UBER RIDES SQL REPORT (WITH QUERIES AND EXPLANATIONS)

1. BASIC DATA CHECKS

```
USE uber_ride;

SELECT COUNT(*)
FROM ncr_ride_bookings;

SELECT *
FROM ncr_ride_bookings
LIMIT 10;

DESCRIBE ncr_ride_bookings;
```

2. CREATING A NEW TABLE WITH MODIFICATIONS (KEEPING THE RAW TABLE AS IT IS)

```
CREATE TABLE uber_bookings AS
SELECT
STR_TO_DATE(`Date`, '%Y-%m-%d') AS ride_date,
STR_TO_DATE(`Time`, '%H:%i:%s') AS ride_time,
```

- This converts the raw date/time strings into proper DATE and TIME formats.

```
REPLACE (`Booking ID`, '"', '') AS booking_id,
`Booking Status` AS booking_status,
REPLACE (`Customer ID`, '"', '') AS customer_id,
```

- Some IDs had double quotes in them → we removed them.

```
`Vehicle Type` AS vehicle_type,
`Pickup Location` AS pickup_location,
`Drop Location` AS drop_location,
```

- These are already text and don't need cleaning.

For all numeric fields (avg_vtat, ride_distance, ratings, etc.):

```
NULLIF(`Avg VTAT`, '') AS avg_vtat,
NULLIF(`Avg CTAT`, '') AS avg_ctat,
```

```
NULLIF(`Cancelled Rides by Customer`, '') AS cancelled_by_customer,
`Reason for cancelling by Customer` AS cust_cancel_reason,

NULLIF(`Cancelled Rides by Driver`, '') AS cancelled_by_driver,
`Driver Cancellation Reason` AS driver_cancel_reason,

NULLIF(`Incomplete Rides`, '') AS incomplete_rides,
`Incomplete Rides Reason` AS incomplete_reason,

NULLIF(`Booking Value`, '') AS booking_value,
NULLIF(`Ride Distance`, '') AS ride_distance,

NULLIF(`Driver Ratings`, '') AS driver_rating,
NULLIF(`Customer Rating`, '') AS customer_rating,
```

This means:

"If the value is an empty string, convert it to NULL."

This avoids problems later when converting to numeric types.

```
`Payment Method` AS payment_method
FROM ncr_ride_bookings;
```

3. DATA CHECKING FROM THE NEW TABLE

```
SELECT *
FROM uber_bookings
LIMIT 10;

SELECT COUNT(*) AS total_rides
FROM uber_bookings;
```

4. CHECKING MISSING VALUES

```
    SELECT
```

```
    SUM(ride_date IS NULL) AS missing_ride_date,
    SUM(ride_time IS NULL) AS missing_ride_time,
    SUM(booking_id IS NULL) AS missing_booking_id,
    SUM(booking_status IS NULL) AS missing_booking_status,
    SUM(customer_id IS NULL) AS missing_customer_id,
    SUM(vehicle_type IS NULL) AS missing_vehicle_type,
    SUM(pickup_location IS NULL) AS missing_pickup_location,
    SUM(drop_location IS NULL) AS missing_drop_location,
    SUM(avg_vtat IS NULL) AS missing_avg_vtat,
    SUM(avg_ctat IS NULL) AS missing_avg_ctat,
    SUM(cancelled_by_customer IS NULL) AS missing_cancelled_by_customer,
    SUM(cust_cancel_reason IS NULL) AS missing_cust_cancel_reason,
    SUM(cancelled_by_driver IS NULL) AS missing_cancelled_by_driver,
    SUM(driver_cancel_reason IS NULL) AS missing_driver_cancel_reason,S
    SUM(incomplete_rides IS NULL) AS missing_incomplete_rides,
    SUM(incomplete_reason IS NULL) AS missing_incomplete_reason,
    SUM(booking_value IS NULL) AS missing_booking_value,
    SUM(ride_distance IS NULL) AS missing_ride_distance,
    SUM(driver_rating IS NULL) AS missing_driver_rating,
    SUM(customer_rating IS NULL) AS missing_customer_rating,
    SUM(payment_method IS NULL) AS missing_payment_method
FROM uber_bookings;
```

5. CREATING 4 SEGMENTS AND CALCULATING THEIR COUNTS

A) COMPLETED RIDES

```
SELECT *

FROM uber_bookings

WHERE booking_value IS NOT NULL;


SELECT COUNT(*)

FROM uber_bookings

WHERE booking_value IS NOT NULL;
```

- A completed ride always has a booking_value (fare).
- So, checking booking_value IS NOT NULL gives all rides that were completed.
- First query shows the rows, second query counts them.

B) NO DRIVER FOUND

```
SELECT *

FROM uber_bookings

WHERE booking_status = 'No Driver Found';


SELECT COUNT(*) AS no_driver_found
```

```
FROM uber_bookings
WHERE booking_status = 'No Driver Found';
```

- These are rides where no driver accepted the request.
- First query lists all such rides.
- Second query gives the total count.

C) CANCELLED BY CUSTOMER

```
SELECT *
FROM uber_bookings
WHERE cancelled_by_customer = 1;

SELECT COUNT(*) AS cancelled_by_customer
FROM uber_bookings
WHERE cancelled_by_customer = 1;
```

- cancelled_by_customer is a flag (1 = Yes).
- These queries show/count all rides where the customer cancelled the trip.

D) CANCELLED BY DRIVER

```
SELECT *
FROM uber_bookings
WHERE cancelled_by_driver = 1;

SELECT COUNT(*) AS cancelled_by_driver
FROM uber_bookings
WHERE cancelled_by_driver = 1;
```

- cancelled_by_driver = 1 means the driver canceled the ride.
- First query shows details, second counts them.

E) INCOMPLETE RIDES

```
SELECT *
FROM uber_bookings
WHERE incomplete_rides = 1;

SELECT COUNT(*) AS incomplete_rides
```

```
FROM uber_bookings
WHERE incomplete_rides = 1;
```

- Incomplete rides = trip started but did not finish.
- These queries identify all such rides and count them.

- These queries simply filter the dataset by different outcome types—Completed, Cancelled, No Driver Found, and Incomplete—to both view the rows and count how many occurred.

6. TRIMMING LEADING/TRAILING SPACES FROM TEXT COLUMNS AND CONVERTING EMPTY STRINGS TO NULL

```
UPDATE uber_bookings
SET
  booking_status = NULLIF(TRIM(booking_status), ''),
  vehicle_type = NULLIF(TRIM(vehicle_type), ''),
  pickup_location = NULLIF(TRIM(pickup_location), ''),
  drop_location = NULLIF(TRIM(drop_location), ''),
  cust_cancel_reason = NULLIF(TRIM(cust_cancel_reason), ''),
  driver_cancel_reason = NULLIF(TRIM(driver_cancel_reason), ''),
  incomplete_reason = NULLIF(TRIM(incomplete_reason), ''),
  payment_method = NULLIF(TRIM(payment_method), '');
```

❖ What TRIM does

Removes unwanted spaces like:

- " UPI " → "UPI"
- " Auto " → "Auto"

❖ What NULLIF does

If the value becomes '' (empty string) → convert to NULL.

❖ Why wrapped inside a transaction?

To make sure:

- All text fields are cleaned safely
- No partial updates happen

7. QUICK CHECKS (PREVIEW 10 CLEANED TEXT ROWS)

```
SELECT ride_date, ride_time, booking_id, booking_status, vehicle_type, pickup_location, drop_location, cust_cancel_reason, driver_cancel_reason, incomplete_reason, payment_method
FROM uber_bookings
LIMIT 10;
```

8. SHOW NULL COUNTS FOR THOSE TEXT COLUMNS TO CONFIRM CHANGES

```
SELECT
    SUM(booking_status IS NULL) AS null_booking_status,
    SUM(vehicle_type IS NULL) AS null_vehicle_type,
    SUM(pickup_location IS NULL) AS null_pickup_location,
    SUM(drop_location IS NULL) AS null_drop_location,
    SUM(cust_cancel_reason IS NULL) AS null_cust_cancel_reason,
    SUM(driver_cancel_reason IS NULL) AS null_driver_cancel_reason,
    SUM(incomplete_reason IS NULL) AS null_incomplete_reason,
    SUM(payment_method IS NULL) AS null_payment_method
FROM uber_bookings;
```

9. TEXT STANDARDIZATION

A) PAYMENT METHOD

```
UPDATE uber_bookings
SET payment_method = CONCAT(UCASE(LEFT(payment_method,1)),SUBSTRING(payment_method,2))
WHERE payment_method IS NOT NULL;

SELECT DISTINCT payment_method
FROM uber_bookings
ORDER BY payment_method;
```

B) BOOKING STATUS

```
UPDATE uber_bookings
SET booking_status = CONCAT(UCASE(LEFT(booking_status,1)),SUBSTRING(booking_status,2))
```

```
WHERE booking_status IS NOT NULL;

SELECT DISTINCT booking_status
FROM uber_bookings
ORDER BY booking_status;
```

C) VEHICLE TYPE

```
UPDATE uber_bookings
SET vehicle_type = CONCAT(UCASE(LEFT(vehicle_type,1)),SUBSTRING(vehicle_type,2))
WHERE vehicle_type IS NOT NULL;

SELECT DISTINCT vehicle_type
FROM uber_bookings
ORDER BY vehicle_type;
```

10. CLEAN CANCELLATION AND INCOMPLETE FLAGS (REMOVE 0 VALUES)

```
UPDATE uber_bookings
SET cancelled_by_customer = NULL
WHERE cancelled_by_customer = 0;

UPDATE uber_bookings
SET cancelled_by_driver = NULL
WHERE cancelled_by_driver = 0;

UPDATE uber_bookings
SET incomplete_rides = NULL
WHERE incomplete_rides = 0;

SELECT
   SUM(cancelled_by_customer = 1) AS cust_cancel,
   SUM(cancelled_by_driver = 1) AS driver_cancel,
   SUM(incomplete_rides = 1) AS incomplete
FROM uber_bookings;
```

11. ADDING ride_type COLUMN AND POPULATE THE COLUMN

```
ALTER TABLE uber_bookings
ADD COLUMN ride_type VARCHAR(30);

UPDATE uber_bookings
SET ride_type =
        CASE
                WHEN booking_status = 'Completed' THEN 'Completed'
                WHEN booking_status = 'No Driver Found' THEN 'No Driver Found'
                WHEN incomplete_rides = 1 THEN 'Incomplete'
                 WHEN cancelled_by_customer = 1 THEN 'Cancelled by Customer'
                WHEN cancelled_by_driver = 1 THEN 'Cancelled by Driver'
        ELSE 'Other'
        END;
```

This takes the messy, inconsistent raw data and converts it into a clean, standardized ride outcome label.

I created ride_type to combine all ride outcomes into 5 clean, correct categories so EDA becomes accurate, easy, and consistent.

```
SELECT DISTINCT ride_type
FROM uber_bookings;
```

12. ADDING TIME-BASED FEATURES

A) ADDING "ride_hour"

```
ALTER TABLE uber_bookings
ADD COLUMN ride_hour INT;

UPDATE uber_bookings
SET ride_hour = HOUR(ride_time);
```

B) ADDING "ride_weekday"

```
ALTER TABLE uber_bookings
ADD COLUMN ride_weekday INT;

UPDATE uber_bookings
SET ride_weekday = WEEKDAY(ride_date);
```

C) ADDING "ride_month"

```
ALTER TABLE uber_bookings
ADD COLUMN ride_month INT;

UPDATE uber_bookings
SET ride_month = MONTH(ride_date);

SELECT ride_date, ride_time, ride_hour, ride_weekday, ride_month
FROM uber_bookings
LIMIT 10;
```

- Adding these features make time-pattern analysis possible.
- Ride-hailing data always depends heavily on *time*, so if I don't extract these fields, you can't answer any hour-wise, weekday-wise, or month-wise insights during EDA.

- **ride_hour** → Understand *hourly demand* & *peak times*.
- **ride_weekday** → Understand *weekday behaviour* & *travel patterns*.
- **ride_month** → Understand *long-term trends* & *seasonality*.

13. NUMERIC CLEANING

A) CONVERTING booking_value TO PROPER NUMERIC

```
UPDATE uber_bookings
SET booking_value = NULLIF(booking_value, '');
```

B) CONVERTING ride_distance TO NUMERIC

```
UPDATE uber_bookings
SET ride_distance = NULLIF(ride_distance, '');
```

C) CONVERTING RATINGS TO NUMERIC

```
UPDATE uber_bookings
SET driver_rating = NULLIF(driver_rating, ''),
customer_rating = NULLIF(customer_rating, '');

SELECT booking_value, ride_distance, driver_rating, customer_rating
FROM uber_bookings
LIMIT 10;
```

14. LAST CHECK

```
DESCRIBE uber_bookings;
```

Here I noticed something important:

Almost ALL numeric columns were still stored as:

- LONGTEXT
- TEXT
- VARCHAR

Meaning:

- numbers were being stored as *strings.*
- MySQL did NOT automatically detect numeric datatypes.
- calculations (SUM, AVG, MIN, MAX) would be wrong or blocked.
- Python would read them as object type (string), causing errors.
- plots and statistics in EDA would be inaccurate.

So, I had to convert everything manually to the correct datatype.

A) CONVERTING NUMERIC METRICS

```
ALTER TABLE uber_bookings
MODIFY avg_vtat DECIMAL(10,2);

ALTER TABLE uber_bookings
MODIFY avg_ctat DECIMAL(10,2);

ALTER TABLE uber_bookings
```

```
MODIFY booking_value DECIMAL(10,2);


ALTER TABLE uber_bookings
MODIFY ride_distance DECIMAL(10,2);


ALTER TABLE uber_bookings
MODIFY driver_rating DECIMAL(3,2);


ALTER TABLE uber_bookings
MODIFY customer_rating DECIMAL(3,2);
```

So, MySQL and Python treat them as numbers, not text.
This allows:

- numeric aggregation
- correct EDA
- meaningful visualization
- correct sorting
- correct filtering

B) CONVERTING FLAG COLUMNS TO TNIYINT (0-1)

```
ALTER TABLE uber_bookings
MODIFY cancelled_by_customer TINYINT;


ALTER TABLE uber_bookings
MODIFY cancelled_by_driver TINYINT;


ALTER TABLE uber_bookings
MODIFY incomplete_rides TINYINT;
```

This:

- Saves storage
- Makes logic checks faster
- Clean binary representation

C) CONVERTING booking_id AND customer_id TO VARCHAR(15)

```
ALTER TABLE uber_bookings
MODIFY booking_id VARCHAR(15);

ALTER TABLE uber_bookings
MODIFY customer_id VARCHAR(15);
```

IDs should NEVER be longtext.

IDs are:

- fixed length
- textual, not numeric
- stable identifiers

So, converting them will:

- faster indexing
- less memory
- cleaner schema
- prevents accidental numeric conversion
- makes queries efficient

SUMMARY

When I ran DESCRIBE uber_bookings, I saw:

- ❖ All numbers stored as TEXT
- ❖ All flags stored as TEXT
- ❖ All IDs stored as LONGTEXT

This is bad practice.

So, I fixed:

- numbers → DECIMAL
- flags → TINYINT
- IDs → VARCHAR

This gave a proper, professional-quality database table that Python could analyze correctly.

```
DESCRIBE uber_bookings;
```