**DBMS [Day - 4]**

UID: 24MCI10204

Name: Rahul Saxena

Branch: 24MCA – AI & ML

**Question 1:** You are working with a PostgreSQL-based banking system with the following tables:

* Accounts(account\_id, customer\_name, balance)
* Transactions(txn\_id, account\_id, txn\_type, amount, txn\_date)

You must implement logic to automate transfers and ensure secure and consistent transaction processing.

**Tasks:**

1. **Stored Procedure**:
   * Write a **stored procedure** transfer\_funds with parameters IN from\_account, IN to\_account, and IN amount. It should:
     + Begin a transaction.
     + Deduct the amount from the sender’s account.
     + Add the amount to the receiver’s account.
     + Insert a record into the Transactions table for both accounts.
     + Commit the transaction only if both updates succeed; otherwise, roll back.
2. **Trigger:**
   * Create a **BEFORE DELETE trigger** on Accounts that prevents deletion of an account if its balance is not zero. Display an appropriate error message if the condition fails.
3. **Security Enforcement:**
   * Write SQL statements to:
     + Create a role bank\_clerk with privileges to SELECT and INSERT on Transactions but not DELETE.
     + Grant this role to a user named clerk\_user.
     + Revoke INSERT privileges from clerk\_user later as a security precaution.
4. **SQL Injection Protection**:
   * Explain how the stored procedure can be protected from SQL Injection if user input is involved.
   * Give an example of a vulnerable dynamic SQL statement and a secure alternative using parameterized queries.

**Answer:**

**Stored Procedure:**

CREATE OR REPLACE FUNCTION transfer\_funds(

IN from\_account INT,

IN to\_account INT,

IN amount NUMERIC

) RETURNS VOID AS $$

BEGIN

BEGIN

IF (SELECT balance FROM Accounts WHERE account\_id = from\_account) < amount THEN

RAISE EXCEPTION 'Insufficient balance in sender account.';

END IF;

UPDATE Accounts

SET balance = balance - amount

WHERE account\_id = from\_account;

UPDATE Accounts

SET balance = balance + amount

WHERE account\_id = to\_account;

INSERT INTO Transactions(account\_id, txn\_type, amount, txn\_date)

VALUES (from\_account, 'debit', amount, CURRENT\_DATE);

INSERT INTO Transactions(account\_id, txn\_type, amount, txn\_date)

VALUES (to\_account, 'credit', amount, CURRENT\_DATE);

EXCEPTION

WHEN OTHERS THEN

RAISE NOTICE 'Transaction failed: %', SQLERRM;

ROLLBACK;

RETURN;

END;

COMMIT;

END;

$$ LANGUAGE plpgsql;

**Trigger: Prevent Deletion of Account with Non-Zero Balance**

**Create Trigger Function**

CREATE OR REPLACE FUNCTION prevent\_delete\_nonzero\_balance()

RETURNS TRIGGER AS $$

BEGIN

IF OLD.balance <> 0 THEN

RAISE EXCEPTION 'Cannot delete account %: Balance is not zero.', OLD.account\_id;

END IF;

RETURN OLD;

END;

$$ LANGUAGE plpgsql;

**Create Trigger**

CREATE TRIGGER prevent\_account\_deletion

BEFORE DELETE ON Accounts

FOR EACH ROW

EXECUTE FUNCTION prevent\_delete\_nonzero\_balance();

**Security Enforcement**

**-> Create the Role bank\_clerk with Privileges**

CREATE ROLE bank\_clerk;

GRANT SELECT, INSERT ON Transactions TO bank\_clerk;

**-> Create the User and Assign Role**

CREATE USER clerk\_user WITH PASSWORD 'securepassword';

GRANT bank\_clerk TO clerk\_user;

**-> Revoke INSERT Privileges from clerk\_user Later**

REVOKE INSERT ON Transactions FROM clerk\_user;

**SQL Injection Protection**

**How to Protect Stored Procedure from SQL Injection**

* **Use parameterized queries** inside stored procedures and avoid EXECUTE with dynamic SQL unless necessary.
* When dynamic SQL is required, **always use format() and quote\_literal() or quote\_ident()** to sanitize inputs.

**Vulnerable Example**

CREATE OR REPLACE FUNCTION get\_transactions(account TEXT)

RETURNS SETOF Transactions AS $$

DECLARE

query TEXT;

BEGIN

query := 'SELECT \* FROM Transactions WHERE account\_id = ' || account;

RETURN QUERY EXECUTE query;

END;

$$ LANGUAGE plpgsql;

**Secure Alternative Using Parameterized Query**

CREATE OR REPLACE FUNCTION get\_transactions\_secure(account INT)

RETURNS SETOF Transactions AS $$

BEGIN

RETURN QUERY

SELECT \* FROM Transactions WHERE account\_id = account;

END;

$$ LANGUAGE plpgsql;

**Question 2:** A startup is using **MongoDB** to manage customer product reviews. Each review document includes:

{

  "\_id": ObjectId,

  "product\_id": "P123",

  "customer\_name": "Rahul Saxena",

  "rating": 4.5,

  "review": "Great product!",

  "review\_date": "2025-05-12"

}

**Tasks:**

1. **CRUD Operations in MongoDB:**
   * Write MongoDB commands to:
     + Insert a new review document.
     + Update a review by a specific customer for a given product.
     + Retrieve all reviews for product\_id = "P123" with a rating above 4, sorted by review\_date.
     + Delete all reviews older than one year.
2. **NoSQL vs SQL:**
   * Compare how the same review data would be stored in an SQL database (provide schema example) and discuss pros and cons of each approach in terms of **scalability, flexibility, and consistency.**
3. **MongoDB Indexing**:
   * Create an index on product\_id and rating fields.
   * Explain how this improves query performance for frequent review retrievals.
4. **Database Backup & Encryption:**
   * Explain a MongoDB strategy to:
     + Encrypt sensitive fields like customer\_name.
     + Perform automated backups and recovery in case of data loss.
5. **Access Control:**
   * Demonstrate how to:
     + Create a MongoDB user with read-only access to the reviews collection.
     + Show the commands to enforce role-based access control.

**Answer:**

**CRUD Operations in MongoDB**

**Insert a new review document**

db.reviews.insertOne({

product\_id: "P123",

customer\_name: "Rahul Saxena",

rating: 4.5,

review: "Great product!",

review\_date: ISODate("2025-05-12")

});

**Update a Review by a Specific Customer for a Given Product**

db.reviews.updateOne(

{ product\_id: "P123", customer\_name: "Rahul Saxena" },

{ $set: { rating: 4.8, review: "Even better after a week of use!" } }

);

**Retrieve All Reviews for product\_id = "P123" with rating > 4, Sorted by review\_date**

db.reviews.find(

{ product\_id: "P123", rating: { $gt: 4 } }

).sort({ review\_date: -1 });

**Delete All Reviews Older Than One Year**

const oneYearAgo = new Date();

oneYearAgo.setFullYear(oneYearAgo.getFullYear() - 1);

db.reviews.deleteMany({

review\_date: { $lt: oneYearAgo }

});

**NoSQL vs SQL**

|  |  |  |
| --- | --- | --- |
| Feature | MongoDB (NoSQL) | SQL (Relational) |
| **Schema** | Flexible, schema-less | Rigid schema, requires migrations |
| **Scalability** | Easily horizontally scalable | Mostly vertical, harder horizontal scaling |
| **Flexibility** | Can store arrays, nested objects natively | Requires normalization / foreign keys |
| **Consistency** | Eventual consistency, needs handling in code | Strong ACID consistency |
| **Use Case** | Rapid iteration, unstructured or semi-structured | Structured, relational data with strict rules |

**MongoDB Indexing**

**Create Index on product\_id and rating:**

db.reviews.createIndex({ product\_id: 1, rating: -1 });

**How it Improves Performance:**

* **Reduces scan time** when filtering and sorting based on product\_id and rating.
* Helps MongoDB **quickly locate documents** during queries like:

db.reviews.find({ product\_id: "P123", rating: { $gt: 4 } })

**Database Backup & Encryption**

**Encrypt Sensitive Fields like customer\_name**

Use **Client-Side Field Level Encryption (CSFLE)**:

* Configure encryption keys via **KMS (e.g., AWS KMS, Azure Key Vault)**

**Automated Backups and Recovery**

**Strategies:**

* Use **MongoDB Atlas** automated backups (daily snapshots with point-in-time recovery).
* Or, for self-hosted:

**Access Control**

**Create a Read-Only User on reviews Collection**

use admin;

db.createUser({

user: "readonly\_user",

pwd: "readonly123",

roles: [

{

role: "read",

db: "yourDatabase"

}

]

});

**Role-Based Access Control Example**

db.createRole({

role: "reviewManager",

privileges: [

{

resource: { db: "yourDatabase", collection: "reviews" },

actions: ["find", "insert", "update"]

}

],

roles: []

});

db.createUser({

user: "manager\_user",

pwd: "man@123",

roles: ["reviewManager"]

});