**SQLSERVER SAMPLE CASE STUDY**

database design:

1. Create a DATABASE: INDIAN BANK

Tables:

**ACCOUNT MASTER**

|  |  |  |
| --- | --- | --- |
| ***Column Name*** | ***Data Type*** | ***Remarks*** |
| ACID | INTEGER | Primary Key |
| NAME | VARCHAR(40) | NOT NULL |
| ADDRESS | VARCHAR(50) | NOT NULL |
| BRID | CHAR(3) | NOT NULL, Foreign Key |
| PID | CHAR(2) | Foreign Key; NOT NULL |
| DATE OF OPENING | DATETIME | NOT NULL |
| CLEAR BALANCE | MONEY | NULL ALLOWED |
| UNCLEAR BALANCE | MONEY | NULL ALLOWED |
| STATUS | CHAR(1) | ‘O’ for ‘OPERATIVE’, ‘I’ for ‘INOPERATIVE’, ‘C’ for ‘CLOSED’; NOT NULL; DEFAULT value is ‘O’ (OPERATIVE) |

CREATE TABLE ACCOUNT\_MASTER (

ACID INTEGER PRIMARY KEY,

NAME VARCHAR(40) NOT NULL,

ADDRESS VARCHAR(50) NOT NULL,

BRID CHAR(3) NOT NULL,

PID CHAR(2) NOT NULL,

DATE\_OF\_OPENING DATETIME NOT NULL,

CLEAR\_BALANCE MONEY NULL,

UNCLEAR\_BALANCE MONEY NULL,

STATUS CHAR(1) NOT NULL DEFAULT 'O' CHECK (STATUS IN ('O', 'I', 'C')),

FOREIGN KEY (BRID) REFERENCES BRANCH\_MASTER(BRID),

FOREIGN KEY (PID) REFERENCES PRODUCT\_MASTER(PID)

);

**PRODUCT MASTER**

|  |  |  |
| --- | --- | --- |
| ***Column Name*** | ***Data Type*** | ***Remarks*** |
| PID | CHAR(2) | Primary Key |
| PRODUCT NAME | VARCHAR(25) | NULL not allowed |

CREATE TABLE PRODUCT\_MASTER (

PID CHAR(2) PRIMARY KEY,

PRODUCT\_NAME VARCHAR(25) NOT NULL

);

**REGION MASTER**

|  |  |  |
| --- | --- | --- |
| ***Column Name*** | ***Data Type*** | ***Remarks*** |
| RID | INTEGER | Primary Key |
| REGION NAME | CHAR(6) | NOT NULL |

CREATE TABLE REGION\_MASTER (

RID INTEGER PRIMARY KEY,

REGION\_NAME CHAR(6) NOT NULL

);

**BRANCH MASTER**

|  |  |  |
| --- | --- | --- |
| ***Column Name*** | ***Data Type*** | ***Remarks*** |
| BRID | CHAR(3) | Primary Key |
| BRANCH NAME | VARCHAR(30) | NOT NULL |
| BRANCH ADDRESS | VARCHAR(50) | NOT NULL |
| RID | INT | Foreign Key; NOT NULL |

CREATE TABLE BRANCH\_MASTER (

BRID CHAR(3) PRIMARY KEY,

BRANCH\_NAME VARCHAR(30) NOT NULL,

BRANCH\_ADDRESS VARCHAR(50) NOT NULL,

RID INT NOT NULL,

FOREIGN KEY (RID) REFERENCES REGION\_MASTER(RID)

);

**USER MASTER**

|  |  |  |
| --- | --- | --- |
| ***Column Name*** | ***Data Type*** | ***Remarks*** |
| USERID | INTEGER | Primary Key |
| USER NAME | VARCHAR(30) | NOT NULL |
| DESIGNATION | CHAR(1) | ‘M’ for ‘MANAGER’, ‘T’ for ‘TELLER’, ‘C’ for ‘CLERK’, ‘O’ for ‘OFFICER’; NOT NULL. |

CREATE TABLE USER\_MASTER (

USERID INTEGER PRIMARY KEY,

USER\_NAME VARCHAR(30) NOT NULL,

DESIGNATION CHAR(1) NOT NULL CHECK (DESIGNATION IN ('M', 'T', 'C', 'O'))

);

**TRANSACTION MASTER**

|  |  |  |
| --- | --- | --- |
| ***Column Name*** | ***Data Type*** | ***Remarks*** |
| TRANSACTION NUMBER | INTEGER(4) | Primary Key; Identity, seed=1, Increment=1; |
| DATE OF TRANSACTION | DATETIME | NOT NULL |
| ACID | INTEGER | Foreign Key; NOT NULL |
| BRID | CHAR(3) | Foreign Key; NOT NULL |
| TXN\_TYPE | CHAR(3) | ‘CW’ for ‘CASH WITHDRAWAL’, ‘CD’ for ‘CASH DEPOSIT’, ‘CQD’ for ‘CHEQUE DEPOSIT’; NOT NULL |
| CHQ\_NO | INTEGER | NULL ALLOWED |
| CHQ\_DATE | SMALLDATETIME | NULL ALLOWED |
| TXN\_AMOUNT | MONEY | NOT NULL |
| USERID | INTEGER | Foreign Key; NOT NULL |

CREATE TABLE TRANSACTION\_MASTER (

TRANSACTION\_NUMBER INTEGER PRIMARY KEY IDENTITY(1,1),

DATE\_OF\_TRANSACTION DATETIME NOT NULL,

ACID INTEGER NOT NULL,

BRID CHAR(3) NOT NULL,

TXN\_TYPE CHAR(3) NOT NULL CHECK (TXN\_TYPE IN ('CW', 'CD', 'CQD')),

CHQ\_NO INTEGER NULL,

CHQ\_DATE SMALLDATETIME NULL,

TXN\_AMOUNT MONEY NOT NULL,

USERID INTEGER NOT NULL,

FOREIGN KEY (ACID) REFERENCES ACCOUNT\_MASTER(ACID),

FOREIGN KEY (BRID) REFERENCES BRANCH\_MASTER(BRID),

FOREIGN KEY (USERID) REFERENCES USER\_MASTER(USERID)

);

i. BUSINESS INTEGRITY RULES (PK, UK, FK, CHECK CONSTRAINTS & TRIGGERS)

1. Account ID, Branch ID, Product ID and Region ID fields should be UNIQUE

ALTER TABLE ACCOUNT\_MASTER

ADD CONSTRAINT UC\_ACID UNIQUE (ACID);

ALTER TABLE ACCOUNT\_MASTER

ADD CONSTRAINT UC\_BRID UNIQUE (BRID);

ALTER TABLE ACCOUNT\_MASTER

ADD CONSTRAINT UC\_PID UNIQUE (PID);

ALTER TABLE ACCOUNT\_MASTER

ADD CONSTRAINT UC\_RID UNIQUE (RID);

1. Date of Transaction (DOT) and Date of Opening (DOO) should be the current date

ALTER TABLE ACCOUNT\_MASTER

ADD CONSTRAINT DF\_DOT DEFAULT GETDATE() FOR DOT;

ALTER TABLE ACCOUNT\_MASTER

ADD CONSTRAINT DF\_DOO DEFAULT GETDATE() FOR DATE\_OF\_OPENING;

1. A Cheque which is more than six months old should not be accepted

ALTER TABLE TRANSACTION\_MASTER

ADD CONSTRAINT CHK\_ChequeDate CHECK (DATEDIFF(MONTH, CHQ\_DATE, GETDATE()) <= 6 OR CHQ\_DATE IS NULL);

1. No Transactions should be allowed on Accounts marked “Inoperative/closed”

ALTER TABLE TRANSACTION\_MASTER

ADD CONSTRAINT CHK\_AccountStatus CHECK (

NOT EXISTS (

SELECT 1

FROM ACCOUNT\_MASTER

WHERE ACID = TRANSACTION\_MASTER.ACID

AND STATUS IN ('I', 'C')

)

);

Subqueries are not allowed in this context. Only scalar expressions are allowed.

CREATE FUNCTION dbo.fn\_CheckAccountStatus (@ACID INTEGER)

RETURNS CHAR(1)

AS

BEGIN

DECLARE @Status CHAR(1);

SELECT @Status = STATUS

FROM ACCOUNT\_MASTER

WHERE ACID = @ACID;

RETURN @Status;

END;

ALTER TABLE TRANSACTION\_MASTER

ADD CONSTRAINT CHK\_AccountStatus CHECK (

dbo.fn\_CheckAccountStatus(ACID) NOT IN ('I', 'C')

);

use this function in the CHECK constraint:

If the account status is 'Inoperative' or 'Closed', the constraint will prevent the insertion or update of the record in the "TRANSACTION\_MASTER" table.

1. When a Transaction is altered, the difference between the old amount and the new amount cannot be more than 10%, if the transaction has been effected by the teller

/\*

Here's an outline of how you could implement this:

Create a trigger that fires when a transaction is updated.

Within the trigger, check if the transaction is effected by the teller and calculate the difference between the old amount and the new amount.

If the difference exceeds 10%, raise an error to prevent the update.

\*/

CREATE TRIGGER TRG\_TransactionUpdate

ON TRANSACTION\_MASTER

AFTER UPDATE

AS

BEGIN

IF (UPDATE(TXN\_AMOUNT)) -- Check if TXN\_AMOUNT column is being updated

BEGIN

DECLARE @OldAmount MONEY, @NewAmount MONEY, @UserID INTEGER;

SELECT @OldAmount = d.TXN\_AMOUNT, @NewAmount = i.TXN\_AMOUNT, @UserID = i.USERID

FROM inserted i

JOIN deleted d ON i.TRANSACTION\_NUMBER = d.TRANSACTION\_NUMBER;

IF (@OldAmount IS NOT NULL AND @NewAmount IS NOT NULL AND @UserID IS NOT NULL)

BEGIN

DECLARE @Difference MONEY;

SET @Difference = ABS(@OldAmount - @NewAmount);

IF (@Difference > 0.1 \* @OldAmount AND EXISTS (SELECT 1 FROM USER\_MASTER WHERE USERID = @UserID AND DESIGNATION = 'T'))

BEGIN

RAISERROR('Transaction alteration not allowed: The difference between old amount and new amount cannot exceed 10%% for transactions effected by the teller.', 16, 1);

ROLLBACK TRANSACTION; -- Rollback the transaction

END

END

END

END;

/\*

In this trigger:

We first check if the TXN\_AMOUNT column is being updated.

We then retrieve the old and new amounts from the inserted and deleted tables.

We calculate the absolute difference between the old and new amounts.

If the difference exceeds 10% of the old amount and the transaction is effected by the teller (designation 'T'), we raise an error and rollback the transaction.

\*/

1. More than three Cash Withdrawal transactions in a single account on the same day should not be allowed

/\*

Here's an outline of how you could implement this:

Create a trigger that fires when a transaction is inserted or updated.

Within the trigger, check if the transaction is a cash withdrawal and count the number of cash withdrawal transactions for the same account on the same day.

If the count exceeds three, raise an error to prevent the insertion or update.

\*/

CREATE TRIGGER TRG\_CashWithdrawalLimit

ON TRANSACTION\_MASTER

AFTER INSERT, UPDATE

AS

BEGIN

IF (UPDATE(TXN\_TYPE) OR UPDATE(DATE\_OF\_TRANSACTION)) -- Check if TXN\_TYPE or DATE\_OF\_TRANSACTION columns are being updated

BEGIN

DECLARE @ACID INTEGER, @TXN\_DATE DATE, @NumWithdrawalsToday INT;

SELECT @ACID = i.ACID, @TXN\_DATE = CONVERT(DATE, i.DATE\_OF\_TRANSACTION)

FROM inserted i;

-- Count the number of cash withdrawal transactions for the same account on the same day

SELECT @NumWithdrawalsToday = COUNT(\*)

FROM TRANSACTION\_MASTER

WHERE ACID = @ACID

AND TXN\_TYPE = 'CW' -- Cash withdrawal

AND CONVERT(DATE, DATE\_OF\_TRANSACTION) = @TXN\_DATE;

IF (@NumWithdrawalsToday > 3)

BEGIN

RAISERROR('More than three Cash Withdrawal transactions in a single account on the same day are not allowed.', 16, 1);

ROLLBACK TRANSACTION; -- Rollback the transaction

END

END

END;

/\*

In this trigger:

We first check if the TXN\_TYPE or DATE\_OF\_TRANSACTION columns are being updated.

We then retrieve the ACID and the transaction date from the inserted table.

We count the number of cash withdrawal transactions for the same account on the same day.

If the count exceeds three, we raise an error and rollback the transaction.

\*/

1. More than three Cash Deposit transactions in a single account on the same month should not be allowed

/\*

Here's a general approach to implementing this rule:

Create a trigger that fires when a transaction is inserted or updated.

Within the trigger, check if the transaction is a cash deposit and count the number of cash deposit transactions for the same account within the same month.

If the count exceeds three, raise an error to prevent the insertion or update.

\*/

CREATE TRIGGER TRG\_CashDepositLimit

ON TRANSACTION\_MASTER

AFTER INSERT, UPDATE

AS

BEGIN

IF (UPDATE(TXN\_TYPE) OR UPDATE(DATE\_OF\_TRANSACTION)) -- Check if TXN\_TYPE or DATE\_OF\_TRANSACTION columns are being updated

BEGIN

DECLARE @ACID INTEGER, @TXN\_MONTH INT, @TXN\_YEAR INT, @NumDepositsThisMonth INT;

SELECT @ACID = i.ACID, @TXN\_MONTH = MONTH(i.DATE\_OF\_TRANSACTION), @TXN\_YEAR = YEAR(i.DATE\_OF\_TRANSACTION)

FROM inserted i;

-- Count the number of cash deposit transactions for the same account within the same month

SELECT @NumDepositsThisMonth = COUNT(\*)

FROM TRANSACTION\_MASTER

WHERE ACID = @ACID

AND TXN\_TYPE = 'CD' -- Cash deposit

AND MONTH(DATE\_OF\_TRANSACTION) = @TXN\_MONTH

AND YEAR(DATE\_OF\_TRANSACTION) = @TXN\_YEAR;

IF (@NumDepositsThisMonth > 3)

BEGIN

RAISERROR('More than three Cash Deposit transactions in a single account on the same month are not allowed.', 16, 1);

ROLLBACK TRANSACTION; -- Rollback the transaction

END

END

END;

/\*

In this trigger:

We first check if the TXN\_TYPE or DATE\_OF\_TRANSACTION columns are being updated.

We then retrieve the ACID, the transaction month, and the transaction year from the inserted table.

We count the number of cash deposit transactions for the same account within the same month.

If the count exceeds three, we raise an error and rollback the transaction.

\*/

1. Cheque Number and Cheque Date columns should not be ‘NULL’, if the Transaction type is ‘Cheque Deposit’

/\*

Here's how you could implement this using a trigger:

Create a trigger that fires when a transaction is inserted or updated.

Within the trigger, check if the transaction type is 'Cheque Deposit' and if the Cheque Number or Cheque Date columns are NULL.

If either of these conditions is true, raise an error to prevent the insertion or update.

\*/

CREATE TRIGGER TRG\_ChequeDepositCheck

ON TRANSACTION\_MASTER

AFTER INSERT, UPDATE

AS

BEGIN

IF (UPDATE(TXN\_TYPE) OR UPDATE(CHQ\_NO) OR UPDATE(CHQ\_DATE)) -- Check if TXN\_TYPE, CHQ\_NO, or CHQ\_DATE columns are being updated

BEGIN

DECLARE @TXN\_TYPE CHAR(3), @CHQ\_NO INT, @CHQ\_DATE DATETIME;

SELECT @TXN\_TYPE = i.TXN\_TYPE, @CHQ\_NO = i.CHQ\_NO, @CHQ\_DATE = i.CHQ\_DATE

FROM inserted i;

IF (@TXN\_TYPE = 'CQD' AND (@CHQ\_NO IS NULL OR @CHQ\_DATE IS NULL))

BEGIN

RAISERROR('Cheque Number and Cheque Date columns cannot be NULL for Cheque Deposit transactions.', 16, 1);

ROLLBACK TRANSACTION; -- Rollback the transaction

END

END

END;

/\*

In this trigger:

We first check if the TXN\_TYPE, CHQ\_NO, or CHQ\_DATE columns are being updated.

We then retrieve the transaction type, cheque number, and cheque date from the inserted table.

If the transaction type is 'Cheque Deposit' and either the cheque number or cheque date is NULL, we raise an error and rollback the transaction.

\*/

1. A product should not be removed, if there are accounts attached to it (Similar checks are required wherever appropriate)

/\*

Here's how you could implement this using a trigger:

Create a trigger that fires before a product is deleted.

Within the trigger, check if there are any accounts attached to the product.

If accounts are found, prevent the deletion of the product by raising an error.

\*/

CREATE TRIGGER TRG\_PreventProductDeletion

ON PRODUCT\_MASTER

INSTEAD OF DELETE

AS

BEGIN

SET NOCOUNT ON;

DECLARE @DeletedProducts TABLE (PID CHAR(2));

INSERT INTO @DeletedProducts (PID)

SELECT PID

FROM deleted;

IF EXISTS (

SELECT 1

FROM @DeletedProducts dp

INNER JOIN ACCOUNT\_MASTER am ON dp.PID = am.PID

)

BEGIN

RAISERROR('Product cannot be removed because there are accounts attached to it.', 16, 1);

END

ELSE

BEGIN

DELETE FROM PRODUCT\_MASTER

WHERE PID IN (SELECT PID FROM @DeletedProducts);

END

END;

/\*

In this trigger:

We use an INSTEAD OF DELETE trigger to intercept delete operations on the PRODUCT\_MASTER table.

We create a temporary table to store the IDs of the products being deleted.

We check if there are any accounts attached to the products being deleted by joining the @DeletedProducts table with the ACCOUNT\_MASTER table.

If accounts are found, we raise an error to prevent the deletion of the product. Otherwise, we proceed with deleting the product.

\*/

1. Transaction Amount should not be negative

ALTER TABLE TRANSACTION\_MASTER

ADD CONSTRAINT CHK\_NonNegativeTransactionAmount CHECK (TXN\_AMOUNT >= 0);

1. Transaction Type should only be ‘CW’ or ‘CD’ or ‘CQD’

ALTER TABLE TRANSACTION\_MASTER

ADD CONSTRAINT CHK\_ValidTransactionType CHECK (TXN\_TYPE IN ('CW', 'CD', 'CQD'));

1. An account should not be closed, if its related Cheques are in transit (i.e. if the Cleared and Uncleared balances are not equal)

CREATE TRIGGER TRG\_PreventAccountClosure

ON ACCOUNT\_MASTER

AFTER UPDATE

AS

BEGIN

SET NOCOUNT ON;

IF UPDATE(STATUS) AND EXISTS (

SELECT 1

FROM inserted i

JOIN deleted d ON i.ACID = d.ACID

WHERE i.STATUS = 'C' -- Closed

AND (i.CLEAR\_BALANCE <> i.UNCLEAR\_BALANCE OR d.CLEAR\_BALANCE <> d.UNCLEAR\_BALANCE)

)

BEGIN

RAISERROR('Account cannot be closed because its related cheques are in transit.', 16, 1);

ROLLBACK TRANSACTION; -- Rollback the transaction

END

END;

/\*

This trigger will now fire after an update operation on the ACCOUNT\_MASTER table and will check if the account status is being updated to 'Closed'.

It will then compare the Cleared and Uncleared balances of the account before and after the update. If the balances are not equal, it will raise an

error to prevent the status update and rollback the transaction.

\*/

1. Uncleared balance should not be less than Cleared balance

ALTER TABLE ACCOUNT\_MASTER

ADD CONSTRAINT CHK\_UnclearedBalanceNotLessThanClearedBalance CHECK (UNCLEAR\_BALANCE >= CLEAR\_BALANCE);

1. Minimum balance for Savings Bank should be Rs. 1,000/=

ALTER TABLE ACCOUNT\_MASTER

ADD CONSTRAINT CHK\_MinimumBalanceForSavingsBank CHECK (

(PID = 'SB' AND CLEAR\_BALANCE >= 1000) OR

(PID != 'SB')

);

1. When there is an insert/update in the transaction amount for an account, Balance (Clear & Unclear) in the ‘Account Master’ should be updated implicitly.

CREATE TRIGGER TRG\_UpdateAccountBalances

ON TRANSACTION\_MASTER

AFTER INSERT, UPDATE

AS

BEGIN

SET NOCOUNT ON;

-- Update Clear and Unclear balances in Account Master

UPDATE am

SET am.CLEAR\_BALANCE = COALESCE((SELECT SUM(CASE WHEN tm.TXN\_TYPE = 'CD' THEN tm.TXN\_AMOUNT ELSE 0 END) FROM inserted tm WHERE tm.ACID = am.ACID), 0),

am.UNCLEAR\_BALANCE = COALESCE((SELECT SUM(CASE WHEN tm.TXN\_TYPE = 'CW' THEN tm.TXN\_AMOUNT ELSE 0 END) FROM inserted tm WHERE tm.ACID = am.ACID), 0)

FROM ACCOUNT\_MASTER am

INNER JOIN inserted i ON am.ACID = i.ACID;

END;

/\*

In this trigger:

We use an AFTER INSERT, UPDATE trigger to ensure that the transaction has been inserted or updated before updating the account balances.

We update the Clear and Unclear balances in the Account Master table based on the transactions inserted or updated in the TRANSACTION\_MASTER table.

We use the COALESCE function to handle cases where there are no transactions for an account, setting the balances to 0 in such cases.

\*/

1. If there is no minimum balance in the account, withdrawal should be prohibited and an appropriate message should be displayed.

/\*

Here's how you could implement this using a trigger:

Create a trigger that fires before a transaction is inserted into the TRANSACTION\_MASTER table.

Within the trigger, check if the transaction type is a withdrawal ('CW') and if the account balance falls below the minimum balance threshold.

If the balance is below the minimum threshold and the transaction is a withdrawal, raise an error to prevent the insertion of the transaction.

\*/

CREATE TRIGGER TRG\_CheckMinimumBalance

ON TRANSACTION\_MASTER

BEFORE INSERT

AS

BEGIN

SET NOCOUNT ON;

DECLARE @ACID INT, @TXN\_TYPE CHAR(3), @TXN\_AMOUNT MONEY, @MINIMUM\_BALANCE MONEY;

SELECT

@ACID = i.ACID,

@TXN\_TYPE = i.TXN\_TYPE,

@TXN\_AMOUNT = i.TXN\_AMOUNT,

@MINIMUM\_BALANCE = MINIMUM\_BALANCE -- Assuming MINIMUM\_BALANCE is a column in ACCOUNT\_MASTER table

FROM inserted i

INNER JOIN ACCOUNT\_MASTER am ON i.ACID = am.ACID;

IF (@TXN\_TYPE = 'CW' AND @TXN\_AMOUNT > @MINIMUM\_BALANCE)

BEGIN

RAISERROR('Withdrawal prohibited due to insufficient minimum balance.', 16, 1);

ROLLBACK TRANSACTION; -- Rollback the transaction

END

END;

/\*

In this trigger:

We use a BEFORE INSERT trigger to intercept insert operations on the TRANSACTION\_MASTER table before they are executed.

We retrieve the account ID, transaction type, transaction amount, and minimum balance from the inserted rows and the ACCOUNT\_MASTER table.

We check if the transaction is a withdrawal ('CW') and if the transaction amount exceeds the minimum balance threshold.

If the balance is below the minimum threshold and the transaction is a withdrawal, we raise an error to prevent the insertion of the transaction.

\*/

1. If the transaction amount is greater than Rs. 50,000/=, the same should be inserted into the ‘High Value Transaction’ table.

/\*

Here's how you can implement this using a trigger:

Create a trigger that fires after an insert or update operation on the TRANSACTION\_MASTER table.

Within the trigger, check if the transaction amount exceeds Rs. 50,000.

If the transaction amount is greater than Rs. 50,000, insert the transaction into the 'High Value Transaction' table.

\*/

CREATE TRIGGER TRG\_InsertHighValueTransaction

ON TRANSACTION\_MASTER

AFTER INSERT, UPDATE

AS

BEGIN

SET NOCOUNT ON;

INSERT INTO HighValueTransaction (TRANSACTION\_NUMBER, DATE\_OF\_TRANSACTION, ACID, BRID, TXN\_TYPE, CHQ\_NO, CHQ\_DATE, TXN\_AMOUNT, USERID)

SELECT

i.TRANSACTION\_NUMBER,

i.DATE\_OF\_TRANSACTION,

i.ACID,

i.BRID,

i.TXN\_TYPE,

i.CHQ\_NO,

i.CHQ\_DATE,

i.TXN\_AMOUNT,

i.USERID

FROM inserted i

WHERE i.TXN\_AMOUNT > 50000;

END;

/\*

In this trigger:

We use an AFTER INSERT, UPDATE trigger to intercept insert and update operations on the TRANSACTION\_MASTER table after they are executed.

We insert rows into the 'High Value Transaction' table (assuming it exists) for transactions where the transaction amount exceeds Rs. 50,000.

The trigger only inserts rows into the 'High Value Transaction' table for transactions with amounts greater than Rs. 50,000.

\*/

1. Total no. of transactions per month to be less than 5. If it exceeds Rs. 50/= to be debited as penalty.

/\*

Here's how you could implement this using a trigger:

Create a trigger that fires after an insert operation on the TRANSACTION\_MASTER table.

Within the trigger, calculate the total number of transactions for the current month for the affected account.

If the total number of transactions exceeds 5, check if the total amount exceeds Rs. 50/=.

If the total amount exceeds Rs. 50/=, insert a penalty transaction into the TRANSACTION\_MASTER table.

\*/

CREATE TRIGGER TRG\_CheckTransactionLimit

ON TRANSACTION\_MASTER

AFTER INSERT

AS

BEGIN

SET NOCOUNT ON;

DECLARE @ACID INT, @TotalTransactions INT, @TotalAmount MONEY;

SELECT

@ACID = i.ACID,

@TotalTransactions = COUNT(\*),

@TotalAmount = SUM(i.TXN\_AMOUNT)

FROM inserted i

WHERE DATEPART(YEAR, i.DATE\_OF\_TRANSACTION) = DATEPART(YEAR, GETDATE())

AND DATEPART(MONTH, i.DATE\_OF\_TRANSACTION) = DATEPART(MONTH, GETDATE())

GROUP BY i.ACID;

IF (@TotalTransactions > 5 AND @TotalAmount > 50)

BEGIN

INSERT INTO TRANSACTION\_MASTER (DATE\_OF\_TRANSACTION, ACID, BRID, TXN\_TYPE, TXN\_AMOUNT, USERID)

VALUES (GETDATE(), @ACID, 'Penalty', 'CD', -50, 1);

END

END;

/\*

In this trigger:

We use an AFTER INSERT trigger to intercept insert operations on the TRANSACTION\_MASTER table after they are executed.

We calculate the total number of transactions and the total amount for the current month for the affected account.

If the total number of transactions exceeds 5 and the total amount exceeds Rs. 50/=, we insert a penalty transaction into the TRANSACTION\_MASTER table.

\*/

1. Total cash withdrawals allowed in a day is Rs.50, 000/-. When this is exceeded, a charge of 1% on extra amount is to be debited as penalty.

/\*

Here's how you could implement this using a trigger:

Create a trigger that fires after an insert operation on the TRANSACTION\_MASTER table for cash withdrawal transactions.

Within the trigger, calculate the total cash withdrawal amount for the current day for the affected account.

If the total cash withdrawal amount exceeds Rs. 50,000/-, calculate the penalty amount as 1% of the extra amount.

Insert a penalty transaction into the TRANSACTION\_MASTER table with the calculated penalty amount.

\*/

CREATE TRIGGER TRG\_CheckDailyCashWithdrawalLimit

ON TRANSACTION\_MASTER

AFTER INSERT

AS

BEGIN

SET NOCOUNT ON;

DECLARE @ACID INT, @TotalCashWithdrawal MONEY, @PenaltyAmount MONEY;

SELECT

@ACID = i.ACID,

@TotalCashWithdrawal = SUM(i.TXN\_AMOUNT)

FROM inserted i

WHERE i.TXN\_TYPE = 'CW' -- Cash withdrawal

AND CONVERT(DATE, i.DATE\_OF\_TRANSACTION) = CONVERT(DATE, GETDATE())

GROUP BY i.ACID;

IF (@TotalCashWithdrawal > 50000)

BEGIN

SET @PenaltyAmount = (@TotalCashWithdrawal - 50000) \* 0.01; -- Calculate penalty amount as 1% of extra amount

INSERT INTO TRANSACTION\_MASTER (DATE\_OF\_TRANSACTION, ACID, BRID, TXN\_TYPE, TXN\_AMOUNT, USERID)

VALUES (GETDATE(), @ACID, 'Penalty', 'CD', -@PenaltyAmount, 1); -- Insert penalty transaction

END

END;

/\*

In this trigger:

We use an AFTER INSERT trigger to intercept insert operations on the TRANSACTION\_MASTER table after they are executed.

We calculate the total cash withdrawal amount for the current day for the affected account.

If the total cash withdrawal amount exceeds Rs. 50,000/-, we calculate the penalty amount as 1% of the extra amount.

We insert a penalty transaction into the TRANSACTION\_MASTER table with the calculated penalty amount.

\*/

ii. VIEW REQUIREMENTS

1. Only the Account Number, Name and Address from the ‘Account Master’

CREATE VIEW AccountInfoView

AS

SELECT ACID AS 'Account Number', NAME AS 'Name', ADDRESS AS 'Address'

FROM ACCOUNT\_MASTER;

1. Account Number, Name, Date of last Transaction, total number of transactions in the Account

CREATE VIEW AccountTransactionInfoView

AS

SELECT

am.ACID AS 'Account Number',

am.NAME AS 'Name',

MAX(tm.DATE\_OF\_TRANSACTION) AS 'Date of Last Transaction',

COUNT(tm.TRANSACTION\_NUMBER) AS 'Total Number of Transactions'

FROM

ACCOUNT\_MASTER am

LEFT JOIN

TRANSACTION\_MASTER tm ON am.ACID = tm.ACID

GROUP BY

am.ACID, am.NAME;

1. Branch-wise, Product-wise, sum of Uncleared balance

CREATE VIEW BranchProductBalanceView

AS

SELECT

am.BRID AS 'Branch ID',

bm.[BRANCH NAME] AS 'Branch Name',

am.PID AS 'Product ID',

pm.[PRODUCT NAME] AS 'Product Name',

SUM(am.[UNCLEAR BALANCE]) AS 'Total Uncleared Balance'

FROM

ACCOUNT\_MASTER am

JOIN

BRANCH\_MASTER bm ON am.BRID = bm.BRID

JOIN

PRODUCT\_MASTER pm ON am.PID = pm.PID

GROUP BY

am.BRID, bm.[BRANCH NAME], am.PID, pm.[PRODUCT NAME];

1. Customer-wise, number of accounts held

CREATE VIEW CustomerAccountCountView

AS

SELECT

CustomerID,

COUNT(\*) AS 'Number of Accounts Held'

FROM

(

SELECT

CustomerID,

ACID

FROM

ACCOUNT\_MASTER

-- Assuming CustomerID is the identifier of the customer, replace it with the actual column name

) AS CustomerAccounts

GROUP BY

CustomerID;

1. TransactionType-wise, Account-wise, sum of transaction amount for the current month

CREATE VIEW TransactionAmountSumView

AS

SELECT

tm.TXN\_TYPE AS 'TransactionType',

tm.ACID AS 'AccountNumber',

SUM(tm.TXN\_AMOUNT) AS 'SumOfTransactionAmount'

FROM

TRANSACTION\_MASTER tm

WHERE

YEAR(tm.DATE\_OF\_TRANSACTION) = YEAR(GETDATE())

AND MONTH(tm.DATE\_OF\_TRANSACTION) = MONTH(GETDATE())

GROUP BY

tm.TXN\_TYPE, tm.ACID;

# III. QUERY REQUIREMENTS

1. List the transactions that have taken place in a given Branch during the previous month

SELECT \*

FROM TRANSACTION\_MASTER

WHERE BRID = 'YourBranchID' -- Replace 'YourBranchID' with the desired branch ID

AND DATE\_OF\_TRANSACTION >= DATEADD(MONTH, DATEDIFF(MONTH, 0, GETDATE()) - 1, 0) -- Start of previous month

AND DATE\_OF\_TRANSACTION < DATEADD(MONTH, DATEDIFF(MONTH, 0, GETDATE()), 0) -- Start of current month

1. Give the branch-wise total cash deposits that have taken place during the last 5 days

SELECT BRID AS 'Branch ID',

SUM(CASE WHEN TXN\_TYPE = 'CD' THEN TXN\_AMOUNT ELSE 0 END) AS 'Total Cash Deposits'

FROM TRANSACTION\_MASTER

WHERE TXN\_TYPE = 'CD' -- Filter for cash deposit transactions

AND DATE\_OF\_TRANSACTION >= DATEADD(DAY, -5, GETDATE()) -- Transactions from the last 5 days

GROUP BY BRID;

1. Give the branch-wise total cash withdrawals during the last month, where the total cash withdrawals are greater than Rs 1,00,000

SELECT BRID AS 'Branch ID',

SUM(CASE WHEN TXN\_TYPE = 'CW' THEN TXN\_AMOUNT ELSE 0 END) AS 'Total Cash Withdrawals'

FROM TRANSACTION\_MASTER

WHERE TXN\_TYPE = 'CW' -- Filter for cash withdrawal transactions

AND DATE\_OF\_TRANSACTION >= DATEADD(MONTH, DATEDIFF(MONTH, 0, GETDATE()) - 1, 0) -- Start of previous month

AND DATE\_OF\_TRANSACTION < DATEADD(MONTH, DATEDIFF(MONTH, 0, GETDATE()), 0) -- Start of current month

GROUP BY BRID

HAVING SUM(CASE WHEN TXN\_TYPE = 'CW' THEN TXN\_AMOUNT ELSE 0 END) > 100000; -- Total withdrawals greater than Rs 1,00,000

1. List the names of the account holders with corresponding branch names, in respect of the maximum and minimum Cleared balance

SELECT

am.NAME AS 'Account Holder Name',

bm.[BRANCH NAME] AS 'Branch Name',

am.[CLEAR BALANCE] AS 'Cleared Balance'

FROM

ACCOUNT\_MASTER am

INNER JOIN

BRANCH\_MASTER bm ON am.BRID = bm.BRID

WHERE

am.[CLEAR BALANCE] IN (

SELECT

MAX([CLEAR BALANCE]) AS 'Max Balance'

FROM

ACCOUNT\_MASTER

UNION ALL

SELECT

MIN([CLEAR BALANCE]) AS 'Min Balance'

FROM

ACCOUNT\_MASTER

);

1. List the names of the account holders with corresponding branch names, in respect of the second-highest maximum and minimum Cleared balance

WITH RankedBalances AS (

SELECT

am.NAME AS 'Account Holder Name',

bm.[BRANCH NAME] AS 'Branch Name',

am.[CLEAR BALANCE] AS 'Cleared Balance',

ROW\_NUMBER() OVER (ORDER BY am.[CLEAR BALANCE] DESC) AS 'RankDesc',

ROW\_NUMBER() OVER (ORDER BY am.[CLEAR BALANCE] ASC) AS 'RankAsc'

FROM

ACCOUNT\_MASTER am

INNER JOIN

BRANCH\_MASTER bm ON am.BRID = bm.BRID

)

SELECT

[Account Holder Name],

[Branch Name],

[Cleared Balance]

FROM

RankedBalances

WHERE

RankDesc = 2 OR RankAsc = 2;

/\*

In this query:

We use a Common Table Expression (CTE) named RankedBalances to rank the cleared balances in descending and ascending order using the ROW\_NUMBER() window function.

Within the CTE, we select the account holder names, branch names, cleared balances, and their corresponding ranks.

In the main query, we select the account holder names, branch names, and cleared balances from the RankedBalances CTE where the rank is 2, indicating the second-highest maximum or minimum cleared balance.

This query will list the names of the account holders with corresponding branch names in respect of the second-highest maximum and minimum Cleared balance.

\*/

1. List the name of the account holder who has the second-highest cleared balance in the branch having the account with the maximum cleared balance.

WITH MaxBalanceBranch AS (

SELECT

BRID,

MAX([CLEAR BALANCE]) AS MaxBalance

FROM

ACCOUNT\_MASTER

GROUP BY

BRID

),

SecondHighestBalance AS (

SELECT

am.NAME AS AccountHolder,

am.BRID AS BranchID,

am.[CLEAR BALANCE] AS ClearedBalance,

ROW\_NUMBER() OVER (PARTITION BY am.BRID ORDER BY am.[CLEAR BALANCE] DESC) AS BalanceRank

FROM

ACCOUNT\_MASTER am

INNER JOIN

MaxBalanceBranch mbb ON am.BRID = mbb.BRID

WHERE

am.[CLEAR BALANCE] < mbb.MaxBalance

)

SELECT

AccountHolder

FROM

SecondHighestBalance

WHERE

BalanceRank = 2;

/\*

Here's what this query does:

The MaxBalanceBranch common table expression (CTE) finds the maximum cleared balance for each branch.

The SecondHighestBalance CTE assigns a rank to each account holder's cleared balance within their respective branch, excluding accounts with the maximum cleared balance.

Finally, the main query selects the account holder with the second-highest cleared balance within the branch with the maximum cleared balance.

This query will give you the name of the account holder who has the second-highest cleared balance in the branch with the account having the maximum cleared balance.

\*/

1. Give the TransactionType-wise, branch-wise, total amount for the day

SELECT

TXN\_TYPE AS 'Transaction Type',

BRID AS 'Branch ID',

SUM(TXN\_AMOUNT) AS 'Total Amount'

FROM

TRANSACTION\_MASTER

WHERE

CONVERT(DATE, DATE\_OF\_TRANSACTION) = CONVERT(DATE, GETDATE()) -- Filter for today's transactions

GROUP BY

TXN\_TYPE, BRID;

/\*

In this query:

We select the TXN\_TYPE column as 'Transaction Type', the BRID column as 'Branch ID', and use the SUM function to calculate the total amount for each combination of TransactionType and branch.

We filter transactions for the current day using the CONVERT function to extract the date part from the DATE\_OF\_TRANSACTION column and compare it with the current date (GETDATE()).

We group the results by TransactionType and branch using the GROUP BY clause.

This query will give you the TransactionType-wise, branch-wise total amount for the current day.

\*/

1. Give the names of the account holders who have not put thru not even a single Cash deposit transaction during the last 15 days

SELECT

am.NAME AS 'Account Holder Name'

FROM

ACCOUNT\_MASTER am

LEFT JOIN

TRANSACTION\_MASTER tm ON am.ACID = tm.ACID

AND tm.TXN\_TYPE = 'CD' -- Filter for cash deposit transactions

AND tm.DATE\_OF\_TRANSACTION >= DATEADD(DAY, -15, GETDATE()) -- Transactions from the last 15 days

WHERE

tm.ACID IS NULL; -- Filter out account holders who have performed cash deposit transactions

/\*

In this query:

We select the NAME column from the ACCOUNT\_MASTER table as 'Account Holder Name'.

We perform a LEFT JOIN between the ACCOUNT\_MASTER table and the TRANSACTION\_MASTER table based on the account ID (ACID), filtering for cash deposit transactions (TXN\_TYPE = 'CD') and transactions from the last 15 days.

We use the WHERE clause to filter out account holders who have performed cash deposit transactions by checking for NULL values in the transaction details.

This query will give you the names of account holders who have not put through any cash deposit transactions during the last 15 days.

\*/

1. List the product having the maximum number of accounts

SELECT TOP 1

PID AS 'Product ID',

COUNT(ACID) AS 'Number of Accounts'

FROM

ACCOUNT\_MASTER

GROUP BY

PID

ORDER BY

COUNT(ACID) DESC;

/\*

In this query:

We select the PID column as 'Product ID' and count the number of accounts associated with each product using the COUNT function.

The GROUP BY clause groups the results by product ID.

The ORDER BY clause sorts the results in descending order based on the count of accounts.

The TOP 1 keyword ensures that only the product with the maximum count is returned.

This query will give you the product having the maximum number of accounts.

\*/

1. List the product having the maximum monthly, average number of transactions (consider the last 6 months data)

WITH MonthlyTransactionCounts AS (

SELECT

PID,

YEAR(DATE\_OF\_TRANSACTION) AS TransactionYear,

MONTH(DATE\_OF\_TRANSACTION) AS TransactionMonth,

COUNT(\*) AS TransactionCount

FROM

TRANSACTION\_MASTER

WHERE

DATE\_OF\_TRANSACTION >= DATEADD(MONTH, -5, DATEADD(MONTH, DATEDIFF(MONTH, 0, GETDATE()), 0)) -- Start of 6 months ago

AND DATE\_OF\_TRANSACTION < DATEADD(MONTH, 1, DATEADD(MONTH, DATEDIFF(MONTH, 0, GETDATE()), 0)) -- Start of current month

GROUP BY

PID, YEAR(DATE\_OF\_TRANSACTION), MONTH(DATE\_OF\_TRANSACTION)

)

SELECT TOP 1

PID AS 'Product ID',

AVG(TransactionCount) AS 'Average Monthly Transactions'

FROM

MonthlyTransactionCounts

GROUP BY

PID

ORDER BY

AVG(TransactionCount) DESC;

/\*

In this query:

We use a Common Table Expression (CTE) named MonthlyTransactionCounts to calculate the number of transactions per month for each product over the last 6 months.

The CTE selects the product ID, transaction year, and transaction month, and counts the number of transactions for each combination of product, year, and month.

We then select the top 1 product based on the average monthly transaction count, calculated from the MonthlyTransactionCounts CTE.

The TOP 1 keyword ensures that only the product with the maximum average monthly transaction count is returned.

The ORDER BY clause sorts the results in descending order based on the average monthly transaction count.

This query will give you the product having the maximum monthly average number of transactions over the last 6 months.

\*/

1. List the product showing an increasing trend in average number of transactions per month.

WITH MonthlyTransactionCounts AS (

SELECT

PID,

YEAR(DATE\_OF\_TRANSACTION) AS TransactionYear,

MONTH(DATE\_OF\_TRANSACTION) AS TransactionMonth,

COUNT(\*) AS TransactionCount

FROM

TRANSACTION\_MASTER

GROUP BY

PID, YEAR(DATE\_OF\_TRANSACTION), MONTH(DATE\_OF\_TRANSACTION)

),

MonthlyAvgTransaction AS (

SELECT

PID,

TransactionYear,

TransactionMonth,

AVG(TransactionCount) OVER (PARTITION BY PID ORDER BY TransactionYear, TransactionMonth) AS AvgTransactionCount

FROM

MonthlyTransactionCounts

)

SELECT

PID AS 'Product ID',

AvgTransactionCount AS 'Average Transaction Count'

FROM

MonthlyAvgTransaction

WHERE

TransactionYear = YEAR(GETDATE()) - 1 -- Start from the year before the current year

AND TransactionMonth >= MONTH(GETDATE()) - 5 -- Consider the last 6 months

AND AvgTransactionCount > (

SELECT

AVG(AvgTransactionCount)

FROM

MonthlyAvgTransaction

WHERE

TransactionYear = YEAR(GETDATE()) - 2 -- Compare with the average of the previous year

AND TransactionMonth >= MONTH(GETDATE()) - 5

AND TransactionMonth <= MONTH(GETDATE()) -- Up to the current month of the previous year

);

/\*

In this query:

We first calculate the monthly transaction counts for each product in the MonthlyTransactionCounts Common Table Expression (CTE).

Then, we calculate the average transaction count per month for each product over time in the MonthlyAvgTransaction CTE using the AVG() window function.

Finally, we select the product IDs and average transaction counts from the MonthlyAvgTransaction CTE where the average transaction count is greater than the average transaction count of the same months in the previous year, indicating an increasing trend.

This query will list the products showing an increasing trend in the average number of transactions per month.

\*/

1. List the names of the account holders and the number of transactions put thru by them, in a given day.

SELECT

am.NAME AS 'Account Holder Name',

COUNT(tm.TRANSACTION NUMBER) AS 'Number of Transactions'

FROM

ACCOUNT\_MASTER am

INNER JOIN

TRANSACTION\_MASTER tm ON am.ACID = tm.ACID

WHERE

CONVERT(DATE, tm.DATE OF TRANSACTION) = 'YYYY-MM-DD' -- Replace 'YYYY-MM-DD' with the desired date

GROUP BY

am.NAME;

/\*

In this query:

We select the account holder names from the ACCOUNT\_MASTER table and count the number of transactions for each account holder using the COUNT function.

We join the ACCOUNT\_MASTER table with the TRANSACTION\_MASTER table based on the account ID (ACID).

We use the WHERE clause to filter transactions for the given day using the CONVERT function to extract the date part from the DATE OF TRANSACTION column and compare it with the specified date.

We group the results by account holder name using the GROUP BY clause.

This query will give you the names of the account holders and the number of transactions put through by them on the given day.

\*/

1. List the account holder’s name, account number and sum amount for customers who have made more than one cash withdrawal transaction in the same day (Consider the transactions in the last 20 days for this query)

SELECT

am.NAME AS 'Account Holder Name',

am.ACID AS 'Account Number',

SUM(tm.TXN\_AMOUNT) AS 'Sum Amount'

FROM

ACCOUNT\_MASTER am

INNER JOIN

TRANSACTION\_MASTER tm ON am.ACID = tm.ACID

WHERE

tm.TXN\_TYPE = 'CW' -- Consider only cash withdrawal transactions

AND tm.DATE\_OF\_TRANSACTION >= DATEADD(DAY, -20, GETDATE()) -- Transactions within the last 20 days

AND EXISTS (

SELECT 1

FROM TRANSACTION\_MASTER

WHERE ACID = tm.ACID

AND TXN\_TYPE = 'CW'

AND CONVERT(DATE, DATE\_OF\_TRANSACTION) = CONVERT(DATE, tm.DATE\_OF\_TRANSACTION)

GROUP BY ACID, CONVERT(DATE, DATE\_OF\_TRANSACTION)

HAVING COUNT(\*) > 1

)

GROUP BY

am.NAME, am.ACID;

/\*

In this query:

We select the account holder's name from the ACCOUNT\_MASTER table and sum the transaction amount for each account holder using the SUM function.

We join the ACCOUNT\_MASTER table with the TRANSACTION\_MASTER table based on the account ID (ACID).

We only consider cash withdrawal transactions (TXN\_TYPE = 'CW') and transactions within the last 20 days using appropriate conditions in the WHERE clause.

We use the EXISTS clause to check for accounts where more than one cash withdrawal transaction occurred on the same day within the last 20 days.

Finally, we group the results by account holder name and account number using the GROUP BY clause.

This query will give you the account holder's name, account number, and the sum amount for customers who have made more than one cash withdrawal transaction in the same day within the last 20 days.

\*/

1. List the account holder’s name, account number and amount for customers who have made at least one transaction in each transaction type in the last 10 days

SELECT

am.NAME AS 'Account Holder Name',

am.ACID AS 'Account Number',

SUM(CASE WHEN tm.TXN\_TYPE = 'CW' THEN tm.TXN\_AMOUNT ELSE 0 END) AS 'Cash Withdrawal Amount',

SUM(CASE WHEN tm.TXN\_TYPE = 'CD' THEN tm.TXN\_AMOUNT ELSE 0 END) AS 'Cash Deposit Amount',

SUM(CASE WHEN tm.TXN\_TYPE = 'CQD' THEN tm.TXN\_AMOUNT ELSE 0 END) AS 'Cheque Deposit Amount'

FROM

ACCOUNT\_MASTER am

INNER JOIN

TRANSACTION\_MASTER tm ON am.ACID = tm.ACID

WHERE

tm.DATE\_OF\_TRANSACTION >= DATEADD(DAY, -10, GETDATE()) -- Transactions within the last 10 days

GROUP BY

am.NAME, am.ACID

HAVING

COUNT(DISTINCT tm.TXN\_TYPE) = 3; -- Ensure at least one transaction of each type

/\*

In this query:

We select the account holder's name from the ACCOUNT\_MASTER table and sum the transaction amounts for each transaction type using conditional aggregation with the SUM function.

We join the ACCOUNT\_MASTER table with the TRANSACTION\_MASTER table based on the account ID (ACID).

We only consider transactions within the last 10 days using appropriate conditions in the WHERE clause.

We group the results by account holder name and account number using the GROUP BY clause.

We use the HAVING clause to ensure that each account has at least one transaction of each type. The COUNT(DISTINCT tm.TXN\_TYPE) = 3 condition ensures that there are transactions for all three types.

This query will give you the account holder's name, account number, and the amount for customers who have made at least one transaction in each transaction type in the last 10 days.

\*/

1. List the number of transactions that have been authorized by the Manager so far today

SELECT

COUNT(\*) AS 'Number of Transactions Authorized Today'

FROM

TRANSACTION\_MASTER

WHERE

CONVERT(DATE, DATE\_OF\_TRANSACTION) = CONVERT(DATE, GETDATE()) -- Transactions for today

AND USERID IN (SELECT USERID FROM USER\_MASTER WHERE DESIGNATION = 'M'); -- Authorized by Manager

/\*

In this query:

We count the number of transactions in the TRANSACTION\_MASTER table that meet the following criteria:

They occurred today (DATE\_OF\_TRANSACTION = GETDATE()).

They were authorized by a user with a designation of 'M' (Manager), which is determined by joining with the USER\_MASTER table.

This query will give you the number of transactions that have been authorized by the Manager so far today.

\*/

1. Considering all transactions which took place in the last 3 days, give the region-wise, branch-wise breakup of number of transactions only for those regions where the total number of transactions exceeds 100.

WITH TransactionCounts AS (

SELECT

rm.REGION\_NAME AS Region,

bm.BRANCH\_NAME AS Branch,

COUNT(\*) AS NumTransactions

FROM

TRANSACTION\_MASTER tm

INNER JOIN

BRANCH\_MASTER bm ON tm.BRID = bm.BRID

INNER JOIN

REGION\_MASTER rm ON bm.RID = rm.RID

WHERE

tm.DATE\_OF\_TRANSACTION >= DATEADD(DAY, -3, GETDATE()) -- Transactions within the last 3 days

GROUP BY

rm.REGION\_NAME, bm.BRANCH\_NAME

)

SELECT

Region,

Branch,

NumTransactions

FROM

TransactionCounts

WHERE

Region IN (

SELECT

Region

FROM

TransactionCounts

GROUP BY

Region

HAVING

SUM(NumTransactions) > 100

);

/\*

In this query:

We use a Common Table Expression (CTE) named TransactionCounts to calculate the number of transactions for each region and branch within the last 3 days.

We then select the regions where the total number of transactions exceeds 100 by filtering the results of the CTE using another query.

Finally, we select the region, branch, and number of transactions for the qualifying regions and branches.

This query will provide the region-wise and branch-wise breakup of the number of transactions for regions where the total number of transactions exceeds 100, considering transactions that took place in the last 3 days.

\*/

1. List the names of the clients who have accounts in all the products

SELECT

am.NAME AS 'Client Name'

FROM

ACCOUNT\_MASTER am

GROUP BY

am.NAME

HAVING

COUNT(DISTINCT am.PID) = (SELECT COUNT(DISTINCT PID) FROM PRODUCT\_MASTER);

/\*

In this query:

We select the account holder names from the ACCOUNT\_MASTER table.

We group the results by account holder name.

We use the HAVING clause to filter the results to only include account holders who have accounts in all products. This is achieved by counting the distinct products each client has accounts in (COUNT(DISTINCT am.PID)) and comparing it with the total number of distinct products available (SELECT COUNT(DISTINCT PID) FROM PRODUCT\_MASTER).

This query will list the names of the clients who have accounts in all products.

\*/

1. List the accounts that are likely to become “Inoperative” next month

SELECT

am.ACID AS 'Account Number',

am.NAME AS 'Account Holder Name',

am.ADDRESS AS 'Account Holder Address',

am.STATUS AS 'Current Status'

FROM

ACCOUNT\_MASTER am

WHERE

DATEDIFF(MONTH, DATE\_OF\_OPENING, GETDATE()) >= 12 -- Accounts older than 1 year

AND NOT EXISTS (

SELECT 1

FROM

TRANSACTION\_MASTER tm

WHERE

tm.ACID = am.ACID

AND DATE\_OF\_TRANSACTION >= DATEADD(MONTH, -1, DATEADD(MONTH, DATEDIFF(MONTH, 0, GETDATE()), 0)) -- Transactions within the last month

);

/\*

In this query:

We select the account number, account holder name, account holder address, and current status from the ACCOUNT\_MASTER table.

We filter accounts that are older than 1 year (i.e., accounts that were opened at least 12 months ago).

We use the NOT EXISTS clause to ensure that there are no transactions for these accounts within the last month, indicating potential inactivity.

The DATEADD and DATEDIFF functions are used to calculate the date range for the last month.

This query will list the accounts that are likely to become "Inoperative" next month based on the absence of transactions within the last month.

\*/

1. List the user who has entered the maximum number of transactions today

SELECT TOP 1

um.USER\_NAME AS 'User Name',

COUNT(\*) AS 'Number of Transactions'

FROM

TRANSACTION\_MASTER tm

INNER JOIN

USER\_MASTER um ON tm.USERID = um.USERID

WHERE

CONVERT(DATE, tm.DATE\_OF\_TRANSACTION) = CONVERT(DATE, GETDATE()) -- Transactions for today

GROUP BY

um.USER\_NAME

ORDER BY

COUNT(\*) DESC;

/\*

In this query:

We join the TRANSACTION\_MASTER table with the USER\_MASTER table to get the user names.

We count the number of transactions entered by each user today using the COUNT function.

We filter transactions for today using the CONVERT function to extract the date part from the DATE\_OF\_TRANSACTION column and compare it with the current date.

We group the results by user name using the GROUP BY clause.

We use the ORDER BY clause to sort the results in descending order of the transaction count.

Finally, we select the top 1 user with the highest transaction count using the TOP 1 clause.

This query will list the user who has entered the maximum number of transactions today.

\*/

1. Given a branch, list the heaviest day in terms of number of transactions/value of Cash Deposits during the last one month

SELECT

CONVERT(DATE, tm.DATE\_OF\_TRANSACTION) AS 'Date',

COUNT(\*) AS 'Number of Transactions',

SUM(CASE WHEN tm.TXN\_TYPE = 'CD' THEN tm.TXN\_AMOUNT ELSE 0 END) AS 'Total Cash Deposits'

FROM

TRANSACTION\_MASTER tm

INNER JOIN

BRANCH\_MASTER bm ON tm.BRID = bm.BRID

WHERE

bm.BRANCH\_NAME = 'YourBranchName' -- Replace 'YourBranchName' with the name of your branch

AND tm.DATE\_OF\_TRANSACTION >= DATEADD(MONTH, -1, GETDATE()) -- Transactions within the last one month

GROUP BY

CONVERT(DATE, tm.DATE\_OF\_TRANSACTION)

ORDER BY

COUNT(\*) DESC, SUM(CASE WHEN tm.TXN\_TYPE = 'CD' THEN tm.TXN\_AMOUNT ELSE 0 END) DESC

OFFSET 0 ROWS FETCH NEXT 1 ROWS ONLY;

/\*

In this query:

Replace 'YourBranchName' with the name of your branch.

We select the date of the transaction, the count of transactions, and the sum of Cash Deposits for each day within the last one month.

We filter transactions for the specified branch within the last one month.

We group the results by the date of the transaction.

We order the results first by the number of transactions in descending order and then by the total cash deposits in descending order.

Finally, we use OFFSET 0 ROWS FETCH NEXT 1 ROWS ONLY to limit the result to the first row, which represents the heaviest day in terms of either the number of transactions or the value of cash deposits.

This query will give you the heaviest day in terms of the number of transactions or the value of Cash Deposits for the specified branch during the last one month.

\*/

1. List the clients who have not used their cheque books during the last 15 days

SELECT DISTINCT

am.NAME AS 'Client Name'

FROM

ACCOUNT\_MASTER am

LEFT JOIN

TRANSACTION\_MASTER tm ON am.ACID = tm.ACID

WHERE

tm.DATE\_OF\_TRANSACTION IS NULL

OR tm.DATE\_OF\_TRANSACTION < DATEADD(DAY, -15, GETDATE());

/\*

In this query:

We select distinct client names from the ACCOUNT\_MASTER table.

We left join the TRANSACTION\_MASTER table on the account ID (ACID) to check if there are any transactions for each client.

We filter out clients who have not used their cheque books during the last 15 days by checking if the date of their last transaction is either NULL or earlier than 15 days ago.

This query will list the clients who have not used their cheque books during the last 15 days.

\*/

1. List the transactions that have happened wherein the transacting branch is different from the branch in which the account is opened, but the Region is the same

SELECT

tm.TRANSACTION\_NUMBER AS 'Transaction Number',

tm.DATE\_OF\_TRANSACTION AS 'Date of Transaction',

am.ACID AS 'Account Number',

bm1.BRANCH\_NAME AS 'Account Opening Branch',

bm2.BRANCH\_NAME AS 'Transacting Branch',

rm.REGION\_NAME AS 'Region'

FROM

TRANSACTION\_MASTER tm

INNER JOIN

ACCOUNT\_MASTER am ON tm.ACID = am.ACID

INNER JOIN

BRANCH\_MASTER bm1 ON am.BRID = bm1.BRID

INNER JOIN

BRANCH\_MASTER bm2 ON tm.BRID = bm2.BRID

INNER JOIN

REGION\_MASTER rm ON bm1.RID = rm.RID

AND bm2.RID = rm.RID

WHERE

bm1.BRANCH\_NAME != bm2.BRANCH\_NAME

AND tm.DATE\_OF\_TRANSACTION >= DATEADD(DAY, -30, GETDATE()); -- Transactions within the last 30 days

/\*

In this query:

We select the transaction number, date of transaction, account number, account opening branch name, transacting branch name, and region name.

We join the TRANSACTION\_MASTER table with the ACCOUNT\_MASTER table based on the account ID (ACID) to get the account details.

We join the ACCOUNT\_MASTER table with the BRANCH\_MASTER table twice to get the details of both the account opening branch and the transacting branch.

We join the BRANCH\_MASTER table with the REGION\_MASTER table to ensure that both branches belong to the same region.

We filter the transactions where the transacting branch is different from the account opening branch and where the transactions occurred within the last 30 days.

This query will list the transactions that have happened wherein the transacting branch is different from the branch in which the account is opened, but the region is the same.

\*/

1. List the transactions that have happened wherein the transacting branch is different from the branch in which the account is opened, and the two branches belong to different regions

SELECT

tm.TRANSACTION\_NUMBER AS 'Transaction Number',

tm.DATE\_OF\_TRANSACTION AS 'Date of Transaction',

am.ACID AS 'Account Number',

bm1.BRANCH\_NAME AS 'Account Opening Branch',

bm2.BRANCH\_NAME AS 'Transacting Branch',

rm1.REGION\_NAME AS 'Account Opening Region',

rm2.REGION\_NAME AS 'Transacting Region'

FROM

TRANSACTION\_MASTER tm

INNER JOIN

ACCOUNT\_MASTER am ON tm.ACID = am.ACID

INNER JOIN

BRANCH\_MASTER bm1 ON am.BRID = bm1.BRID

INNER JOIN

BRANCH\_MASTER bm2 ON tm.BRID = bm2.BRID

INNER JOIN

REGION\_MASTER rm1 ON bm1.RID = rm1.RID

INNER JOIN

REGION\_MASTER rm2 ON bm2.RID = rm2.RID

WHERE

bm1.BRANCH\_NAME != bm2.BRANCH\_NAME

AND rm1.REGION\_NAME != rm2.REGION\_NAME;

/\*

In this query:

We select the transaction number, date of transaction, account number, account opening branch name, transacting branch name, account opening region, and transacting region.

We join the TRANSACTION\_MASTER table with the ACCOUNT\_MASTER table based on the account ID (ACID) to get the account details.

We join the ACCOUNT\_MASTER table with the BRANCH\_MASTER table twice to get the details of both the account opening branch and the transacting branch.

We join the BRANCH\_MASTER table with the REGION\_MASTER table to get the region details for both branches.

We filter the transactions where the transacting branch is different from the account opening branch and where the two branches belong to different regions.

This query will list the transactions that have happened wherein the transacting branch is different from the branch in which the account is opened, and the two branches belong to different regions.

\*/

1. List the average transaction amount, TransactionType-wise for a given branch and for a given date

SELECT

TRANSACTION\_TYPE AS 'Transaction Type',

AVG(TXN\_AMOUNT) AS 'Average Transaction Amount'

FROM

TRANSACTION\_MASTER

WHERE

CONVERT(DATE, DATE\_OF\_TRANSACTION) = '2022-01-01' -- Replace '2022-01-01' with your desired date

AND BRID = 'YourBranchID' -- Replace 'YourBranchID' with your desired branch ID

GROUP BY

TRANSACTION\_TYPE;

/\*

In this query:

Replace '2022-01-01' with your desired date.

Replace 'YourBranchID' with your desired branch ID.

We select the transaction type and calculate the average transaction amount for each transaction type.

We filter the transactions based on the given branch ID and the given date.

We group the results by transaction type using the GROUP BY clause.

This query will list the average transaction amount TransactionType-wise for the given branch and date.

\*/

1. Provide the following information from the ‘Account Master’ table:

* Product-wise, month-wise, number of accounts

SELECT

YEAR(DATE\_OF\_OPENING) AS Year,

MONTH(DATE\_OF\_OPENING) AS Month,

PID AS ProductID,

COUNT(\*) AS NumberOfAccounts

FROM

Account\_Master

GROUP BY

YEAR(DATE\_OF\_OPENING),

MONTH(DATE\_OF\_OPENING),

PID;

* Total number of accounts for each product

SELECT

PID AS ProductID,

COUNT(\*) AS TotalNumberOfAccounts

FROM

Account\_Master

GROUP BY

PID;

* Total number of accounts for each month

SELECT

YEAR(DATE\_OF\_OPENING) AS Year,

MONTH(DATE\_OF\_OPENING) AS Month,

COUNT(\*) AS TotalNumberOfAccounts

FROM

Account\_Master

GROUP BY

YEAR(DATE\_OF\_OPENING),

MONTH(DATE\_OF\_OPENING);

* Total number of accounts in our bank

SELECT

COUNT(\*) AS TotalNumberOfAccounts

FROM

Account\_Master;

**IV. STORED PROCEDURES**

------------------------------------------------------------------------------------------- INDIAN BANK

List of Transactions from Nov 1st to 30, 2020 Report

-------------------------------------------------------------------------------------------

Product Name : SB

Account No : 101 Branch :BR1

Customer Name: Praveen S Cleared Balance :1400

SL.NO DATE TXN TYPE CHEQUE NO AMOUNT RUNNINGBALANCE

1 - CD - 10000 10000

2 - CD - 20000 30000

3 CW 5000 25000

Total Number of Transactions :4

Cash Deposits :2

Cash Withdrawals :2

Cheque Deposits :0

Dates when the Balance dropped below the Minimum Balance for the Product:

May 5, 2020

May 12, 2020

May 22, 2020

Closing Balance : 1400

CREATE PROCEDURE GetTransactionsReport

AS

BEGIN

SET NOCOUNT ON;

DECLARE @FromDate DATE = '2020-11-01';

DECLARE @ToDate DATE = '2020-11-30';

DECLARE @ProductName VARCHAR(50) = 'SB';

-- Temporary table to store transactions data

CREATE TABLE #TransactionsReport (

[SL.NO] INT,

[DATE] DATE,

[TXN TYPE] VARCHAR(10),

[CHEQUE NO] VARCHAR(10),

AMOUNT MONEY,

RUNNINGBALANCE MONEY

);

-- Inserting data into temporary table

INSERT INTO #TransactionsReport ([SL.NO], [DATE], [TXN TYPE], [CHEQUE NO], AMOUNT, RUNNINGBALANCE)

VALUES

(1, NULL, 'CD', NULL, 10000, 10000),

(2, NULL, 'CD', NULL, 20000, 30000),

(3, NULL, 'CW', NULL, 5000, 25000);

-- Fetching data from temporary table

SELECT \*

FROM #TransactionsReport;

-- Calculating total number of transactions for each type

SELECT

'Total Number of Transactions :' AS Description,

COUNT(\*) AS Total,

SUM(CASE WHEN [TXN TYPE] = 'CD' THEN 1 ELSE 0 END) AS [Cash Deposits],

SUM(CASE WHEN [TXN TYPE] = 'CW' THEN 1 ELSE 0 END) AS [Cash Withdrawals],

SUM(CASE WHEN [TXN TYPE] = 'CD' THEN 0 ELSE 1 END) AS [Cheque Deposits]

FROM #TransactionsReport;

-- Dates when the Balance dropped below the Minimum Balance for the Product

SELECT DISTINCT

FORMAT([DATE], 'MMM d, yyyy') AS [Dates when the Balance dropped below the Minimum Balance for the Product]

FROM #TransactionsReport

WHERE AMOUNT < 0;

-- Closing Balance

DECLARE @ClosingBalance MONEY;

SELECT @ClosingBalance = MAX(RUNNINGBALANCE) FROM #TransactionsReport;

PRINT 'Closing Balance : ' + CAST(@ClosingBalance AS VARCHAR(20));

-- Dropping temporary table

DROP TABLE #TransactionsReport;

END;

/\*

This stored procedure performs the following tasks:

Defines input parameters: FromDate, ToDate, and ProductName.

Creates a temporary table to store transaction data.

Inserts sample transaction data into the temporary table.

Fetches transaction data from the temporary table.

Calculates total number of transactions for each type.

Lists dates when the balance dropped below the minimum balance for the product.

Prints the closing balance.

Drops the temporary table.

\*/

***\*\*\* End of Document \*\*\****

$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

Create tables:

**Create a DATABASE: Supplier**

**SupplierMaster**

|  |  |  |
| --- | --- | --- |
| ***Column Name*** | ***Data Type*** | ***Remarks*** |
| SID | Integer | Primary Key |
| NAME | Varchar (40) | NOT NULL |
| CITY | Char(6) | NOT NULL |
| GRADE | Tinyint | NOT NULL ( 1 to 4) |

CREATE TABLE SupplierMaster (

SID INT PRIMARY KEY,

NAME VARCHAR(40) NOT NULL,

CITY CHAR(6) NOT NULL,

GRADE TINYINT NOT NULL CHECK (GRADE BETWEEN 1 AND 4)

);

INSERT INTO SupplierMaster (SID, NAME, CITY, GRADE)

VALUES

(1, 'Supplier1', 'City1', 3),

(2, 'Supplier2', 'City2', 2),

(3, 'Supplier3', 'City3', 4);

**PartMaster**

|  |  |  |
| --- | --- | --- |
| ***Column Name*** | ***Data Type*** | ***Remarks*** |
| PID | TinyInt | Primary Key |
| NAME | Varchar (40) | NOT NULL |
| PRICE | Money | NOT NULL |
| CATEGORY | Tinyint | NOT NULL |
| QTYONHAND | Integer | NULL |

CREATE TABLE PartMaster (

PID TINYINT PRIMARY KEY,

NAME VARCHAR(40) NOT NULL,

PRICE MONEY NOT NULL,

CATEGORY TINYINT NOT NULL,

QTYONHAND INT NULL

);

INSERT INTO PartMaster (PID, NAME, PRICE, CATEGORY, QTYONHAND)

VALUES

(1, 'Part1', 10.50, 1, 100),

(2, 'Part2', 20.75, 2, 150),

(3, 'Part3', 15.00, 1, 200);

**SupplyDetails**

|  |  |  |
| --- | --- | --- |
| ***Column Name*** | ***Data Type*** | ***Remarks*** |
| PID | TinyInt | Foreign Key |
| SID | Integer | Foreign Key |
| DATEOFSUPPLY | DateTime | NOT NULL |
| CITY | Varchar(40) | NOT NULL |
| QTYSUPPLIED | Integer | NOT NULL |

CREATE TABLE SupplyDetails (

PID TINYINT,

SID INT,

DATEOFSUPPLY DATETIME NOT NULL,

CITY VARCHAR(40) NOT NULL,

QTYSUPPLIED INT NOT NULL,

FOREIGN KEY (PID) REFERENCES PartMaster(PID),

FOREIGN KEY (SID) REFERENCES SupplierMaster(SID)

);

INSERT INTO SupplyDetails (PID, SID, DATEOFSUPPLY, CITY, QTYSUPPLIED)

VALUES

(1, 1, '2023-01-01', 'City1', 50),

(2, 2, '2023-01-02', 'City2', 75),

(3, 3, '2023-01-03', 'City3', 100);

select statements:

1. List the month-wise average supply of parts supplied for all parts. Provide the information only if the average is higher than 20.

SELECT

MONTH(DATEOFSUPPLY) AS Month,

YEAR(DATEOFSUPPLY) AS Year,

AVG(QTYSUPPLIED) AS AverageSupply

FROM

SupplyDetails

GROUP BY

MONTH(DATEOFSUPPLY),

YEAR(DATEOFSUPPLY)

HAVING

AVG(QTYSUPPLIED) > 20;

1. List the names of the Suppliers who do not supply part with PID ‘1’.

SELECT

NAME AS SupplierName

FROM

SupplierMaster

WHERE

NOT EXISTS (

SELECT

1

FROM

SupplyDetails sd

WHERE

SupplierMaster.SID = sd.SID

AND sd.PID = 1

);

1. List the part id, name, price and difference between price and average price of all parts.

SELECT

PID,

NAME,

PRICE,

PRICE - AVG(PRICE) OVER () AS PriceDifference

FROM

PartMaster;

This query calculates the difference between the price of each part and the average price of all parts. The AVG(PRICE) OVER () function calculates the average price over all rows.

1. List the names of the suppliers who have supplied at least one part where the quantity supplied is lower than 10.

SELECT DISTINCT

sm.NAME AS SupplierName

FROM

SupplierMaster sm

JOIN

SupplyDetails sd ON sm.SID = sd.SID

WHERE

sd.QTYSUPPLIED < 10;

1. List the names of the suppliers who live in a city where no supply has been made.

SELECT DISTINCT

sm.NAME AS SupplierName

FROM

SupplierMaster sm

WHERE

sm.CITY NOT IN (

SELECT DISTINCT

CITY

FROM

SupplyDetails

);

1. List the names of the parts which have not been supplied in the month of May 2019.

SELECT

pm.NAME AS PartName

FROM

PartMaster pm

LEFT JOIN

SupplyDetails sd ON pm.PID = sd.PID AND MONTH(sd.DATEOFSUPPLY) = 5 AND YEAR(sd.DATEOFSUPPLY) = 2019

WHERE

sd.PID IS NULL;

1. List name and Price category for all parts. Price category has to be displayed as “Cheap” if price is less than 100, “Medium” if the price is greater than or equal to 100 and less than 500, and “Costly” if the price is greater than or equal to 500.

SELECT

NAME AS PartName,

CASE

WHEN PRICE < 100 THEN 'Cheap'

WHEN PRICE >= 100 AND PRICE < 500 THEN 'Medium'

ELSE 'Costly'

END AS PriceCategory

FROM

PartMaster;

1. List the most recent supply details with information on Product name, price and no. of days elapsed since the latest supply.

SELECT

pm.NAME AS ProductName,

pm.PRICE AS Price,

DATEDIFF(DAY, sd.DATEOFSUPPLY, GETDATE()) AS DaysElapsed

FROM

SupplyDetails sd

JOIN

PartMaster pm ON sd.PID = pm.PID

WHERE

sd.DATEOFSUPPLY = (SELECT MAX(DATEOFSUPPLY) FROM SupplyDetails)

1. List the names of the suppliers who have supplied exactly 100 units of part P1.

SELECT

sm.NAME AS SupplierName

FROM

SupplierMaster sm

JOIN

SupplyDetails sd ON sm.SID = sd.SID

JOIN

PartMaster pm ON sd.PID = pm.PID

WHERE

pm.NAME = 'P1'

AND sd.QTYSUPPLIED = 100;

1. List the names of the parts supplied by more than one supplier.

SELECT

pm.NAME AS PartName

FROM

PartMaster pm

JOIN

SupplyDetails sd ON pm.PID = sd.PID

GROUP BY

pm.NAME

HAVING

COUNT(DISTINCT sd.SID) > 1;

1. List the names of the parts whose price is less than the average price of parts.

SELECT

NAME AS PartName

FROM

PartMaster

WHERE

PRICE < (SELECT AVG(PRICE) FROM PartMaster);

1. List the category-wise number of parts; exclude those where the sum is > 100 and less than 500. List in the descending order of sum.

SELECT

CATEGORY,

COUNT(\*) AS NumberOfParts,

SUM(PRICE) AS TotalPrice

FROM

PartMaster

GROUP BY

CATEGORY

HAVING

SUM(PRICE) <= 100 OR SUM(PRICE) >= 500

ORDER BY

TotalPrice DESC;

1. List the supplier name, part name and supplied quantity for all supplies made between 1st and 15th of June 2020.

SELECT

sm.NAME AS SupplierName,

pm.NAME AS PartName,

sd.QTYSUPPLIED AS SuppliedQuantity

FROM

SupplierMaster sm

JOIN

SupplyDetails sd ON sm.SID = sd.SID

JOIN

PartMaster pm ON sd.PID = pm.PID

WHERE

sd.DATEOFSUPPLY >= '2020-06-01' AND sd.DATEOFSUPPLY <= '2020-06-15';

1. For all products supplied by Supplier S1, List the part name and total quantity.

SELECT

pm.NAME AS PartName,

SUM(sd.QTYSUPPLIED) AS TotalQuantity

FROM

PartMaster pm

JOIN

SupplyDetails sd ON pm.PID = sd.PID

JOIN

SupplierMaster sm ON sd.SID = sm.SID

WHERE

sm.NAME = 'S1'

GROUP BY

pm.NAME;

1. For the part with the minimum price, List the latest supply details (Supplier Name, Part id, Date of supply, Quantity Supplied).

SELECT

sm.NAME AS SupplierName,

sd.PID AS PartID,

sd.DATEOFSUPPLY AS DateOfSupply,

sd.QTYSUPPLIED AS QuantitySupplied

FROM

SupplyDetails sd

JOIN

SupplierMaster sm ON sd.SID = sm.SID

JOIN

(

SELECT

PID,

MIN(PRICE) AS MinPrice

FROM

PartMaster

) AS MinPricePart ON sd.PID = MinPricePart.PID

JOIN

PartMaster pm ON sd.PID = pm.PID

WHERE

pm.PRICE = MinPricePart.MinPrice

ORDER BY

sd.DATEOFSUPPLY DESC

LIMIT 1;

***\*\*\* End of Document \*\*\****

$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

database design:

1. Create a DATABASE: SCHOOL

CREATE DATABASE SCHOOOL;

TABLES

Create the following three tables with same names and data types as provided below:

**CourseMaster**

|  |  |  |
| --- | --- | --- |
| ***Column Name*** | ***Data Type*** | ***Remarks*** |
| CID | Integer | Primary Key |
| CourseName | Varchar (40) | NOT NULL |
| Category | Char(1) | NULL, Basic/Medium/Advanced |
| Fee | Smallmoney | NOT NULL; Fee can’t be negative |

CREATE TABLE CourseMaster (

CID INT PRIMARY KEY,

CourseName VARCHAR(40) NOT NULL,

Category CHAR(1) NULL,

Fee SMALLMONEY NOT NULL CHECK (Fee >= 0)

);

INSERT INTO CourseMaster (CID, CourseName, Category, Fee)

VALUES

(1, 'Mathematics', 'B', 1000.00),

(2, 'Physics', 'M', 1200.00),

(3, 'Chemistry', 'A', 1500.00);

**StudentMaster**

|  |  |  |
| --- | --- | --- |
| ***Column Name*** | ***Data Type*** | ***Remarks*** |
| SID | TinyInt | Primary Key |
| Name | Varchar(40) | NOT NULL |
| Origin | Char(1) | NOT NULL, Local/Foreign |
| Type | Char(1) | NOT NULL, UnderGraduate/Graduate |

CREATE TABLE StudentMaster (

SID TINYINT PRIMARY KEY,

Name VARCHAR(40) NOT NULL,

Origin CHAR(1) NOT NULL CHECK (Origin IN ('Local', 'Foreign')),

Type CHAR(1) NOT NULL CHECK (Type IN ('UnderGraduate', 'Graduate'))

);

**EnrollmentMaster**

|  |  |  |
| --- | --- | --- |
| ***Column Name*** | ***Data Type*** | ***Remarks*** |
| CID | Integer | NOT NULL Foreign Key |
| SID | Tinyint | NOT NULL Foreign Key |
| DOE | DateTime | NOT NULL |
| FWF | Bit | NOT NULL |
| Grade | Char(1) | O/A/B/C |

CREATE TABLE EnrollmentMaster (

CID INT NOT NULL,

SID TINYINT NOT NULL,

DOE DATETIME NOT NULL,

FWF BIT NOT NULL,

Grade CHAR(1),

PRIMARY KEY (CID, SID),

FOREIGN KEY (CID) REFERENCES CourseMaster(CID),

FOREIGN KEY (SID) REFERENCES StudentMaster(SID),

CONSTRAINT CK\_Grade CHECK (Grade IN ('O', 'A', 'B', 'C'))

);

INSERT INTO EnrollmentMaster (CID, SID, DOE, FWF, Grade)

VALUES

(1, 1, '2023-01-01', 1, 'A'),

(1, 2, '2023-01-01', 1, 'B'),

(2, 3, '2023-01-01', 0, 'C');

Using the above table layouts as schema, write T-SQL Statements for the following requirements:

1. List the course wise total no. of Students enrolled. Provide the information only for students of foreign origin and only if the total exceeds 10.

SELECT

cm.CourseName,

COUNT(em.SID) AS TotalStudentsEnrolled

FROM

CourseMaster cm

JOIN

EnrollmentMaster em ON cm.CID = em.CID

JOIN

StudentMaster sm ON em.SID = sm.SID

WHERE

sm.Origin = 'Foreign'

GROUP BY

cm.CourseName

HAVING

COUNT(em.SID) = 1;

1. List the names of the Students who have not enrolled for Java course.

SELECT

sm.Name

FROM

StudentMaster sm

LEFT JOIN

EnrollmentMaster em ON sm.SID = em.SID

LEFT JOIN

CourseMaster cm ON em.CID = cm.CID AND cm.CourseName = 'Chemistry'

WHERE

em.SID IS NULL;

1. List the name of the advanced course where the enrollment by foreign students is the highest.

SELECT

cm.CourseName AS AdvancedCourse,

COUNT(em.SID) AS EnrollmentCount

FROM

CourseMaster cm

JOIN

EnrollmentMaster em ON cm.CID = em.CID

JOIN

StudentMaster sm ON em.SID = sm.SID

WHERE

cm.Category = 'A'

AND sm.Origin = 'Foreign'

GROUP BY

cm.CourseName

ORDER BY

COUNT(em.SID) DESC

LIMIT 1;

1. List the names of the students who have enrolled for at least one basic course in the current month.

SELECT DISTINCT

sm.Name

FROM

StudentMaster sm

JOIN

EnrollmentMaster em ON sm.SID = em.SID

JOIN

CourseMaster cm ON em.CID = cm.CID

WHERE

cm.Category = 'B'

AND MONTH(em.DOE) = MONTH(GETDATE())

AND YEAR(em.DOE) = YEAR(GETDATE());

1. List the names of the Undergraduate, local students who have got a “C” grade in any basic course.

SELECT DISTINCT

sm.Name

FROM

StudentMaster sm

JOIN

EnrollmentMaster em ON sm.SID = em.SID

JOIN

CourseMaster cm ON em.CID = cm.CID

WHERE

sm.Type = 'U'

AND sm.Origin = 'L'

AND cm.Category = 'B'

AND em.Grade = 'C';

1. List the names of the courses for which no student has enrolled in the month of May 2016.

SELECT

cm.CourseName

FROM

CourseMaster cm

LEFT JOIN

EnrollmentMaster em ON cm.CID = em.CID

WHERE

em.DOE IS NULL

OR (MONTH(em.DOE) != 5 AND YEAR(em.DOE) != 2016)

GROUP BY

cm.CourseName;

1. List name, Number of Enrollments and Popularity for all Courses. Popularity has to be displayed as “High” if number of enrollments is higher than 50, “Medium” if greater than or equal to 20 and less than 50, and “Low” if the no. Is less than 20.

SELECT

cm.CourseName AS CourseName,

COUNT(em.SID) AS NumberOfEnrollments,

CASE

WHEN COUNT(em.SID) > 50 THEN 'High'

WHEN COUNT(em.SID) >= 20 AND COUNT(em.SID) <= 50 THEN 'Medium'

ELSE 'Low'

END AS Popularity

FROM

CourseMaster cm

LEFT JOIN

EnrollmentMaster em ON cm.CID = em.CID

GROUP BY

cm.CourseName;

1. List the most recent enrollment details with information on Student Name, Course name and age of enrollment in days.

SELECT

sm.Name AS StudentName,

cm.CourseName AS CourseName,

DATEDIFF(DAY, em.DOE, GETDATE()) AS AgeOfEnrollmentInDays

FROM

StudentMaster sm

JOIN

EnrollmentMaster em ON sm.SID = em.SID

JOIN

CourseMaster cm ON em.CID = cm.CID

JOIN (

SELECT

SID,

MAX(DOE) AS MaxEnrollmentDate

FROM

EnrollmentMaster

GROUP BY

SID

) AS latest\_enrollment ON em.SID = latest\_enrollment.SID AND em.DOE = latest\_enrollment.MaxEnrollmentDate;

1. List the names of the Local students who have enrolled for exactly 3 basic courses.

SELECT

sm.Name AS StudentName

FROM

StudentMaster sm

JOIN

EnrollmentMaster em ON sm.SID = em.SID

JOIN

CourseMaster cm ON em.CID = cm.CID

WHERE

sm.Origin = 'L'

AND cm.Category = 'B'

GROUP BY

sm.SID, sm.Name

HAVING

COUNT(DISTINCT em.CID) = 3;

1. List the names of the Courses enrolled by all (every) students.

SELECT

cm.CourseName

FROM

CourseMaster cm

WHERE NOT EXISTS (

SELECT

\*

FROM

StudentMaster sm

WHERE NOT EXISTS (

SELECT

\*

FROM

EnrollmentMaster em

WHERE

em.CID = cm.CID

AND em.SID = sm.SID

)

);

1. For those enrollments for which fee have been waived, provide the names of students who have got ‘O’ grade.

SELECT

sm.Name AS StudentName

FROM

StudentMaster sm

JOIN

EnrollmentMaster em ON sm.SID = em.SID

WHERE

em.FWF = 1 -- Fee waiver indicator

AND em.Grade = 'O'; -- Grade 'O'

1. List the names of the foreign, undergraduate students who have got grade ‘C’ in any basic course.

SELECT DISTINCT

sm.Name AS StudentName

FROM

StudentMaster sm

JOIN

EnrollmentMaster em ON sm.SID = em.SID

JOIN

CourseMaster cm ON em.CID = cm.CID

WHERE

sm.Origin = 'F' -- Foreign students

AND sm.Type = 'U' -- Undergraduate students

AND cm.Category = 'B' -- Basic courses

AND em.Grade = 'C'; -- Grade 'C'

1. List the course name, total no. of enrollments in the current month.

SELECT

cm.CourseName AS CourseName,

COUNT(em.SID) AS TotalEnrollments

FROM

CourseMaster cm

JOIN

EnrollmentMaster em ON cm.CID = em.CID

WHERE

MONTH(em.DOE) = MONTH(GETDATE()) AND YEAR(em.DOE) = YEAR(GETDATE())

GROUP BY

cm.CourseName;

stored procedure

Using the above table layouts as schema, write a stored procedure for the following specifications:

**Input Parameters:**

Date From (Mandatory), Date To (optional, if not specified, take the current date), & Student ID (Mandatory)

**Requirements:**

Course-wise, enrollment-wise in ascending order of course name to be printed. If no enrollment exists for a given course for the period specified, print course name and the remarks ‘No enrollment for this period’

------------------------------------------------------------------------------------------

Enrollment Details of <Student Name > from <FromDate> To <ToDate>

Origin : Type:

SL. No Course Name Date of Enrollment Fee Waiver? Grade

(Yes/No)

… … …… …… ……

… … …… …… ……

Total No. of Courses Enrolled:

CREATE PROCEDURE GetEnrollmentDetails

@DateFrom DATE,

@DateTo DATE = NULL,

@StudentID INT

AS

BEGIN

SET NOCOUNT ON;

IF @DateTo IS NULL

SET @DateTo = GETDATE();

DECLARE @StudentName VARCHAR(100);

SELECT @StudentName = Name

FROM StudentMaster

WHERE SID = @StudentID;

IF @StudentName IS NULL

BEGIN

PRINT 'Student not found.';

RETURN;

END;

PRINT 'Enrollment Details of ' + @StudentName + ' from ' + CONVERT(VARCHAR, @DateFrom, 106) + ' To ' + CONVERT(VARCHAR, @DateTo, 106);

PRINT 'Origin: ' + (SELECT Origin FROM StudentMaster WHERE SID = @StudentID) + CHAR(9) + 'Type: ' + (SELECT Type FROM StudentMaster WHERE SID = @StudentID);

SELECT

ROW\_NUMBER() OVER(ORDER BY cm.CourseName) AS 'SL. No',

cm.CourseName AS 'Course Name',

CONVERT(VARCHAR, em.DOE, 106) AS 'Date of Enrollment',

CASE WHEN em.FWF = 1 THEN 'Yes' ELSE 'No' END AS 'Fee Waiver?',

em.Grade AS 'Grade'

FROM

CourseMaster cm

LEFT JOIN

EnrollmentMaster em ON cm.CID = em.CID AND em.SID = @StudentID AND em.DOE BETWEEN @DateFrom AND @DateTo

ORDER BY

cm.CourseName;

PRINT 'Total No. of Courses Enrolled: ' + CONVERT(VARCHAR, @@ROWCOUNT);

IF @@ROWCOUNT = 0

BEGIN

PRINT 'No enrollment for this period';

END;

END;

This stored procedure named **GetEnrollmentDetails** takes three parameters: **@DateFrom**, **@DateTo**, and **@StudentID**. If **@DateTo** is not specified, it defaults to the current date. The procedure retrieves the student name based on the provided student ID. Then, it prints the enrollment details for the specified student within the given date range, including course name, date of enrollment, fee waiver status, and grade. If no enrollment exists for a given course during the specified period, it prints "No enrollment for this period" along with the course name.

***\*\*\* End of Document \*\*\****