Final Project
Ride-Sharing Application

Assignment1

1.Business case:

The Ride-Sharing Application is a platform designed to connect passengers with drivers for convenient, affordable, and efficient transportation. Much like popular services such as Uber and Bolt, the application allows passengers to book rides based on their location, while drivers can accept requests and earn money for providing transportation.

Key Features of the System:

1. **User registration and login (Driver/Passenger)**
   - Users register with a username, email, password, and role (Driver or Passenger).
2. **Trip creation and booking**
   - Drivers create trips, and passengers can book them.
3. **Price calculation**
   - The system should allow for the calculation of trip prices based on certain parameters.
4. **Rating and review system**
   - Passengers can rate trips and leave reviews.
5. **Payment management**
   - Payments are linked to bookings and are processed after the trip is completed.

Entity-Relationship Information

Entity: User

- - user_id (PK)
- - username
- - email
- - password
- - role

Entity: Ride

- - ride_id (PK)

- - driver_id (FK)
- - origin
- - destination
- - departure_time
- - available_seats

Entity: Booking

- - booking_id (PK)
- - ride_id (FK)
- - passenger_id (FK)
- - booking_time
- - status

Entity: Payment

- - payment_id (PK)
- - booking_id (FK)
- - amount
- - payment_time

Entity: Review

- - review_id (PK)
- - ride_id (FK)
- - passenger_id (FK)
- - rating
- - comment

2. Constraints Definition

Primary Keys:

- **Primary Keys** are defined for each table to uniquely identify records. For instance:
    - user_id in the **User** table.
    - ride_id in the **Ride** table.
    - booking_id in the **Booking** table.
    - payment_id in the **Payment** table.
    - review_id in the **Review** table.

Foreign Keys:

- **Foreign Keys** are used to create relationships between tables:

    - driver_id in the **Ride** table references user_id in the **User** table, establishing a relationship between rides and drivers.
    - ride_id in the **Booking** table references ride_id in the **Ride** table, indicating which ride a booking refers to.
    - passenger_id in the **Booking** table references user_id in the **User** table, indicating which passenger made the booking.
    - booking_id in the **Payment** table references booking_id in the **Booking** table, linking payments to bookings.
    - ride_id in the **Review** table references ride_id in the **Ride** table, linking reviews to rides.
    - passenger_id in the **Review** table references user_id in the **User** table, linking reviews to passengers.

3. Relationship and Cardinality

Relationships:

- **One-to-Many**:
    - A **User** can have many **Rides** (as a driver), but each **Ride** can have only one **Driver**. This is a one-to-many relationship between the **User** and **Ride** tables.
    - A **User** can have many **Bookings**, but each **Booking** is made by one **Passenger**. This is a one-to-many relationship between the **User** and **Booking** tables.
    - A **Ride** can have many **Bookings**, but each **Booking** refers to only one **Ride**. This is a one-to-many relationship between the **Ride** and **Booking** tables.
    - A **Booking** can have one **Payment**, but each **Payment** refers to only one **Booking**. This is a one-to-one relationship between the **Booking** and **Payment** tables.
    - A **Ride** can have many **Reviews**, but each **Review** refers to one **Ride**. This is a one-to-many relationship between the **Ride** and **Review** tables.

Cardinality:

- **One-to-Many** is used for relationships where one entity (such as a user or a ride) can be related to multiple instances of another entity (such as bookings or reviews).
- **One-to-One** is used for the relationship between **Booking** and **Payment**, where each booking can have only one associated payment.

4.Normal form

All tables are already in **Third Normal Form (3NF)** because:

- All attributes are atomic.
- Every non-prime attribute is fully functionally dependent on the primary key.
- There are no transitive dependencies between non-prime attributes.

5.Candidate and Super keys, Non-prime attributes

1. User

- **Candidate key**: user_id
- **Super keys**:
  - {user_id}
  - {user_id, username}, {user_id, email}, {user_id, password}, {user_id, role},
- **Non-prime attributes**: username, email, password, role

---

2. Ride

- **Candidate key**: ride_id
- **Super keys**:
  - {ride_id}
  - {ride_id, driver_id}, {ride_id, origin}
- **Non-prime attributes**: driver_id, origin, destination, departure, available_seats

---

3. Booking

- **Candidate key**: booking_id
- **Super keys**:
  - {booking_id}
  - {booking_id, ride_id}, {booking_id, passenger_id}
- **Non-prime attributes**: ride_id, passenger_id, booking_time, status

---

4. Payment

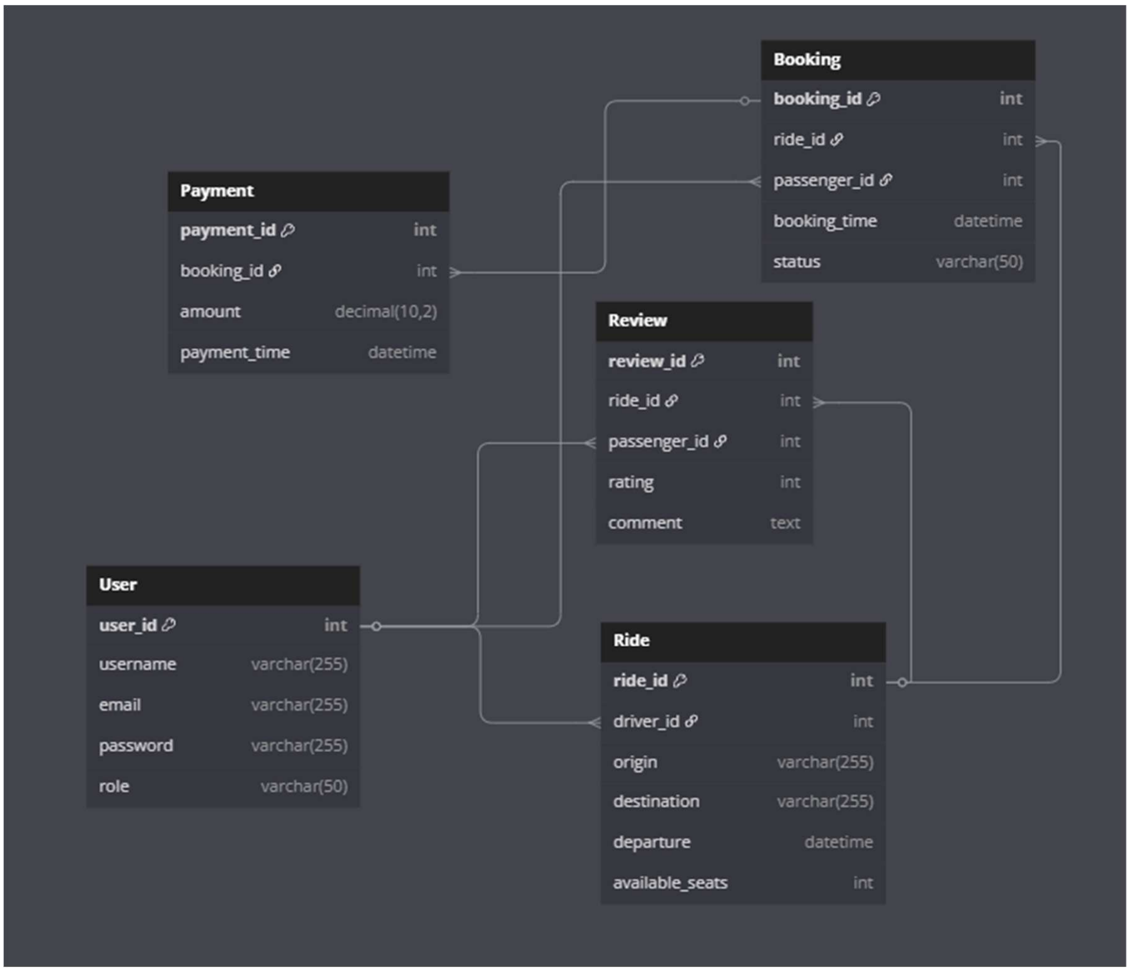
- **Candidate key**: payment_id
- **Super keys**:
    - {payment_id}
    - {payment_id, booking_id}, {payment_id, amount}
- **Non-prime attributes**: booking_id, amount, payment_time

---

5. Review

- **Candidate key**: review_id
- **Super keys**:
    - {review_id}
    - {review_id, ride_id}, {review_id, passenger_id}
- **Non-prime attributes**: ride_id, passenger_id, rating, comment

ER-DIAGRAM


Booking
booking_id int
ride_id int
passenger_id int
booking_time datetime
status varchar(50)
Payment
payment_id int
booking_id int
amount decimal(10,2)
payment_time datetime
Review
review_id int
ride_id int
passenger_id int
rating int
comment text
User
user_id int
username varchar(255)
email varchar(255)
password varchar(255)
role varchar(50)
Ride
ride_id int
driver_id int
origin varchar(255)
destination varchar(255)
departure datetime
available_seats int

Assignment2

1.Code for creating tables:

Creating User table

```
CREATE TABLE User (

   user_id INT AUTO_INCREMENT PRIMARY KEY,

   username VARCHAR(255) NOT NULL,

   email VARCHAR(255) NOT NULL,

   password VARCHAR(255) NOT NULL,

   role VARCHAR(50) NOT NULL

);
```

Columns:

- user_id: An integer that uniquely identifies each user. It is set as the **Primary Key** and will auto-increment with each new user.
- username: A string that stores the username of the user. It is defined as **NOT NULL** to ensure every user has a unique name.
- email: A string for the user's email. It is also **NOT NULL**, ensuring that every user has a valid email.
- password: A string that stores the user's password. It is **NOT NULL**.
- role: A string that specifies whether the user is a driver or a passenger.

Creating Ride table

```
CREATE TABLE Ride (

   ride_id INT AUTO_INCREMENT PRIMARY KEY,

   driver_id INT,

   origin VARCHAR (255) NOT NULL,

   destination VARCHAR (255) NOT NULL,
   departure_time DATETIME NOT NULL,

   available_seats INT NOT NULL,

   FOREIGN KEY (driver_id) REFERENCES User(user_id)

);
```

Columns:

- ride_id: An integer that uniquely identifies each ride. It is set as the **Primary Key** and auto-increments with each new ride.
- driver_id: An integer that references the user_id from the **User** table, indicating which user (driver) is offering the ride. This is a **Foreign Key**.
- origin: A string that stores the origin location of the ride.
- destination: A string that stores the destination location of the ride.
- departure_time: A **DATETIME** data type that stores the scheduled departure time of the ride.
- available_seats: An integer representing the number of available seats for passengers.

Creating Booking table

```
CREATE TABLE Booking (

    booking_id INT AUTO_INCREMENT PRIMARY KEY,

    ride_id INT,

    passenger_id INT,

    booking_time DATETIME NOT NULL,

    status VARCHAR(50) NOT NULL,

    FOREIGN KEY (ride_id) REFERENCES Ride(ride_id),

    FOREIGN KEY (passenger_id) REFERENCES User(user_id)

);
```

Columns:

- booking_id: An integer that uniquely identifies each booking. It is set as the **Primary Key**.
- ride_id: An integer that references the ride_id from the **Ride** table, indicating which ride the passenger has booked. This is a **Foreign Key**.
- passenger_id: An integer that references the user_id from the **User** table, indicating which passenger made the booking. This is a **Foreign Key**.
- booking_time: A **DATETIME** value that stores when the booking was made.
- status: A string that indicates the status of the booking (e.g., confirmed, cancelled).

Creating Payment table

```
CREATE TABLE Payment (

   payment_id INT AUTO_INCREMENT PRIMARY KEY,

   booking_id INT UNIQUE,

   amount DECIMAL(10, 2) NOT NULL,

   payment_time DATETIME NOT NULL,

   FOREIGN KEY (booking_id) REFERENCES Booking(booking_id)

);
```

Columns:

- payment_id: An integer that uniquely identifies each payment. It is set as the **Primary Key**.
- booking_id: A unique integer that references the booking_id from the **Booking** table, indicating which booking the payment corresponds to. This is a **Foreign Key**.
- amount: A **DECIMAL** value that stores the amount paid for the booking (with up to two decimal places).
- payment_time: A **DATETIME** value that records the time the payment was made.

Creating Review table

```
CREATE TABLE Review (
    review_id INT AUTO_INCREMENT PRIMARY KEY,
    ride_id INT,
    passenger_id INT,
    rating INT NOT NULL,
    comment TEXT,
    FOREIGN KEY (ride_id) REFERENCES Ride(ride_id),
    FOREIGN KEY (passenger_id) REFERENCES User(user_id)
);
```

Columns:

- review_id: An integer that uniquely identifies each review. It is set as the **Primary Key**.
- ride_id: An integer that references the ride_id from the **Ride** table, indicating which ride the review is associated with. This is a **Foreign Key**.
- passenger_id: An integer that references the user_id from the **User** table, indicating which passenger left the review. This is a **Foreign Key**.
- rating: An integer that stores the rating given by the passenger (e.g., 1 to 5).
- comment: A **TEXT** field that stores the comment given by the passenger in the review.

2. Constraints Definition

Primary Keys:

- **Primary Keys** are defined for each table to uniquely identify records. For instance:
    - user_id in the **User** table.
    - ride_id in the **Ride** table.
    - booking_id in the **Booking** table.
    - payment_id in the **Payment** table.
    - review_id in the **Review** table.

Foreign Keys:

- **Foreign Keys** are used to create relationships between tables:

  - driver_id in the **Ride** table references user_id in the **User** table, establishing a relationship between rides and drivers.
  - ride_id in the **Booking** table references ride_id in the **Ride** table, indicating which ride a booking refers to.
  - passenger_id in the **Booking** table references user_id in the **User** table, indicating which passenger made the booking.
  - booking_id in the **Payment** table references booking_id in the **Booking** table, linking payments to bookings.
  - ride_id in the **Review** table references ride_id in the **Ride** table, linking reviews to rides.
  - passenger_id in the **Review** table references user_id in the **User** table, linking reviews to passengers.

3. Relationship and Cardinality

Relationships:

- **One-to-Many**:
  - A **User** can have many **Rides** (as a driver), but each **Ride** can have only one **Driver**. This is a one-to-many relationship between the **User** and **Ride** tables.
  - A **User** can have many **Bookings**, but each **Booking** is made by one **Passenger**. This is a one-to-many relationship between the **User** and **Booking** tables.
  - A **Ride** can have many **Bookings**, but each **Booking** refers to only one **Ride**. This is a one-to-many relationship between the **Ride** and **Booking** tables.
  - A **Booking** can have one **Payment**, but each **Payment** refers to only one **Booking**. This is a one-to-one relationship between the **Booking** and **Payment** tables.
  - A **Ride** can have many **Reviews**, but each **Review** refers to one **Ride**. This is a one-to-many relationship between the **Ride** and **Review** tables.

Cardinality:

- **One-to-Many** is used for relationships where one entity (such as a user or a ride) can be related to multiple instances of another entity (such as bookings or reviews).
- **One-to-One** is used for the relationship between **Booking** and **Payment**, where each booking can have only one associated payment.

Using Alter command

Adding new column

```
ALTER TABLE User

ADD COLUMN phone_number VARCHAR(20);
```

Changing data type of column

```
ALTER TABLE User

MODIFY COLUMN username VARCHAR(500);
```

Changing name of column

```
ALTER TABLE Booking

RENAME COLUMN status TO booking_status;
```

Deleting column

```
ALTER TABLE User

DROP COLUMN phone_number;
```

Adding foreign key

```
ALTER TABLE Payment

ADD CONSTRAINT fk_payment_type FOREIGN KEY (payment_type) REFERENCES
PaymentType(payment_type_id);
```

Using insert into command

```
INSERT INTO User (username, email, password, role) VALUES

('user1', 'user1@example.com', 'password1', 'passenger'),

('user2', 'user2@example.com', 'password2', 'driver'),

('user3', 'user3@example.com', 'password3', 'passenger'),

('user4', 'user4@example.com', 'password4', 'driver'),

('user5', 'user5@example.com', 'password5', 'passenger'),

('user6', 'user6@example.com', 'password6', 'passenger'),

('user7', 'user7@example.com', 'password7', 'driver'),

('user8', 'user8@example.com', 'password8', 'passenger'),

('user9', 'user9@example.com', 'password9', 'driver'),

('user10', 'user10@example.com', 'password10', 'passenger'),

('user11', 'user11@example.com', 'password11', 'driver'),

('user12', 'user12@example.com', 'password12', 'passenger'),

('user13', 'user13@example.com', 'password13', 'passenger'),

('user14', 'user14@example.com', 'password14', 'driver'),

('user15', 'user15@example.com', 'password15', 'passenger'),

('user16', 'user16@example.com', 'password16', 'driver'),

('user17', 'user17@example.com', 'password17', 'passenger'),

('user18', 'user18@example.com', 'password18', 'passenger'),

('user19', 'user19@example.com', 'password19', 'driver'),

('user20', 'user20@example.com', 'password20', 'passenger'),

('user21', 'user21@example.com', 'password21', 'driver'),

('user22', 'user22@example.com', 'password22', 'passenger'),
('user23', 'user23@example.com', 'password23', 'driver'),

('user24', 'user24@example.com', 'password24', 'passenger'),

('user25', 'user25@example.com', 'password25', 'driver'),

('user26', 'user26@example.com', 'password26', 'passenger'),

('user27', 'user27@example.com', 'password27', 'driver'),

('user28', 'user28@example.com', 'password28', 'passenger'),

('user29', 'user29@example.com', 'password29', 'driver'),

('user30', 'user30@example.com', 'password30', 'passenger');
```

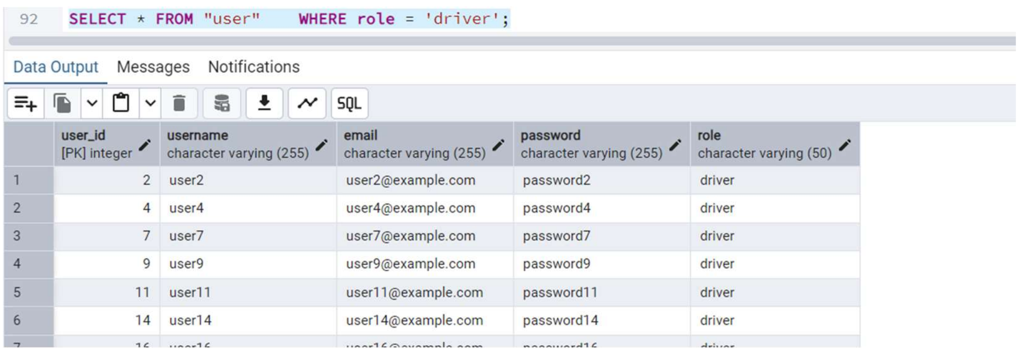
Assignment3

1. WHERE Filtration

SELECT * FROM USER WHERE ROLE = 'DRIVER';

Condition: ROLE = 'DRIVER'

Purpose: This query filters the users to return only those whose role is 'DRIVER'. In the context of a ride-sharing application, this query would be used to fetch all drivers available in the system. It could be used to manage driver data, assign rides, or track driver performance.



```
SELECT * FROM "user"    WHERE role = 'driver';
```

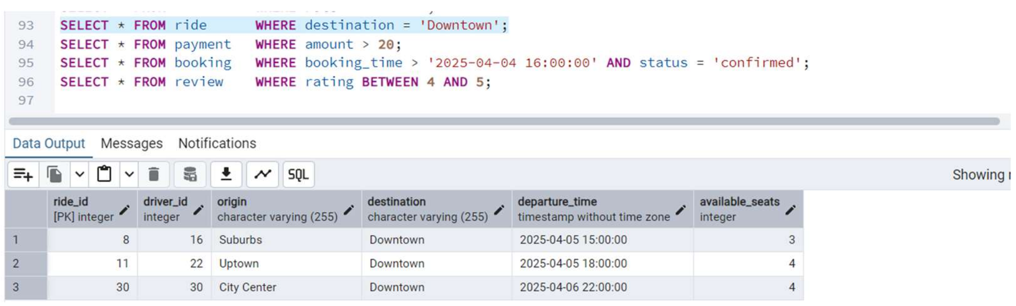
Data Output | Messages | Notifications

	user_id [PK] integer	username character varying (255)	email character varying (255)	password character varying (255)	role character varying (50)
1	2	user2	user2@example.com	password2	driver
2	4	user4	user4@example.com	password4	driver
3	7	user7	user7@example.com	password7	driver
4	9	user9	user9@example.com	password9	driver
5	11	user11	user11@example.com	password11	driver
6	14	user14	user14@example.com	password14	driver
7	16	user16	user16@example.com	password16	driver

SELECT * FROM RIDE WHERE DESTINATION = 'DOWNTOWN';

Condition: DESTINATION = 'DOWNTOWN'

Purpose: This query filters the ride records to return all rides that are headed to "Downtown". This is useful in situations where the company wants to identify or manage all rides going to a specific location, such as for routing, scheduling, or providing promotions to downtown-bound passengers.



```
SELECT * FROM ride       WHERE destination = 'Downtown';
SELECT * FROM payment    WHERE amount > 20;
SELECT * FROM booking    WHERE booking_time > '2025-04-04 16:00:00' AND status = 'confirmed';
SELECT * FROM review     WHERE rating BETWEEN 4 AND 5;
```

Data Output | Messages | Notifications

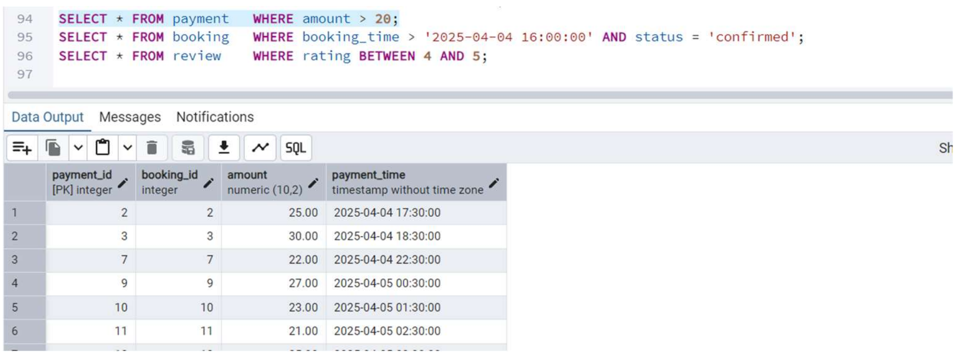
Showing

	ride_id [PK] integer	driver_id integer	origin character varying (255)	destination character varying (255)	departure_time timestamp without time zone	available_seats integer
1	8	16	Suburbs	Downtown	2025-04-05 15:00:00	3
2	11	22	Uptown	Downtown	2025-04-05 18:00:00	4
3	30	30	City Center	Downtown	2025-04-06 22:00:00	4

SELECT * FROM PAYMENT WHERE AMOUNT > 20;

Condition: amount > 20

Purpose: This query filters the payment records to only include those where the payment amount is greater than 20. This could be used to analyze larger transactions or identify high-value rides. It might also be useful for applying specific rules or promotions to higher-paying customers.



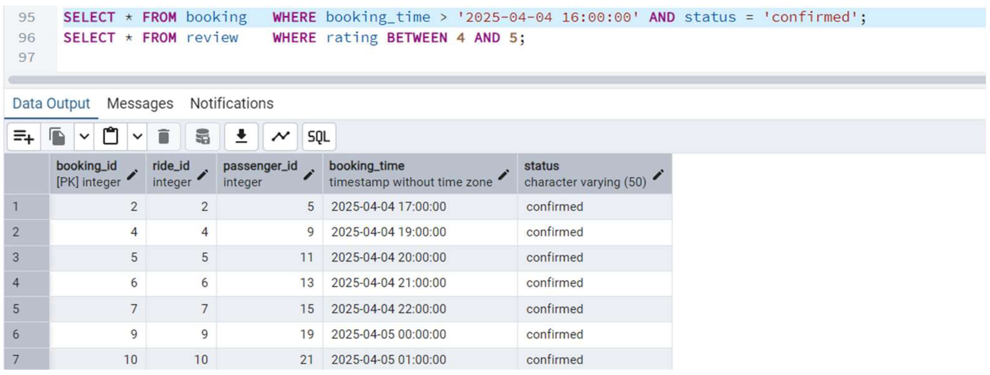
```
SELECT * FROM payment   WHERE amount > 20;
SELECT * FROM booking   WHERE booking_time > '2025-04-04 16:00:00' AND status = 'confirmed';
SELECT * FROM review    WHERE rating BETWEEN 4 AND 5;
```

	payment_id [PK] integer	booking_id integer	amount numeric (10,2)	payment_time timestamp without time zone
1	2	2	25.00	2025-04-04 17:30:00
2	3	3	30.00	2025-04-04 18:30:00
3	7	7	22.00	2025-04-04 22:30:00
4	9	9	27.00	2025-04-05 00:30:00
5	10	10	23.00	2025-04-05 01:30:00
6	11	11	21.00	2025-04-05 02:30:00

SELECT * FROM BOOKING WHERE BOOKING_TIME> '2025-04-04 16:00:00' AND STATUS = 'CONFIRMED';

Condition: booking_time > '2025-04-04 16:00:00' AND status = 'confirmed'

Purpose: This query filters bookings that have a booking time after April 4, 2025, at 4:00 PM and are confirmed. This is typically used to track upcoming confirmed bookings, which can help with scheduling and customer support, ensuring that only relevant and valid bookings are processed.



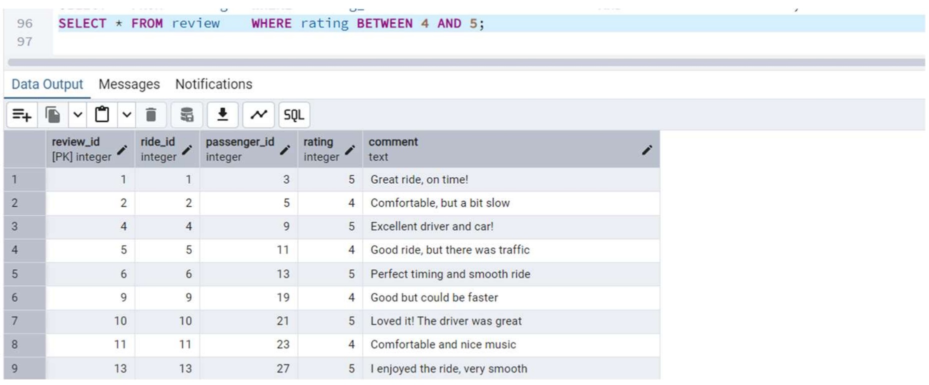
```
SELECT * FROM booking   WHERE booking_time > '2025-04-04 16:00:00' AND status = 'confirmed';
SELECT * FROM review    WHERE rating BETWEEN 4 AND 5;
```

	booking_id [PK] integer	ride_id integer	passenger_id integer	booking_time timestamp without time zone	status character varying (50)
1	2	2	5	2025-04-04 17:00:00	confirmed
2	4	4	9	2025-04-04 19:00:00	confirmed
3	5	5	11	2025-04-04 20:00:00	confirmed
4	6	6	13	2025-04-04 21:00:00	confirmed
5	7	7	15	2025-04-04 22:00:00	confirmed
6	9	9	19	2025-04-05 00:00:00	confirmed
7	10	10	21	2025-04-05 01:00:00	confirmed

SELECT * FROM REVIEW WHERE RATING>=4 AND RATING<=5;

Condition: rating >= 2 AND rating <= 4

Purpose: This query filters reviews to return those where the rating is between 2 and 4, inclusive. In a ride-sharing scenario, this query might be used to monitor average customer satisfaction. It can help identify reviews that are neither excellent nor poor, allowing the company to focus on improving areas where customers have expressed neutral or slightly negative experiences.



```
SELECT * FROM review    WHERE rating BETWEEN 4 AND 5;
```

Data Output Messages Notifications

	review_id [PK] integer	ride_id integer	passenger_id integer	rating integer	comment text
1	1	1	3	5	Great ride, on time!
2	2	2	5	4	Comfortable, but a bit slow
3	4	4	9	5	Excellent driver and car!
4	5	5	11	4	Good ride, but there was traffic
5	6	6	13	5	Perfect timing and smooth ride
6	9	9	19	4	Good but could be faster
7	10	10	21	5	Loved it! The driver was great
8	11	11	23	4	Comfortable and nice music
9	13	13	27	5	I enjoyed the ride, very smooth

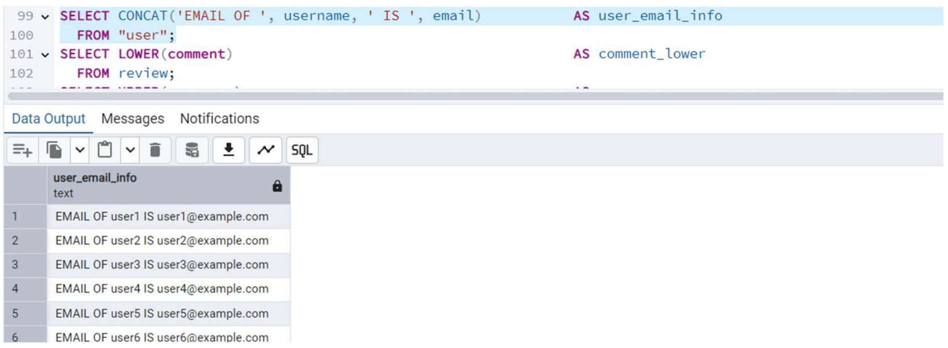
2. String Built-In Functions

SELECT CONCAT('EMAIL OF ', USERNAME , ' IS ', EMAIL) FROM USER;

String Function: CONCAT()

Purpose: The CONCAT() function is used to combine (concatenate) multiple strings into one string. In this case, it combines the text "email of", the username field, the text " is ", and the email field for each user.

Expected Transformation: The output will be a string in the format: "email of <username> is <email>". For example, if the username is "john_doe" and the email is "john.doe@example.com", the resulting output will be "email of john_doe is john.doe@example.com". This is useful for generating readable or formatted information about users, like contact details or user summaries.



```
SELECT CONCAT('EMAIL OF ', username, ' IS ', email)          AS user_email_info
  FROM "user";
SELECT LOWER(comment)                                         AS comment_lower
  FROM review;
```

Data Output   Messages   Notifications

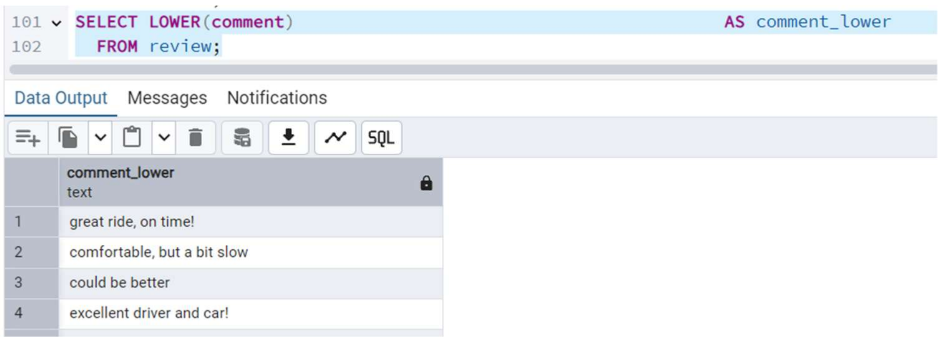
	user_email_info text
1	EMAIL OF user1 IS user1@example.com
2	EMAIL OF user2 IS user2@example.com
3	EMAIL OF user3 IS user3@example.com
4	EMAIL OF user4 IS user4@example.com
5	EMAIL OF user5 IS user5@example.com
6	EMAIL OF user6 IS user6@example.com

SELECT LOWER(COMMENT) FROM REVIEW AS LOWER_COMMENTS;

String Function: lower()

Purpose: The lower() function converts all characters in the specified string (in this case, comment) to lowercase.

Expected Transformation: This transformation will make all characters in the comment field lowercase. For example, if the comment is "GREAT RIDE!", the result will be "great ride!". This function can be useful for standardizing the text data when performing case-insensitive searches or when you want to display user input in a uniform format.



```
SELECT LOWER(comment)                                AS comment_lower
  FROM review;
```

Data Output   Messages   Notifications

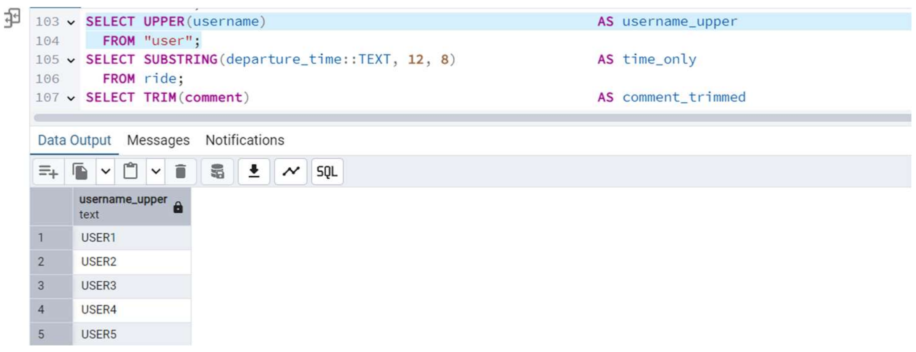
	comment_lower text
1	great ride, on time!
2	comfortable, but a bit slow
3	could be better
4	excellent driver and car!

SELECT UPPER(USERNAME) FROM USER AS UPPER_USERNAME;

String Function: upper()

Purpose: The upper() function converts all characters in the specified string (in this case, username) to uppercase.

Expected Transformation: The username field will be converted to uppercase. For instance, if the username is "john_doe", the result will be "JOHN_DOE". This is useful for consistency in displaying usernames or for comparing data where the case should not matter.



```
SELECT UPPER(username)                                    AS username_upper
  FROM "user";
SELECT SUBSTRING(departure_time::TEXT, 12, 8)             AS time_only
  FROM ride;
SELECT TRIM(comment)                                      AS comment_trimmed
```

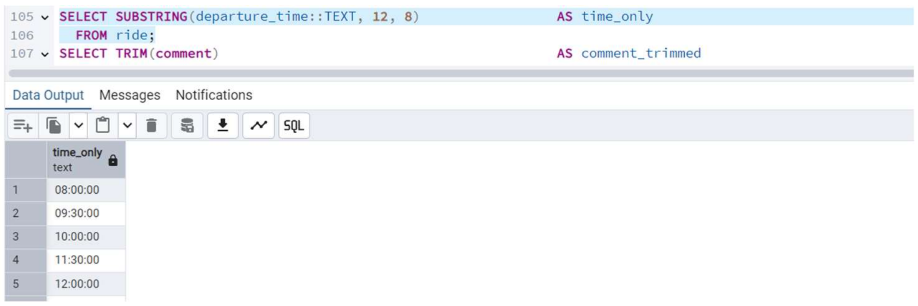
	username_upper text
1	USER1
2	USER2
3	USER3
4	USER4
5	USER5

SELECT SUBSTRING(DEPARTURE_TIME, 11, 20) FROM RIDE;

String Function: SUBSTRING()

Purpose: The SUBSTRING() function is used to extract a portion of the string starting from a specified position. Here, it starts at position 11 of the departure_time field and extracts the next 20 characters.

Expected Transformation: If departure_time is in a format like "2025-04-05 14:30:00", this query will extract the time part, starting from the 11th character, giving "14:30:00". This function is typically used to isolate a part of a field (such as extracting just the time from a datetime field), which is helpful for scheduling, filtering, or reporting purposes where only the time of departure is needed.



```
SELECT SUBSTRING(departure_time::TEXT, 12, 8)             AS time_only
  FROM ride;
SELECT TRIM(comment)                                      AS comment_trimmed
```

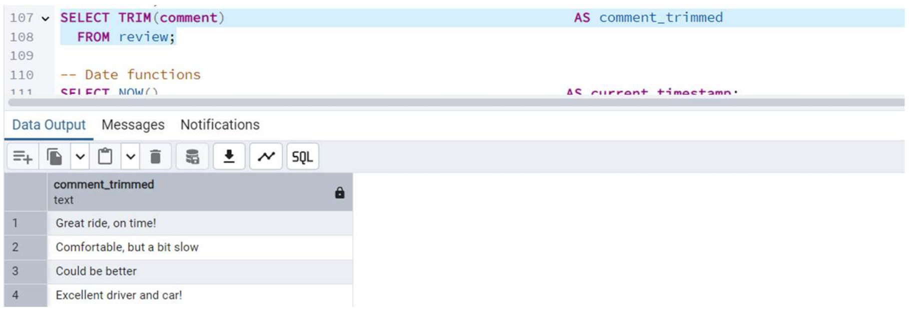
	time_only text
1	08:00:00
2	09:30:00
3	10:00:00
4	11:30:00
5	12:00:00

SELECT TRIM(COMMENT) FROM REVIEW;

String Function: trim()

Purpose: The trim() function removes any leading and trailing whitespace characters from the string.

Expected Transformation: If a comment contains extra spaces before or after the actual text, the trim() function will remove them. For example, if the comment is " Great service! ", the output will be "Great service!". This ensures that the data is clean, consistent, and properly formatted, which is especially important when storing or displaying reviews, as extraneous spaces could affect data processing or presentation.



```
SELECT TRIM(comment)                                    AS comment_trimmed
  FROM review;

-- Date functions
```

Data Output Messages Notifications

	comment_trimmed text
1	Great ride, on time!
2	Comfortable, but a bit slow
3	Could be better
4	Excellent driver and car!

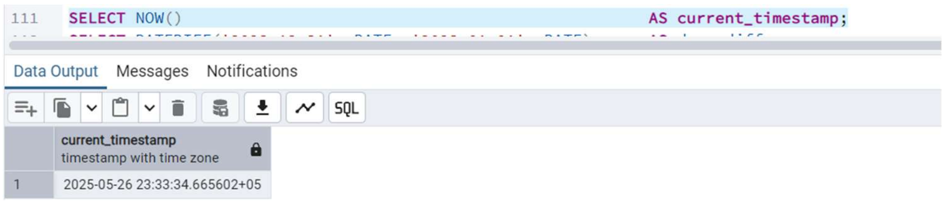
3. Date Built-In Functions

SELECT NOW();

Date Function: NOW()

How it works: The NOW() function returns the current date and time in the system's default timezone.

Relevance: In a ride-sharing or booking system, the NOW() function can be used to track the current timestamp for various operations, such as recording when a booking was made, checking the status of a ride, or sending notifications to users. It's crucial for time-sensitive operations like tracking ride arrivals, booking confirmations, or monitoring ride durations.



```
SELECT NOW()                                    AS current_timestamp;
```

Data Output Messages Notifications

	current_timestamp timestamp with time zone
1	2025-05-26 23:33:34.665602+05

SELECT DATEDIFF('2023-12-31', '2023-01-01') AS DAYS_DIFFERENCE;

Date Function: DATEDIFF()

How it works: The DATEDIFF() function calculates the difference between two dates and returns the result as the number of days. The first date ('2023-12-31') is subtracted from the second ('2023-01-01'), returning the difference in days.

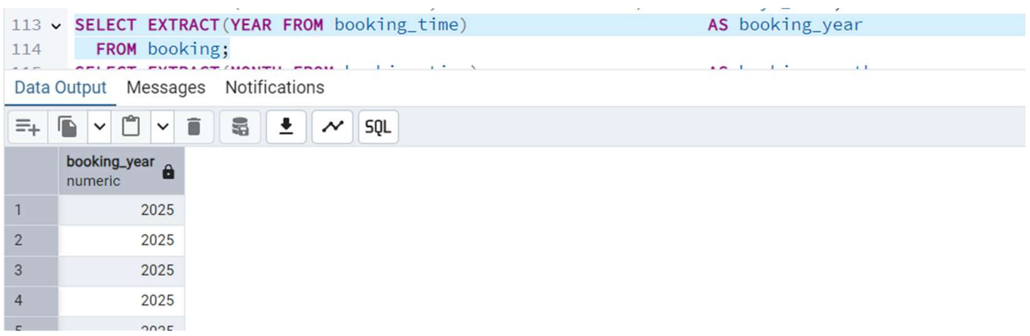
Relevance: In a ride-sharing system, this function can be used to calculate the number of days between a user's booking date and the date of ride completion, which is useful for calculating charges, cancellations, or providing services like reminders or offers for upcoming rides. It can also be applied to monitor how long a booking or ride has been pending or scheduled.

SELECT YEAR(BOOKING_TIME) AS BOOKING_YEAR FROM BOOKING;

Date Function: YEAR()

How it works: The YEAR() function extracts the year part from a given date (in this case, from the BOOKING_TIME column).

Relevance: This query can be used to track and categorize bookings by year, which can help in reporting, analytics, or performance tracking over time. For example, a ride-sharing business may want to track the number of bookings made in a given year, analyze trends in demand, or forecast future growth.



```
SELECT EXTRACT(YEAR FROM booking_time)                AS booking_year
  FROM booking;
```

	booking_year numeric
1	2025
2	2025
3	2025
4	2025

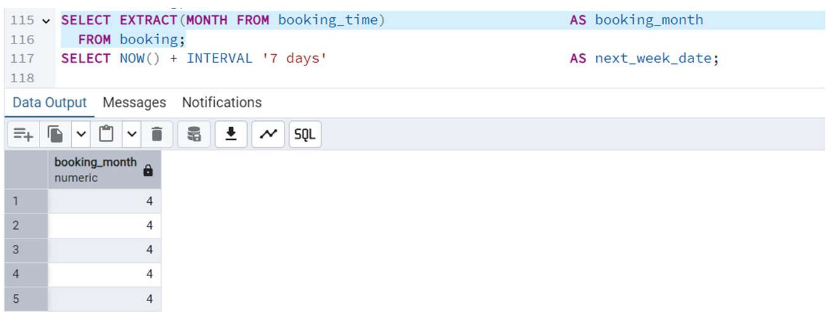
SELECT MONTH(BOOKING_TIME) AS BOOKING_MONTH FROM BOOKING;

Date Function: MONTH()

How it works: The MONTH() function extracts the month part from a given date (in this case, from the BOOKING_TIME column).

Relevance: This query can be used to analyze booking trends on a monthly basis. For example, a ride-sharing business might want to know which months have the highest

demand, track seasonal patterns, or create promotional campaigns during peak months. It can also help in scheduling drivers based on busy periods.



```
SELECT EXTRACT(MONTH FROM booking_time)                    AS booking_month
  FROM booking;
SELECT NOW() + INTERVAL '7 days'                           AS next_week_date;
```

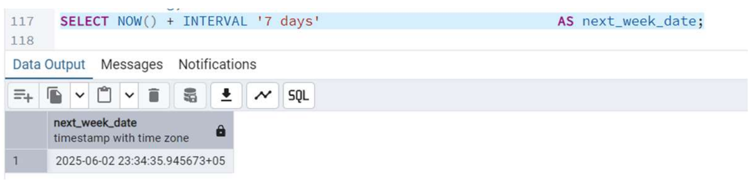
	booking_month numeric
1	4
2	4
3	4
4	4
5	4

SELECT DATE_ADD(NOW(), INTERVAL 7 DAY) AS NEXT_WEEK;

Date Function: DATE_ADD()

How it works: The DATE_ADD() function adds a specified interval (in this case, 7 days) to a given date (in this case, the current date returned by NOW()).

Relevance: This query is useful for projecting future dates based on the current date. In the context of a ride-sharing system, it can be used to forecast the date of an upcoming ride or booking, plan vehicle availability for the following week, or offer promotions or discounts for upcoming rides within a specific time frame. It's also useful for reminders, confirming reservations for the upcoming week, or notifying users about upcoming ride schedules.



```
SELECT NOW() + INTERVAL '7 days'                     AS next_week_date;
```

	next_week_date timestamp with time zone
1	2025-06-02 23:34:35.945673+05

4.Changing Data

UPDATE USER SET ROLE = 'DRIVER' WHERE USER_ID = 3;

Specific Update: The ROLE field of the user with USER_ID = 3 is set to 'DRIVER'.

Condition: USER_ID = 3 — This condition specifies that only the user with ID 3 will be updated.

Affected Rows: Only one row will be updated because the condition targets a single user (ID 3).

Relevance: This query could be used to promote a user (for example, a driver candidate) to the role of a 'DRIVER' in the system. It would typically be triggered when a user has completed the necessary onboarding process and is being assigned the role of a driver within the ride-sharing platform.

UPDATE USER SET EMAIL = 'DELETED' WHERE USER_ID>5 AND USER_ID<10;

Specific Update: The EMAIL field for users with USER_ID between 6 and 9 is updated to 'DELETED'.

Condition: USER_ID > 5 AND USER_ID < 10 — This condition ensures that the query targets users with IDs between 6 and 9 (inclusive of 6 and exclusive of 10).

Affected Rows: This will affect all users with IDs between 6 and 9, which could be 4 rows in total (depending on the actual data in the USER table).

Relevance: This update might be used in a scenario where the users' accounts are being deactivated or their emails are removed (perhaps due to account closure, data privacy reasons, or inactivity). The email is replaced with a placeholder value ("DELETED") to indicate the removal.

UPDATE RIDE SET ORIGIN = 'OUTSKIRTS' WHERE ORIGIN = 'DOWNTOWN';

Specific Update: The ORIGIN field is updated to 'OUTSKIRTS' for all rides where the origin was previously 'DOWNTOWN'.

Condition: ORIGIN = 'DOWNTOWN' — This condition selects all rides that originally had 'DOWNTOWN' as the origin.

Affected Rows: This will affect all rows where the ORIGIN is 'DOWNTOWN'. The exact number of rows depends on how many rides in the database have 'DOWNTOWN' as the origin.

Relevance: This query might be used when the origin of a set of rides is being redefined or changed due to a business decision or operational update. For example, if the company starts classifying downtown rides differently or needs to rename the origin point to reflect a new area (such as the outskirts instead of downtown), this update would reflect that change.

UPDATE BOOKING SET STATUS ='CANCELLED' WHERE BOOKING_TIME>='2025-04-04 21:00:00';

Specific Update: The STATUS field is set back to its previous value (OLD_VALUE) for all bookings where the BOOKING_TIME is on or after '2025-04-04 21:00:00'.

Condition: BOOKING_TIME >= '2025-04-04 21:00:00' — This condition ensures that only bookings made after or at 9:00 PM on April 4, 2025, are affected.

Affected Rows: The number of affected rows depends on how many bookings are scheduled at or after this timestamp.

Relevance: This query might be part of a system where the status of a booking needs to be reverted to its old value. For example, if there was an error in updating the booking status or a temporary change in status (like pending to confirmed), the system might need to restore the status for bookings after a specific time. It could also be part of an emergency or corrective action in response to a problem or system downtime.

UPDATE REVIEW SET RATING = 5 WHERE REVIEW_ID >5;

Specific Update: The RATING field is updated to 5 for all reviews with REVIEW_ID greater than 5.

Condition: REVIEW_ID > 5 — This condition ensures that all reviews with IDs greater than 5 are updated.

Affected Rows: This will update all reviews where the REVIEW_ID is greater than 5. The number of affected rows depends on the total number of reviews in the database with IDs greater than 5.

Relevance: This query could be used to fix an issue, such as a bug or system error, where ratings for certain reviews need to be corrected. It might also be applied in a scenario where the system wants to give high ratings (5 stars) to reviews based on certain conditions, such as if the reviews are older than a certain time and have been flagged for an update.

5. Deleting Data

DELETE FROM PAYMENT WHERE AMOUNT <=20;

Purpose: The query removes payment records where the transaction amount is less than or equal to 20. This may be done to clean up small-value transactions that are not necessary for ongoing analysis or reporting.

Condition: AMOUNT <= 20 — Deletes payments that are small in value, perhaps representing insignificant transactions like small tips or administrative fees.

Impact on the Database: The deletion will reduce the total number of payment records, which might improve query performance and database storage efficiency. However, it also removes historical financial data that could have been useful for comprehensive reporting or audit purposes.

DELETE FROM BOOKING WHERE PASSENGER_ID > 28 AND PASSENGER_ID < 30;

Purpose: The query deletes bookings made by passengers with PASSENGER_ID between 29 and 28. This may be done to remove bookings related to inactive, non-existent, or irrelevant passengers.

Condition: PASSENGER_ID > 28 AND PASSENGER_ID < 30 — Targets passengers with IDs 29 and 28 to delete any bookings they made. This could be part of a data cleanup strategy.

Impact on the Database: Removing these bookings will affect ride schedules, availability, and customer records. If these bookings are tied to other tables, such as payment or ride details, those records might also be affected.

DELETE FROM USER WHERE USER_ID > 28 AND USER_ID < 30;

Purpose: The query deletes user records where the USER_ID is between 29 and 28. This could be part of a data maintenance process where inactive, duplicate, or erroneous user records are removed.

Condition: USER_ID > 28 AND USER_ID < 30 — Deletes users with IDs 29 and 28, potentially in preparation for system cleanup or user account removal.

Impact on the Database: Deleting these users will remove their account details, associated ride history, and payment information. If these users have linked data in other tables, such as bookings or reviews, additional steps might be needed to handle these relationships.

DELETE FROM REVIEW WHERE LENGTH(COMMENT)<=20;

Purpose: This query deletes reviews with short comments (20 characters or fewer). Short, non-descriptive reviews might be seen as unhelpful and could clutter the review system.

Condition: LENGTH(COMMENT) <= 20 — Deletes reviews where the comment length is 20 characters or fewer, which may be considered too brief to provide meaningful feedback.

Impact on the Database: Removing these reviews will improve the quality and relevance of the review data, potentially enhancing the overall user experience. However, it also reduces the volume of available reviews, which could affect the completeness of customer feedback analysis.

DELETE FROM BOOKING WHERE RIDE_ID>15 AND RIDE_ID<18;

Purpose: This query deletes booking records associated with RIDE_ID values between 16 and 17. This could be done to remove bookings related to specific rides that are no longer relevant or have been canceled.

Condition: RIDE_ID > 15 AND RIDE_ID < 18 — Deletes bookings linked to rides with IDs 16 and 17, which might be obsolete, canceled, or related to test data.

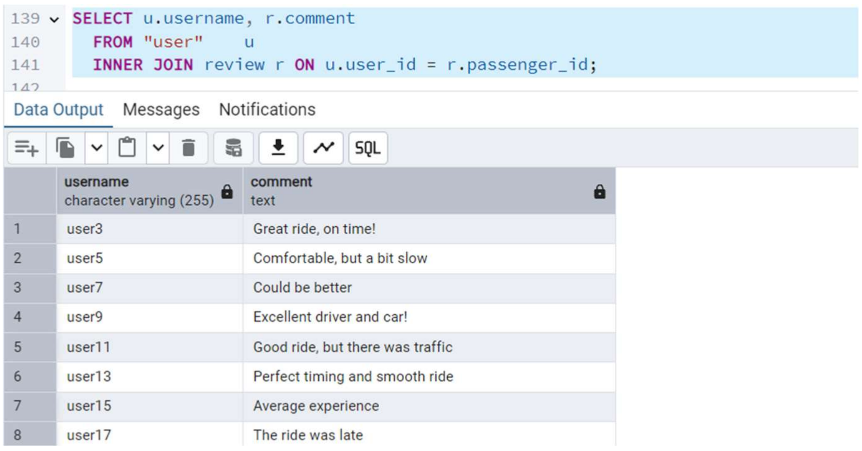
Impact on the Database: The deletion removes records for specific bookings, which could impact ride scheduling, availability, and reporting. If other tables, such as payment or review tables, are related to these bookings, those records may also need to be cleaned up to maintain data integrity.

Assignment4

1.INNER JOIN

SELECT USERNAME, COMMENT FROM USER INNER JOIN REVIEW ON USER_ID = PASSENGER_ID;

- This query connects the user and review tables by matching user.user_id with review.passenger_id, returning usernames and comments only for users who have submitted reviews. The INNER JOIN ensures results include only records with matches in both tables, excluding NULLs and showing reviews linked to valid passengers. In a ride-sharing context, this query is used to analyze passenger feedback, link reviews to specific users, generate satisfaction reports, and identify frequent or vocal reviewers.



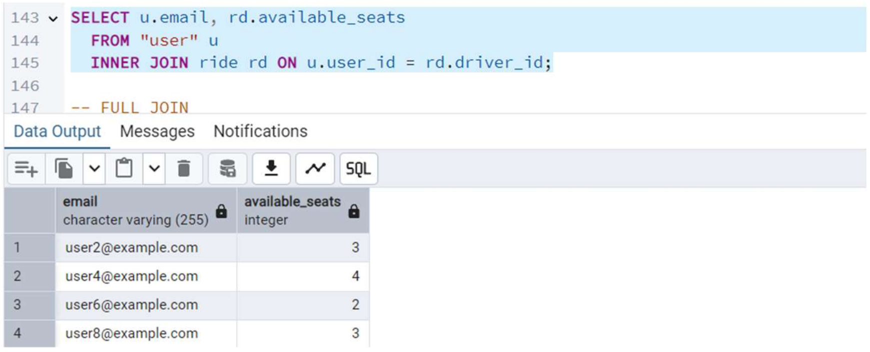
```
SELECT u.username, r.comment
  FROM "user"    u
  INNER JOIN review r ON u.user_id = r.passenger_id;
```

Data Output Messages Notifications

	username character varying (255)	comment text
1	user3	Great ride, on time!
2	user5	Comfortable, but a bit slow
3	user7	Could be better
4	user9	Excellent driver and car!
5	user11	Good ride, but there was traffic
6	user13	Perfect timing and smooth ride
7	user15	Average experience
8	user17	The ride was late

SELECT EMAIL, AVAILABLE_SEATS FROM USER INNER JOIN RIDE ON USER_ID = DRIVER_ID;

This query links the user and ride tables by matching user.user_id with ride.driver_id, returning driver emails and available seats for active rides only. The INNER JOIN ensures results include only active drivers, showing their contact info and ride capacity while excluding non-drivers or inactive rides. In ride-sharing, it helps identify active drivers, monitor availability, contact drivers, match passengers with seats, and analyze driver activity.



```
SELECT u.email, rd.available_seats
  FROM "user" u
  INNER JOIN ride rd ON u.user_id = rd.driver_id;

-- FULL JOIN
```

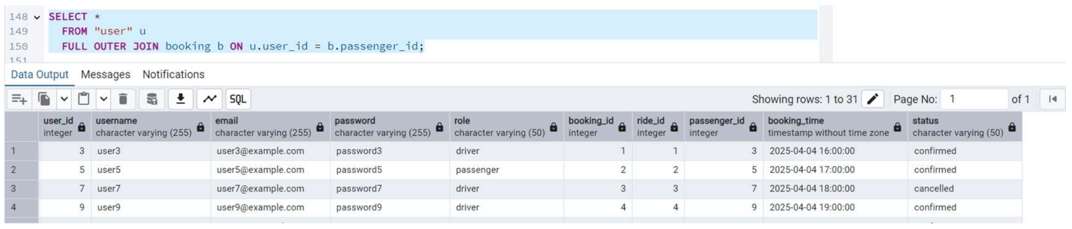
Data Output Messages Notifications

	email character varying (255)	available_seats integer
1	user2@example.com	3
2	user4@example.com	4
3	user6@example.com	2
4	user8@example.com	3

2.FULL JOIN

SELECT * FROM USER FULL JOIN BOOKING ON USER_ID = PASSENGER_ID;

This query connects the user and booking tables by matching user.user_id with booking.passenger_id, returning all records from both tables, including users without bookings and bookings without valid passengers. It shows NULLs where no matches exist, providing a complete view to audit user activity, identify inactive users, detect data issues, generate comprehensive reports, and find orphaned records.



```
SELECT *
  FROM "user" u
  FULL OUTER JOIN booking b ON u.user_id = b.passenger_id;
```

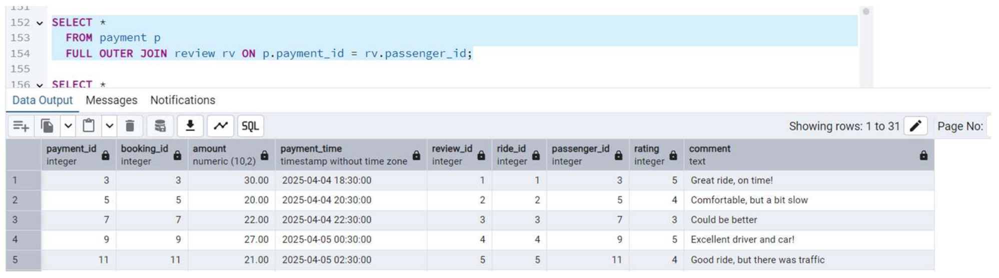
Data Output Messages Notifications

Showing rows: 1 to 31 Page No: 1 of 1

	user_id integer	username character varying (255)	email character varying (255)	password character varying (255)	role character varying (50)	booking_id integer	ride_id integer	passenger_id integer	booking_time timestamp without time zone	status character varying (50)
1	3	user3	user3@example.com	password3	driver	1	1	3	2025-04-04 16:00:00	confirmed
2	5	user5	user5@example.com	password5	passenger	2	2	5	2025-04-04 17:00:00	confirmed
3	7	user7	user7@example.com	password7	driver	3	3	7	2025-04-04 18:00:00	cancelled
4	9	user9	user9@example.com	password9	driver	4	4	9	2025-04-04 19:00:00	confirmed

SELECT * FROM PAYMENT FULL JOIN REVIEW ON PAYMENT_ID = PASSENGER_ID;

This query links the user and booking tables by matching user.user_id with booking.passenger_id, returning all records from both tables, including users without bookings and bookings without valid passengers. It handles NULLs where no matches exist, providing a full view for auditing user activity, identifying inactive users, detecting data issues, generating complete reports, and cleaning up orphaned records.



```
SELECT *
  FROM payment p
  FULL OUTER JOIN review rv ON p.payment_id = rv.passenger_id;

SELECT *
```

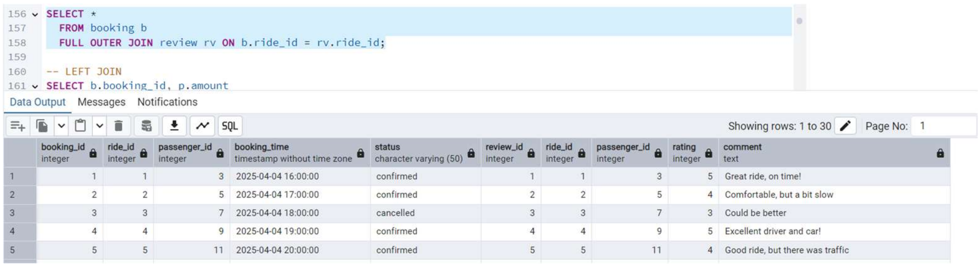
Data Output Messages Notifications

Showing rows: 1 to 31 Page No:

	payment_id integer	booking_id integer	amount numeric (10,2)	payment_time timestamp without time zone	review_id integer	ride_id integer	passenger_id integer	rating integer	comment text
1	3	3	30.00	2025-04-04 18:30:00	1	1	3	5	Great ride, on time!
2	5	5	20.00	2025-04-04 20:30:00	2	2	5	4	Comfortable, but a bit slow
3	7	7	22.00	2025-04-04 22:30:00	3	3	7	3	Could be better
4	9	9	27.00	2025-04-05 00:30:00	4	4	9	5	Excellent driver and car!
5	11	11	21.00	2025-04-05 02:30:00	5	5	11	4	Good ride, but there was traffic

SELECT * FROM BOOKING FULL JOIN REVIEW ON BOOKING.RIDE_ID = REVIEW.RIDE_ID;

This query links the booking and review tables by matching booking.ride_id with review.ride_id, returning all records from both tables, including bookings without reviews and reviews without bookings. It shows NULLs where no matches exist, providing comprehensive data for analyzing review coverage, identifying unreviewed rides, detecting system issues, and cleaning up orphaned records.



```
SELECT *
  FROM booking b
  FULL OUTER JOIN review rv ON b.ride_id = rv.ride_id;

-- LEFT JOIN
SELECT b.booking_id, p.amount
```

	booking_id integer	ride_id integer	passenger_id integer	booking_time timestamp without time zone	status character varying (50)	review_id integer	ride_id integer	passenger_id integer	rating integer	comment text
1	1	1	3	2025-04-04 16:00:00	confirmed	1	1	3	5	Great ride, on time!
2	2	2	5	2025-04-04 17:00:00	confirmed	2	2	5	4	Comfortable, but a bit slow
3	3	3	7	2025-04-04 18:00:00	cancelled	3	3	7	3	Could be better
4	4	4	9	2025-04-04 19:00:00	confirmed	4	4	9	5	Excellent driver and car!
5	5	5	11	2025-04-04 20:00:00	confirmed	5	5	11	4	Good ride, but there was traffic

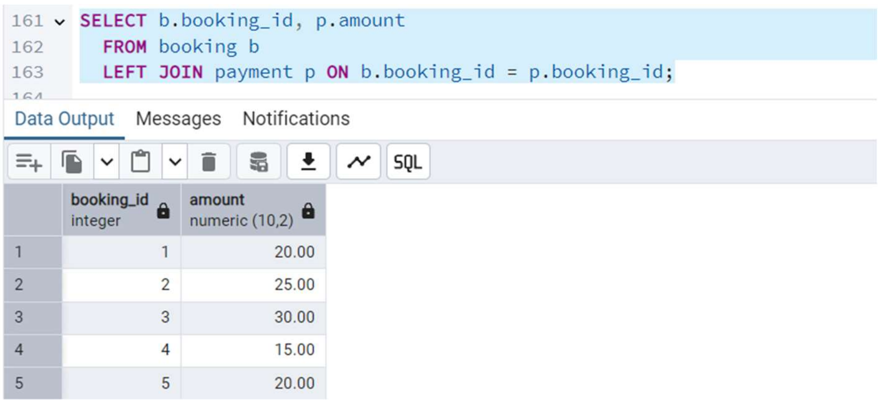
3.LEFT JOIN

SELECT booking.booking_id, amount

FROM booking

LEFT JOIN payment ON booking.booking_id = payment.booking_id;

This query connects the booking and payment tables by matching booking.booking_id with payment.booking_id, returning all bookings including unpaid ones with NULL payment amounts. It ensures every booking appears, tracks payment status, identifies unpaid bookings, and supports accounting and payment monitoring.

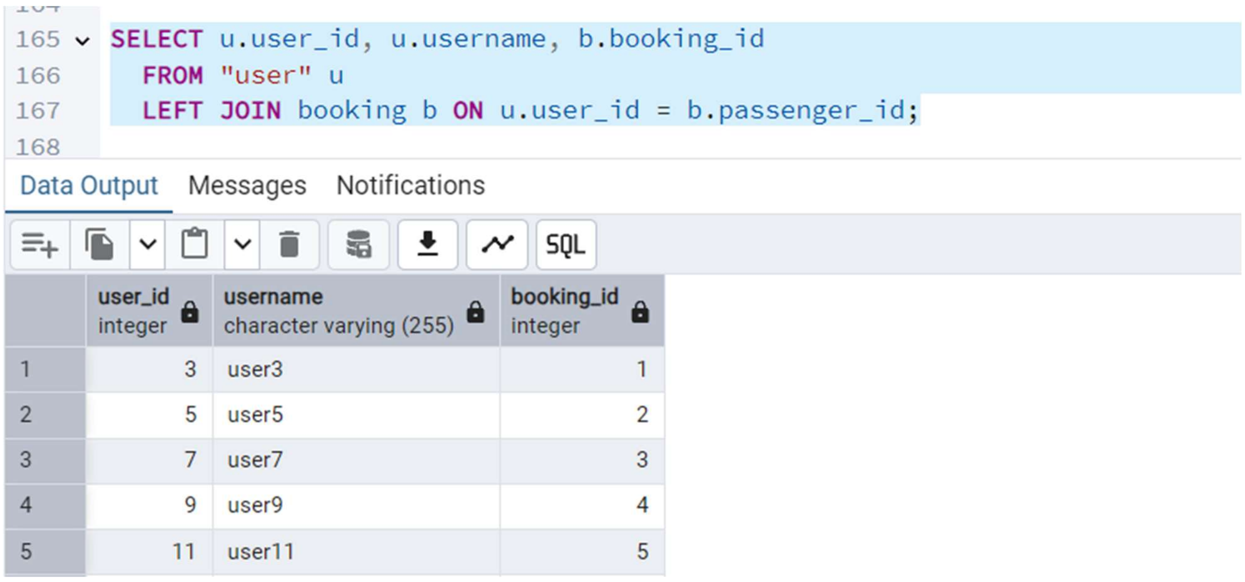


```
SELECT b.booking_id, p.amount
  FROM booking b
  LEFT JOIN payment p ON b.booking_id = p.booking_id;
```

	booking_id integer	amount numeric (10,2)
1	1	20.00
2	2	25.00
3	3	30.00
4	4	15.00
5	5	20.00

SELECT U.USER_ID, U.USERNAME, B.BOOKING_ID
FROM USER U
LEFT JOIN BOOKING B ON U.USER_ID = B.PASSENGER_ID;

This query links the user and booking tables by matching user.user_id with booking.passenger_id, returning all users with their bookings where available, and NULL booking IDs for inactive users. It ensures every user appears, helps track engagement, identify inactive passengers, analyze booking patterns, and segment users by activity.



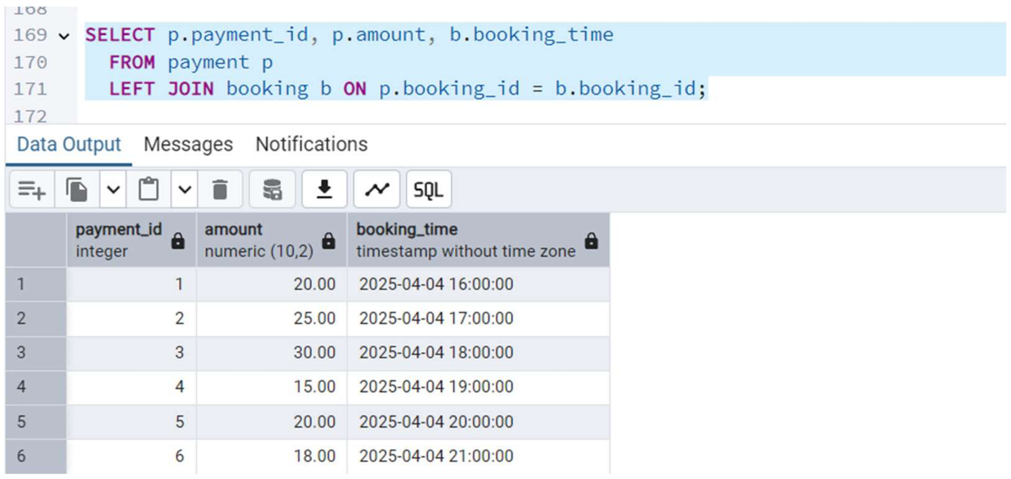
```
SELECT u.user_id, u.username, b.booking_id
  FROM "user" u
  LEFT JOIN booking b ON u.user_id = b.passenger_id;
```

Data Output   Messages   Notifications

	user_id integer	username character varying (255)	booking_id integer
1	3	user3	1
2	5	user5	2
3	7	user7	3
4	9	user9	4
5	11	user11	5

SELECT P.PAYMENT_ID, P.AMOUNT, B.BOOKING_TIME
FROM PAYMENT P
LEFT JOIN BOOKING B ON P.BOOKING_ID = B.BOOKING_ID;

This query connects the payment and booking tables by matching payment.booking_id with booking.booking_id, returning all payments with booking times when available and NULL for orphaned payments. It ensures every payment appears, helps audit payments, identify unassigned payments, reconcile transactions, and detect system issues.



```
SELECT p.payment_id, p.amount, b.booking_time
  FROM payment p
  LEFT JOIN booking b ON p.booking_id = b.booking_id;
```

Data Output   Messages   Notifications

	payment_id integer	amount numeric (10,2)	booking_time timestamp without time zone
1	1	20.00	2025-04-04 16:00:00
2	2	25.00	2025-04-04 17:00:00
3	3	30.00	2025-04-04 18:00:00
4	4	15.00	2025-04-04 19:00:00
5	5	20.00	2025-04-04 20:00:00
6	6	18.00	2025-04-04 21:00:00

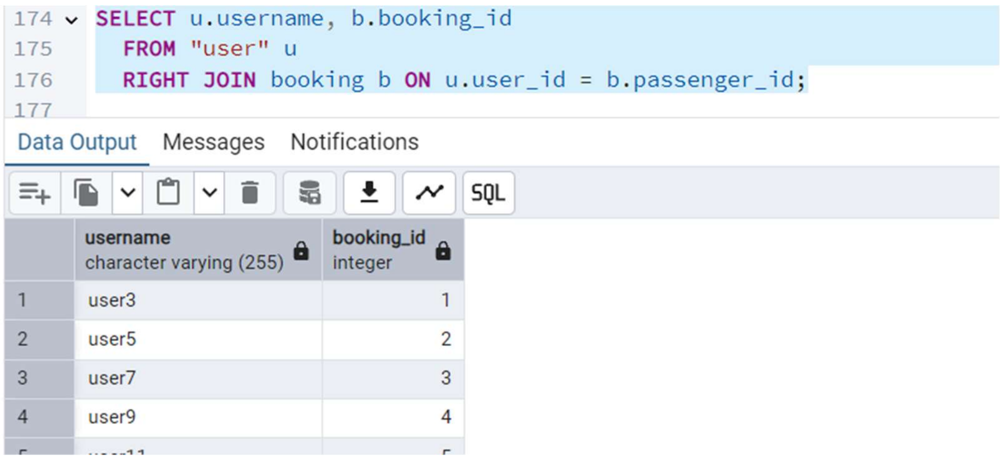
4.RIGHT JOIN

SELECT u.username, b.booking_id

FROM user u

RIGHT JOIN booking b ON u.user_id = b.passenger_id;

This query connects the user and booking tables by matching user.user_id with booking.passenger_id, returning all bookings including those without users with NULL usernames. It ensures all bookings appear, helps validate records, identify orphaned bookings, and audit passenger data.



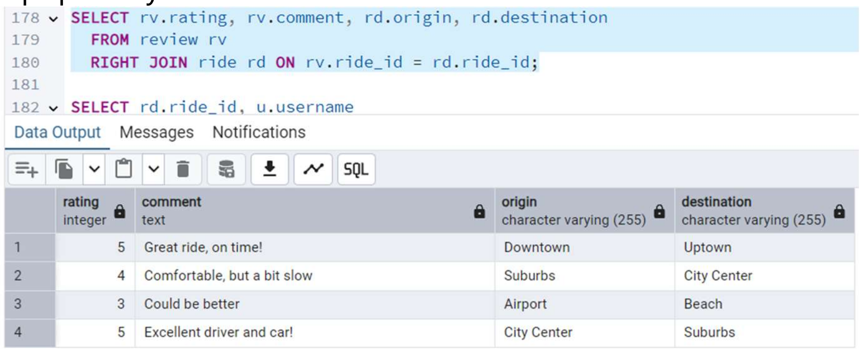
```
SELECT u.username, b.booking_id
  FROM "user" u
  RIGHT JOIN booking b ON u.user_id = b.passenger_id;
```

Data Output | Messages | Notifications

	username character varying (255)	booking_id integer
1	user3	1
2	user5	2
3	user7	3
4	user9	4

SELECT R.RATING, R.COMMENT, D.ORIGIN, D.DESTINATION
FROM REVIEW R
RIGHT JOIN RIDE D ON R.RIDE_ID = D.RIDE_ID;

This query connects the review and ride tables by matching review.ride_id with ride.ride_id, returning all rides including those without reviews with NULL ratings and comments. It ensu res all rides appear, helps track ride satisfaction, identify unreviewed rides, and analyze rout e popularity.



```
SELECT rv.rating, rv.comment, rd.origin, rd.destination
  FROM review rv
  RIGHT JOIN ride rd ON rv.ride_id = rd.ride_id;

SELECT rd.ride_id, u.username
```

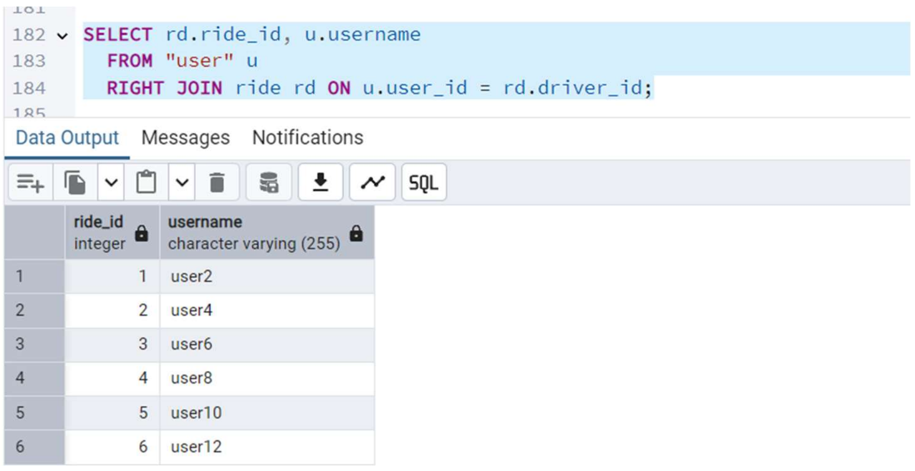
Data Output | Messages | Notifications

	rating integer	comment text	origin character varying (255)	destination character varying (255)
1	5	Great ride, on time!	Downtown	Uptown
2	4	Comfortable, but a bit slow	Suburbs	City Center
3	3	Could be better	Airport	Beach
4	5	Excellent driver and car!	City Center	Suburbs

SELECT R.RIDE_ID, U.USERNAME

FROM USER U
RIGHT JOIN RIDE R ON U.USER_ID = R.DRIVER_ID;

This query connects the user and ride tables by matching user.user_id with ride.driver_id, returning all ridesincluding those without drivers with NULL usernames. It ensures all rides appear, helps validate rides, identify orphaned rides, and audit driver information.



```
SELECT rd.ride_id, u.username
  FROM "user" u
  RIGHT JOIN ride rd ON u.user_id = rd.driver_id;
```

	ride_id integer	username character varying (255)
1	1	user2
2	2	user4
3	3	user6
4	4	user8
5	5	user10
6	6	user12

5.CROSS JOIN

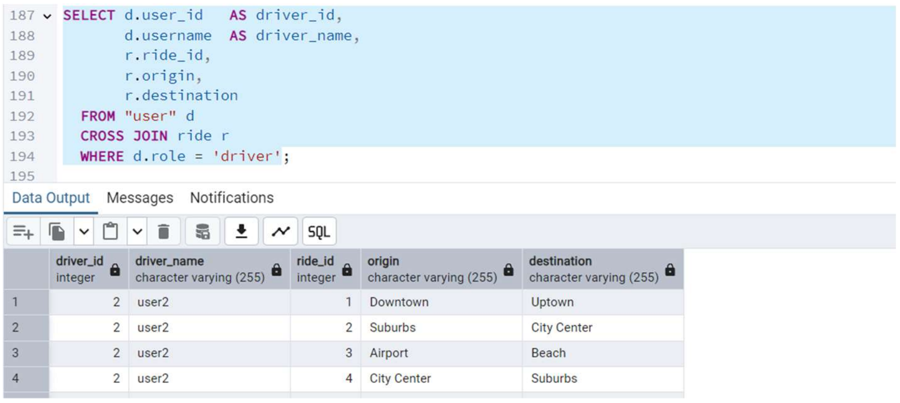
SELECT D.USER_ID AS DRIVER_ID, D.USERNAME AS DRIVER_NAME, R.RIDE_ID, R.ORIGIN, R.DESTINATION

FROM USER D

CROSS JOIN RIDE R

WHERE D.ROLE = 'DRIVER';

○



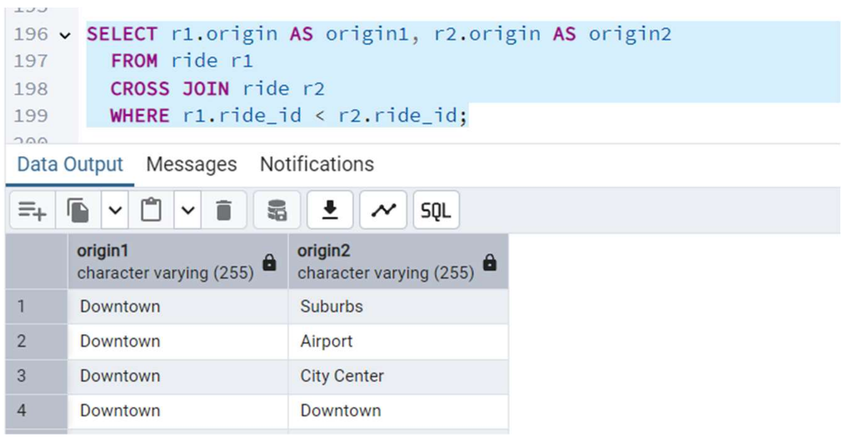
```
SELECT d.user_id   AS driver_id,
       d.username  AS driver_name,
       r.ride_id,
       r.origin,
       r.destination
  FROM "user" d
  CROSS JOIN ride r
  WHERE d.role = 'driver';
```

	driver_id integer	driver_name character varying (255)	ride_id integer	origin character varying (255)	destination character varying (255)
1	2	user2	1	Downtown	Uptown
2	2	user2	2	Suburbs	City Center
3	2	user2	3	Airport	Beach
4	2	user2	4	City Center	Suburbs

- This query creates all possible pairs between drivers and rides by generating a Cartesian product filtered for users with role 'DRIVER'. It returns driver and ride details for every combination, supporting driver assignment planning, system testing, and capacity analysis.

SELECT R1.ORIGIN AS ORIGIN1, R2.ORIGIN AS ORIGIN2
FROM RIDE R1
CROSS JOIN RIDE R2
WHERE R1.RIDE_ID < R2.RIDE_ID;

This query generates unique pairs of ride origins by creating all combinations of rides excluding duplicates and self-pairs. It returns paired origins to analyze route overlaps, plan driver allocation, and identify high-demand areas.



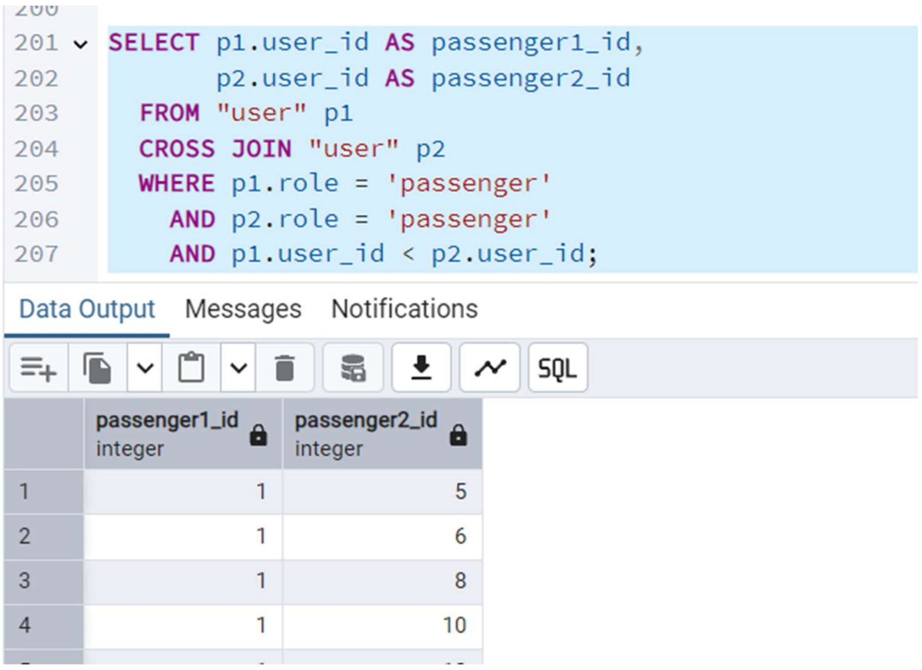
```
SELECT r1.origin AS origin1, r2.origin AS origin2
  FROM ride r1
  CROSS JOIN ride r2
  WHERE r1.ride_id < r2.ride_id;
```

	origin1 character varying (255)	origin2 character varying (255)
1	Downtown	Suburbs
2	Downtown	Airport
3	Downtown	City Center
4	Downtown	Downtown

SELECT P1.USER_ID AS PASSENGER1_ID,
  P1.USERNAME AS PASSENGER1_NAME,

```
   P2.USER_ID AS PASSENGER2_ID,
   P2.USERNAME AS PASSENGER2_NAME
FROM USER P1
CROSS JOIN USER P2
WHERE P1.ROLE = 'PASSENGER'
   AND P2.ROLE = 'PASSENGER'
   AND P1.USER_ID < P2.USER_ID;
```

This query creates all unique pairs of passengers by self-joining users with role 'PASSENGER' while excluding duplicates and self-pairs. It returns passenger pairs to support ride-sharing matching, social features, and passenger analytics.



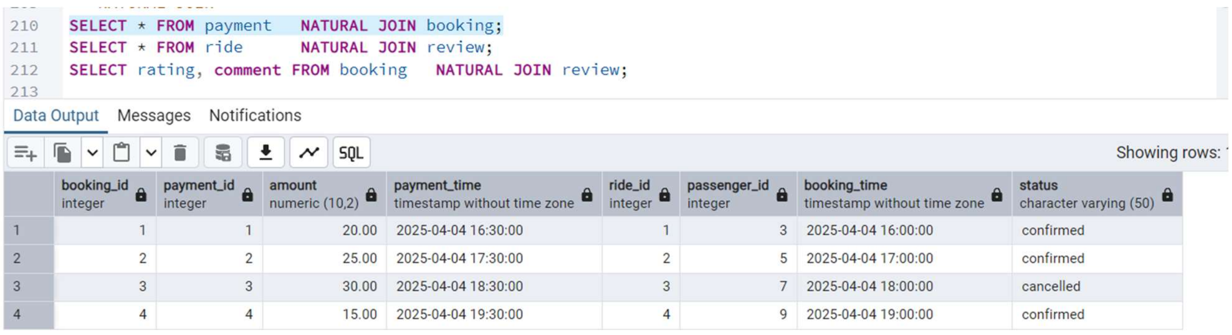
```
SELECT p1.user_id AS passenger1_id,
       p2.user_id AS passenger2_id
  FROM "user" p1
  CROSS JOIN "user" p2
  WHERE p1.role = 'passenger'
    AND p2.role = 'passenger'
    AND p1.user_id < p2.user_id;
```

	passenger1_id integer	passenger2_id integer
1	1	5
2	1	6
3	1	8
4	1	10

6.NATURAL JOIN

SELECT * FROM PAYMENT NATURAL JOIN BOOKING;

This query naturally joins payment and booking tables on booking_id, returning only matched records with combined columns and no duplicates. It helps verify payments against bookings, reconcile finances, and maintain data integrity.

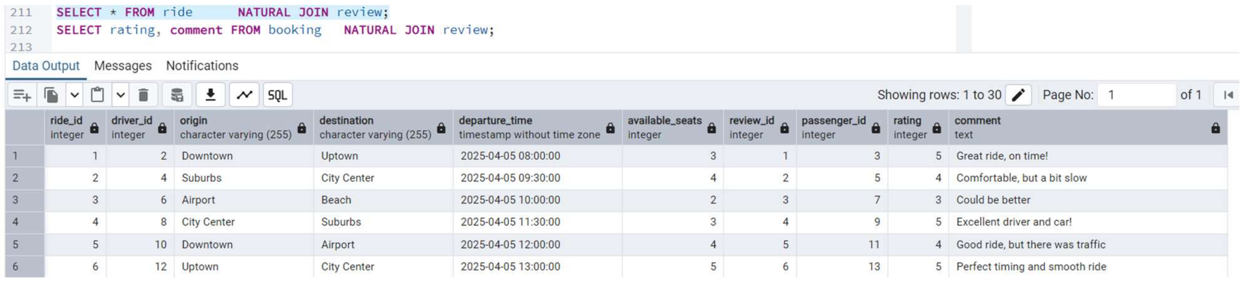


```
SELECT * FROM payment   NATURAL JOIN booking;
SELECT * FROM ride      NATURAL JOIN review;
SELECT rating, comment FROM booking   NATURAL JOIN review;
```

	booking_id integer	payment_id integer	amount numeric (10,2)	payment_time timestamp without time zone	ride_id integer	passenger_id integer	booking_time timestamp without time zone	status character varying (50)
1	1	1	20.00	2025-04-04 16:30:00	1	3	2025-04-04 16:00:00	confirmed
2	2	2	25.00	2025-04-04 17:30:00	2	5	2025-04-04 17:00:00	confirmed
3	3	3	30.00	2025-04-04 18:30:00	3	7	2025-04-04 18:00:00	cancelled
4	4	4	15.00	2025-04-04 19:30:00	4	9	2025-04-04 19:00:00	confirmed

SELECT * FROM RIDE NATURAL JOIN REVIEW;

This query naturally joins ride and review tables on ride_id, returning only rides with reviews and merging shared columns. It may miss accuracy without passenger matching but is used to retrieve reviewed rides.



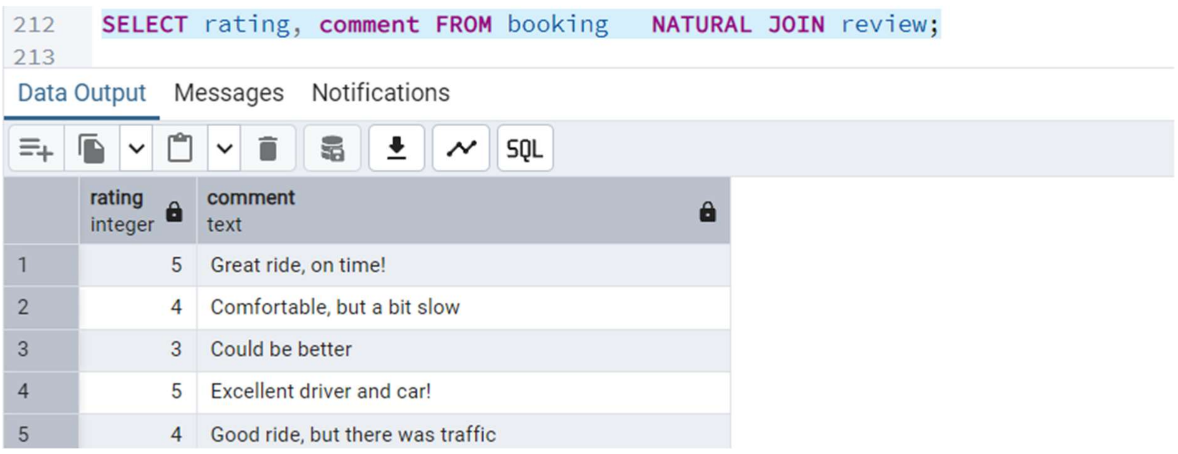
```
SELECT * FROM ride      NATURAL JOIN review;
SELECT rating, comment FROM booking   NATURAL JOIN review;
```

Showing rows: 1 to 30   Page No: 1 of 1

	ride_id integer	driver_id integer	origin character varying (255)	destination character varying (255)	departure_time timestamp without time zone	available_seats integer	review_id integer	passenger_id integer	rating integer	comment text
1	1	2	Downtown	Uptown	2025-04-05 08:00:00	3	1	3	5	Great ride, on time!
2	2	4	Suburbs	City Center	2025-04-05 09:30:00	4	2	5	4	Comfortable, but a bit slow
3	3	6	Airport	Beach	2025-04-05 10:00:00	2	3	7	3	Could be better
4	4	8	City Center	Suburbs	2025-04-05 11:30:00	3	4	9	5	Excellent driver and car!
5	5	10	Downtown	Airport	2025-04-05 12:00:00	4	5	11	4	Good ride, but there was traffic
6	6	12	Uptown	City Center	2025-04-05 13:00:00	5	6	13	5	Perfect timing and smooth ride

SELECT RATING, COMMENT FROM BOOKING NATURAL JOIN REVIEW;

- This query naturally joins booking and review tables on ride_id and passenger_id, returning ratings and comments only for bookings with matching reviews. It ensures accurate linkage of reviews to bookings.



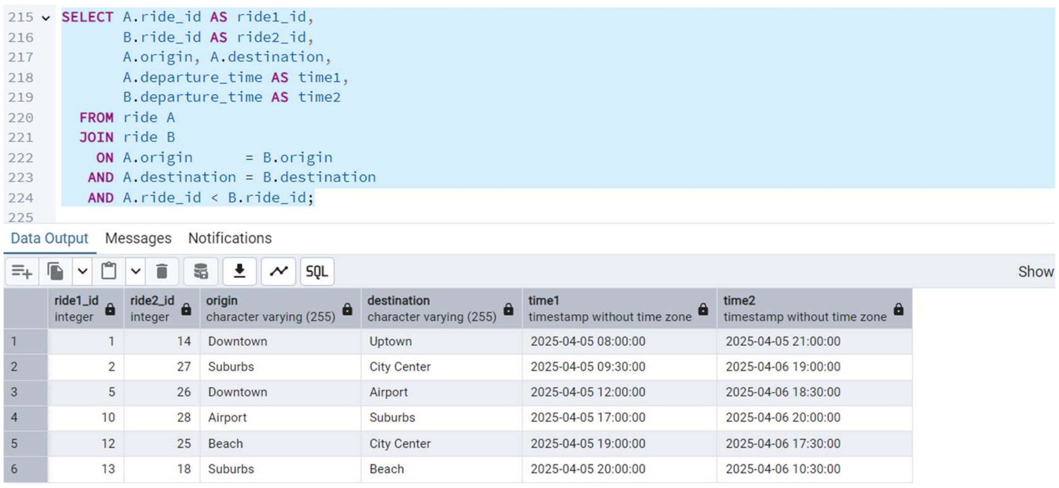
```
SELECT rating, comment FROM booking   NATURAL JOIN review;
```

	rating integer	comment text
1	5	Great ride, on time!
2	4	Comfortable, but a bit slow
3	3	Could be better
4	5	Excellent driver and car!
5	4	Good ride, but there was traffic

7.SELF JOIN

```
SELECT A.RIDE_ID AS RIDE1_ID, A.ORIGIN, A.DESTINATION,
     B.RIDE_ID AS RIDE2_ID,
     A.DEPARTURE_TIME AS TIME1, B.DEPARTURE_TIME AS TIME2
FROM RIDE A
JOIN RIDE B ON A.ORIGIN = B.ORIGIN
        AND A.DESTINATION = B.DESTINATION
        AND A.RIDE_ID < B.RIDE_ID;
```

This query self-joins rides to pair those with identical origin and destination, excluding duplicates and self-matches. It returns paired ride IDs and departure times to identify frequent or duplicate routes.



```
SELECT A.ride_id AS ride1_id,
       B.ride_id AS ride2_id,
       A.origin, A.destination,
       A.departure_time AS time1,
       B.departure_time AS time2
  FROM ride A
  JOIN ride B
    ON A.origin      = B.origin
   AND A.destination = B.destination
   AND A.ride_id < B.ride_id;
```

	ride1_id integer	ride2_id integer	origin character varying (255)	destination character varying (255)	time1 timestamp without time zone	time2 timestamp without time zone
1	1	14	Downtown	Uptown	2025-04-05 08:00:00	2025-04-05 21:00:00
2	2	27	Suburbs	City Center	2025-04-05 09:30:00	2025-04-06 19:00:00
3	5	26	Downtown	Airport	2025-04-05 12:00:00	2025-04-06 18:30:00
4	10	28	Airport	Suburbs	2025-04-05 17:00:00	2025-04-06 20:00:00
5	12	25	Beach	City Center	2025-04-05 19:00:00	2025-04-06 17:30:00
6	13	18	Suburbs	Beach	2025-04-05 20:00:00	2025-04-06 10:30:00

```
SELECT A.USER_ID AS ID1, A.USERNAME AS NAME1,
        B.USER_ID AS ID2, B.USERNAME AS NAME2
FROM USER A
JOIN USER B ON A.USERNAME = B.USERNAME
        AND A.USER_ID < B.USER_ID;
```

This query self-joins the user table to find users with duplicate usernames by matching user names and excluding self-pairs. It returns pairs of duplicate accounts to detect and resolve username collisions.

SELECT U1.USER_ID AS USER1_ID, U1.EMAIL AS EMAIL,U2.USER_ID AS USER2_ID
FROM USER U1
JOIN USER U2 ON U1.EMAIL = U2.EMAIL
AND U1.USER_ID < U2.USER_ID;

This query self-joins the user table to find users sharing the same email by matching emails and excluding self-pairs. It returns pairs of accounts to enforce unique email policies and improve data security.

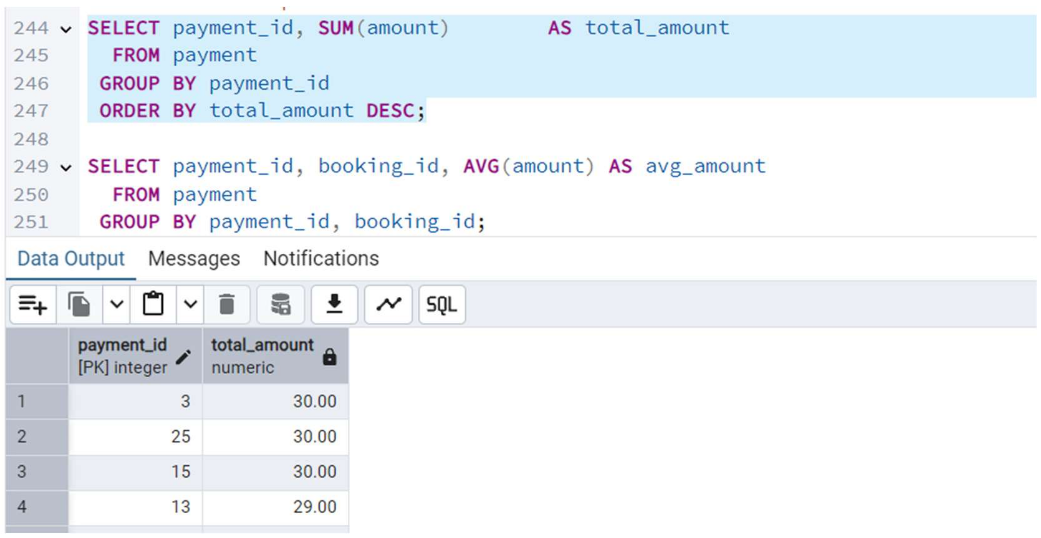
Assignment5

Grouping

1.Basic Grouping with an Aggregate Function

select payment_id, sum(amount) from payment group by payment_id order by amount desc;

Description:
Groups payments by their unique payment_id and calculates the sum of amount for each. Since payment_id is a primary key, this query is redundant (each group contains only one row). Results are sorted by amount in descending order.



```
SELECT payment_id, SUM(amount)        AS total_amount
  FROM payment
 GROUP BY payment_id
 ORDER BY total_amount DESC;

SELECT payment_id, booking_id, AVG(amount) AS avg_amount
  FROM payment
 GROUP BY payment_id, booking_id;
```

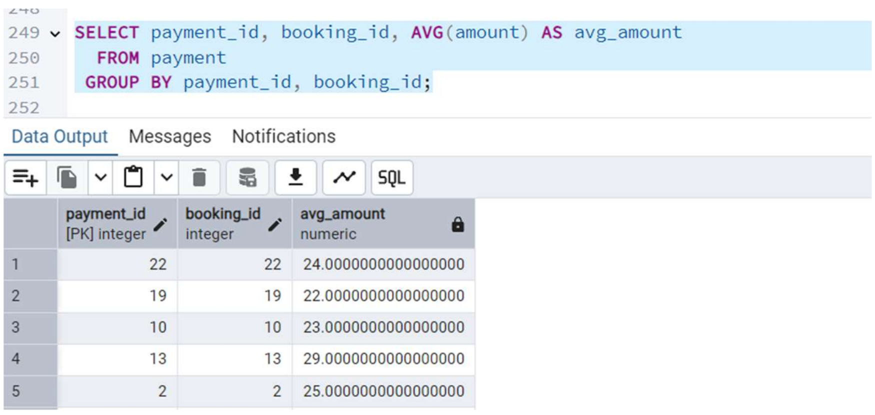
	payment_id [PK] integer	total_amount numeric
1	3	30.00
2	25	30.00
3	15	30.00
4	13	29.00

2.Grouping with Multiple Columns

select payment_id, booking_id, avg(amount) from payment group by payment_id, booking_id;

Description:
Groups payments by both payment_id and booking_id, then calculates the

average amount for each group. Like the first query, this is unnecessary because payment_id is already unique.



```
SELECT payment_id, booking_id, AVG(amount) AS avg_amount
  FROM payment
 GROUP BY payment_id, booking_id;
```

Data Output   Messages   Notifications

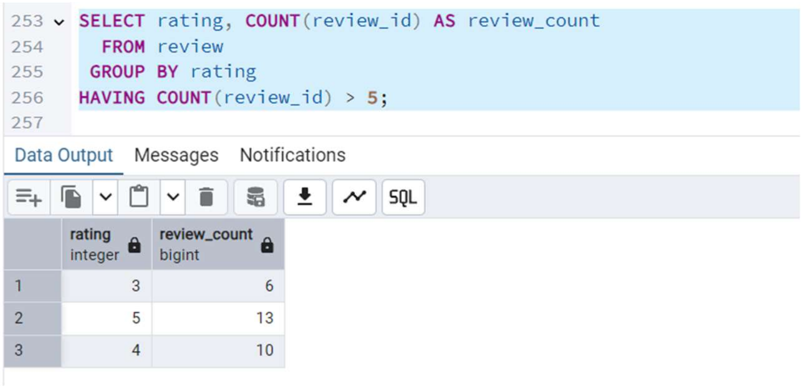
	payment_id [PK] integer	booking_id integer	avg_amount numeric
1	22	22	24.0000000000000000
2	19	19	22.0000000000000000
3	10	10	23.0000000000000000
4	13	13	29.0000000000000000
5	2	2	25.0000000000000000

3.Grouping with HAVING

select rating, count(review_id) from review group by(rating)having count(review_id)>5;

Description:

Groups reviews by rating and counts how many reviews exist for each rating. Filters out ratings with fewer than 5 reviews using HAVING.



```
SELECT rating, COUNT(review_id) AS review_count
  FROM review
 GROUP BY rating
HAVING COUNT(review_id) > 5;
```

Data Output   Messages   Notifications

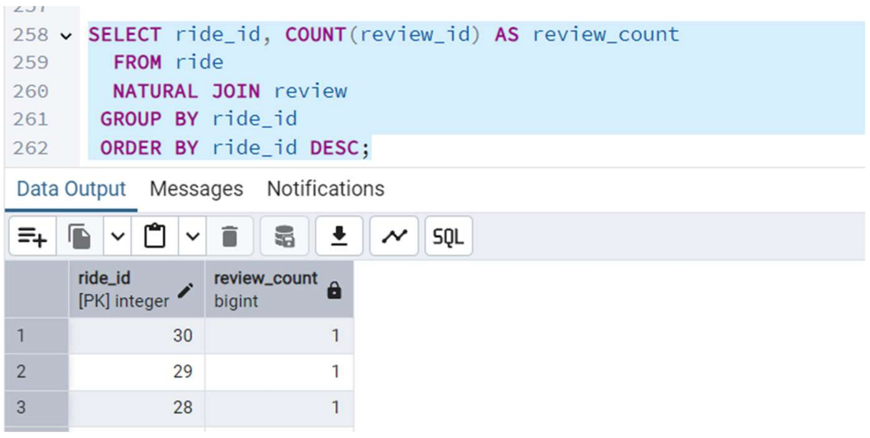
	rating integer	review_count bigint
1	3	6
2	5	13
3	4	10

4.Join and Grouping

select ride_id, count(review_id) from ride natural join review group by ride_id order by ride_id desc;

Description:

Joins the ride and review tables on ride_id and counts the number of reviews for each ride. Results are sorted by ride_id in descending order.



```sql
SELECT ride_id, COUNT(review_id) AS review_count
  FROM ride
  NATURAL JOIN review
 GROUP BY ride_id
 ORDER BY ride_id DESC;
```

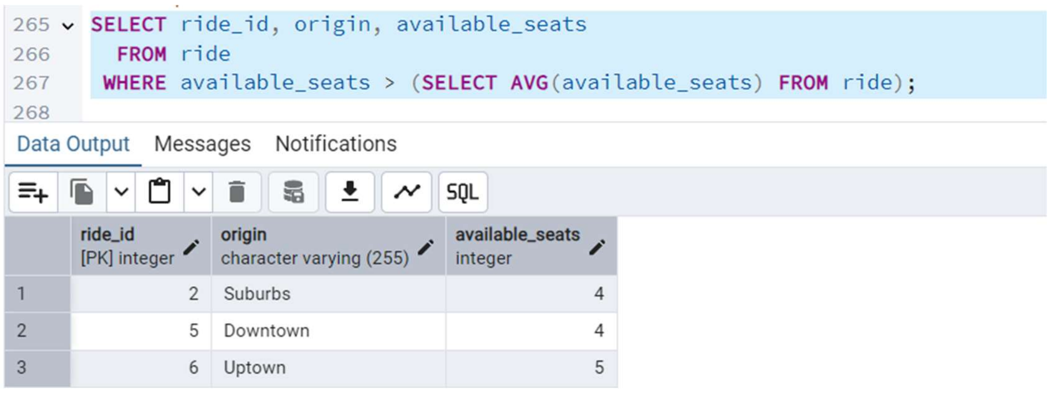
	ride_id [PK] integer	review_count bigint
1	30	1
2	29	1
3	28	1

Queries

1.Scalar Subquery

select ride_id, origin, available_seats from ride where available_seats > (select avg(available_seats) from ride);

Description:
Finds rides where available_seats is greater than the average number of seats across all rides. The subquery calculates the average seats.



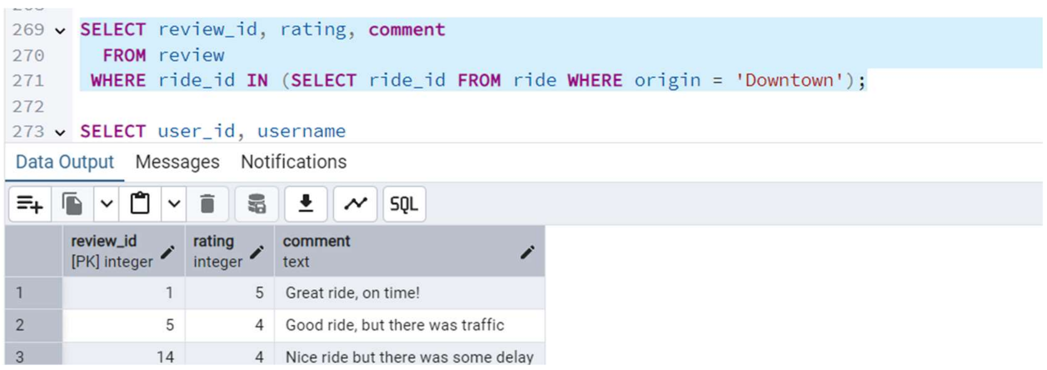
```sql
SELECT ride_id, origin, available_seats
  FROM ride
 WHERE available_seats > (SELECT AVG(available_seats) FROM ride);
```

	ride_id [PK] integer	origin character varying (255)	available_seats integer
1	2	Suburbs	4
2	5	Downtown	4
3	6	Uptown	5

2.Subquery with IN

select review_id, rating, comment from review where ride_id in (select ride_id from ride where origin = 'downtown');

Description:
Selects reviews for rides that originated in 'Downtown'. The subquery returns a list of ride_ids matching the condition.



```
SELECT review_id, rating, comment
  FROM review
 WHERE ride_id IN (SELECT ride_id FROM ride WHERE origin = 'Downtown');

SELECT user_id, username
```

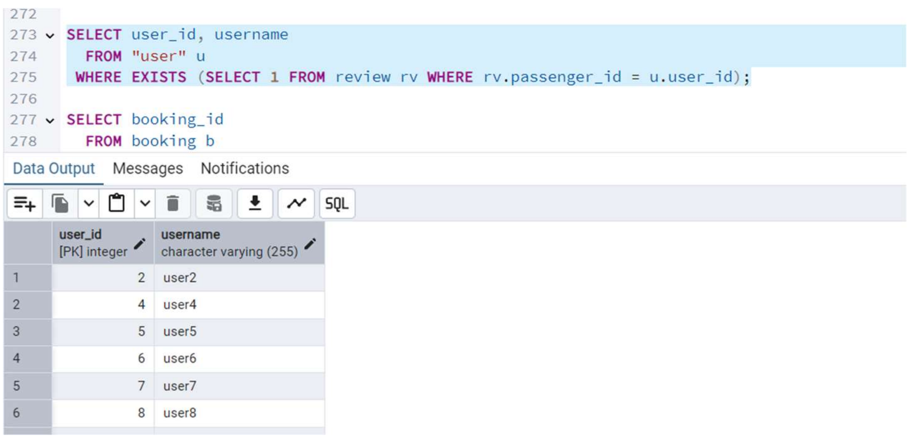
Data Output | Messages | Notifications

	review_id [PK] integer	rating integer	comment text
1	1	5	Great ride, on time!
2	5	4	Good ride, but there was traffic
3	14	4	Nice ride but there was some delay

3.Correlated Subquery

select user_id, username from "user" u where exists (select * from review where passenger_id = u.user_id);

Description:
Finds users who have left at least one review. The subquery checks if a matching passenger_id exists in the review table for each user.



```
SELECT user_id, username
  FROM "user" u
 WHERE EXISTS (SELECT 1 FROM review rv WHERE rv.passenger_id = u.user_id);

SELECT booking_id
  FROM booking b
```

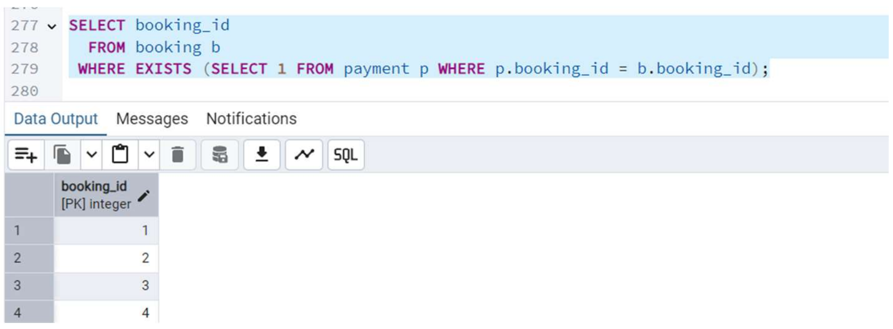
Data Output | Messages | Notifications

	user_id [PK] integer	username character varying (255)
1	2	user2
2	4	user4
3	5	user5
4	6	user6
5	7	user7
6	8	user8

4.Subquery with EXISTS

select booking_id from booking where exists (select 1 from payment where booking_id = booking.booking_id);

Description:
Returns booking_ids that have at least one associated payment. The subquery checks for the existence of a matching record in payment.



```
SELECT booking_id
  FROM booking b
 WHERE EXISTS (SELECT 1 FROM payment p WHERE p.booking_id = b.booking_id);
```

Data Output   Messages   Notifications

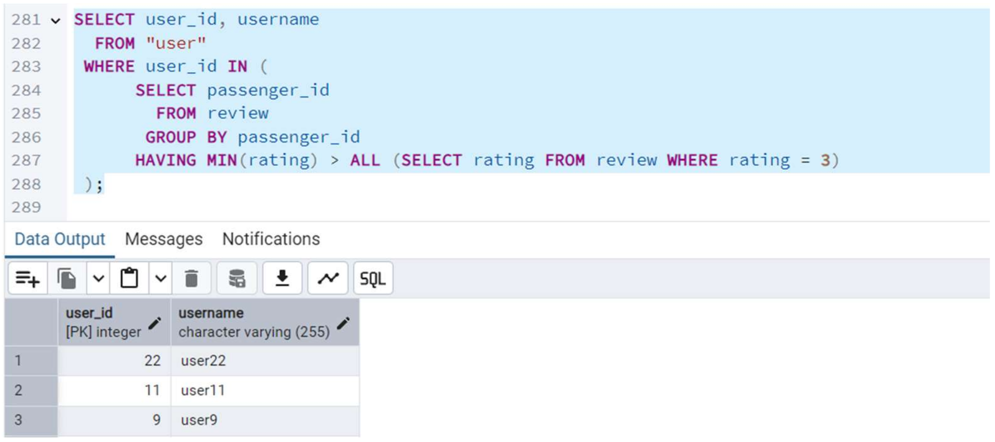
	booking_id [PK] integer
1	1
2	2
3	3
4	4

5.Subquery with ALL

select user_id, username from "user" where user_id in (select passenger_id from review group by passenger_id having min(rating) > all (select rating from review where rating = 3));

Description:

Finds users whose **lowest** review rating is still higher than **all** reviews with a rating of 3. Essentially, selects users who never received a rating of 3 or below.



```
SELECT user_id, username
  FROM "user"
 WHERE user_id IN (
      SELECT passenger_id
        FROM review
       GROUP BY passenger_id
      HAVING MIN(rating) > ALL (SELECT rating FROM review WHERE rating = 3)
 );
```

Data Output   Messages   Notifications

	user_id [PK] integer	username character varying (255)
1	22	user22
2	11	user11
3	9	user9

6.Row Subquery

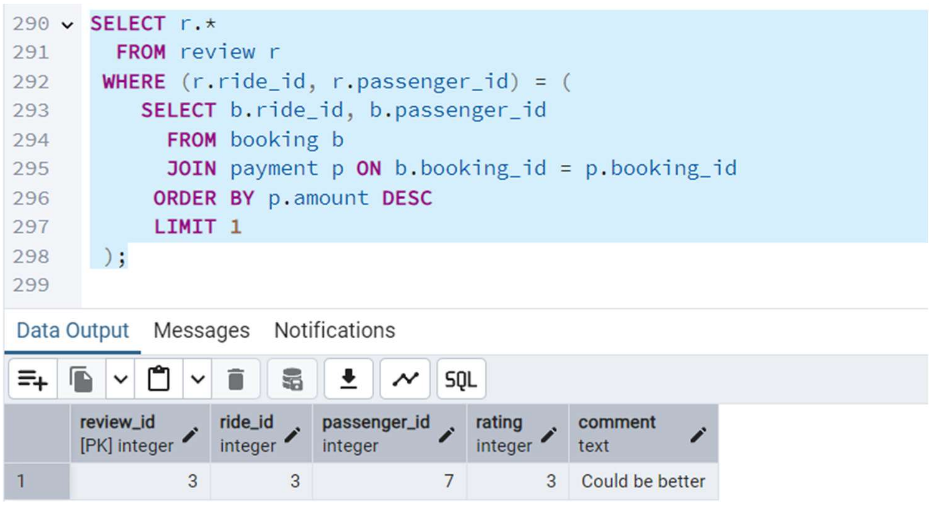
select r.* from review r where (r.ride_id, r.passenger_id) = (select b.ride_id, b.passenger_id

from booking b join payment p on b.booking_id = p.booking_id order by p.amount desc

limit 1);

Description:

This query is designed to identify the most lucrative ride in your system—the one that generated the **highest payment**—and then retrieve the corresponding **customer review**. It does so in two step



```
SELECT r.*
  FROM review r
 WHERE (r.ride_id, r.passenger_id) = (
    SELECT b.ride_id, b.passenger_id
      FROM booking b
      JOIN payment p ON b.booking_id = p.booking_id
     ORDER BY p.amount DESC
     LIMIT 1
 );
```

	review_id [PK] integer	ride_id integer	passenger_id integer	rating integer	comment text
1	3	3	7	3	Could be better

Assignment6

Window functions

select status,payment_time,amount,

lag(amount) over(partition by status order by payment_time),

amount- lag(amount) over(partition by status order by payment_time) as difference

from payment p join booking b on p.booking_id = b.booking_id join "user" u on b.passenger_id = u.user_id

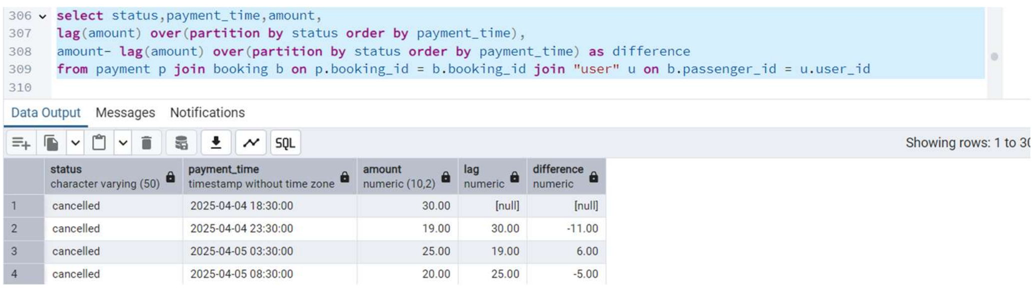
1. Window Function: Payment Amount Comparison by Status

Purpose: Analyze payment trends by comparing each payment with the previous one in the same status category.

Functionality:

- Partitions payments by status and orders them by payment_time.
- Uses LAG() to fetch the previous payment amount in the same status group.
- Calculates the difference between the current and previous payment.

Expected Result: A table showing each payment with its previous amount and the difference between them, grouped by status.



```
select status,payment_time,amount,
lag(amount) over(partition by status order by payment_time),
amount- lag(amount) over(partition by status order by payment_time) as difference
from payment p join booking b on p.booking_id = b.booking_id join "user" u on b.passenger_id = u.user_id
```

Data Output Messages Notifications

Showing rows: 1 to 30

	status character varying (50)	payment_time timestamp without time zone	amount numeric (10,2)	lag numeric	difference numeric
1	cancelled	2025-04-04 18:30:00	30.00	[null]	[null]
2	cancelled	2025-04-04 23:30:00	19.00	30.00	-11.00
3	cancelled	2025-04-05 03:30:00	25.00	19.00	6.00
4	cancelled	2025-04-05 08:30:00	20.00	25.00	-5.00

select amount, avg(amount) over() as average, sum(amount) over() as total,

amount * 100/sum(amount) over() as percentage from payment;

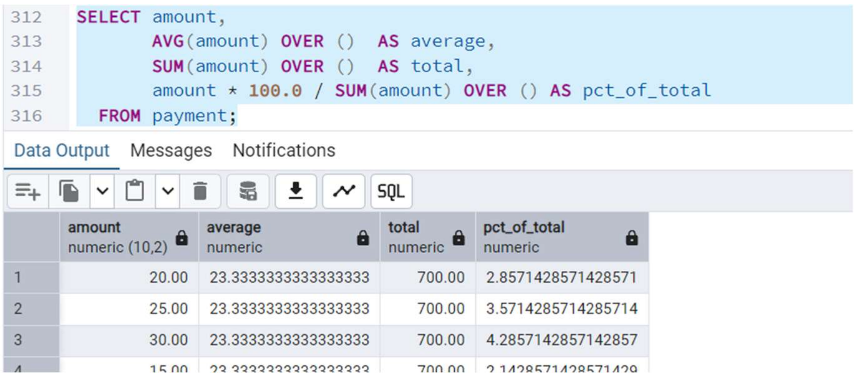
2. Window Function: Payment Statistics Overview

Purpose: Provide an overview of payment statistics, including average, total, and percentage contribution of each payment.

Functionality:

- Computes the average (avg(amount)) and total (sum(amount)) across all payments.
- Calculates the percentage of each payment relative to the total.

Expected Result: A table listing each payment alongside the overall average, total, and its percentage of the total.



```
SELECT amount,
       AVG(amount) OVER ()  AS average,
       SUM(amount) OVER ()  AS total,
       amount * 100.0 / SUM(amount) OVER () AS pct_of_total
  FROM payment;
```

Data Output Messages Notifications

	amount numeric (10,2)	average numeric	total numeric	pct_of_total numeric
1	20.00	23.3333333333333333	700.00	2.8571428571428571
2	25.00	23.3333333333333333	700.00	3.5714285714285714
3	30.00	23.3333333333333333	700.00	4.2857142857142857
4	15.00	23.3333333333333333	700.00	2.1428571428571429

select status,payment_time,amount,

avg(amount) over(partition by status order by payment_time rows between 1 preceding and 1 following)

from payment p join booking b on p.booking_id = b.booking_id join "user" u on b.passenger_id = u.user_id

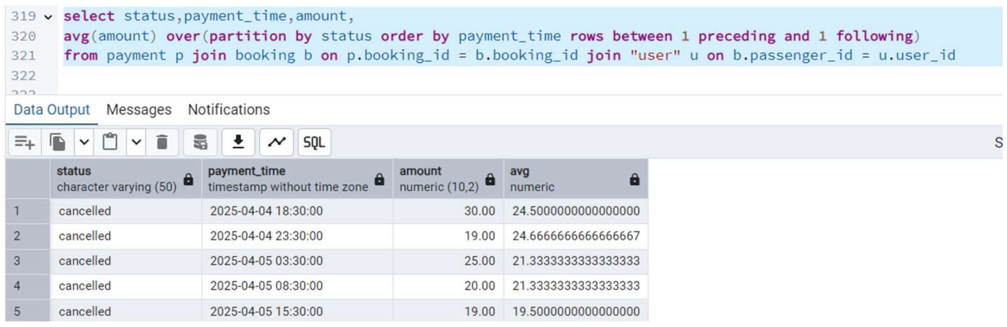
3. Window Function: Moving Average of Payments by Status

Purpose: Calculate a moving average of payment amounts for each status to smooth out fluctuations.

Functionality:

- Partitions payments by status and orders them by payment_time.
- Computes the moving average using a 3-row window (current row, one preceding, and one following).

Expected Result: A table showing payments with a moving average for each status, highlighting trends over time.



```
select status,payment_time,amount,
avg(amount) over(partition by status order by payment_time rows between 1 preceding and 1 following)
from payment p join booking b on p.booking_id = b.booking_id join "user" u on b.passenger_id = u.user_id
```

	status character varying (50)	payment_time timestamp without time zone	amount numeric (10,2)	avg numeric
1	cancelled	2025-04-04 18:30:00	30.00	24.5000000000000000
2	cancelled	2025-04-04 23:30:00	19.00	24.6666666666666667
3	cancelled	2025-04-05 03:30:00	25.00	21.3333333333333333
4	cancelled	2025-04-05 08:30:00	20.00	21.3333333333333333
5	cancelled	2025-04-05 15:30:00	19.00	19.5000000000000000

Set operations

select status, role, sum(amount) as total_amount from payment p

join booking b on p.booking_id = b.booking_id

join "user" u on b.passenger_id = u.user_id

group by grouping sets (

(status),

(role),

  (status, role),

  ()

);

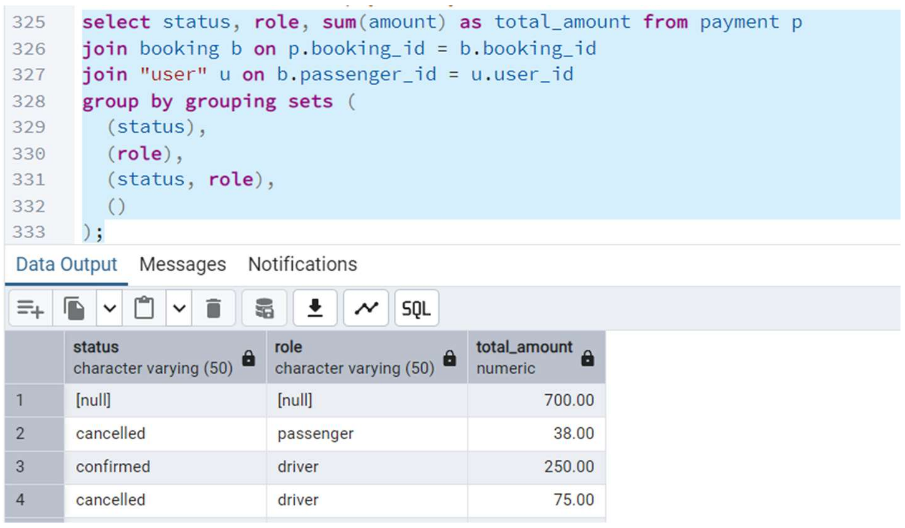
4. Grouping Sets: Payment Amounts by Status and Role

Purpose: Aggregate payment amounts at multiple grouping levels (status, role, and their combination).

Functionality:

- Uses GROUPING SETS to compute sums of amount for:
  - Each status individually.
  - Each role individually.
  - Combinations of status and role.
  - A grand total (empty grouping set).

Expected Result: A table with subtotals and a grand total for payment amounts, broken down by status and role.



```sql
select status, role, sum(amount) as total_amount from payment p
join booking b on p.booking_id = b.booking_id
join "user" u on b.passenger_id = u.user_id
group by grouping sets (
  (status),
  (role),
  (status, role),
  ()
);
```

Data Output   Messages   Notifications

	status character varying (50)	role character varying (50)	total_amount numeric
1	[null]	[null]	700.00
2	cancelled	passenger	38.00
3	confirmed	driver	250.00
4	cancelled	driver	75.00

select role, status, count(rating), grouping(role), grouping(status) from review r join "user" u on

r.passenger_id = u.user_id join booking b on u.user_id = b.passenger_id group by grouping sets(

        (status),

        (role),

        (role,status),

        ()

);

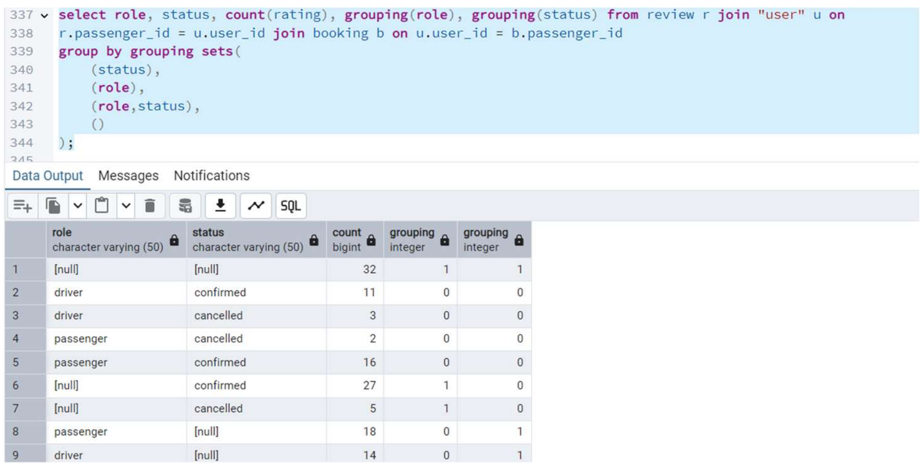
5. Grouping Sets: Review Counts by Role and Status

Purpose: Count reviews at different grouping levels (role, status, and their combinations).

Functionality:

- Uses GROUPING SETS to count reviews for:
  - Each status.
  - Each role.
  - Combinations of role and status.
  - A grand total.
- Includes GROUPING() flags to identify aggregation levels.

Expected Result: A table with review counts at various aggregation levels, including subtotals and a grand total.



```sql
select role, status, count(rating), grouping(role), grouping(status) from review r join "user" u on
r.passenger_id = u.user_id join booking b on u.user_id = b.passenger_id
group by grouping sets(
    (status),
    (role),
    (role,status),
    ()
);
```

	role character varying (50)	status character varying (50)	count bigint	grouping integer	grouping integer
1	[null]	[null]	32	1	1
2	driver	confirmed	11	0	0
3	driver	cancelled	3	0	0
4	passenger	cancelled	2	0	0
5	passenger	confirmed	16	0	0
6	[null]	confirmed	27	1	0
7	[null]	cancelled	5	1	0
8	passenger	[null]	18	0	1
9	driver	[null]	14	0	1

select role,status, sum(available_seats), grouping(role),grouping(status),grouping(available_seats) from ride r join booking b on r.ride_id = b.ride_id join "user" u on b.passenger_id = u.user_id group by cube(role,status,available_seats);

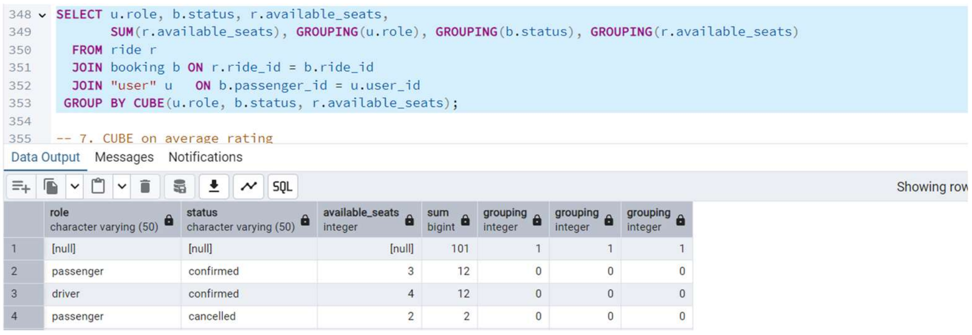
6. Cube: Seat Availability Analysis

Purpose: Analyze seat availability across all combinations of role, status, and seat count.

Functionality:

- Uses `CUBE` to compute sums of `available_seats` for all possible groupings of `role`, `status`, and `available_seats`.
- Includes `GROUPING()` flags to distinguish aggregation levels.

Expected Result: A comprehensive table with seat totals for every combination of role, status, and seat count, including partial and grand totals.


```sql
SELECT u.role, b.status, r.available_seats,
       SUM(r.available_seats), GROUPING(u.role), GROUPING(b.status), GROUPING(r.available_seats)
  FROM ride r
  JOIN booking b ON r.ride_id = b.ride_id
  JOIN "user" u   ON b.passenger_id = u.user_id
 GROUP BY CUBE(u.role, b.status, r.available_seats);

-- 7. CUBE on average rating
```

	role character varying (50)	status character varying (50)	available_seats integer	sum bigint	grouping integer	grouping integer	grouping integer
1	[null]	[null]	[null]	101	1	1	1
2	passenger	confirmed	3	12	0	0	0
3	driver	confirmed	4	12	0	0	0
4	passenger	cancelled	2	2	0	0	0

select role,status,available_seats,avg(rating),grouping(role), grouping(status), grouping(available_seats)

from review natural join ride natural join

booking b join "user" u on b.passenger_id = u.user_id

group by cube(role,status,available_seats);

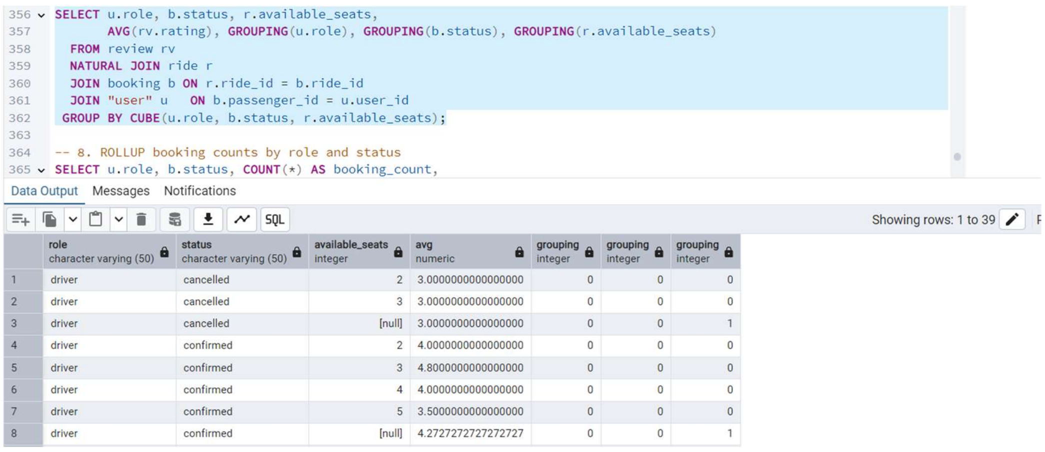
7. Cube: Average Rating Analysis

Purpose: Compute average ratings for all combinations of role, status, and seat availability.

Functionality:

- Uses CUBE to calculate avg(rating) for every grouping of role, status, and available_seats.
- Includes GROUPING() flags to mark aggregated rows.

Expected Result: A table showing average ratings across all possible grouping combinations, with subtotals and grand totals.



```
SELECT u.role, b.status, r.available_seats,
       AVG(rv.rating), GROUPING(u.role), GROUPING(b.status), GROUPING(r.available_seats)
  FROM review rv
  NATURAL JOIN ride r
  JOIN booking b ON r.ride_id = b.ride_id
  JOIN "user" u   ON b.passenger_id = u.user_id
 GROUP BY CUBE(u.role, b.status, r.available_seats);

-- 8. ROLLUP booking counts by role and status
SELECT u.role, b.status, COUNT(*) AS booking_count,
```

Showing rows: 1 to 39

	role character varying (50)	status character varying (50)	available_seats integer	avg numeric	grouping integer	grouping integer	grouping integer
1	driver	cancelled	2	3.0000000000000000	0	0	0
2	driver	cancelled	3	3.0000000000000000	0	0	0
3	driver	cancelled	[null]	3.0000000000000000	0	0	1
4	driver	confirmed	2	4.0000000000000000	0	0	0
5	driver	confirmed	3	4.8000000000000000	0	0	0
6	driver	confirmed	4	4.0000000000000000	0	0	0
7	driver	confirmed	5	3.5000000000000000	0	0	0
8	driver	confirmed	[null]	4.2727272727272727	0	0	1

select role, status, count(*) as booking_count,

grouping(role) , grouping(status)

from booking b join "user" u on b.passenger_id = u.user_id

group by rollup(role, status);

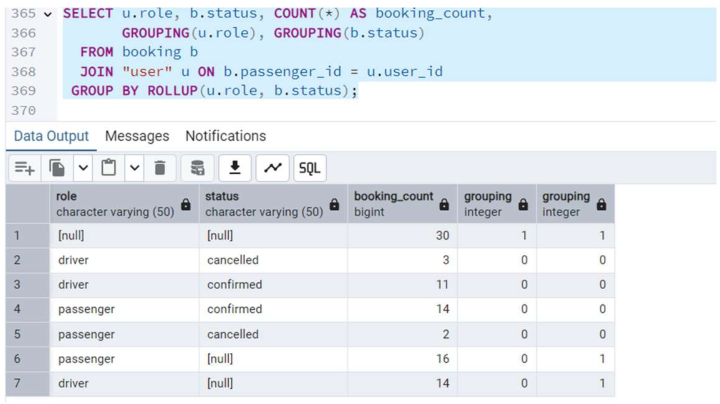
8. Rollup: Booking Counts by Role and Status

Purpose: Count bookings hierarchically by role and status.

Functionality:

- Uses ROLLUP to compute booking counts for:
  - Each role and status combination.
  - Each role (subtotal).
  - A grand total.

- Includes GROUPING() flags to identify aggregation levels.
  Expected Result: A hierarchical table with booking counts by role and status, including subtotals and a grand total.



```
SELECT u.role, b.status, COUNT(*) AS booking_count,
       GROUPING(u.role), GROUPING(b.status)
  FROM booking b
  JOIN "user" u ON b.passenger_id = u.user_id
 GROUP BY ROLLUP(u.role, b.status);
```

Data Output Messages Notifications

	role character varying (50)	status character varying (50)	booking_count bigint	grouping integer	grouping integer
1	[null]	[null]	30	1	1
2	driver	cancelled	3	0	0
3	driver	confirmed	11	0	0
4	passenger	confirmed	14	0	0
5	passenger	cancelled	2	0	0
6	passenger	[null]	16	0	1
7	driver	[null]	14	0	1

select b.passenger_id,count(p.payment_id) as total_payments,

sum(p.amount) as total_amount from payment p

join booking b on p.booking_id = b.booking_id

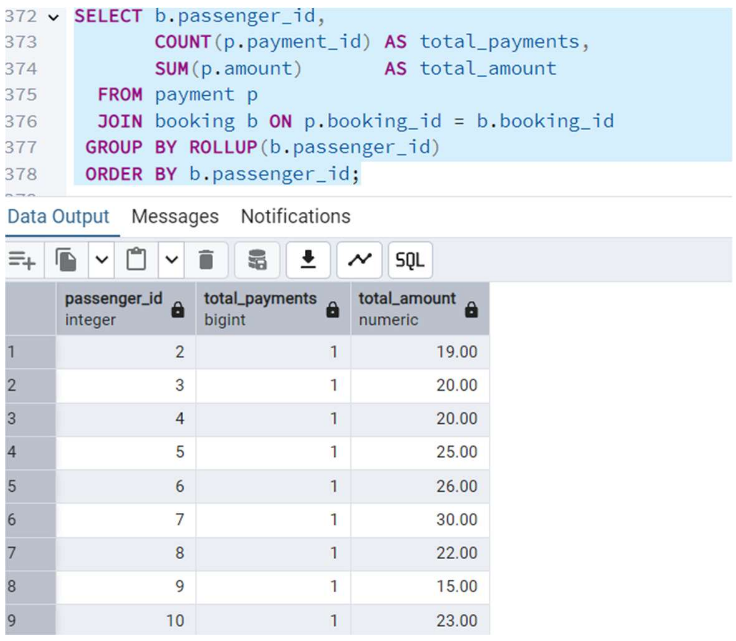
group by rollup (b.passenger_id)

order by b.passenger_id;

9. Rollup: Payment Summary by Passenger

Purpose: Summarize payments by passenger, including subtotals and a grand total.
Functionality:

- Uses ROLLUP to compute:
  - Total payments (COUNT) and amounts (SUM) for each passenger_id.
  - A grand total for all passengers.
- Orders results by passenger_id.
  Expected Result: A table with payment summaries per passenger and a final row for the overall total.



```
SELECT b.passenger_id,
       COUNT(p.payment_id) AS total_payments,
       SUM(p.amount)       AS total_amount
  FROM payment p
  JOIN booking b ON p.booking_id = b.booking_id
 GROUP BY ROLLUP(b.passenger_id)
 ORDER BY b.passenger_id;
```

Data Output Messages Notifications

	passenger_id integer	total_payments bigint	total_amount numeric
1	2	1	19.00
2	3	1	20.00
3	4	1	20.00
4	5	1	25.00
5	6	1	26.00
6	7	1	30.00
7	8	1	22.00
8	9	1	15.00
9	10	1	23.00

select origin, available_seats from ride

where origin like 'b%' intersect select origin, available_seats

from ride where available_seats <5;

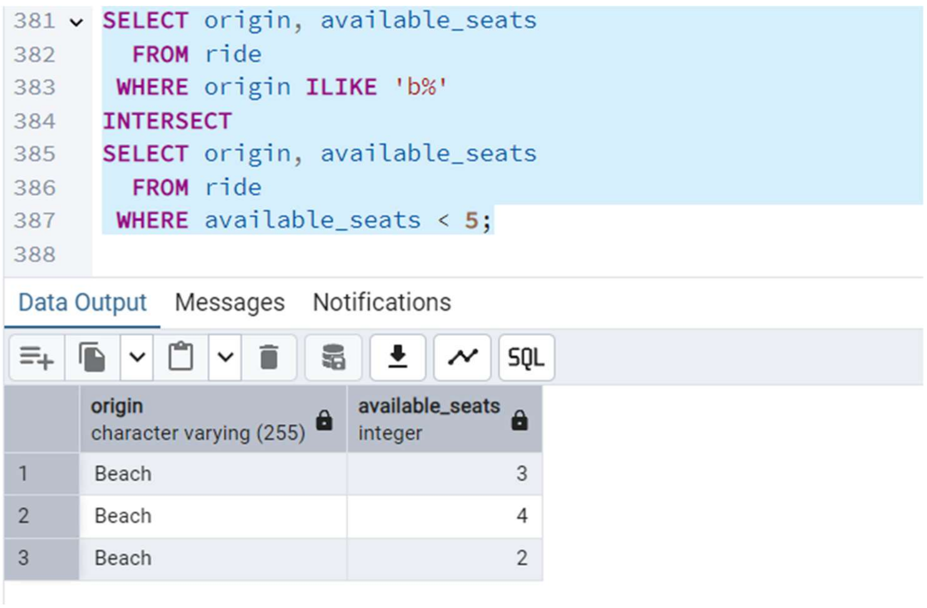
10. Intersect: Rides Starting with 'B' and Fewer than 5 Seats

Purpose: Find rides that meet two conditions: starting location begins with 'b' and having fewer than 5 seats.

Functionality:

- Uses INTERSECT to combine two queries:
  - Rides where origin starts with 'B'.
  - Rides with available_seats < 5.

Expected Result: A table listing rides that satisfy both conditions (origin and seat limit).



```sql
SELECT origin, available_seats
  FROM ride
 WHERE origin ILIKE 'b%'
INTERSECT
SELECT origin, available_seats
  FROM ride
 WHERE available_seats < 5;
```

Data Output   Messages   Notifications

	origin character varying (255)	available_seats integer
1	Beach	3
2	Beach	4
3	Beach	2