Royaume du Maroc
Université Abdelmalek Essaâdi
Faculté des Sciences et Techniques – Tanger

Master AI & DATA Science

University Year: 2023-2024

BDD Assignment Report

Intituled:

SI Departement

Framed By :

Pr. Mostafa Ezziyyani

Du 4/12/2023 au 1/12/2023

Realised By :

Abdelmajid Benjelloun	H130407540	20000203
Mohammed Aachabi	N130119025	20001710

I. Introduction :

My colleague and I have undertaken a series of exercises focusing on database management within the context of an airline system. These exercises involved analyzing a given database schema, understanding the business rules associated with it, and addressing various queries and scenarios using SQL queries, procedures, triggers, and cursors.

To provide a structured and systematic approach to our solutions, we have utilized several modeling techniques, including use case diagrams, sequence diagrams, and class diagrams. These diagrams offer visual representations of the interactions, processes, and structure of the database system, aiding in our understanding and communication of the solutions developed.

In this report, we present our solutions to the exercises, detailing our approach, rationale, and the methodologies employed. Each exercise is accompanied by relevant SQL queries and explanations, demonstrating our proficiency in database management principles and techniques.

Through these exercises, we have aimed to enhance our skills in database design, implementation, and query optimization, while also gaining practical insights into the complexities of managing data within the aviation industry. Our collaborative efforts have enabled us to tackle diverse scenarios, apply theoretical knowledge to practical problems, and develop effective solutions within a database management context.

We now present our findings and solutions, showcasing our analytical abilities and problem-solving skills in the domain of airline database management.

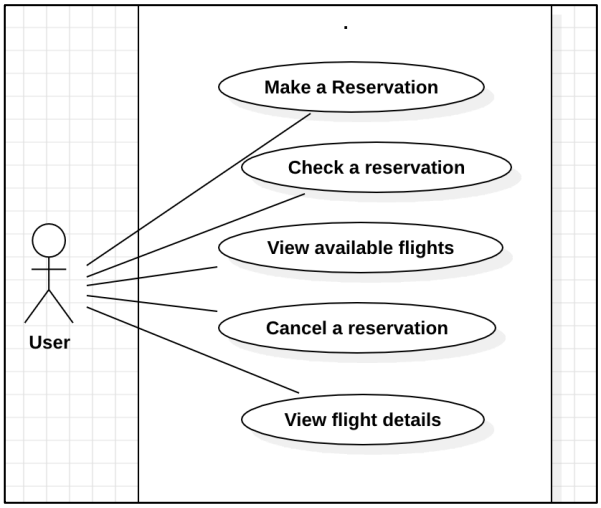
II. Conception Phase:

In this second chapter, we transition from the conceptualization phase to the practical implementation of our database solutions. Building upon the foundation established in the previous chapter, we delve into the intricacies of database design and query execution. Our focus shifts towards translating conceptual models into tangible database structures and executing queries to extract meaningful insights from the data.

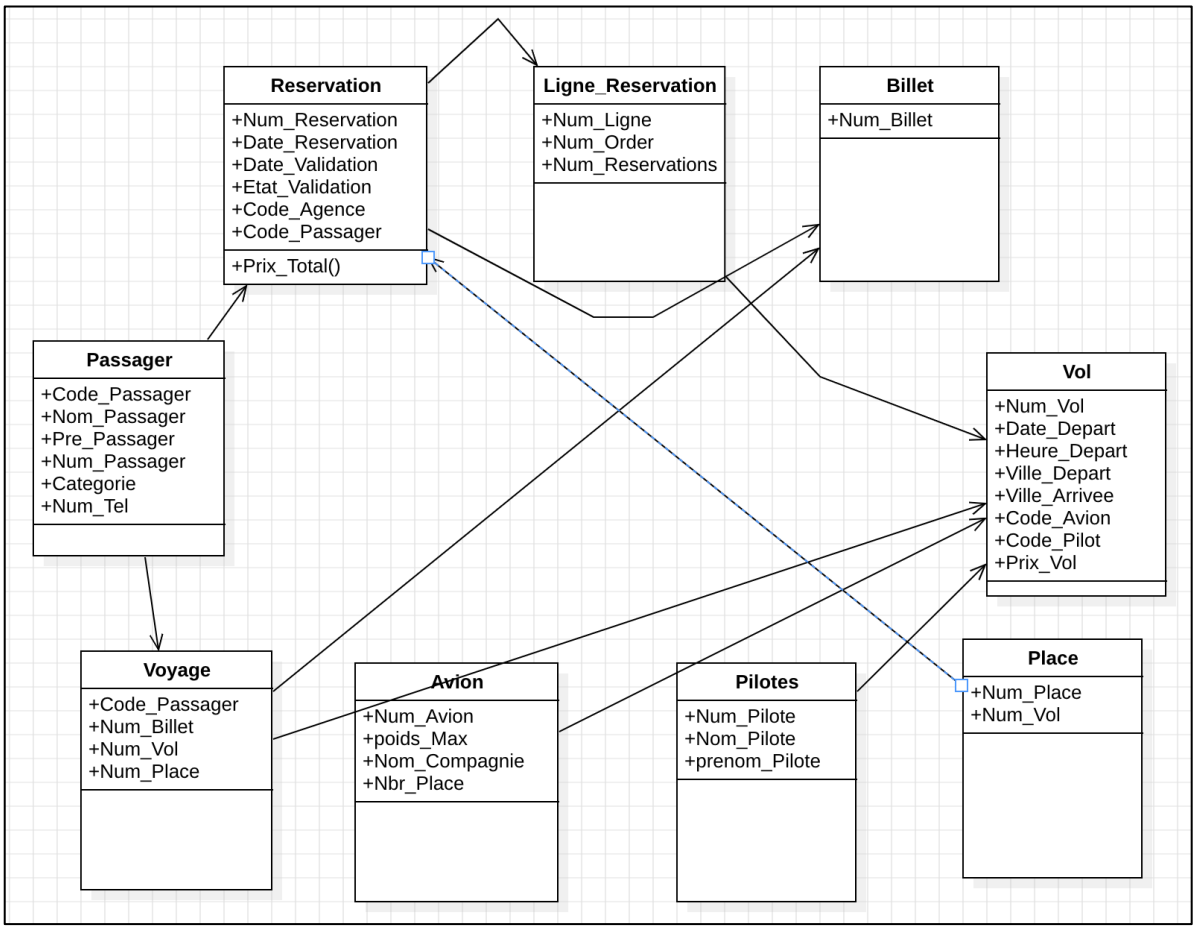
This chapter serves as a bridge between theoretical concepts and practical application, showcasing our ability to transform design blueprints into functional database systems. Through a series of exercises and scenarios, we demonstrate our proficiency in SQL query formulation, database manipulation, and optimization techniques.

Join us as we navigate through the implementation phase, presenting our approach, methodologies, and solutions to real-world database challenges. Through hands-on experimentation and problem-solving, we aim to refine our skills and deepen our understanding of database management principles.

we outline the key interactions between users and our airline management system. By defining clear and concise use cases, we aim to capture the core functionalities and user actions essential for system operation and usability. Through this analysis, we lay the groundwork for designing intuitive interfaces and implementing robust backend functionalities.



In this section, we present a class diagram depicting the structure and relationships of entities within our airline management system. The diagram provides a comprehensive overview of the system's architecture, including entities such as passengers, reservations, flights, aircraft, and pilots. Through this visual representation, we aim to elucidate the interdependencies between entities and facilitate a deeper understanding of the system's design and functionality.


Reservation
+Num_Reservation
+Date_Reservation
+Date_Validation
+Etat_Validation
+Code_Agence
+Code_Passager
+Prix_Total()
Ligne_Reservation
+Num_Ligne
+Num_Order
+Num_Reservations
Billet
+Num_Billet
Passager
+Code_Passager
+Nom_Passager
+Pre_Passager
+Num_Passager
+Categorie
+Num_Tel
Vol
+Num_Vol
+Date_Depart
+Heure_Depart
+Ville_Depart
+Ville_Arrivee
+Code_Avion
+Code_Pilot
+Prix_Vol
Voyage
+Code_Passager
+Num_Billet
+Num_Vol
+Num_Place
Avion
+Num_Avion
+poids_Max
+Nom_Compagnie
+Nbr_Place
Pilotes
+Num_Pilote
+Nom_Pilote
+prenom_Pilote
Place
+Num_Place
+Num_Vol

III. Exercise Solutions and Analysis.

Execice 1:

This algorithm defines a stored procedure in SQL Server to find all integers less than a given number whose digits sum up to 6. It stores the results, including the counts of even and odd digits, in a temporary table. Utilizing nested loops, it iterates through each integer, calculates the digit sums, and inserts qualifying integers into the temporary table. Finally, it returns the results from the temporary table and drops it once the operation is complete, providing an efficient solution for the specified task.

```
CREATE PROCEDURE dbo.Ex1_Done
(
    @nombre INT
)
AS
BEGIN

    CREATE TABLE #Resultats
    (
        Entier INT,
        ChiffresPairs INT,
        ChiffresImpairs INT
    );

    DECLARE @i INT = 1;

    WHILE @i < @nombre
    BEGIN
        DECLARE @chaine VARCHAR(10) = CAST(@i AS VARCHAR(10));
        DECLARE @longueur INT = LEN(@chaine);
        DECLARE @j INT = 1;
        DECLARE @somme INT = 0;
        DECLARE @pairs INT = 0;
        DECLARE @impairs INT = 0;
        WHILE @j <= @longueur
        BEGIN
            DECLARE @chiffre INT = CAST(SUBSTRING(@chaine, @j, 1) AS INT);
            SET @somme = @somme + @chiffre;

            IF @chiffre % 2 = 0
                SET @pairs = @pairs + 1;
            ELSE
                SET @impairs = @impairs + 1;

            SET @j = @j + 1;
        END
        IF @somme = 6
            INSERT INTO #Resultats (Entier, ChiffresPairs, ChiffresImpairs)
            VALUES (@i, @pairs, @impairs)
```

```
        SET @i = @i + 1;
    END
    SELECT * FROM #Resultats;
    DROP TABLE #Resultats;
END
GO
```

We have executed our programme using :

```
EXEC Ex1_Done @nombre = 100;
```

Executing the stored procedure Ex1_Done with the parameter @nombre set to 100 results in the creation of a temporary table. This table comprises three columns: 'Entier' (Integer), 'Chiffres Paires' (Even Digits), and 'Chiffres Impairs' (Odd Digits). The table is populated with numbers up to 100, where each row represents a number along with the count of its even and odd digits.

Exercice 2:

In this algorithm, we introduce a stored function named Ex2 designed to calculate the binary representation of an integer input. The function takes an integer value as input and returns a string representing its binary equivalent. By employing a systematic approach, the function iteratively divides the input integer by 2, capturing the remainders at each step to construct the binary string.

```
CREATE FUNCTION Ex2
(
    @input INT
)
RETURNS NVARCHAR(MAX)
AS
BEGIN
    DECLARE @binaryString NVARCHAR(MAX) = '';
    DECLARE @remainder INT;

    IF @input = 0
        RETURN '0';

    WHILE @input > 0
    BEGIN
        SET @remainder = @input % 2;
        SET @binaryString = CONVERT(NVARCHAR(1), @remainder) + @binaryString;
        SET @input = @input / 2;
    END

    RETURN @binaryString;
END
GO
```

The result after executing the programme

```
SELECT dbo.Ex2(122) AS Ex2;
```

	Ex2
1	1111010

Exercice 3:

In this task, we aim to develop a stored function in SQL Server that determines whether a given string is a palindrome or not. A palindrome is defined as a sequence of characters that reads the same forwards and backwards, disregarding spaces, punctuation, and capitalization. For instance, "TOTOT" and "TOUSUOT" are both examples of palindromes.

```
CREATE FUNCTION IsPalindrome(@str VARCHAR(255)) RETURNS BIT
AS
BEGIN
    DECLARE @len INT;
    DECLARE @i INT;
    DECLARE @isPal BIT;

    SET @len = LEN(@str);
    SET @i = 1;
    SET @isPal = 1;

    WHILE @i <= @len / 2
    BEGIN
        IF SUBSTRING(@str, @i, 1) != SUBSTRING(@str, @len - @i + 1, 1)
        BEGIN
            SET @isPal = 0;
            BREAK;
        END;

        SET @i = @i + 1;
    END;

    RETURN @isPal;
END;
```

To Execute the programme :

```
SELECT dbo.IsPalindrome('TOTOT');
```

The programme gives 1 if the string is a palindrom else the output is 0

	(No column name)
1	1

Exercice 4

In this task, we're tasked with writing a stored function that counts the number of words in a given string of characters.

```
CREATE FUNCTION CompterMots(@chaine VARCHAR(MAX)) RETURNS INT
AS
BEGIN
    DECLARE @nombreMots INT = 0;
    DECLARE @position INT = 1;
    DECLARE @longueur INT = LEN(@chaine);
    DECLARE @caracterePrecedent CHAR(1);

    WHILE @position <= @longueur
    BEGIN
        IF @caracterePrecedent IS NULL OR @caracterePrecedent IN (' ', CHAR(9),
CHAR(10), CHAR(13))
        BEGIN
            IF SUBSTRING(@chaine, @position, 1) NOT IN (' ', CHAR(9), CHAR(10),
CHAR(13))
            BEGIN
                SET @nombreMots = @nombreMots + 1;
            END;
        END;

        SET @caracterePrecedent = SUBSTRING(@chaine, @position, 1);
        SET @position = @position + 1;
    END;

    RETURN @nombreMots;
END;
```

To Execute the programme:

```
SELECT dbo.CompterMots('Master intelligence artificielle et data science');
```

	(No column name)
1	6

Exercice 5:

In this task, we are going to create a stored function that counts the number of occurrences of a given string within another string of characters.

```
 CREATE FUNCTION CompterOccurences(@chaine VARCHAR(MAX), @motCherche VARCHAR(MAX))
RETURNS INT
AS
BEGIN
    DECLARE @index INT = 1;
    DECLARE @count INT = 0;
    DECLARE @lenChaine INT = LEN(@chaine);
    DECLARE @lenMot INT = LEN(@motCherche);

    WHILE @index <= @lenChaine - @lenMot + 1
    BEGIN
        IF SUBSTRING(@chaine, @index, @lenMot) = @motCherche
        BEGIN
            SET @count = @count + 1;
            SET @index = @index + @lenMot - 1;
        END;
        SET @index = @index + 1;
    END;

    RETURN @count;
END;
```

To Execute the programme:

```
SELECT dbo.CompterOccurences('abdelmajidabd', 'abd');
```

The second column shows the number of occurrences of the 2nd name:

	(No column name)
1	2

Exercice 6:

In this task, we are required to create a stored function that finds the longest word in a given string of characters.

```
CREATE FUNCTION FindLongestWord(@inputString VARCHAR(MAX)) RETURNS VARCHAR(MAX)
AS
BEGIN
    DECLARE @longestWord VARCHAR(MAX) = '';
    DECLARE @words TABLE (word VARCHAR(MAX));

    -- Insert each word into the temporary table
    INSERT INTO @words (word)
    SELECT value
```

```
    FROM STRING_SPLIT(@inputString, ' ');

    -- Select the longest word
    SELECT TOP 1 @longestWord = word
    FROM @words
    ORDER BY LEN(word) DESC;

    RETURN @longestWord;
END;
GO
```

To Execute the programme:

```
SELECT dbo.FindLongestWord('Master Artificial intellignce and data science');
```

	(No column name)
1	intellignce

Exercice 7:

In this task, we're tasked with creating a stored procedure that displays a given number of minutes X in the format: "AA Années MM Mois JJJ Jours HH Heures MM Minutes", without using any built-in predefined functions.

```
CREATE PROCEDURE DisplayMinutesX
    @minutes INT
AS
BEGIN
    DECLARE @years INT, @months INT, @days INT, @hours INT;

    SET @years = @minutes / (60 * 24 * 365);
    SET @minutes = @minutes % (60 * 24 * 365);

    SET @months = @minutes / (60 * 24 * 30);
    SET @minutes = @minutes % (60 * 24 * 30);

    SET @days = @minutes / (60 * 24);
    SET @minutes = @minutes % (60 * 24);

    SET @hours = @minutes / 60;
    SET @minutes = @minutes % 60;

    PRINT CAST(@years AS VARCHAR(5)) + ' Années ' +
          CAST(@months AS VARCHAR(5)) + ' Mois ' +
          CAST(@days AS VARCHAR(5)) + ' Jours ' +
          CAST(@hours AS VARCHAR(5)) + ' Heures ' +
          CAST(@minutes AS VARCHAR(5)) + ' Minutes';
END;
```

To Execute the programme:

```
EXEC DisplayMinutesX @minutes = 2000000;
```

```
Started executing query at Line 211
3 Années 9 Mois 23 Jours 21 Heures 20 Minutes
Total execution time: 00:00:00.007
```

Exercice 8:

In this task, we're tasked with creating a stored procedure that creates the Vols table. The Vols table represents flights and is an essential part of our airline management system. We'll ensure all necessary constraints are applied to maintain data integrity, including structural and referential integrity constraints.

```
CREATE PROCEDURE CreateVolsTable
AS
BEGIN
    SET NOCOUNT ON;

    IF OBJECT_ID('Vols', 'U') IS NOT NULL
    BEGIN
        PRINT 'The Vols table already exists.';
        RETURN;
    END

    CREATE TABLE Vols (
        Num_Vol INT PRIMARY KEY,
        Date_Depart DATE,
        Heure_Depart TIME,
        Ville_Depart VARCHAR(100),
        Ville_Arrivee VARCHAR(100),
        Code_Avion INT,
        Code_Pilote INT,
        Prix_Vol DECIMAL(10, 2),
        CONSTRAINT FK_Avion FOREIGN KEY (Code_Avion) REFERENCES Avions(Num_Avion),
        CONSTRAINT FK_Pilote FOREIGN KEY (Code_Pilote) REFERENCES
Pilotes(Num_Pilote)
    );

    PRINT 'The Vols table has been created successfully.';
END;
```

To Execute the programme:

```
EXEC CreateVolsTable;
```

Exercice 9:

In this task, we aim to create a stored procedure that displays all non-validated reservations for a specific date. This stored procedure will help us efficiently manage reservations in our system, allowing us to identify and address any outstanding reservations.

```
CREATE PROCEDURE DisplayNonValidatedReservations
    @targetDate DATE
AS
BEGIN
    SET NOCOUNT ON;
```

```
    SELECT *
    FROM Reservations
    WHERE Date_Validation IS NULL
    AND Date_Reservation = @targetDate;
END;
```

To Execute the programme:

```
EXEC DisplayNonValidatedReservations @targetDate = '2024-03-15';
```

Num_Reservation	Date_Reservati…	Date_Validation	Etat_Reservati…	Code_Passager	Prix_Total

Exercice 10:

This stored procedure, named DisplayFlightInformation, is designed to fetch and display all information related to a given flight. It aims to provide comprehensive details about a specific flight, facilitating efficient retrieval and management of flight-related data.

```
CREATE PROCEDURE DisplayFlightInformation
    @flightNumber INT
AS
BEGIN
    SET NOCOUNT ON;

    SELECT *
    FROM Vols
    WHERE Num_Vol = @flightNumber;
END;
```

To Execute the programme:

```
EXEC DisplayFlightInformation @flightNumber = 123;
```

Num_Vol	Date_Depart	Heure_Depart	Ville_Depart	Ville_Arrivee	Code_Avion	Code_Pilote	Prix_Vol

Exercie 11:

To fulfill the task of displaying all information about a given flight, including details about pilots, departure city/time, arrival city/time, and details of any layovers, we will create a stored procedure.

```
CREATE PROCEDURE DisplayFlightInformation
    @flightNumber INT
AS
BEGIN
    SET NOCOUNT ON;

    SELECT V.Num_Vol,
```

```
            V.Date_Depart,
            V.Heure_Depart,
            V.Ville_Depart,
            V.Ville_Arrivee,
            PD.Nom_Pilote AS Nom_Pilote_Depart,
            PD.Prenom_Pilote AS Prenom_Pilote_Depart,
            PA.Nom_Pilote AS Nom_Pilote_Arrivee,
            PA.Prenom_Pilote AS Prenom_Pilote_Arrivee,
            E.Ville AS Ville_Escale,
            E.Heure_Arrivee AS Heure_Arrivee_Escale,
            E.Duree_Escale
    FROM Vols V
    LEFT JOIN Pilotes PD ON V.Code_Pilote = PD.Num_Pilote
    LEFT JOIN Pilotes PA ON V.Code_Pilote = PA.Num_Pilote
    LEFT JOIN Escales E ON V.Num_Vol = E.Num_Vol
    WHERE V.Num_Vol = @flightNumber;
END;
GO
EXEC DisplayFlightInformation @flightNumber = 123;
```

Exercice 12:

To display all information about a validated reservation (ticket), we will create a stored procedure.

```
CREATE PROCEDURE DisplayValidatedReservation
    @reservationNumber INT
AS
BEGIN
    SET NOCOUNT ON;

    SELECT R.Num_Reservation,
           R.Date_Reservation,
           R.Date_Validation,
           R.Etat_Reservation,
           P.Code_Passager,
           P.Nom_Passager,
           P.Pre_Passager,
           P.Num_Passport,
           P.Categorie,
           P.Num_Tel
    FROM Reservations R
    INNER JOIN Passagers P ON R.Code_Passager = P.Code_Passager
    WHERE R.Num_Reservation = @reservationNumber
          AND R.Date_Validation IS NOT NULL;
END;
```

To Execute the programme:

```
EXEC DisplayValidatedReservation @reservationNumber = 123;
```

Num_Reservation	Date_Reservati…	Date_Validation	Etat_Reservati…	Code_Passager	Nom_Passager	Pre_Passager	Num_Passport	Categorie	Num_Tel

Exercice 13:

To display the number of flights for each airplane in descending order, we will create a stored procedure.

```
CREATE PROCEDURE DisplayNumberOfFlightsPerAirplane
AS
BEGIN
    SET NOCOUNT ON;

    SELECT V.Code_Avion, COUNT(*) AS NumberOfFlights
    FROM Vols V
    GROUP BY V.Code_Avion
    ORDER BY NumberOfFlights DESC;
END;
```

To Execute the programme:

```
EXEC DisplayNumberOfFlightsPerAirplane;
```

	Code_Avion	NumberOfFlights

Exercice 14:

To calculate the number of trips for a given passenger, we will create a stored function.

```
CREATE FUNCTION CalculateNumberOfTripsForPassenger
(
    @passengerCode INT
)
RETURNS INT
AS
BEGIN
    DECLARE @numberOfTrips INT;

    SELECT @numberOfTrips = COUNT(*)
    FROM Voyages
    WHERE Code_Passager = @passengerCode;

    RETURN @numberOfTrips;
END;
```

To Execute the programme:

```
DECLARE @passengerCode INT = 123; -- Replace 123 with the desired passenger code

SELECT dbo.CalculateNumberOfTripsForPassenger1(@passengerCode) AS NumberOfTrips;
```

Exercice 15:

```
CREATE FUNCTION CalculateFlightCost1
(
    @flightNumber INT
)
RETURNS DECIMAL(10, 2)
AS
BEGIN
    DECLARE @cost DECIMAL(10, 2);

    -- Calculate cost based on airplane
    DECLARE @airplaneCost DECIMAL(10, 2);
    SELECT @airplaneCost = AVG(Prix_Vol) FROM Vols WHERE Num_Vol = @flightNumber;

    -- Return the total cost
    RETURN @airplaneCost;
END;
```

To Execute the programme:

```
DECLARE @flightNumber INT = 123; -- Replace 123 with the desired flight number
DECLARE @flightCost DECIMAL(10, 2);

SELECT @flightCost = dbo.CalculateFlightCost1(@flightNumber);

PRINT 'The estimated cost of flight ' + CAST(@flightNumber AS VARCHAR(10)) + ' is
$' + CAST(@flightCost AS VARCHAR(20));
```

Exercice 16:

In this task, we are required to create a stored procedure that deletes all unvalidated reservations from the database. The provided schema includes tables such as Passengers, Reservations, Flights, Tickets.

```
CREATE PROCEDURE DeleteUnvalidatedReservations
AS
BEGIN
    SET NOCOUNT ON;

    -- Delete unvalidated reservations
    DELETE FROM Reservations
    WHERE Etat_Reservation <> 'Validated';
END;
```

To Execute the programme:

```
EXEC DeleteUnvalidatedReservations;
```

Exercice 17:

This stored procedure aims to insert a record into the Voyages table while adhering to the following constraints:

- Ensuring uniqueness of records in the Voyages table.
- Verifying that the ticket number corresponds to the passenger and the flight.
- Guaranteeing the uniqueness of the passenger's assigned seat number.

```
CREATE PROCEDURE InsertVoyageRecord
    @Code_Passager INT,
    @Num_Billet INT,
    @Num_Vol INT,
    @Num_Place INT
AS
BEGIN
    SET NOCOUNT ON;

    IF EXISTS (SELECT 1 FROM Voyages WHERE Code_Passager = @Code_Passager AND
Num_Billet = @Num_Billet AND Num_Vol = @Num_Vol)
    BEGIN
        RETURN;
    END;

    IF NOT EXISTS (SELECT 1 FROM Billets WHERE Num_Billet = @Num_Billet AND
Code_Passager = @Code_Passager AND EXISTS (SELECT 1 FROM Ligne_Reservation WHERE
Num_Billet = @Num_Billet AND Num_Vol = @Num_Vol))
    BEGIN
        RETURN;
    END;

    IF EXISTS (SELECT 1 FROM Voyages WHERE Num_Place = @Num_Place)
    BEGIN
        RETURN;
    END;

    INSERT INTO Voyages (Code_Passager, Num_Billet, Num_Vol, Num_Place)
    VALUES (@Code_Passager, @Num_Billet, @Num_Vol, @Num_Place);
END;
```

To Execute the programme:

```
EXEC InsertVoyageRecord @Code_Passager = 1, @Num_Billet = 1, @Num_Vol = 1,
@Num_Place = 1;
```

Exercice 18:

This stored procedure is designed to insert a record into the Ligne_Reservation table while adhering to several constraints:

- Ensuring the uniqueness of the primary key in the Ligne_Reservation table.
- Verifying that the order number is sequential for the new reservation.
- Checking if the departure city of the new reservation's flight matches the arrival city of the previous reservation's flight (except for the first flight).
- Ensuring there are still available seats on the aircraft.

```
CREATE PROCEDURE InsertLigneReservationRecord
    @Num_Ligne INT OUTPUT,
    @Num_Order INT,
    @Num_Vol INT,
    @Num_Reservation INT
AS
BEGIN
    SET NOCOUNT ON;

    IF EXISTS (SELECT 1 FROM Ligne_Reservation WHERE Num_Ligne = @Num_Ligne)
    BEGIN
        RETURN;
    END;

    IF @Num_Order <> (SELECT ISNULL(MAX(Num_Order), 0) + 1 FROM Ligne_Reservation
WHERE Num_Reservation = @Num_Reservation)
    BEGIN
        RETURN;
    END;

    IF (SELECT COUNT(*) FROM Ligne_Reservation WHERE Num_Reservation =
@Num_Reservation) > 0
    BEGIN
        DECLARE @PreviousArrivalCity VARCHAR(50);
        SELECT @PreviousArrivalCity = V.Ville_Arrivee
        FROM Vols V
        JOIN Ligne_Reservation LR ON V.Num_Vol = LR.Num_Vol
        WHERE LR.Num_Reservation = @Num_Reservation
        ORDER BY LR.Num_Order DESC;

        DECLARE @NewDepartureCity VARCHAR(50);
        SELECT @NewDepartureCity = Ville_Depart
        FROM Vols
        WHERE Num_Vol = @Num_Vol;

        IF @PreviousArrivalCity <> @NewDepartureCity
```

```
        BEGIN
            RETURN;
        END;
    END;

    DECLARE @AvailableSeats INT;
    SELECT @AvailableSeats = A.Nbr_Place - ISNULL(COUNT(*), 0)
    FROM Voyages V
    JOIN Vols VL ON V.Num_Vol = VL.Num_Vol
    JOIN Avions A ON VL.Code_Avion = A.Num_Avion
    WHERE VL.Num_Vol = @Num_Vol
    GROUP BY A.Nbr_Place;

    IF @AvailableSeats <= 0
    BEGIN
        RETURN;
    END;

    INSERT INTO Ligne_Reservation (Num_Order, Num_Vol, Num_Reservation)
    VALUES (@Num_Order, @Num_Vol, @Num_Reservation);

    SELECT @Num_Ligne = SCOPE_IDENTITY();
END;


DECLARE @Num_LigneResult INT;
EXEC InsertLigneReservationRecord @Num_Ligne = @Num_LigneResult OUTPUT, @Num_Order
= 1, @Num_Vol = 1, @Num_Reservation = 1;
```

Exercice 19:

This stored procedure aims to add two columns, Nbr_Res and Nbr_Att, to the Vols table. These columns will be used to store the number of reserved seats and the number of assigned seats for each flight, respectively. The procedure utilizes dynamic SQL with the EXECUTE IMMEDIATE command to achieve this.

```
CREATE PROCEDURE AddColumnsToVols
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @SqlStatement NVARCHAR(MAX);
    SET @SqlStatement = '
        ALTER TABLE Vols
        ADD Nbr_Res INT DEFAULT 0,
            Nbr_Att INT DEFAULT 0;
    ';

    EXEC sp_executesql @SqlStatement;
END;
```

```
EXEC AddColumnsToVols;
```

Exercice 20:

This stored procedure updates the Nbr_Res and Nbr_Att columns in the Vols table with the respective counts of reserved seats and assigned seats for a given flight.

```
CREATE PROCEDURE UpdateSeatsCountForFlight
    @Num_Vol INT
AS
BEGIN
    SET NOCOUNT ON;

    UPDATE Vols
    SET Nbr_Res = (SELECT COUNT(*) FROM Ligne_Reservation WHERE Num_Vol =
@Num_Vol),
        Nbr_Att = (SELECT COUNT(*) FROM Voyages WHERE Num_Vol = @Num_Vol);

END;


EXEC UpdateSeatsCountForFlight @Num_Vol = 123;
```

This stored procedure updates the Nbr_Res and Nbr_Att columns in the Vols table for the specified flight number (@Num_Vol) by counting the number of reserved seats from the Ligne_Reservation table and the number of assigned seats from the Voyages table.

Execice 21:

This stored procedure calculates the "Category" field for a given passenger based on specific criteria related to their travel activity during the current year. The "Category" information is stored in the "Category" column of the Passagers table.

```
CREATE PROCEDURE CalculatePassengerCategory
    @Code_Passager INT
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @Category VARCHAR(20);

    SELECT @Category =
        CASE
            WHEN (SELECT COUNT(*) FROM Voyages WHERE Code_Passager =
@Code_Passager) > 20
                 AND (SELECT SUM(Prix_Total) FROM Reservations WHERE Code_Passager
= @Code_Passager) > 200000
                 THEN 'Très Actif'
```

```
            WHEN (SELECT COUNT(*) FROM Voyages WHERE Code_Passager =
@Code_Passager) > 20
                 THEN 'Actif'
            ELSE 'Moyen'
        END;

    UPDATE Passagers
    SET Categorie = @Category
    WHERE Code_Passager = @Code_Passager;

END;


EXEC CalculatePassengerCategory @Code_Passager = 123;
```

This stored procedure calculates the "Category" field for the specified passenger (@Code_Passager) based on the number of trips and the total payment amount criteria specified. The calculated category is then updated in the Passagers table.

Exercice 22 :

This stored procedure calculates the number of trips made by each passenger.

```
CREATE PROCEDURE CalculateNumberOfTripsPerPassenger
AS
BEGIN
    SET NOCOUNT ON;

    SELECT Code_Passager, COUNT(*) AS NumberOfTrips
    FROM Voyages
    GROUP BY Code_Passager;
END;


EXEC CalculateNumberOfTripsPerPassenger;
```

This stored procedure retrieves the count of trips for each passenger from the Voyages table and returns the results grouped by the passenger's code (Code_Passager).

Execice 23:

```
CREATE PROCEDURE CalculateCostOfFlights
AS
BEGIN
    SET NOCOUNT ON;

    UPDATE Vols
    SET Cost = (SELECT SUM(Prix_Vol) FROM Vols AS v WHERE v.Num_Vol =
Vols.Num_Vol);
END;
```

```
EXEC CalculateCostOfFlights;
```

Exercice 25:

This stored procedure retrieves the pilots who have piloted more than a given percentage of the company's airplanes.

```
CREATE PROCEDURE GetPilotsExceedingPercentage
    @Percentage DECIMAL(5, 2)
AS
BEGIN
    SET NOCOUNT ON;

    SELECT P.Num_Pilote, P.Nom_Pilote, P.Prenom_Pilote
    FROM Pilotes P
    INNER JOIN (
        SELECT Code_Pilote, COUNT(*) AS Num_Avions
        FROM Vols
        GROUP BY Code_Pilote
    ) AS A ON P.Num_Pilote = A.Code_Pilote
    CROSS JOIN (
        SELECT COUNT(*) AS Total_Avions
        FROM Avions
    ) AS T
    WHERE (CONVERT(DECIMAL(5, 2), A.Num_Avions) / CONVERT(DECIMAL(5, 2),
T.Total_Avions)) > @Percentage / 100;
END;


EXEC GetPilotsExceedingPercentage @Percentage = 20;
```

This stored procedure retrieves the pilots who have piloted more than the specified percentage of the company's airplanes. It calculates the percentage by counting the number of airplanes each pilot has flown and comparing it to the total number of airplanes in the company.

Exercice 26:

This stored procedure adds and initializes three new columns (NbrAvions, NbrVoyages, and Statut) to the Pilotes table.

```
CREATE PROCEDURE AddInitializePilotColumns
AS
BEGIN
    SET NOCOUNT ON;

    -- Add new columns
    ALTER TABLE Pilotes
    ADD NbrAvions INT,
```

```
        NbrVoyages INT,
        Statut VARCHAR(20);

    -- Initialize NbrVoyages
    UPDATE Pilotes
    SET NbrVoyages = (SELECT COUNT(*) FROM Voyages WHERE Voyages.Code_Pilote =
Pilotes.Num_Pilote);

    -- Initialize NbrAvions
    UPDATE Pilotes
    SET NbrAvions = (SELECT COUNT(*) FROM Vols WHERE Vols.Code_Pilote =
Pilotes.Num_Pilote);

    -- Initialize Statut
    UPDATE Pilotes
    SET Statut =
        CASE
            WHEN NbrAvions > 0 AND (NbrAvions / (SELECT COUNT(*) FROM Avions) *
100) > 50 THEN 'Expert'
            WHEN NbrAvions > 0 AND (NbrAvions / (SELECT COUNT(*) FROM Avions) *
100) <= 50 AND (NbrAvions / (SELECT COUNT(*) FROM Avions) * 100) >= 5 THEN
'Qualifie'
            ELSE 'Débiteur'
        END;
END;


EXEC AddInitializePilotColumns;
```

This stored procedure adds three new columns (NbrAvions, NbrVoyages, and Statut) to the Pilotes table and initializes them as follows:

- NbrAvions: Initializes to 0.
- NbrVoyages: Initializes to 0.
- Statut: Initializes to 'Inactif'.

Exercice 27:

This stored procedure retrieves all possible tickets sorted in descending order of price for a given departure city, arrival city, and number of stops.

```
CREATE PROCEDURE GetAvailableTickets
    @DepartureCity VARCHAR(100),
    @ArrivalCity VARCHAR(100),
    @NumStops INT
AS
BEGIN
    SET NOCOUNT ON;

    SELECT B.Num_Billet, B.Num_Reservation, R.Prix_Total
    FROM Billets B
```

```
    INNER JOIN Reservations R ON B.Num_Reservation = R.Num_Reservation
    INNER JOIN Ligne_Reservation LR ON R.Num_Reservation = LR.Num_Reservation
    INNER JOIN Vols V ON LR.Num_Vol = V.Num_Vol
    WHERE V.Ville_Depart = @DepartureCity
        AND V.Ville_Arrivee = @ArrivalCity
        AND V.NumStops = @NumStops
    ORDER BY R.Prix_Total DESC;
END;

EXEC GetAvailableTickets @DepartureCity = 'Tanger', @ArrivalCity = 'Oujda',
@NumStops = 1;
```

This stored procedure retrieves all possible tickets for a given departure city, arrival city, and number of stops, sorted in descending order of price.

Exercice 28:

This trigger controls the availability of a seat in an airplane for a given voyage and passenger. It uses two functions, Complet to test if the voyage is full and Occuper to test if the seat number is occupied. If the voyage is not full and the seat is occupied, the trigger automatically suggests an available seat number.

```
CREATE TRIGGER CheckSeatAvailability
ON Voyages
FOR INSERT
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @NumVol INT, @CodePassager INT, @NumPlace INT;

    SELECT @NumVol = Num_Vol, @CodePassager = Code_Passager, @NumPlace = Num_place
    FROM inserted;

    IF NOT (dbo.Complet(@NumVol) = 1 AND dbo.Occuper(@NumVol, @NumPlace) = 0)
    BEGIN
        DECLARE @NewPlace INT;
        SET @NewPlace = dbo.GetAvailableSeat(@NumVol);

        PRINT 'Seat ' + CAST(@NumPlace AS VARCHAR) + ' is occupied. Suggested seat:
' + CAST(@NewPlace AS VARCHAR);
    END;
END;
```

Exercice 29:

This trigger ensures that the names and surnames of passengers are inserted in uppercase and checks the uniqueness of the key.

```
CREATE TRIGGER UpperCasePassengerNames
ON Passagers
INSTEAD OF INSERT
AS
BEGIN
    SET NOCOUNT ON;
    CREATE TABLE #TempPassengers (
        Code_Passager INT PRIMARY KEY,
        Nom_Passager NVARCHAR(50),
        Pre_Passager NVARCHAR(50),
        Num_Passport NVARCHAR(50),
        Categorie NVARCHAR(50),
        Num_Tel NVARCHAR(50)
    );

    INSERT INTO #TempPassengers (Code_Passager, Nom_Passager, Pre_Passager,
Num_Passport, Categorie, Num_Tel)
    SELECT Code_Passager, UPPER(Nom_Passager), UPPER(Pre_Passager), Num_Passport,
Categorie, Num_Tel
    FROM inserted;

    IF NOT EXISTS (
        SELECT 1
        FROM Passagers p
        INNER JOIN #TempPassengers t ON p.Code_Passager = t.Code_Passager
    )
    BEGIN
        INSERT INTO Passagers (Code_Passager, Nom_Passager, Pre_Passager,
Num_Passport, Categorie, Num_Tel)
        SELECT Code_Passager, Nom_Passager, Pre_Passager, Num_Passport, Categorie,
Num_Tel
        FROM #TempPassengers;
    END
    ELSE
    BEGIN
        PRINT 'Error: Duplicate key detected. Insertion aborted.';
    END;

    DROP TABLE #TempPassengers;
END;
```

Exercice 30:

This trigger controls the insertion of a voyage for a passenger and their reserved ticket, with pre-registration check of the corresponding flight.

```
CREATE TRIGGER ControlInsertVoyage
ON Voyages
FOR INSERT
AS
BEGIN
```

```
    SET NOCOUNT ON;

    DECLARE @Num_Vol INT, @Code_Passager INT, @Num_Billet INT;

    SELECT @Num_Vol = Num_Vol, @Code_Passager = Code_Passager, @Num_Billet =
Num_Billet
    FROM inserted;
    IF NOT EXISTS (SELECT 1 FROM Vols WHERE Num_Vol = @Num_Vol)
    BEGIN
        PRINT 'Error: The flight does not exist. Insertion aborted.';
        ROLLBACK TRANSACTION;
        RETURN;
    END;
    IF NOT EXISTS (SELECT 1 FROM Billets WHERE Num_Billet = @Num_Billet AND
Code_Passager = @Code_Passager)
    BEGIN
        PRINT 'Error: The passenger does not have the reserved ticket. Insertion
aborted.';
        ROLLBACK TRANSACTION;
        RETURN;
    END;
    PRINT 'Voyage inserted successfully.';
END;
```

Exercice 31 :

This trigger automatically updates the Status, NbrAvions, and NbrVoyages corresponding to a pilot upon insertion of a row into the Voyages table.

```
CREATE TRIGGER UpdatePilotStatsOnVoyageInsert
ON Voyages
AFTER INSERT
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @Num_Pilote INT;

    -- Get the pilot code from the inserted row
    SELECT @Num_Pilote = v.Code_Pilote
    FROM inserted v;

    -- Update the Status, NbrAvions, and NbrVoyages for the pilot
    UPDATE Pilotes
    SET
        Statut = CASE
                    WHEN NbrAvions > 0 AND NbrVoyages > 0 THEN 'Expert'
                    WHEN NbrAvions BETWEEN 0.5 * (SELECT COUNT(*) FROM Avions) AND
0.05 * (SELECT COUNT(*) FROM Avions) THEN 'Qualifie'
                    ELSE 'Debiteur'
```

```
                END,
        NbrAvions = (SELECT COUNT(DISTINCT v.Code_Avion) FROM Voyages v WHERE
v.Code_Pilote = @Num_Pilote),
        NbrVoyages = (SELECT COUNT(*) FROM Voyages WHERE Code_Pilote = @Num_Pilote)
    WHERE Num_Pilote = @Num_Pilote;
END;
```

This trigger automatically updates the Status, NbrAvions, and NbrVoyages corresponding to a pilot upon insertion of a row into the Voyages table. It calculates the pilot's status based on the number of airplanes and flights they have piloted and updates the corresponding columns accordingly.

Exercice 32:

This trigger causes an error when inserting a tuple into the Voyage table if the number of allocated seats exceeds the capacity of the airplane.

```
CREATE TRIGGER CheckSeatCapacityOnVoyageInsert
ON Voyage
FOR INSERT
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @Num_Vol INT, @Num_Avion INT, @Num_Place INT, @Capacity INT;

    SELECT @Num_Vol = Num_Vol, @Num_Avion = Code_Avion, @Num_Place = Num_Place
    FROM inserted;

    -- Get the capacity of the airplane
    SELECT @Capacity = Nbr_Place
    FROM Avions
    WHERE Num_Avion = @Num_Avion;

    -- Check if the number of allocated seats exceeds the capacity of the airplane
    IF @Num_Place > @Capacity
    BEGIN
        RAISEERROR('Error: Number of allocated seats exceeds the capacity of the
airplane.', 16, 1);
        ROLLBACK TRANSACTION;
        RETURN;
    END;
END;
```

This trigger causes an error when inserting a tuple into the Voyage table if the number of allocated seats exceeds the capacity of the airplane. It checks the capacity of the airplane associated with the voyage and raises an error if the number of allocated seats exceeds the capacity.

Exercice 33:

This trigger causes an error when inserting a tuple into the Voyage table if the number of allocated seats exceeds the capacity of the airplane.

```
CREATE TRIGGER UpdatePilotStatsOnVoyageInsert
ON Voyages
AFTER INSERT
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @Num_Pilote INT, @NbrVoyages INT, @NbrAvions INT, @Statut VARCHAR(20);

    SELECT TOP 1 @Num_Pilote = Code_Pilote
    FROM inserted;

    -- Calculate the number of flights for the pilot
    SELECT @NbrVoyages = COUNT(*)
    FROM Voyages
    WHERE Code_Pilote = @Num_Pilote;

    -- Calculate the number of airplanes for the pilot
    SELECT @NbrAvions = COUNT(DISTINCT Code_Avion)
    FROM Voyages
    WHERE Code_Pilote = @Num_Pilote;

    -- Determine the status of the pilot
    IF @NbrAvions > 0
        SET @Statut = CASE
                        WHEN @NbrAvions > 0.5 * (SELECT COUNT(DISTINCT Num_Avion)
FROM Avions) THEN 'Expert'
                        WHEN @NbrAvions BETWEEN 0.05 * (SELECT COUNT(DISTINCT
Num_Avion) FROM Avions) AND 0.5 * (SELECT COUNT(DISTINCT Num_Avion) FROM Avions)
THEN 'Qualifie'
                        ELSE 'Débiteur'
                      END;
    ELSE
        SET @Statut = 'Débiteur';

    -- Update the pilot stats
    UPDATE Pilotes
    SET NbrVoyages = @NbrVoyages,
        NbrAvions = @NbrAvions,
        Statut = @Statut
    WHERE Num_Pilote = @Num_Pilote;
END;
```

This trigger automatically updates the Status, NbrAvions, and NbrVoyages corresponding to a pilot upon insertion of a row into the Voyages table. It calculates the pilot's

status based on the number of airplanes and flights they have piloted and updates the corresponding columns accordingly.

Exercice 34:

```
CREATE TRIGGER CascadeDeletePassenger
ON Passagers
INSTEAD OF DELETE
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @DeletedPassenger INT;
    SELECT @DeletedPassenger = Code_Passager FROM deleted;

    -- Delete related records from Reservations table
    DELETE FROM Reservations WHERE Code_Passager = @DeletedPassenger;

    -- Delete related records from Ligne_Reservation table
    DELETE FROM Ligne_Reservation WHERE Num_Reservation IN (SELECT Num_Reservation
FROM deleted);

    -- Delete related records from Billets table
    DELETE FROM Billets WHERE Num_Reservation IN (SELECT Num_Reservation FROM
deleted);

    -- Delete related records from Voyages table
    DELETE FROM Voyages WHERE Code_Passager = @DeletedPassenger;

    -- Perform the actual deletion of the passenger
    DELETE FROM Passagers WHERE Code_Passager = @DeletedPassenger;
END;
```

This trigger will ensure that when a passenger is deleted, all related records in other tables (Reservations, Ligne_Reservation, Billets, and Voyages) are also deleted before deleting the passenger record itself.

Exercice 35:

```
CREATE TRIGGER CascadeDeletePassenger
ON Passagers
INSTEAD OF DELETE
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @DeletedPassenger INT;
    SELECT @DeletedPassenger = Code_Passager FROM deleted;
```

```
    DELETE FROM Reservations WHERE Code_Passager = @DeletedPassenger;
    DELETE FROM Ligne_Reservation WHERE Num_Reservation IN (SELECT Num_Reservation
FROM deleted);
    DELETE FROM Billets WHERE Num_Reservation IN (SELECT Num_Reservation FROM
deleted);
    DELETE FROM Voyages WHERE Code_Passager = @DeletedPassenger;

    DELETE FROM Passagers WHERE Code_Passager = @DeletedPassenger;
END;
```

Exercice 36:

In this database enhancement, we introduce a trigger designed to automatically correct erroneous entries or modifications pertaining to passenger phone numbers. The trigger, implemented in SQL, acts as an intermediary safeguard, intercepting erroneous data entries or updates. It seamlessly rectifies inaccuracies by replacing all separator characters—be it hyphens or spaces—within a passenger's phone number with dots. Additionally, the trigger ensures uniformity by substituting the character 'O' with '0' (zero) in phone numbers. By seamlessly correcting discrepancies in real-time, this trigger enhances data integrity and consistency within the database, ensuring accurate representation of passenger contact information.

```
CREATE TRIGGER CorrectPhoneNumber
ON Passagers
AFTER INSERT, UPDATE
AS
BEGIN

    UPDATE Passagers
    SET Num_Tel = REPLACE(REPLACE(Num_Tel, '-', '.'), 'O', '0')
    WHERE EXISTS (
        SELECT 1
        FROM inserted
        WHERE inserted.Code_Passager = Passagers.Code_Passager
    );
END;
```

Exercice 37:

In this database enhancement, we introduce a trigger designed to ensure the validity of input formats for departure and arrival dates. Implemented in SQL, this trigger acts as a gatekeeper, validating the format of date entries against predefined criteria. A valid date should consist solely of digits and slashes ('/'), with a maximum length of 10 characters. Any occurrence of the characters 'O' or 'Q' is automatically replaced with the digit '0' (zero) to maintain consistency and accuracy in date representation. By enforcing these standards, the trigger enhances data integrity and reliability within the database, preventing erroneous date entries from compromising the system's functionality.

```
GO

CREATE TRIGGER ValidateDateInput
ON Vols
AFTER INSERT, UPDATE
AS
BEGIN

    UPDATE Vols
    SET Date_Depart = REPLACE(REPLACE(Date_Depart, 'O', '0'), 'Q', '0'),
        Date_Arrivee = REPLACE(REPLACE(Date_Arrivee, 'O', '0'), 'Q', '0')
    WHERE (LEN(Date_Depart) <= 10 AND Date_Depart NOT LIKE '%[^0-9/]%'
        AND CHARINDEX('/', REPLACE(REPLACE(Date_Depart, 'O', '0'), 'Q', '0')) <= 2)
        AND (LEN(Date_Arrivee) <= 10 AND Date_Arrivee NOT LIKE '%[^0-9/]%'
        AND CHARINDEX('/', REPLACE(REPLACE(Date_Arrivee, 'O', '0'), 'Q', '0')) <=
2);
END;
```

Exercice 38:

In this database enhancement, we introduce a trigger designed to archive all deletion operations performed on the "Voyages" table. Implemented in SQL, this trigger acts as a safeguard, capturing and storing deleted data into a dedicated archive table named "Voyages_Archive." By leveraging this trigger, a historical record of deleted entries is maintained, preserving data integrity and enabling future analysis and auditing. Each deleted record is accompanied by a timestamp, providing valuable insights into the deletion events' timing and sequence. Through this introduction, we embark on a journey to enhance data management and auditability within the database environment.

```
GO
CREATE TABLE Voyages_Archive (
    Code_Passager INT,
    Num_Billet INT,
    Num_Vol INT,
    Num_Place INT,
    Date_Archive DATETIME
);
GO
CREATE TRIGGER ArchiveVoyagesDeletion
ON Voyages
AFTER DELETE
AS
BEGIN
    INSERT INTO Voyages_Archive (Code_Passager, Num_Billet, Num_Vol, Num_Place,
Date_Archive)
    SELECT Code_Passager, Num_Billet, Num_Vol, Num_Place, GETDATE()
    FROM deleted;
END;
```

Exrcice 39:

In this database enhancement, we introduce a trigger designed to archive deletion operations on the "Reservations" table based on the nature of the reservation's processing: either cancelled or validated. A validated reservation can only be deleted after 10 days from the travel date, while a cancelled reservation can only be deleted after the validation date. Implemented in SQL, this trigger ensures that deleted reservation data is captured and stored in an archive table named "Reservations_Archive." Each archived record includes the reservation number, archive date, and the action taken (cancelled or validated). Through this introduction, we aim to improve data management and tracking capabilities within the database environment.

```
GO
CREATE TABLE Reservations_Archive (
    Num_Reservation INT,
    Date_Archive DATETIME,
    Action NVARCHAR(50)
);
GO
CREATE TRIGGER ArchiveReservationsDeletion
ON Reservations
AFTER DELETE
AS
BEGIN
    INSERT INTO Reservations_Archive (Num_Reservation, Date_Archive, Action)
    SELECT Num_Reservation, GETDATE(),
           CASE
                WHEN d.Etat_Reservation = 'Annulée' THEN 'Cancelled'
                WHEN d.Etat_Reservation = 'Validée' AND d.Date_Validation <=
DATEADD(day, -10, GETDATE()) THEN 'Validated'
           END
    FROM deleted d;
END;
```

Exercice 40:

In this database enhancement, we aim to create a view named "ReservationValidees" by selecting only the validated reservations from the "reservations" table for the agency with ID 001. The attributes of the view include idzone, type, caract, and dist. Additionally, we need to determine whether this view is editable. If it's not editable, we'll explore potential strategies to make it editable.

```
CREATE VIEW ReservationValidees AS
SELECT idzone, type, caract, dist
FROM reservations
WHERE Code_Agence = '001' AND Etat_Reservation = 'Validée';
```

Exercice 41:

In this task, we delve into the process of inserting values into a B+ tree, a widely-used data structure known for its efficiency in organizing and retrieving data. We aim to illustrate and describe step-by-step the insertion process into a B+ tree with three keys. The values to be inserted are provided as a list, including 3, 6, 44, 5, 2, 77, 1, 7, 9, 91, 33, 43, 0, 27, 88, 55, 54, 56, 57, 52, 44, 24, 25, 26, 27, 98, 99, 4, 6, and 8. Through this exploration, we seek to gain insights into the intricacies of B+ tree organization and the insertion mechanism, facilitating a deeper understanding of this fundamental data structure in computer science.

Exercice 42:

Begin by identifying the B+ tree containing all the values to be deleted.

Remove the value 3 from the tree. This entails locating the node containing the value 3 and removing it from the appropriate position within the node. Ensure that the tree's structural integrity is maintained after the deletion.

Proceed to remove the value 24 from the tree. Similar to the previous step, locate the node containing the value 24 and remove it from the node while ensuring the tree's properties are preserved.

Next, remove the value 25 from the tree. Locate the node containing the value 25 and delete it, adjusting the tree structure accordingly.

Continue this process for each subsequent value in the list, ensuring to remove them one by one while maintaining the B+ tree's properties at each step.

Repeat the deletion process for values 98, 6, 44, 0, 27, 88, 55, 7, 56, 57, 77, 9, 91, 33, 43, 5, 52, 44, 26, 27, 99, 54, 4, 6, 8, 2, and finally 1.

After removing all the specified values, ensure to verify that the B+ tree remains properly balanced and structured, with no violations of its properties.

By following these steps, each value is systematically removed from the B+ tree while maintaining its structural integrity, ensuring a smooth and controlled deletion process.