have registrations with a 'Confirmed' status.

**Query Scenario 6: Count the total number of events.**

SELECT COUNT(\*) AS total\_events FROM Events;

In this query, we want to find out the total number of events available in our database. The query calculates the total number of events stored in the 'Events' table using the COUNT function.

**1. SELECT:** We specify the columns we want to retrieve in the SELECT clause:

* + **total\_events:** total records in the event table

**2. FROM:** We specify the tables from which we are querying data, which is the events table.

The query will execute and produce a result set with one column , ‘total\_events’ , that has the total number of events.

**Query Scenario 7: Calculate the average capacity of resources available for each event.**

SELECT

e.event\_name,

AVG(r.resource\_capacity) AS avg\_resource\_capacity

FROM

Events e

INNER JOIN

Resources r ON e.event\_location = r.resource\_name

GROUP BY

e.event\_name;

In this query scenario, we want to calculate the average capacity of resources available for each event. To achieve this, we use an INNER JOIN between the 'Events' and 'Resources' tables based on the event location (event\_location) and resource name (resource\_name). Then, we apply the AVG function to calculate the average resource capacity for each event.

1. **SELECT**: We specify the columns we want to retrieve in the SELECT clause:
   * **e.event\_name**: The name of the event.
   * **AVG(r.resource\_capacity) AS avg\_resource\_capacity**: The average capacity of resources for each event. We use the AVG function to calculate the average resource capacity from the 'Resources' table. The alias 'avg\_resource\_capacity' is used for clarity.
2. **FROM**: We specify two tables, 'Events' and 'Resources', that we are querying from. We give them aliases 'e' and 'r', respectively.
3. **INNER JOIN**: We use an INNER JOIN to combine records from the 'Events' and 'Resources' tables based on the event location in the 'Events' table (e.event\_location) and the resource name in the 'Resources' table (r.resource\_name). This ensures that we only include events with resources available.
4. **ON**: We specify the condition for the JOIN, which is that the event location in the 'Events' table should match the resource name in the 'Resources' table.
5. **GROUP BY**: We use the GROUP BY clause to group the results by event\_name. This groups the average resource capacities for each event.

The query will execute and produce a result set with columns 'event\_name' and 'avg\_resource\_capacity', showing each event along with the average capacity of resources available for that event.

**Query Scenario 8: Find the event with the earliest date and the one with the latest date.**

SELECT

MIN(event\_date) AS earliest\_event\_date,

MAX(event\_date) AS latest\_event\_date

FROM

Events;

In this query scenario, we want to find the event with the earliest date and the one with the latest date among all events. To achieve this, we use the MIN and MAX functions to find the minimum (earliest) and maximum (latest) event dates from the 'Events' table.

1. **SELECT**: We use the SELECT clause to specify the columns we want to retrieve:
   * **MIN(event\_date) AS earliest\_event\_date**: The earliest event date. We use the MIN function to find the minimum event date in the 'Events' table. The alias 'earliest\_event\_date' is used for clarity.
   * **MAX(event\_date) AS latest\_event\_date**: The latest event date. We use the MAX function to find the maximum event date in the 'Events' table. The alias 'latest\_event\_date' is used for clarity.
2. **FROM**: We specify the 'Events' table that we are querying from.

Since there is no need for JOINs or GROUP BY in this scenario, the query will calculate the earliest and latest event dates directly from the 'Events' table.

The query will execute and produce a result set with two columns, 'earliest\_event\_date' and 'latest\_event\_date', showing the earliest and latest event dates among all events.

**Query Scenario 9: List all events with their respective attendee count and order them by the number of attendees in descending order.**

SELECT

e.event\_name,

COUNT(a.attendee\_id) AS num\_attendees

FROM

Events e

LEFT JOIN

Attendees a ON e.event\_id = a.event\_id

GROUP BY

e.event\_name

ORDER BY

num\_attendees DESC;

In this query scenario, we want to list all events along with the count of attendees for each event. We also want to order the events based on the number of attendees in descending order. To achieve this, we use a LEFT JOIN between the 'Events' and 'Attendees' tables, apply the COUNT function to count the number of attendees for each event, and then use the ORDER BY clause to sort the events by the number of attendees in descending order.

1. **SELECT**: We use the SELECT clause to specify the columns we want to retrieve:
   * **e.event\_name**: The name of the event.
   * **COUNT(a.attendee\_id) AS num\_attendees**: The number of attendees for each event. We use the COUNT function to count the number of attendee IDs (attendee\_id) in the 'Attendees' table. The alias 'num\_attendees' is used for clarity.
2. **FROM**: We specify two tables, 'Events' and 'Attendees', that we are querying from. We give them aliases 'e' and 'a', respectively.
3. **LEFT JOIN**: We use a LEFT JOIN to include all events from the 'Events' table, regardless of whether they have corresponding records in the 'Attendees' table.
4. **ON**: We specify the condition for the JOIN, which is that the event\_id in the 'Events' table (e.event\_id) should match the event\_id in the 'Attendees' table (a.event\_id).
5. **GROUP BY**: We use the GROUP BY clause to group the results by event\_name. This groups the count of attendees for each event.
6. **ORDER BY**: We use the ORDER BY clause to sort the results in descending order based on the number of attendees (num\_attendees).

The query will execute and produce a result set with columns 'event\_name' and 'num\_attendees', showing each event along with the count of attendees for that event. The events will be ordered based on the number of attendees, with the event with the highest number of attendees at the top.

**Query Scenario 10: Retrieve all events along with their organizers' names, ordered by the event date in ascending order.**

SELECT

e.event\_name,

e.event\_date,

o.organizer\_name

FROM

Events e

INNER JOIN

Organizers o ON e.event\_id = o.organizer\_id

ORDER BY

e.event\_date ASC;

In this query scenario, we want to retrieve all events along with their respective organizers' names. We also want to order the events based on the event date in ascending order. To achieve this, we use an INNER JOIN between the 'Events' and 'Organizers' tables based on the event\_id and organizer\_id. Then, we use the ORDER BY clause to sort the events by the event date in ascending order.

1. **SELECT**: We use the SELECT clause to specify the columns we want to retrieve:
   * **e.event\_name**: The name of the event.
   * **e.event\_date**: The date of the event.
   * **o.organizer\_name**: The name of the organizer.
2. **FROM**: We specify two tables, 'Events' and 'Organizers', that we are querying from. We give them aliases 'e' and 'o', respectively.
3. **INNER JOIN**: We use an INNER JOIN to combine records from the 'Events' and 'Organizers' tables based on the event\_id and organizer\_id. This ensures that we only include events that have corresponding organizers.
4. **ON**: We specify the condition for the JOIN, which is that the event\_id in the 'Events' table (e.event\_id) should match the organizer\_id in the 'Organizers' table (o.organizer\_id).
5. **ORDER BY**: We use the ORDER BY clause to sort the results in ascending order based on the event\_date.

The query will execute and produce a result set with columns 'event\_name', 'event\_date', and 'organizer\_name', showing each event along with the organizer's name. The events will be ordered based on the event date, with the earliest event date at the top.

**Query Scenario 11: List the resources that have not been assigned to any event.**

SELECT

r.resource\_name

FROM

Resources r

LEFT JOIN

Events e ON r.resource\_name = e.event\_location

WHERE

e.event\_id IS NULL;

In this query scenario, we want to list all resources that have not been assigned to any event. To achieve this, we use a LEFT JOIN between the 'Resources' and 'Events' tables on the resource\_name and event\_location. Then, we use a WHERE clause to filter out the resources that do not have corresponding events (i.e., events where the event\_location is NULL).

1. **SELECT**: We use the SELECT clause to specify the columns we want to retrieve:
   * **r.resource\_name**: The name of the resource.
2. **FROM**: We specify the 'Resources' table that we are querying from. We give it the alias 'r'.
3. **LEFT JOIN**: We use a LEFT JOIN to include all resources from the 'Resources' table, regardless of whether they have corresponding events.
4. **ON**: We specify the condition for the JOIN, which is that the resource\_name in the 'Resources' table (r.resource\_name) should match the event\_location in the 'Events' table (e.event\_location).
5. **WHERE**: We use the WHERE clause to filter the results and include only the resources where the event\_location in the 'Events' table is NULL. This means that the resource has not been assigned to any event.

The query will execute and produce a result set with the column 'resource\_name', showing all resources that have not been assigned to any event.

**Query Scenario 12: Find the event with the most pending registrations.**

SELECT

e.event\_name,

COUNT(r.registration\_id) AS num\_pending\_registrations

FROM

Events e

LEFT JOIN

Attendees a ON e.event\_id = a.event\_id

LEFT JOIN

Registrations r ON a.attendee\_id = r.attendee\_id

WHERE

r.registration\_status = 'Pending'

GROUP BY

e.event\_name

ORDER BY

num\_pending\_registrations DESC

LIMIT 1;

In this query scenario, we want to find the event with the most pending registrations. To achieve this, we use LEFT JOINs between the 'Events', 'Attendees', and 'Registrations' tables to include all events along with their registrations. Then, we use a WHERE clause to filter the results and include only the registrations with a 'Pending' status. Next, we use the GROUP BY clause to group the results by event\_name and apply the COUNT function to count the number of pending registrations for each event. Finally, we use the ORDER BY clause to sort the events in descending order based on the number of pending registrations, and the LIMIT 1 clause to retrieve only the top event with the most pending registrations.

1. **SELECT**: We use the SELECT clause to specify the columns we want to retrieve:
   * **e.event\_name**: The name of the event.
   * **COUNT(r.registration\_id) AS num\_pending\_registrations**: The number of pending registrations for each event. We use the COUNT function to count the number of registration IDs (registration\_id) in the 'Registrations' table where the registration\_status is 'Pending'. The alias 'num\_pending\_registrations' is used for clarity.
2. **FROM**: We specify three tables, 'Events', 'Attendees', and 'Registrations', that we are querying from. We give them aliases 'e', 'a', and 'r', respectively.
3. **LEFT JOINs**: We use LEFT JOINs to combine records from the 'Events', 'Attendees', and 'Registrations' tables based on event\_id and attendee\_id.
4. **ON**: We specify the conditions for the JOINs. The event\_id in the 'Events' table (e.event\_id) should match the event\_id in the 'Attendees' table (a.event\_id), and the attendee\_id in the 'Attendees' table (a.attendee\_id) should match the attendee\_id in the 'Registrations' table (r.attendee\_id).
5. **WHERE**: We use the WHERE clause to filter the results and include only the registrations with a 'Pending' status.
6. **GROUP BY**: We use the GROUP BY clause to group the results by event\_name. This groups the count of pending registrations for each event.
7. **ORDER BY**: We use the ORDER BY clause to sort the results in descending order based on the number of pending registrations (num\_pending\_registrations).
8. **LIMIT 1**: We use the LIMIT 1 clause to retrieve only the top event with the most pending registrations.

The query will execute and produce a result set with columns 'event\_name' and 'num\_pending\_registrations', showing the event with the most pending registrations at the top.

**Query Scenario 13: Creating a view for all confirmed registrations:**

CREATE VIEW ConfirmedRegistrations AS

SELECT

e.event\_name,

a.attendee\_name,

r.registration\_date,

r.registration\_status

FROM

Events e

INNER JOIN

Attendees a ON e.event\_id = a.event\_id

INNER JOIN

Registrations r ON a.attendee\_id = r.attendee\_id

WHERE

r.registration\_status = 'Confirmed';

In this query scenario, we create a view called 'ConfirmedRegistrations' that contains information about all confirmed registrations. A view is a virtual table based on the result of a SELECT query, which can be used later in other queries as if it were an actual table.

1. **CREATE VIEW**: We use the CREATE VIEW statement to create a view. After the VIEW keyword, we provide the name of the view we want to create, which is 'ConfirmedRegistrations' in this case.
2. **AS**: We use the AS keyword to define the SELECT query that forms the view.
3. **SELECT**: We specify the columns we want to include in the view:
   * **e.event\_name**: The name of the event.
   * **a.attendee\_name**: The name of the attendee.
   * **r.registration\_date**: The registration date.
   * **r.registration\_status**: The registration status.
4. **FROM**: We specify three tables, 'Events', 'Attendees', and 'Registrations', that we are querying from. We give them aliases 'e', 'a', and 'r', respectively.
5. **INNER JOINs**: We use INNER JOINs to combine records from the 'Events', 'Attendees', and 'Registrations' tables based on event\_id and attendee\_id. This ensures that we include only confirmed registrations.
6. **ON**: We specify the conditions for the JOINs. The event\_id in the 'Events' table (e.event\_id) should match the event\_id in the 'Attendees' table (a.event\_id), and the attendee\_id in the 'Attendees' table (a.attendee\_id) should match the attendee\_id in the 'Registrations' table (r.attendee\_id).
7. **WHERE**: We use the WHERE clause to filter the results and include only the registrations with a 'Confirmed' status.

After executing this query, the 'ConfirmedRegistrations' view will be created, containing the details of all confirmed registrations. This view can be used in subsequent queries to retrieve information about confirmed registrations without having to write the same complex query repeatedly.