SQL MINI PROJECT REPORT: CREDIT CARD TRANSACTION

Submitted By:

Name: Shravani Pawar - 25030422084

Atulya Singh - 25030422029

Mahek Parekh - 25030422145

Batch: BBA Section A

Submitted To:

Dr Pradnya Patil

Assistant Professor

Symbiosis Artificial Intelligence Institute Symbiosis International (Deemed University)

TABLE OF CONTENTS

Introduction and Objectives

1. Overview

The Credit Card Management System is a relational database project designed to simulate the backend infrastructure of a financial institution's credit card division. It provides a robust schema for managing the end-to-end lifecycle of credit card operations, from customer onboarding to transaction processing and dispute resolution.

This project demonstrates a normalized database architecture capable of handling complex financial relationships and ensuring data integrity through strict constraints.

2. Core Objectives

The primary goals of this database design are:

- **Data Integrity:** Enforcing relationships between customers, cards, and merchants using Foreign Keys.
- **Financial Tracking:** Accurately recording outgoing transactions and incoming payments.
- **Lifecycle Management:** Handling post-transaction events such as reward calculation and dispute filing.
- **Analytics Readiness:** Structuring data to allow for easy querying of spending habits, risk analysis, and merchant categories.

3. Usage Scenarios

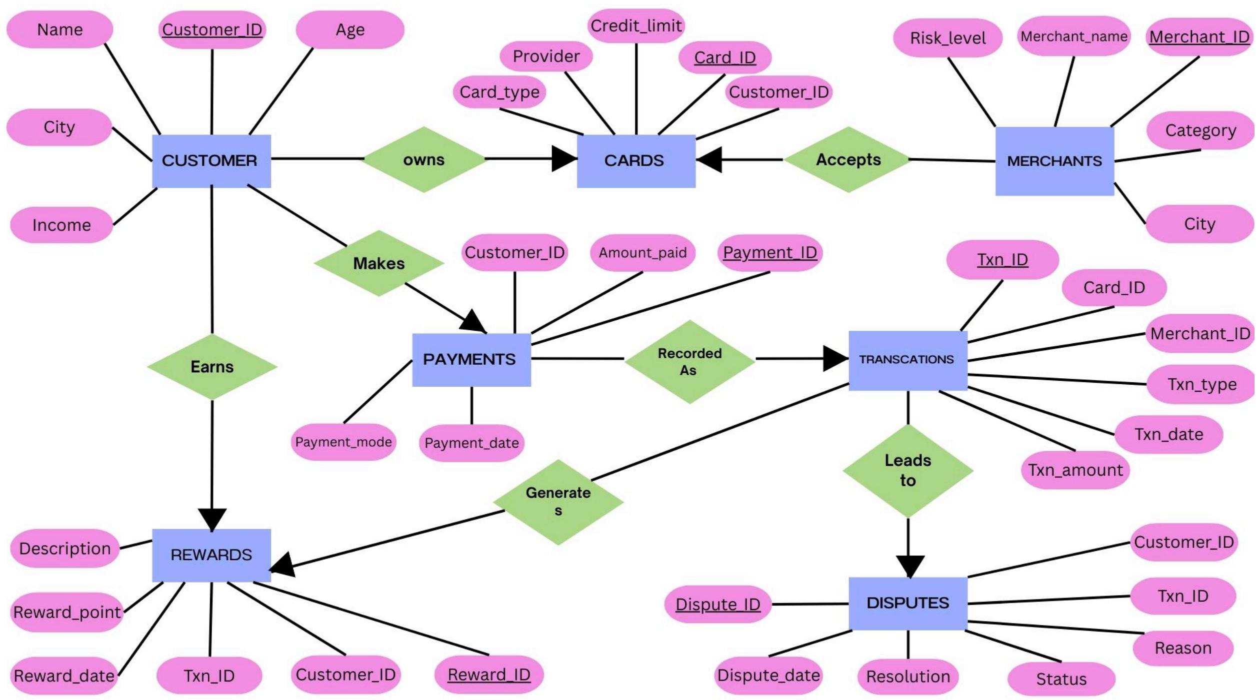
This project is suitable for:

- **Backend Prototyping:** Developing the data layer for a Fintech application.
- **Data Analysis:** Practicing SQL queries for customer segmentation, churn prediction, and fraud analysis.
- **Educational Reference:** Learning database normalization, constraint management, and DDL/DML operations.

Table and Attributes

Sr.No	Table	Attributes	Description
1.	Cards	Card_ID, Customer_ID, Card_Type, Provider, Credit_Limit	Contains information about the credit/debit cards issued to customers.
2.	Disputes	Dispute_ID, Txn_ID, Customer_ID, Reason, Dispute_Date, Status, Resolution	Manages records of contested transactions filed by customers.
3.	Merchants	Merchant_ID, Merchant_Name, Category, Risk_Level, City	Lists various vendors or businesses where transactions can occur.
4.	Customers	Customer_ID, Name, Age, City, Income	Stores personal and demographic details for each unique customer.
5.	Payments	Payment_ID, Customer_ID, Amount_Paid, Payment_mode, Payment_date	Records the payments made by customers and mode of payment.
6.	Rewards	Reward_ID, Customer_ID, Txn_ID, Reward_Points, Reward_Date	Tracks the points or benefits earned by customers from their transactions.
7.	Transactions	Txn_ID, Card_ID, Merchant_ID, Txn_Amount, Txn_Date, Txn_Type	Logs every financial activity involving a card.

Entity Relationship Diagram-



SECTION I: DATA DEFINITION LANGUAGE (DDL)

1.1 Database Initialization (CREATE DATABASE)

The script begins by establishing the container environment.

Code Block:

```
CREATE DATABASE creditcard_system;
USE creditcard_system;
```

Code Structure & Usage Breakdown:

Component	Syntax Element	Description
Command	CREATE DATABASE	Allocates disk space and system catalog entries for a new database.
Identifier	creditcard_system	The unique namespace for this project.
Context	USE	Switches the active session context to this specific database.

1.2 Table Construction (CREATE TABLE):

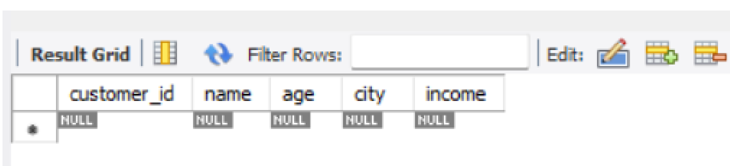
Code Block:

```
CREATE TABLE Customers (
    customer_id INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    age INT,
    city VARCHAR(100),
    income DECIMAL(12,2)
);
```

Code Breakdown & Logic Table

Line Item	Technical Specification	Usage Context
customer_id	INT PRIMARY KEY	**Unique Identifier.** Ensures no two customers have the same ID. Serves as the parent key for joins.
name	VARCHAR(100) NOT NULL	**Mandatory Field.** Stores customer full name. NOT NULL prevents anonymous records.
income	DECIMAL(12,2)	**Financial Precision.** Allows up to 12 digits total, with 2 decimal places for cents.

Output:



1.3 Table Alteration (ALTER TABLE) :

Code Block:
ALTER TABLE NewCustomers
DROP COLUMN age;

Code Breakdown & Logic Table

Alter	DROP COLUMN	Removes the age column permanently.	**Irreversible.** Data in that column is lost.

Output:

customer_id	name	city	income
1	John Doe	New York	75000.00
2	Sarah Smith	Chicago	62000.00
3	Michael Brown	San Francisco	120000.00
4	Emily Johnson	Los Angeles	90000.00
5	David Lee	Houston	110000.00
6	Olivia Davis	Seattle	58000.00
7	Daniel Wilson	Miami	95000.00
8	Sophia Martinez	Boston	70000.00
NULL	NULL	NULL	NULL

1.4 Renaming a table (RENAME):

Code Block- Rename:

```
RENAME TABLE Customers to NewCustomers;
SELECT * FROM NewCustomers;
```

Code Breakdown & Logic Table

Operation	Syntax	Description	Impact
Rename	RENAME TABLE	Changes the entity name from Customers to NewCustomers.	**High.** Breaks any existing queries using the old name.

Output:

customer_id	name	age	city	income
1	John Doe	32	New York	75000.00
2	Sarah Smith	28	Chicago	62000.00
3	Michael Brown	45	San Francisco	120000.00
4	Emily Johnson	36	Los Angeles	90000.00
5	David Lee	50	Houston	110000.00
6	Olivia Davis	26	Seattle	58000.00
7	Daniel Wilson	40	Miami	95000.00
8	Sophia Martinez	30	Boston	70000.00
NULL	NULL	NULL	NULL	NULL

SECTION II: DATA MANIPULATION LANGUAGE (DML) OPERATIONS

This section analyzes how data is injected and modified within the schema.

2.1 Inserting values (INSERT):

Code Block (Excerpt):

```
INSERT INTO Customers VALUES
(1, 'John Doe', 32, 'New York', 75000),
...
(8, 'Sophia Martinez', 30, 'Boston', 70000);
```

Structure, Usage, and Example Analysis

Feature	Explanation	Example Scenario
Structure	Bulk Insertion	Inserting multiple rows in a single command for performance efficiency.
Usage	Seeding	Initializing the database with dummy data for testing.
Integrity	Order Matters	Customers must be inserted *before* Cards because Cards relies on customer_id.

Output:

customer_id	name	age	city	income
1	John Doe	32	New York	75000.00
2	Sarah Smith	28	Chicago	62000.00
3	Michael Brown	45	San Francisco	120000.00
4	Emily Johnson	36	Los Angeles	90000.00
5	David Lee	50	Houston	110000.00
6	Olivia Davis	26	Seattle	58000.00
7	Daniel Wilson	40	Miami	95000.00
8	Sophia Martinez	30	Boston	70000.00

2.2 Record Modification (INSERT AND DELETE):

Code Block- Insert:

```
INSERT INTO Customers(customer_id, name) VALUES (9, 'Atulya Singh');
```

Code Block- Delete:

```
DELETE FROM Customers WHERE customer_id= 9;
```

Code Breakdown & Logic Table

Operation	Syntax	Usage Context
Insertion	INSERT INTO ... VALUES	Adding a new user named 'Atulya Singh'. Note: partial insert (age/city will be NULL).
Deletion	DELETE FROM ... WHERE	**Critical Operation.** Removes the record. The WHERE clause is vital to prevent deleting the whole table.

Output:

Insert:

customer_id	name	age	city	income
1	John Doe	32	New York	75000.00
2	Sarah Smith	28	Chicago	62000.00
3	Michael Brown	45	San Francisco	120000.00
4	Emily Johnson	36	Los Angeles	90000.00
5	David Lee	50	Houston	110000.00
6	Olivia Davis	26	Seattle	58000.00
7	Daniel Wilson	40	Miami	95000.00
8	Sophia Martinez	30	Boston	70000.00
9	Atulya Singh	NULL	NULL	NULL
NULL	NULL	NULL	NULL	NULL

Delete:

customer_id	name	age	city	income
1	John Doe	32	New York	75000.00
2	Sarah Smith	28	Chicago	62000.00
3	Michael Brown	45	San Francisco	120000.00
4	Emily Johnson	36	Los Angeles	90000.00
5	David Lee	50	Houston	110000.00
6	Olivia Davis	26	Seattle	58000.00
7	Daniel Wilson	40	Miami	95000.00
8	Sophia Martinez	30	Boston	70000.00
NULL	NULL	NULL	NULL	NULL

2.3 Schema Evolution - Update

Code Block: Update

UPDATE NewCustomers

SET income= 100000

WHERE customer_id = 1;

SELECT * FROM NewCustomers;

Code Breakdown & Logic Table

Operation	Syntax	Description	Impact
Update	SET income=	Modifies the value of an existing row.	Changes business logic (credit limits might need recalculation).

Output:

customer_id	name	city	income
1	John Doe	New York	75000.00
2	Sarah Smith	Chicago	62000.00
3	Michael Brown	San Francisco	120000.00
4	Emily Johnson	Los Angeles	90000.00
5	David Lee	Houston	110000.00
6	Olivia Davis	Seattle	58000.00
7	Daniel Wilson	Miami	95000.00
8	Sophia Martinez	Boston	70000.00
9	Atulya Singh	NULL	NULL
NULL	NULL	NULL	NULL

SECTION III: DATA QUERY LANGUAGE (DQL) & ANALYTICS

This section analyzes the logic used to extract insights from the data.

3.1 Aggregation & Grouping

Code Block:

```
SELECT
  MONTH(txn_date) AS month,
  SUM(txn_amount) AS total_spent
FROM Transactions
GROUP BY month
ORDER BY month;
```

Structure, Usage, and Example Analysis

Feature	Explanation	Usage
Aggregation	SUM(txn_amount)	Calculates total volume of money spent.
Function	MONTH(txn_date)	Extracts just the integer month (1-12) from the full date.
Grouping	GROUP BY month	Groups transactions by extracted month.

Output:

month	total_spent
1	669.25
2	1705.25
3	1045.49

3.2 Complex Joins (Left, Right, Inner)

The script utilizes all three major join types to demonstrate different data retrieval strategies.

A. Right Join (Payments -> Customers)

Code Block:

```
SELECT p.payment_id, p.amount_paid, c.name
FROM Payments p
RIGHT JOIN NewCustomers c ON p.customer_id = c.customer_id;
```

Output:

payment_id	amount_paid	name
1	250.00	John Doe
2	100.00	Sarah Smith
3	500.00	Michael Brown
4	300.00	Emily Johnson
5	450.00	David Lee
6	120.00	Olivia Davis
7	600.00	Daniel Wilson
8	350.00	Sophia Martinez
NULL	NULL	Atulya Singh

- **Usage:** Returns **ALL** customers, even those who have *not* made a payment.
- **Why?** To identify inactive users or those who haven't paid bills yet.

B. Left Join (Cards -> Transactions)

Code Block:

```
SELECT cd.card_id,  t.txn_amount
FROM Cards cd
LEFT JOIN Transactions t

ON cd.card_id = t.card_id;
```

Output:

card_id	card_type	credit_limit	txn_id	txn_amount
1	Credit	10000.00	1	120.50
1	Credit	10000.00	5	5.25
2	Credit	15000.00	2	540.00
3	Debit	5000.00	3	8.75
4	Credit	20000.00	4	1500.00
5	Credit	12000.00	6	200.00
6	Debit	7000.00	7	999.99
7	Credit	8000.00	8	45.50
8	Credit	15000.00	NULL	NULL

- **Usage:** Returns **ALL** cards, even those that have never been used.
- **Why?** To find "dormant" cards that have been issued but have zero transactions.

C. Inner Join (Customers -> Rewards)

Code Block:

```
SELECT c.name, r.reward_points
FROM NewCustomers c
INNER JOIN Rewards r ON c.customer_id = r.customer_id;
```

Output:

name	reward_points	reward_date
John Doe	12	2025-01-10 10:20:00
Sarah Smith	54	2025-01-12 14:25:00
John Doe	5	2025-01-20 08:15:00
Emily Johnson	20	2025-01-22 13:50:00
Olivia Davis	50	2025-01-25 16:05:00
Daniel Wilson	4	2025-01-28 11:35:00
Michael Brown	1	2025-01-15 09:05:00
David Lee	20	2025-01-22 13:55:00

- **Usage:** Returns **ONLY** customers who have earned rewards.
- **Why?** Strict filtering. Customers with 0 rewards are excluded from this report.

3.3 Subquery Logic

Code Block:

```
SELECT *
FROM Transactions
WHERE txn_amount > (
   SELECT AVG(txn_amount)
   FROM Transactions
);
```

Code Breakdown & Logic Table

Component	Logic	Description

Inner Query	SELECT AVG(txn_amount)	First, calculates the average global transaction size (e.g., $500).
Outer Query	WHERE txn_amount >	Filters the main table to show only transactions *larger* than that calculated average.
Usage	Outlier Analysis	Used to identify "High Value" transactions relative to the norm.

Output:

txn_id	card_id	merchant_id	txn_amount	txn_date	txn_type	status
2	2	2	540.00	2025-01-12 14:20:00	Purchase	Success
4	4	4	1500.00	2025-02-18 17:30:00	Purchase	Flagged
7	6	6	999.99	2025-03-25 16:00:00	Purchase	Success
NULL	NULL	NULL	NULL	NULL	NULL	NULL

SECTION IV: ADVANCED DATABASE OBJECTS

4.1 Database Views

Views are virtual tables used to simplify complex queries or restrict data access.

Code Block 1 (Basic View):

```
CREATE VIEW Newcustomer_basic_view AS
SELECT customer_id, name, city, income
FROM NewCustomers;
```

Output:

customer_id	name	city	income
1	John Doe	New York	75000.00
2	Sarah Smith	Chicago	62000.00
3	Michael Brown	San Francisco	120000.00
4	Emily Johnson	Los Angeles	90000.00
5	David Lee	Houston	110000.00
6	Olivia Davis	Seattle	58000.00
7	Daniel Wilson	Miami	95000.00
8	Sophia Martinez	Boston	70000.00
9	Atulya Singh	NULL	NULL

- **Purpose:** Security. This view hides the internal customer_id or sensitive data if we exclude specific columns (though here it includes them). It simplifies SELECT * FROM NewCustomers queries.

Code Block 2 (Joined View):

```
CREATE VIEW payment_history_view AS
SELECT p.payment_id, c.name, p.amount_paid, p.payment_date
FROM Payments p
JOIN NewCustomers c ON p.customer_id = c.customer_id;
```

Output:

	payment_id	name	amount_paid	payment_date
▶	1	John Doe	250.00	2025-01-11 11:00:00
	2	Sarah Smith	100.00	2025-01-13 12:00:00
	3	Michael Brown	500.00	2025-01-17 16:45:00
	4	Emily Johnson	300.00	2025-01-23 10:30:00
	5	David Lee	450.00	2025-01-24 09:15:00
	6	Olivia Davis	120.00	2025-01-26 14:20:00
	7	Daniel Wilson	600.00	2025-01-29 15:50:00
	8	Sophia Martinez	350.00	2025-01-30 10:40:00

- **Purpose:** Abstraction. Instead of writing the JOIN syntax every time, a data analyst can simply run SELECT * FROM payment_history_view. It pre-packages the join logic.

4.2 Stored Procedures

Stored Procedures are saved SQL code that can be reused with parameters.

Code Block:

```
DELIMITER $$
CREATE PROCEDURE get_customer_transactions(IN cust_id INT)
BEGIN
  SELECT t.txn_id, t.txn_amount, t.txn_date, t.status
  FROM Transactions t
  JOIN Cards cd ON t.card_id = cd.card_id
  WHERE cd.customer_id = cust_id;
END $$
DELIMITER ;
```

Code Breakdown & Logic Table

Component	Syntax	Description
Delimiter	DELIMITER $$	Changes the "end of line" marker so the procedure doesn't prematurely end the script.
Parameter	IN cust_id INT	Accepts an input

		(Customer ID) when called.
Logic	JOIN ... WHERE	Links transactions to cards, then cards to the specific customer ID provided.

Execution:

CALL get_customer_transactions(1);

Output:

	txn_id	txn_amount	txn_date	status
▶	1	120.50	2025-01-10 10:15:00	Success
	5	5.25	2025-02-20 08:10:00	Success
	2	540.00	2025-01-12 14:20:00	Success

- **Result:** This executes the complex join logic specifically for Customer #1 ("John Doe"), returning his specific transaction history.

SECTION V: FRAUD DETECTION AND MONITORING

This section introduces two critical DQL patterns essential for financial risk analysis: identifying unusual high-value transactions and detecting potentially fraudulent velocity (frequency) spikes.

5.1 High-Value Transaction Screening

This query identifies all transactions exceeding a specific high-value threshold (100,000). Such transactions typically require manual review, flagging them for potential fraud or compliance checks.

Code Block:

```
SELECT
    t.txn_id,
    t.txn_amount,
    t.txn_date,
    t.status,
    c.card_id,
    c.customer_id
FROM Transactions t
JOIN Cards c ON t.card_id = c.card_id
WHERE t.txn_amount > 100000
ORDER BY t.txn_amount DESC;
```

Code Breakdown & Logic Table

Component	Syntax	Description
WHERE Clause	t.txn_amount > 100000	Hard-coded filter to instantly isolate records exceeding the acceptable limit.
JOIN Cards	ON t.card_id = c.card_id	Necessary to link the transaction back to the issuing card and the **customer_id** for immediate account action.
ORDER BY	t.txn_amount DESC	Sorts results to show the

		largest, and thus highest-risk, transactions first.

Output:

	txn_id	txn_amount	txn_date	status	card_id	customer_id
▶	4	150000.00	2025-02-18 17:30:00	Flagged	4	3
	5	100005.25	2025-02-20 08:10:00	Flagged	1	1

5.2 Velocity and Frequency Analysis

This query measures the **velocity** of transactions—the number of events occurring within a specific time window (in this case, daily). An unexpected spike in daily_transaction_count can signal a "card testing" attack or compromised credentials.

Code Block:

```
SELECT
   customer_id,
   DATE(txn_date) AS txn_day,
   COUNT(*) AS daily_transaction_count
FROM Transactions
JOIN Cards ON Transactions.card_id = Cards.card_id
GROUP BY Cards.customer_id, DATE(Transactions.txn_date);
```

Code Breakdown & Logic Table

Component	Syntax	Description
DATE	DATE(txn_date)	Temporal Grouping. Extracts only the date component, discarding the time, to group transactions by full day.
COUNT	COUNT(*)	Aggregation. Counts the total number of transactions that fall into the combined customer_id

		and txn_day group.
GROUP BY	GROUP BY Cards.customer_id, DATE(Transactions.txn_date);	Ensures the count is specific to one customer on one specific day.

Output:

customer_id	txn_day	daily_transaction_count
1	2025-01-10	1
1	2025-01-12	1
2	2025-01-15	1
3	2025-02-18	1
1	2025-02-20	1
4	2025-02-22	1
5	2025-03-25	1
6	2025-03-28	1

LEARNING OUTCOMES

This project provided comprehensive hands-on experience across the entire database management lifecycle, yielding mastery in both foundational and advanced SQL concepts.

- **Entity Relationship (ER) Mapping:** Established robust one-to-many relationships using Foreign Key constraints to ensure referential integrity, preventing data anomalies and redundancy.

- **Data Type Selection:** Appropriately utilized specialized data types, such as DECIMAL(12,2) for financial precision and DATETIME for accurate time-series logging, which is critical for financial applications.

- **Complex Join Operations:** Demonstrated strategic use of various join types for targeted analysis:
    - **LEFT JOIN:** Identifying dormant accounts or cards without associated transactions.
    - **RIGHT JOIN:** Ensuring complete customer coverage, regardless of their payment history.
    - **INNER JOIN:** Filtering for specific intersection sets (e.g., customers who have earned rewards).

- **Data Aggregation:** Executed advanced statistical queries using aggregate functions (SUM, AVG) combined with the GROUP BY clause to generate essential business reports, such as monthly total spending volume.

- **Database Views:** Created virtual tables (Newcustomer_basic_view, payment_history_view) to simplify common or complex query logic, providing an abstraction layer for application developers and enhancing security by restricting direct table access.

- **Stored Procedures:** Developed and utilized parameterized stored procedures (get_customer_transactions) to centralize complex, reusable business logic, improving execution performance and minimizing redundant code at the application level.

CONCLUSION

The creditcard_system database script represents a fully functional prototype for a financial transaction system.

1. **Robustness:** It employs strict typing (DECIMAL for money, DATETIME for logs) and Referencing (FOREIGN KEY) to ensure data quality.
2. **Flexibility:** The inclusion of LEFT/RIGHT joins demonstrates an ability to handle incomplete data sets (e.g., customers without payments).
3. **Scalability:** The use of VIEWS and STORED PROCEDURES indicates a design ready for application integration, allowing backend developers to interact with the database via abstract interfaces rather than raw table queries.
4. **Auditability:** Tables like Disputes and Transactions provide a clear paper trail for all financial activities.

This architecture allows for comprehensive reporting on customer behavior, merchant risk profiles, and financial health.