## 1. Selecting the Right Database

```
USE portfolio_projects;
```

- Selects the database where the project will be executed.
- Make sure we are working in the correct database to avoid accidentally modifying the wrong data.

---

## 2. Creating a Staging Table

```
CREATE TABLE transactions_staging LIKE transactions;
```

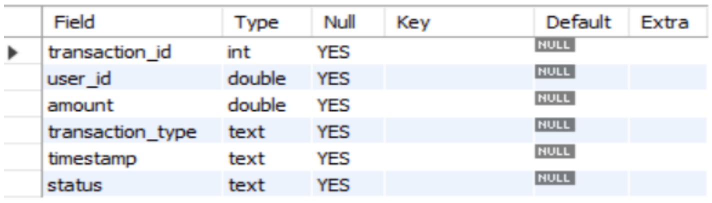
- Creates a staging table named (transactions_staging) with the same structure as the original transactions table with the help of LIKE function it copies the structure (columns, data types etc)
- Staging tables are used to clean and transform data without affecting the original dataset.

---

## 3. Checking the Structure of the Staging Table

```
DESC transactions_staging;
```

- Displays the structure of the transactions_staging table. The DESC (or DESCRIBE) command lists all columns with their data types and constraints.



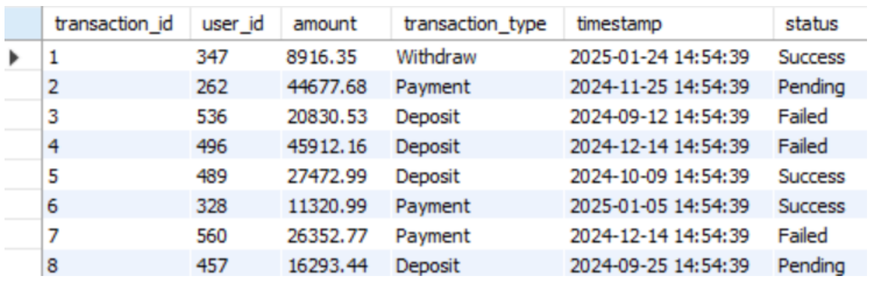
| Field | Type | Null | Key | Default | Extra |
|---|---|---|---|---|---|
| transaction_id | int | YES | | NULL | |
| user_id | double | YES | | NULL | |
| amount | double | YES | | NULL | |
| transaction_type | text | YES | | NULL | |
| timestamp | text | YES | | NULL | |
| status | text | YES | | NULL | |

---

## 4. Inserting Data into the Staging Table

```
INSERT INTO transactions_staging
SELECT * FROM transactions;
```

- **This query** copies all data from the transactions table into the transactions_staging table.
- The INSERT INTO … SELECT statement selects all rows from the original table and inserts them into the staging table.



| transaction_id | user_id | amount | transaction_type | timestamp | status |
|---|---|---|---|---|---|
| 1 | 347 | 8916.35 | Withdraw | 2025-01-24 14:54:39 | Success |
| 2 | 262 | 44677.68 | Payment | 2024-11-25 14:54:39 | Pending |
| 3 | 536 | 20830.53 | Deposit | 2024-09-12 14:54:39 | Failed |
| 4 | 496 | 45912.16 | Deposit | 2024-12-14 14:54:39 | Failed |
| 5 | 489 | 27472.99 | Deposit | 2024-10-09 14:54:39 | Success |
| 6 | 328 | 11320.99 | Payment | 2025-01-05 14:54:39 | Success |
| 7 | 560 | 26352.77 | Payment | 2024-12-14 14:54:39 | Failed |
| 8 | 457 | 16293.44 | Deposit | 2024-09-25 14:54:39 | Pending |

---

### 5. Verifying Data Consistency

```
SELECT *
FROM transactions AS t1
LEFT JOIN transactions_staging AS t2
ON t1.transaction_id = t2.transaction_id
WHERE t2.transaction_id IS NULL;
```

- Checks if any rows in the original transactions table are missing in the transactions_staging table.
- Ensuring no data was lost during the copying process.

---

### 6. Attempting to Add a Primary Key

```
ALTER TABLE transactions_staging
MODIFY transaction_id INT NOT NULL,
ADD PRIMARY KEY(transaction_id);
```

- Adding a primary key constraint to the transaction_id column.
- The MODIFY clause ensures the column is not nullable, and the ADD PRIMARY KEY clause sets it as the primary key.
- This query failed at first because of duplicates values in transaction_id column so first we have to remove those duplicates values before applying primary key constraint

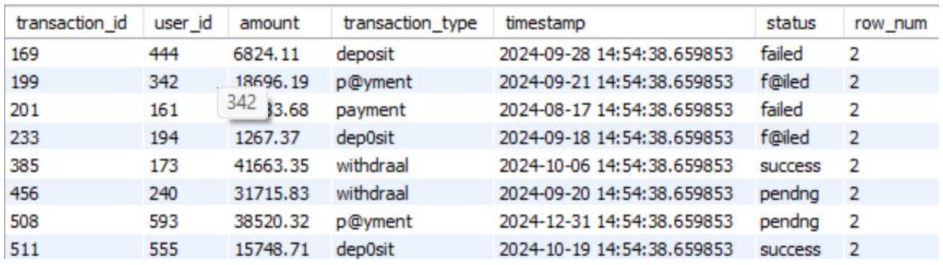
---

### 7. Checking for Duplicate Rows

```
WITH duplicate_rows AS (
  SELECT *,
    ROW_NUMBER() OVER(PARTITION BY transaction_id, user_id, amount, transaction_type,
timestamp, status ORDER BY transaction_id) AS row_num
```

```
  FROM transactions_staging
)
SELECT * FROM duplicate_rows
WHERE row_num > 1;
```

- Cheching for duplicate rows in the transactions_staging table using CTEs.
- The ROW_NUMBER() window function assigns a unique number to each row within a group of duplicates. Rows with row_num > 1 are duplicates.



| transaction_id | user_id | amount | transaction_type | timestamp | status | row_num |
|---|---|---|---|---|---|---|
| 169 | 444 | 6824.11 | deposit | 2024-09-28 14:54:38.659853 | failed | 2 |
| 199 | 342 | 18696.19 | p@yment | 2024-09-21 14:54:38.659853 | f@iled | 2 |
| 201 | 161 | 342 3.68 | payment | 2024-08-17 14:54:38.659853 | failed | 2 |
| 233 | 194 | 1267.37 | dep0sit | 2024-09-18 14:54:38.659853 | f@iled | 2 |
| 385 | 173 | 41663.35 | withdraal | 2024-10-06 14:54:38.659853 | success | 2 |
| 456 | 240 | 31715.83 | withdraal | 2024-09-20 14:54:38.659853 | pendng | 2 |
| 508 | 593 | 38520.32 | p@yment | 2024-12-31 14:54:38.659853 | pendng | 2 |
| 511 | 555 | 15748.71 | dep0sit | 2024-10-19 14:54:38.659853 | success | 2 |

---

## 8. Adding a temporary id column

```
ALTER TABLE transactions_staging
ADD COLUMN temp_id INT AUTO_INCREMENT PRIMARY KEY;
```

- Adds a temporary unique identifier (temp_id) to help delete duplicate rows.
- The AUTO_INCREMENT property ensures each row gets a unique value, and the PRIMARY KEY constraint enforces uniqueness.

---

## 9. Deleting Duplicate Rows

```
DELETE t1
FROM transactions_staging AS t1
JOIN transactions_staging AS t2
  ON t1.transaction_id = t2.transaction_id
  AND t1.user_id = t2.user_id
  AND t1.amount = t2.amount
  AND t1.transaction_type = t2.transaction_type
  AND t1.timestamp = t2.timestamp
  AND t1.status = t2.status
```

```
WHERE t1.temp_id > t2.temp_id;
```

- Deletes duplicate rows from the transactions_staging table.
- The query uses a self-join to compare rows and deletes the one with the higher temp_id (indicating it's a duplicate).

---

## 10. Removing the temporary id column

```
ALTER TABLE transactions_staging
DROP COLUMN temp_id;
```

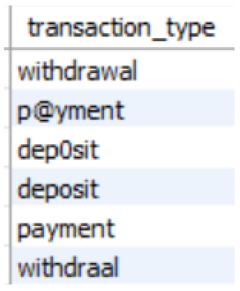
- Removes the temporary identifier column (temp_id) after duplicates are deleted.

---

## 11. Cleaning the transaction_type Column

```
SELECT DISTINCT transaction_type FROM transactions_staging;
```

- Identifies inconsistent values in the transaction_type column.
- The DISTINCT keyword lists all unique values in the column.



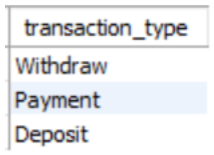
| transaction_type |
|---|
| withdrawal |
| p@yment |
| dep0sit |
| deposit |
| payment |
| withdraal |

### Standardizing transaction_type Values

```
UPDATE transactions_staging
SET transaction_type = CASE
  WHEN transaction_type = 'withdrawal' OR transaction_type = 'withdraal' THEN 'Withdraw'
  WHEN transaction_type = 'p@yment' THEN 'Payment'
  WHEN transaction_type = 'dep0sit' THEN 'Deposit'
  ELSE transaction_type
END;
```

- Standardizes inconsistent values in the transaction_type column.

- The CASE statement converts inconsistent values to their standardized forms.



| transaction_type |
|---|
| Withdraw |
| Payment |
| Deposit |

**Formats transaction types to start with an uppercase letter.**

SELECT transaction_type,

CONCAT(UCASE(LEFT(transaction_type, 1)), SUBSTRING(transaction_type, 2))

FROM transactions_staging;

**Applies the capitalization formatting to all rows**

UPDATE transactions_staging

SET transaction_type = CONCAT(UCASE(LEFT(transaction_type, 1)), SUBSTRING(transaction_type, 2));

**Checks if any values still need fixing**

SELECT transaction_type FROM transactions_staging WHERE transaction_type != CONCAT(UCASE(LEFT(transaction_type, 1)), SUBSTRING(transaction_type, 2));

---

## 15. Cleaning amount column

**This query identifies the amount range in amoun column**

SELECT MIN(amount), MAX(amount) FROM transactions_staging;

**Checks for incorrect or inconsistent values in the amount column**

SELECT DISTINCT amount FROM transactions_staging ORDER BY 1;

---

## 16. Cleaning the status Column

**Identifies inconsistent values in the status column.**

**SELECT DISTINCT `status` FROM transactions_staging;**

**Checks to see if inconsistent data is formatted correctly before updating**

```
SELECT DISTINCT `status`,
        CASE
                WHEN `status` = 'success' OR `status` = 'succes' THEN 'Success'
                WHEN `status` = 'pending' OR `status` = 'pendng' THEN 'Pending'
    WHEN `status` = 'f@iled' OR `status` = 'failed' THEN 'Failed'
    ELSE `status`
    END AS formatted_status
FROM transactions_staging;
```

Updates the standardized values in transaction_staging table

```
UPDATE transactions_staging
SET `status` =
        CASE
                WHEN `status` = 'success' OR `status` = 'succes' THEN 'Success'
                WHEN `status` = 'pending' OR `status` = 'pendng' THEN 'Pending'
    WHEN `status` = 'f@iled' OR `status` = 'failed' THEN 'Failed'
    ELSE `status`
    END;
```

Checking to see all status values are standardized or not.

```
SELECT DISTINCT status FROM transactions_staging;
```

---

## 17. Cleaning the timestamp Column

Checks for any inconsistent date formats or missing values and orders it by timestamp column ascending.

```
SELECT DISTINCT `timestamp` FROM transactions_staging ORDER BY 1;
```

Identifies the earliest and latest transaction timestamps.

```
SELECT MIN(`timestamp`), MAX(`timestamp`) FROM transactions_staging;
```

**Checks the data type of the timestamp column.**

```
DESC transactions_staging;
```

**Checks the timestamp column has formatted to the DATETIME format or not**

```
ALTER TABLE transactions_staging MODIFY `timestamp` DATETIME;
```

**Updates empty values with NULL to ensure correct data conversion**

```
UPDATE transactions_staging SET `timestamp` = NULL WHERE timestamp = ' ';
```

---

**Summary**

This documentation outlines the step-by-step process of cleaning and preparing the transactions table for analysis. Each query serves a specific purpose, from creating a staging table to standardizing values and removing duplicates. By following this process, the data is made consistent, accurate, and ready for further analysis.