Enhancing E-commerce Efficiency: Database Implementation for Streamlined Transactions

Sasank Dattu Talluri
Computer Science Department
University at Buffalo, The state University of Newyork
Buffalo, Newyork, United States
sasankda@buffalo.edu

Bala Seshagiri Prasad Munagala
Computer Science Department
University at Buffalo, The state University of Newyork
Buffalo, Newyork, United States bmunagal@buffalo.edu

***Abstract*— In the fast-evolving e-commerce world, effective transaction management is critical for businesses to prosper. This project aims to alleviate the constraints of manual approaches like Excel spreadsheets in processing transactional data by building a specialized database solution. The major goal is to streamline transaction management operations and give useful information to stakeholders. Target users include ecommerce managers, customer service teams, financial experts, and marketing analysts who use the database for various objectives. Database administrators are responsible for assuring the database's stability, security, and performance. Overall, this initiative intends to help e-commerce enterprises survive in the digital marketplace by leveraging data-driven insights and improved transaction administration.**

Keywords*— *E-commerce, Transaction management, Database implementation, Database Administration.

I. Problem Statement

In the realm of e-commerce, managing transactions efficiently is crucial for business success. The problem lies in the reliance on manual methods or basic spreadsheets to handle transactional data, leading to potential errors, inefficiencies, and limited scalability. Therefore, the aim of this project is to implement a robust database solution tailored for e-commerce transactions, addressing the following questions:

How can a database streamline transaction management compared to using Excel files?

What are the specific features and functionalities required in the database to support e-commerce transactions effectively?

What are the potential benefits of transitioning from Excel files to a dedicated database for e-commerce transaction management?

II. Background and Significance

The background of this problem stems from the exponential growth of e-commerce and the subsequent surge in transaction volumes. Despite the advancements in technology, many businesses still rely on outdated methods, such as Excel spreadsheets, to manage their transactional data. This reliance poses significant challenges, including errors, inefficiencies, and scalability issues. Moreover, manual processes hinder real-time analysis and decisionmaking, ultimately impacting the overall performance and competitiveness of e-commerce businesses. Hence, implementing a dedicated database solution tailored for transaction management is crucial to address these challenges and unlock the full potential of e-commerce operations.

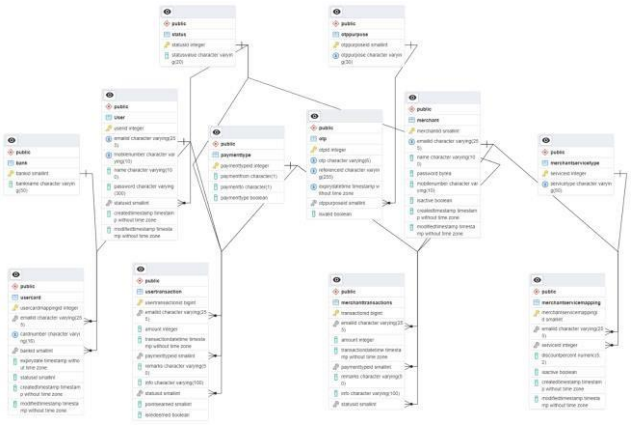
III. Potential Contribution and Importance

The implementation of a dedicated database for ecommerce transactions holds immense potential in revolutionizing how businesses operate in the digital marketplace. By centralizing transactional data, automating processes, and providing real-time insights, the database can significantly enhance operational efficiency, improve decision-making, and elevate the overall customer experience. Furthermore, the database lays the foundation for scalability, enabling businesses to handle growing transaction volumes seamlessly. Ultimately, the project's contribution lies in empowering e-commerce businesses to thrive in a competitive landscape by harnessing the power of data-driven insights and streamlined transaction management.

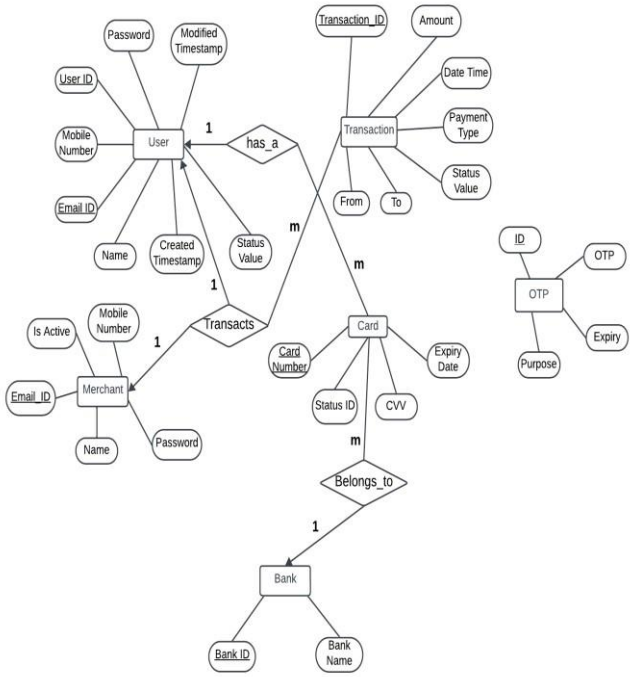
IV. Target Users and Administration

The primary users of the database include e-commerce managers, customer support teams, finance professionals, and marketing analysts. These stakeholders rely on the database for various purposes, such as monitoring transactional activities, resolving customer inquiries, financial reporting, and designing marketing campaigns. In a real-life scenario, a dedicated team of database administrators would administer the database. These administrators would be responsible for database maintenance, security, performance optimization, and troubleshooting technical issues. For instance, an IT professional within the e-commerce company could serve as the database administrator, ensuring the smooth functioning and reliability of the database infrastructure.

V. E/R Digram Crows E/R diagram:



Chen's E/R diagram:



VI. RELATION SCHEMA

Status(StatusId, StatusValue)
StatusId: Unique identifier for a status entry.
StatusValue: Descriptive value representing the status.

StatusId is chosen as the primary key because it uniquely identifies each status entry. It's an auto-incrementing field, ensuring uniqueness and easy retrieval.

User(UserId, EmailId, Mobile Number, Name, Password, StatusId, CreatedTimestamp, ModifiedTimestamp)
FOREIGN KEY REFERENCES Status(StatusId)
UserId: Unique identifier for a user.
EmailId: Unique email address of the user.
MobileNumber: Unique mobile number of the user.
Name: Name of the user.
Password: Password of the user.
StatusId: Foreign key referencing the status of the user.
CreatedTimestamp: Timestamp indicating when the user was created
ModifiedTimestamp: Timestamp indicating when the user was last modified.

UserId is chosen as the primary key because it uniquely identifies each user. It's an auto-incrementing field, ensuring uniqueness and easy retrieval.

Bank(BankId, BankName)
BankId: Unique identifier for a bank.
BankName: Name of the bank.

BankId is chosen as the primary key because it uniquely identifies each bank entry. It's an auto-incrementing field, ensuring uniqueness and easy retrieval.

UserCard(UserCardMappingId, EmailId, CardNumber, BankId, ExpiryDate, StatusId, CreatedTimestamp, ModifiedTimestamp)
fk_UserCard_EmailId REFERENCES User(EmailId)
fk_UserCard_BankId REFERENCES Bank(BankId)
UNIQUE(EmailId, CardNumber)
UserCardMappingId: Unique identifier for a user card mapping.
EmailId: Email address associated with the user.
CardNumber: Card number associated with the user. BankId: Foreign key referencing the bank associated with the card.
ExpiryDate: Expiry date of the card.
StatusId: Status of the card.
CreatedTimestamp: Timestamp indicating when the card mapping was created.
ModifiedTimestamp: Timestamp indicating when the card mapping was last modified.

UserCardMappingId is chosen as the primary key because it uniquely identifies each user card mapping. It's an auto-incrementing field, ensuring uniqueness and easy retrieval.

Merchant(MerchantId, EmailId, Name, Password, MobileNumber, IsActive, CreatedTimestamp, ModifiedTimestamp)
MerchantId: Unique identifier for a merchant.
EmailId: Unique email address of the merchant.
Name: Name of the merchant.
Password: Password of the merchant.
MobileNumber: Mobile number of the merchant.
IsActive: Indicates whether the merchant is active.
CreatedTimestamp: Timestamp indicating when the merchant was created.
ModifiedTimestamp: Timestamp indicating when the merchant was last modified.

MerchantId is chosen as the primary key because it uniquely identifies each merchant. It's an auto-incrementing field, ensuring uniqueness and easy retrieval.

MerchantServiceType(ServiceId, ServiceType)
UNIQUE(ServiceType)
ServiceId: Unique identifier for a service type.
ServiceType: Descriptive value representing the type of service.

MerchantServiceMapping(MerchantServiceMappingId, EmailId, ServiceId, DiscountPercent, IsActive, CreatedTimestamp, ModifiedTimestamp)
CONSTRAINT fk_MerchantSM_EmailId REFERENCES Merchant(EmailId) NOT NULL fk_ServiceId REFERENCES
MerchantServiceType(ServiceId) NOT NULL
UNIQUE (EmailId, ServiceId)
MerchantServiceMappingId: Unique identifier for a merchant service mapping.
EmailId: Email address of the merchant associated with the service.

ServiceId: Foreign key referencing the service type offered by the merchant.
DiscountPercent: Percentage discount offered by the merchant for the service.
IsActive: Indicates whether the service mapping is active.
CreatedTimestamp: Timestamp indicating when the service mapping was created.
ModifiedTimestamp: Timestamp indicating when the service mapping was last modified.

PaymentType(PaymentTypeId, PaymentFrom, PaymentTo, PaymentType)
PaymentTypeId: Unique identifier for a payment type.
PaymentFrom: Indicates the source of the payment.
PaymentTo: Indicates the destination of the payment.
PaymentType: Indicates the type of payment.

UserTransaction(UserTransactionId, EmailId, Amount, TransactionDateTime, PaymentTypeId, Remarks, Info, StatusId, PointsEarned, IsRedeemed)
fk_UserTransaction_EmailId References User(EmailId) NOT NULL
fk_UserT_PaymentTypeId References PaymentType(PaymentTypeId) NOT NULL
fk_UserT_StatusId References Status(StatusId) NOT NULL
UserTransactionId: Unique identifier for a user transaction.
EmailId: Email address of the user associated with the transaction.
Amount: Amount of the transaction.
TransactionDateTime: Timestamp indicating when the transaction occurred.
PaymentTypeId: Foreign key referencing the type of payment used for the transaction.
Remarks: Additional remarks or notes for the transaction.
Info: Additional information about the transaction.
StatusId: Foreign key referencing the status of the transaction.
PointsEarned: Points earned by the user for the transaction.
IsRedeemed: Indicates whether the transaction has been redeemed.

MerchantTransactions(TransactionId, EmailId, Amount, TransactionDateTime, PaymentTypeId, Remarks, Info, StatusId)
fk_MerchantT_EmailId References Merchant(EmailId) NOT NULL
fk_MerchantT_PaymentTypeId References PaymentType(PaymentTypeId) NOT NULL
fk_MerchantT_StatusId References Status(StatusId) NOT NULL
TransactionId: Unique identifier for a merchant transaction.
EmailId: Email address of the merchant associated with the transaction.
Amount: Amount of the transaction.
TransactionDateTime: Timestamp indicating when the transaction occurred.
PaymentTypeId: Foreign key referencing the type of payment used for the transaction.
Remarks: Additional remarks or notes for the transaction.
Info: Additional information about the transaction.
StatusId: Foreign key referencing the status of the transaction.

MerchantServiceMappingId is chosen as the primary key because it uniquely identifies each merchant service mapping. It's an auto-incrementing field, ensuring uniqueness and easy retrieval.

EmailId is relates each service mapping to the corresponding merchant, ensuring that each mapping is associated with a valid merchant.

OTPPurpose((OTPPurposeId, OTPPurpose)
OTPPurposeId: Unique identifier for an OTP purpose.
OTPPurpose: Descriptive value representing the purpose of the OTP.

OTPPurposeId is chosen as the primary key because it uniquely identifies each OTP purpose. It's an autoincrementing field, ensuring uniqueness and easy retrieval.

OTP(OTPId, OTP, ReferenceId, ExpiryDateTime, OTPPurposeId, IsValid) fk_OTPPurposeId REFERENCES OTPPurpose(OTPPurposeId) NOT NULL UNIQUE (OTP, ReferenceId, ExpiryDateTime) OTPId: Unique identifier for an OTP.
OTP: One-time password generated for authentication.
ReferenceId: Reference identifier for the OTP.
ExpiryDateTime: Timestamp indicating when the OTP expires.
OTPPurposeId: Foreign key referencing the purpose of the OTP.
IsValid: Indicates whether the OTP is valid.

OTPId is chosen as the primary key because it uniquely identifies each OTP. It's an auto-incrementing field, ensuring uniqueness and easy retrieval.

Foreign Key-OTPPurposeId relates each OTP to the corresponding OTP purpose, ensuring that each OTP is associated with a valid purpose.

Default Values and Nullable Attributes:

StatusId, UserId, BankId, ServiceId, PaymentTypeId, UserTransactionId, TransactionId, OTPPurposeId, OTPId: Primary key fields with auto-increment and cannot be null.

EmailId: Unique identifiers that cannot be null.

MobileNumber: Nullable attributes that can be set to null if not provided.

Remarks: Nullable attribute that can be set to null if not provided.

ModifiedTimestamp: Nullable attribute indicating the last modification time.

IsRedeemed, IsActive, IsValid: Boolean attributes with default values.

CreatedTimestamp, TransactionDateTime, ExpiryDateTime: Timestamp attributes with default values representing creation time or current time.

DiscountPercent, PointsEarned: Decimal attributes with default values.

Password: Attribute storing sensitive data that cannot be null.

Info: Attribute containing additional information that cannot be null.

StatusValue, ServiceType, PaymentFrom, PaymentTo, OTPPurpose: Descriptive attributes that cannot be null.

CardNumber: Attribute containing sensitive information that cannot be null.

Actions on Foreign Keys:

StatusId in User, UserTransaction, MerchantTransactions: When a status entry is deleted, the corresponding foreign key in these tables should ideally be set to NULL or restricted based on business logic.

EmailId in UserCard, MerchantServiceMapping, UserTransaction, MerchantTransactions: When a user or merchant is deleted, the corresponding entries in dependent tables can be set to NULL, deleted (cascade), or restricted based on business logic.

PostgreSQL queries:

```
CREATE TABLE Status (
    StatusId SERIAL PRIMARY KEY,
    StatusValue VARCHAR(20) NOT NULL
);

CREATE TABLE "User" (
    UserId SERIAL PRIMARY KEY,
    EmailId VARCHAR(255) UNIQUE NOT NULL,
    MobileNumber VARCHAR(10) UNIQUE NOT
NULL,
    Name VARCHAR(100) NOT NULL,
    Password VARCHAR(300) NOT NULL,
    StatusId SMALLINT REFERENCES
Status(StatusId) NOT NULL,
    CreatedTimestamp TIMESTAMP DEFAULT
CURRENT_TIMESTAMP NOT NULL,
    ModifiedTimestamp TIMESTAMP
);

CREATE TABLE Bank (
    BankId SMALLSERIAL PRIMARY KEY,
    BankName VARCHAR(50) NOT NULL
);

CREATE TABLE UserCard (
    UserCardMappingId SERIAL PRIMARY KEY,
    EmailId VARCHAR(255) NOT NULL
REFERENCES "User"(EmailId),
    CardNumber VARCHAR(16) NOT NULL,
    BankId SMALLINT NOT NULL REFERENCES
Bank(BankId),
    ExpiryDate TIMESTAMP NOT NULL,
    StatusId SMALLINT NOT NULL,
    CreatedTimestamp TIMESTAMP DEFAULT
CURRENT_TIMESTAMP NOT NULL,
    ModifiedTimestamp TIMESTAMP,
    CONSTRAINT unq_Email_Card
UNIQUE(EmailId, CardNumber)
);

CREATE TABLE Merchant (
    MerchantId SMALLSERIAL PRIMARY KEY,
    EmailId VARCHAR(255) UNIQUE NOT NULL,
    Name VARCHAR(100) NOT NULL,
    Password BYTEA NOT NULL,
    MobileNumber VARCHAR(10),
    IsActive BOOLEAN DEFAULT TRUE NOT NULL,
    CreatedTimestamp TIMESTAMP DEFAULT
CURRENT_TIMESTAMP NOT NULL,
    ModifiedTimestamp TIMESTAMP
);

CREATE TABLE MerchantServiceType (
    ServiceId SERIAL PRIMARY KEY,
    ServiceType VARCHAR(50) UNIQUE NOT NULL
);

CREATE TABLE MerchantServiceMapping (
    MerchantServiceMappingId SMALLSERIAL
PRIMARY KEY,
    EmailId VARCHAR(255) REFERENCES
Merchant(EmailId) NOT NULL,
    ServiceId INTEGER REFERENCES
MerchantServiceType(ServiceId) NOT NULL,
    DiscountPercent DECIMAL(5,2) DEFAULT 0
NOT NULL,
    IsActive BOOLEAN DEFAULT TRUE NOT NULL,
    CreatedTimestamp TIMESTAMP DEFAULT
CURRENT_TIMESTAMP,
    ModifiedTimestamp TIMESTAMP,
    CONSTRAINT unq_EmailId_ServiceId UNIQUE
(EmailId, ServiceId)
);

CREATE TABLE PaymentType (
    PaymentTypeId SERIAL PRIMARY KEY,
    PaymentFrom CHAR(1) CHECK(PaymentFrom
IN ('B','M','W')) NOT NULL,
    PaymentTo CHAR(1) CHECK(PaymentTo IN
('B','M','W')) NOT NULL,
    PaymentType BOOLEAN NOT NULL
);

CREATE TABLE UserTransaction (
    UserTransactionId BIGSERIAL PRIMARY
KEY,
    EmailId VARCHAR(255) REFERENCES
"User"(EmailId) NOT NULL,
```

```
    Amount MONEY CHECK(Amount!=0) NOT NULL,
    TransactionDateTime TIMESTAMP DEFAULT
CURRENT_TIMESTAMP NOT NULL,
    PaymentTypeId SMALLINT REFERENCES
PaymentType(PaymentTypeId) NOT NULL,
    Remarks VARCHAR(50),
    Info VARCHAR(100) NOT NULL,
    StatusId SMALLINT REFERENCES
Status(StatusId) NOT NULL,
    PointsEarned SMALLINT
CHECK(PointsEarned>=0) DEFAULT 0 NOT NULL,
IsRedeemed BOOLEAN DEFAULT FALSE NOT
NULL
);

CREATE TABLE MerchantTransactions (
    TransactionId BIGSERIAL PRIMARY KEY,
    EmailId VARCHAR(255) REFERENCES
Merchant(EmailId) NOT NULL,
    Amount MONEY CHECK(Amount>0) NOT NULL,
    TransactionDateTime TIMESTAMP DEFAULT
CURRENT_TIMESTAMP NOT NULL,
    PaymentTypeId SMALLINT REFERENCES
PaymentType(PaymentTypeId) NOT NULL,
    Remarks VARCHAR(50),
    Info VARCHAR(100) NOT NULL,
    StatusId SMALLINT REFERENCES
Status(StatusId) NOT NULL
);

CREATE TABLE OTPPurpose (
    OTPPurposeId SMALLSERIAL PRIMARY KEY,
    OTPPurpose VARCHAR(30) UNIQUE NOT NULL
);

CREATE TABLE OTP (
    OTPId SERIAL PRIMARY KEY,
    OTP VARCHAR(6) DEFAULT
CURRENT_TIMESTAMP + INTERVAL '5 minutes'
NOT NULL,
    ReferenceId VARCHAR(255) NOT NULL,
    ExpiryDateTime TIMESTAMP NOT NULL,
    OTPPurposeId SMALLINT REFERENCES
OTPPurpose(OTPPurposeId) NOT NULL,
    IsValid BOOLEAN DEFAULT TRUE NOT NULL,
    CONSTRAINT uq_OTP_ExpiryDateTime UNIQUE
(OTP, ReferenceId, ExpiryDateTime)
);
```

VII. NORMALIZATION (CHECKING FOR BCNF)

User:

UserId → {EmailId, MobileNumber, Name, Password, StatusValue, CreatedTimestamp, ModifiedTimestamp}

EmailId → {UserId}

MobileNumber → {UserId}

Both UserId and EmailId are candidate keys. Hence, User is already in BCNF.

Bank:

BankId → {BankName}BankId is the candidate key. So, Bank is already in BCNF.

UserCard:

{EmailId, CardNumber} →{BankId, ExpiryDate, StatusId, CreatedTimestamp, ModifiedTimestamp}

EmailId →{CardNumber}

CardNumber →{EmailId}

Both {EmailId, CardNumber} and CardNumber are candidate keys. UserCard is in BCNF.

Merchant:

MerchantId →{EmailId, Name, Password, MobileNumber, IsActive, CreatedTimestamp, ModifiedTimestamp}

EmailId →{MerchantId}

MerchantId and EmailId are candidate keys. Merchant is in BCNF.

MerchantServiceMapping:

MerchantServiceMappingId → {EmailId, ServiceType, DiscountPercent, IsActive, CreatedTimestamp, ModifiedTimestamp}

EmailId →{MerchantServiceMappingId}

MerchantServiceMappingId and EmailId are candidate keys. MerchantServiceMapping is in BCNF.

UserTransaction:

UserTransactio→{EmailId, Amount, TransactionDateTime, PaymentFrom, PaymentTo, PaymentType, Remarks, Info, Status, PointsEarned, IsRedeemed}

EmailId →{UserTransactionId}

UserTransactionId and EmailId are candidate keys. UserTransaction is in BCNF.

MerchantTransactions:

TransactionId → {EmailId, Amount, TransactionDateTime, PaymentFrom, PaymentTo, PaymentType, Remarks, Info, Status}

EmailId →{TransactionId}

TransactionId and EmailId are candidate keys. MerchantTransactions is in BCNF.

OTP:

OTPId → {OTP, ReferenceId, ExpiryDateTime, OTPPurpose, IsValid}

OTPId is the candidate key. So, OTP is in BCNF.

All relations are already in BCNF. Therefore, no further decomposition is needed. Explanation for not transforming to 3NF:

The relations are already in BCNF, which is a higher normal form than 3NF. BCNF eliminates all redundancy and dependency anomalies, making it a preferable normalization level. Therefore, there's no need to decompose them further into 3NF.

VIII. INDEXING

In our database, we have two tables UserTransactions and MerchantTransaction whose record count is around 10000 and 4000 respectively and these are the tables which will be used too frequently since transactions happen all the time. When dealing with tables containing a large number of records, queries may become slow due to full table scans. We optimized query performance by creating appropriate indexes on columns frequently used in WHERE clauses, JOIN conditions, and ORDER BY clauses.

1. User Transaction Table (10,000 records):
Since this table likely sees a high volume of queries, especially for retrieving transactions based on EmailId, PaymentTypeId, and StatusId, we created composite indexes on these columns.

```
CREATE INDEX
idx_user_transaction_email_payment_status
ON UserTransaction (EmailId, PaymentTypeId,
StatusId);
```

1. Merchant Transactions Table (4,000 records):
Similar to the User Transaction table, we created composite indexes on columns frequently used in queries, such as EmailId, PaymentTypeId, and StatusId.

```
CREATE INDEX
idx_merchant_transactions_email_payment_sta
tus ON MerchantTransactions (EmailId,
PaymentTypeId, StatusId);
```

2. UserCard Table (3,689 records):
This table might be frequently queried based on EmailId and CardNumber. Therefore, we can create indexes on these columns.

```
CREATE INDEX
idx_user_card_email_card_number ON UserCard
(EmailId, CardNumber);
```

Query Performance before indexing:

3/5/2024 8:54:47 am	4000	265 msec
Date	Rows affected	Duration

Copy | Copy to Query Editor

```
SELECT mt.*, m.*
FROM MerchantTransactions mt
INNER JOIN Merchant m ON mt.EmailId = m.EmailId;
```

Messages
Successfully run. Total query runtime: 265 msec. 4000 rows affected.

Query Performance after indexing:

3/5/2024 9:23:38 am	4000	216 msec
Date	Rows affected	Duration

Copy | Copy to Query Editor

```
SELECT mt.*, m.*
FROM MerchantTransactions mt
INNER JOIN Merchant m ON mt.EmailId = m.EmailId;
```

Messages
Successfully run. Total query runtime: 216 msec. 4000 rows affected.

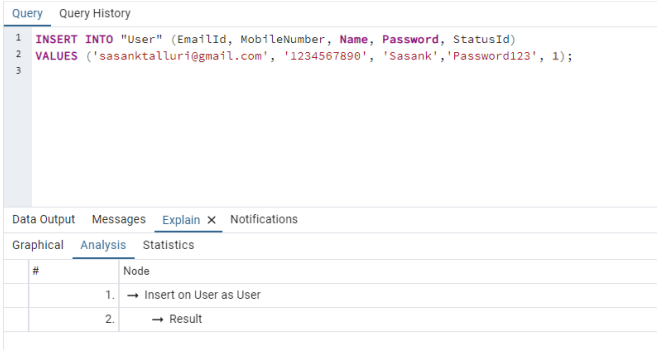
As you can see fetch time is decreased from 265 ms to 216 ms. By creating these indexes, we aim to improve query performance for frequently accessed columns and reduce the need for full table scans, thus enhancing overall database performance.

IX. QUERYING

1. Inserting Operation:

Insert a new user into the "User" table

```
INSERT INTO "User" (EmailId, MobileNumber,
Name, Password, StatusId)
VALUES ('sasanktalluri@gmail.com',
'1234567890', 'Sasank','Password123', 1);
```

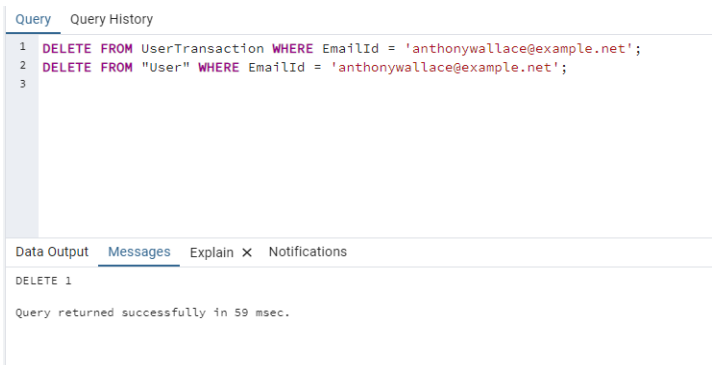


2. Deleting Operation:
Delete a user and associated transactions from the "User" and "UserTransaction" tables

```
DELETE FROM UserTransaction WHERE EmailId =
'anthonywallace@example.net';
```

3. Deleting Operation:

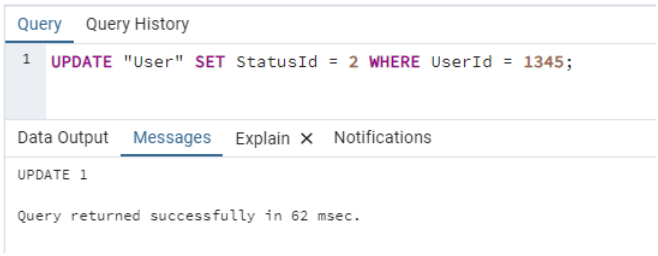
```
DELETE FROM "User" WHERE EmailId =
'anthonywallace@example.net';
```



4. Updating Operation:

Update the status of a user in the "User" table:

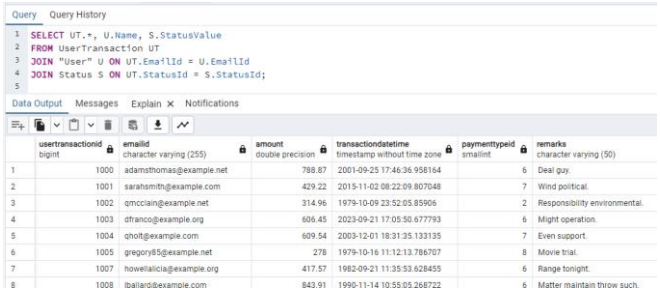
```
UPDATE "User" SET StatusId = 2 WHERE UserId = 1345;
```



5. Join Query:

Retrieve user transactions along with user and status details

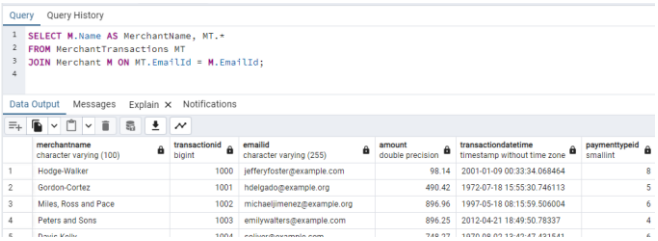
```
SELECT UT.*, U.Name, S.StatusValue
FROM UserTransaction UT
JOIN "User" U ON UT.EmailId = U.EmailId
JOIN Status S ON UT.StatusId = S.StatusId;
```



6. Join Query:

Retrieve merchant transactions along with merchant details

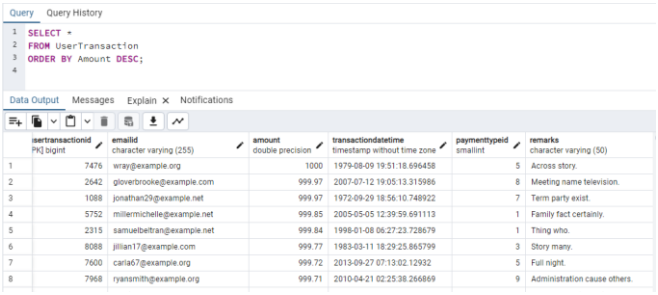
```
SELECT M.Name AS MerchantName, MT.*
FROM MerchantTransactions MT
JOIN Merchant M ON MT.EmailId = M.EmailId;
```



7. Order By Query:

Retrieve user transactions ordered by transaction amount:

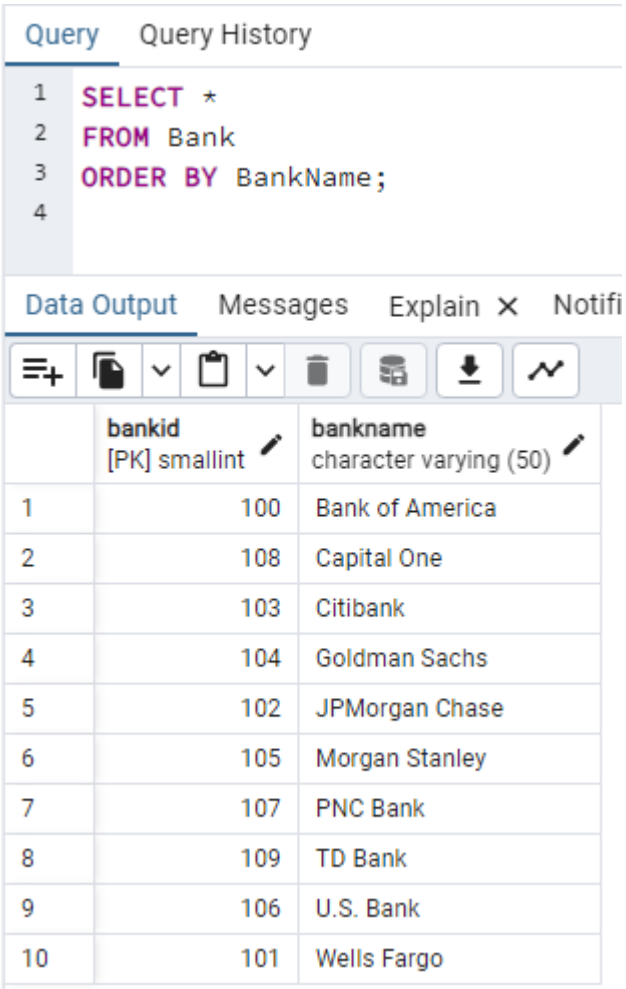
```
SELECT *
FROM UserTransaction
ORDER BY Amount DESC;
```



8. Order By Query:

Retrieve banks ordered by bank name

```
SELECT *
FROM Bank
ORDER BY BankName;
```



	bankid [PK] smallint	bankname character varying (50)
1	100	Bank of America
2	108	Capital One
3	103	Citibank
4	104	Goldman Sachs
5	102	JPMorgan Chase
6	105	Morgan Stanley
7	107	PNC Bank
8	109	TD Bank
9	106	U.S. Bank
10	101	Wells Fargo

9. Group By Query:

Retrieve the total transaction amount for each user

```
SELECT EmailId, SUM(Amount) AS TotalAmount
FROM UserTransaction
GROUP BY EmailId;
```

Query Query History

```
SELECT EmailId, SUM(Amount) AS TotalAmount
FROM UserTransaction
GROUP BY EmailId;
```

Data Output Messages Explain Notifications

	emailid character varying (255)	totalamount double precision
1	sandra32@example.org	1150.6
2	ramirezpatrick@example.com	1776.37
3	asavage@example.org	954.62
4	sheilabrewer@example.com	166.84
5	mcleancole@example.org	1511.31
6	michael58@example.com	815.5799999999999
7	james27@example.com	2009.02
8	danielle27@example.org	1297.56
9	ncameron@example.org	549.19

10. Group By Query:

Retrieve the count of transactions for each merchant:

```
SELECT EmailId, COUNT(*) AS
TransactionCount
FROM MerchantTransactions
GROUP BY EmailId;
```

Query Query History

```
SELECT EmailId, COUNT(*) AS TransactionCount
FROM MerchantTransactions
GROUP BY EmailId;
```

Data Output Messages Explain Notifications

	emailid character varying (255)	transactioncount bigint
1	malonederek@example.com	1
2	alyssamacdonald@example.net	2
3	anewman@example.org	4
4	tammy97@example.com	5
5	davisteresa@example.net	5
6	cynthia07@example.com	6
7	bellchristina@example.net	7
8	collierjohn@example.net	3
9	rroberts@example.org	2

11. Group By Query:

Retrieve the count of transactions for each merchant:

```
SELECT EmailId, COUNT(*) AS
TransactionCount
FROM UserTransactions
GROUP BY EmailId;
```

Query Query History

```
SELECT EmailId, COUNT(*) AS TransactionCount
FROM UserTransaction
GROUP BY EmailId;
```

Data Output Messages Explain Notifications

	emailid character varying (255)	transactioncount bigint
1	sandra32@example.org	2
2	ramirezpatrick@example.com	3
3	asavage@example.org	2
4	sheilabrewer@example.com	1
5	mcleancole@example.org	3

12. Subquery:

Retrieve users with transactions exceeding a certain amount:

```
SELECT Emailid FROM "User"
WHERE EmailId IN (SELECT EmailId FROM
UserTransaction WHERE Amount > 500);
```

Query Query History

```
SELECT Emailid
FROM "User"
WHERE EmailId IN (SELECT EmailId FROM UserTransaction WHERE Amount > 500);
```

Data Output Messages Explain Notifications

	emailid character varying (255)
1	sarahmartinez@example.com
2	johnsonkathryn@example.org
3	johnmartinez@example.net
4	jamie35@example.com
5	karenfields@example.com
6	ygibson@example.com
7	alexisjensen@example.org
8	vbarnes@example.com

13. Subquery:

Retrieve merchants with transactions exceeding a certain amount.

```
SELECT Name
FROM Merchant
WHERE EmailId IN (SELECT EmailId FROM
MerchantTransactions WHERE Amount > 900);
```

Query Query History

```
SELECT Name
FROM Merchant
WHERE EmailId IN (SELECT EmailId FROM MerchantTransactions WHERE Amount > 900);
```

Data Output Messages Explain Notifications

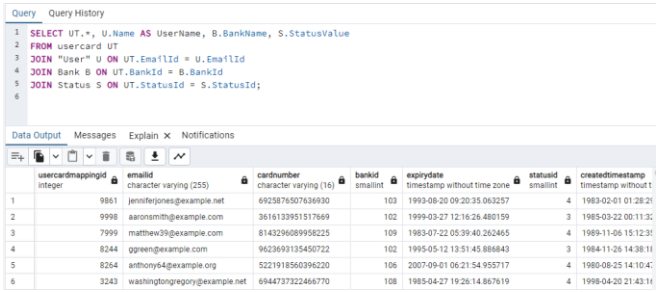
	name character varying (100)
1	Harris-Yates
2	Hall-Morgan
3	Ayala, Garcia and Massey
4	Ramirez Group
5	Ramsey, Cox and Gonzalez
6	Mitchell, Montes and Moran

14. Join Query:

Retrieve user card details along with user, bank, and status details.

```
SELECT UT.*, U.Name AS UserName,
B.BankName, S.StatusValue
```

```
FROM usercard UT
JOIN "User" U ON UT.EmailId = U.EmailId
JOIN Bank B ON UT.BankId = B.BankId
JOIN Status S ON UT.StatusId = S.StatusId;
```



15. Aggregation with Having Clause:

Retrieve merchants with a total transaction amount greater than a certain threshold.

```
SELECT M.Name, SUM(MT.Amount) AS
TotalAmount
FROM MerchantTransactions MT
JOIN Merchant M ON MT.EmailId = M.EmailId
GROUP BY M.Name
HAVING SUM(MT.Amount) > 1000;
```

Query | Query History

```
SELECT M.Name, SUM(MT.Amount) AS TotalAmount
FROM MerchantTransactions MT
JOIN Merchant M ON MT.EmailId = M.EmailId
GROUP BY M.Name
HAVING SUM(MT.Amount) > 1000;
```

Data Output | Messages | Explain | Notifications

	name character varying (100)	totalamount double precision
1	Stewart Group	1890.7600000000002
2	Wright, Wilcox and Huffman	3781.94
3	Williams LLC	7262.490000000001
4	Peters and Sons	9817.17
5	Matthews Ltd	2044.0500000000002
6	Lambert, Mahoney and Greer	3150.92
7	Harvey-Pacheco	5079.65
8	Woodward Ltd	3544.92

16. Using DISTINCT:

Retrieve distinct service types offered by merchants:

```
SELECT DISTINCT ServiceType
FROM MerchantServiceType;
```

Query | Query History

```
SELECT DISTINCT ServiceType
FROM MerchantServiceType;
```

Data Output | Messages | Explain | Notifications

	servicetype character varying (50)
1	any
2	her
3	statement
4	draw
5	here
6	interest
7	majority
8	order
9	resource
10	effect

X. Problematic Query Analysis

Three queries whose cost is greater than 200 ms.

1. Retrieve user transactions along with user and status details

```
SELECT UT.*, U.Name, S.StatusValue
FROM UserTransaction UT
JOIN "User" U ON UT.EmailId = U.EmailId
JOIN Status S ON UT.StatusId = S.StatusId;
```

3/5/2024 2:40:38 pm	9998	235 msec
Date	Rows affected	Duration

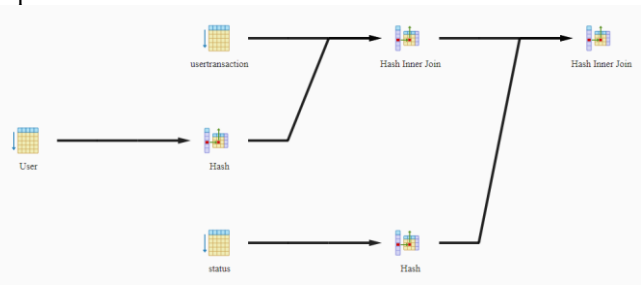
Copy | Copy to Query Editor

```
SELECT UT.*, U.Name, S.StatusValue
FROM UserTransaction UT
JOIN "User" U ON UT.EmailId = U.EmailId
JOIN Status S ON UT.StatusId = S.StatusId;
```

Messages

```
Successfully run. Total query runtime: 235 msec. 9998 rows
affected.
```

Explain Tool:



Data Output | Messages | Explain | Notifications

Graphical | Analysis | Statistics

#	Node
1.	→ Hash Inner Join **Hash Cond**: (ut.statusid = s.statusid)
2.	→ Hash Inner Join **Hash Cond**: ((ut.emailid)::text = (u.emailid)::text)
3.	→ Seq Scan on usertransaction as ut
4.	→ Hash
5.	→ Seq Scan on User as u
6.	→ Hash

Plan to optimize:
Create indexes on the "EmailId" column of the "User" table and the "StatusId" column of the "Status" table, as they are used in join conditions.
Optimize the query by adding additional filters to reduce the number of rows scanned.

2. Retrieve user card details along with user, bank, and status details.

```
SELECT UT.*, U.Name AS UserName,
B.BankName, S.StatusValue
FROM usercard UT
JOIN "User" U ON UT.EmailId = U.EmailId
JOIN Bank B ON UT.BankId = B.BankId
JOIN Status S ON UT.StatusId = S.StatusId;
```

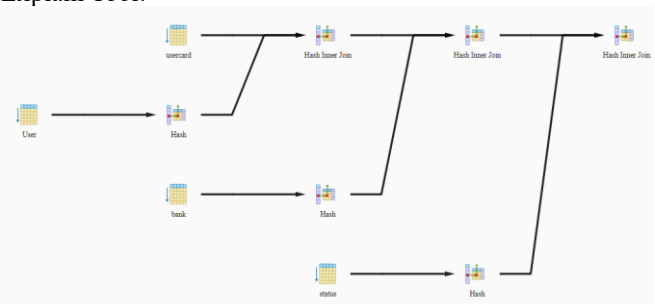
Cost:

3/5/2024 2:50:13 pm	3689	202 msec
Date	Rows affected	Duration

Copy | Copy to Query Editor

```
SELECT UT.*, U.Name AS UserName, B.BankName, S.StatusVa
FROM usercard UT
JOIN "User" U ON UT.EmailId = U.EmailId
JOIN Bank B ON UT.BankId = B.BankId
JOIN Status S ON UT.StatusId = S.StatusId;
```

Explain Tool:



Graphical | Analysis | Statistics

#	Node
1.	→ Hash Inner Join **Hash Cond**: (ut.statusid = s.statusid)
2.	→ Hash Inner Join **Hash Cond**: (ut.bankid = b.bankid)
3.	→ Hash Inner Join **Hash Cond**: ((ut.emailid)::text = (u.emailid)::text)
4.	→ Seq Scan on usercard as ut
5.	→ Hash
6.	→ Seq Scan on User as u
7.	→ Hash
8.	→ Seq Scan on bank as b
9.	→ Hash
10.	→ Seq Scan on status as s

Plan to optimize:
Create indexes on the "EmailId" column of the "User" table, the "BankId" column of the "Bank" table, and the "StatusId" column of the "Status" table, as they are used in join conditions.
Optimize the query by adding additional filters to reduce the number of rows scanned.

3. Retrieve users with transactions exceeding a certain amount:

```
SELECT *
FROM "User"
WHERE EmailId IN (SELECT EmailId FROM
UserTransaction WHERE Amount > 500);
```

Cost:

3/5/2024 2:57:31 pm	3086	232 msec
Date	Rows affected	Duration

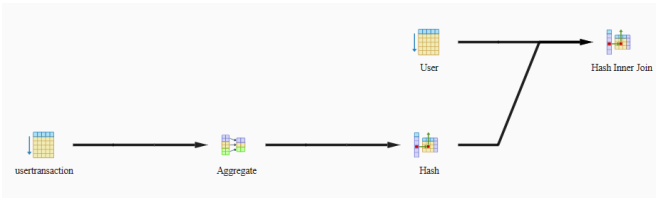
Copy | Copy to Query Editor

```
SELECT *
FROM "User"
WHERE EmailId IN (SELECT EmailId FROM UserTransaction WHI
```

Messages

```
Successfully run. Total query runtime: 232 msec. 3086 rows
affected.
```

Explain Tool:



#	Node
1.	→ Hash Inner Join **Hash Cond**: (("User".emailid)::text = (usertransaction.emailid)::text)
2.	→ Seq Scan on User as User
3.	→ Hash
4.	→ Aggregate
5.	→ Seq Scan on usertransaction as usertransaction **Filter**: (amount > '500'::double precision)

Plan to improve:

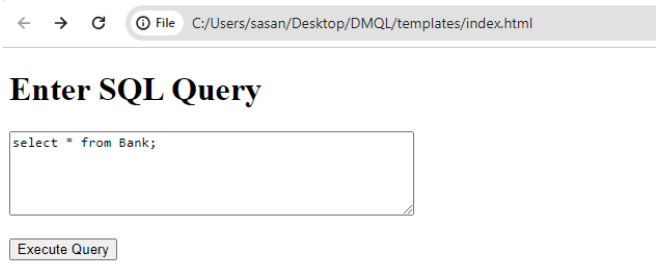
Create an index on the "EmailId" column of the "UserTransaction" table, as it is used in the subquery.

Optimize the query by adding additional filters to reduce the number of rows scanned.

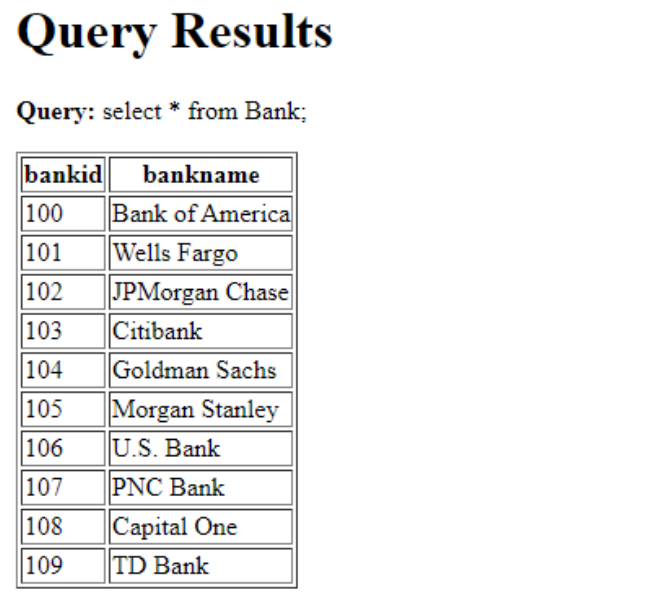
Note: These three are the queries which has highest cost. These might be potential problematic queries.

XI. Demo for User Interface.

Developed a running website and hosted locally which allows user to enter a query and if user enters on execute query, It lands on new page which display the query results in tables.



In the above demo, I entered a select query and I hit execute query button.



Query Results

Query: select * from Bank;

bankid	bankname
100	Bank of America
101	Wells Fargo
102	JPMorgan Chase
103	Citibank
104	Goldman Sachs
105	Morgan Stanley
106	U.S. Bank
107	PNC Bank
108	Capital One
109	TD Bank

The data fetched from the Database is displayed as shown in above picture.

Contribution:

Name	UBIT	Contribution%
Sasank Dattu Talluri	sasankda	50
Bala Seshagiri Prasad Munagala	bmunagal	50