

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

# 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.

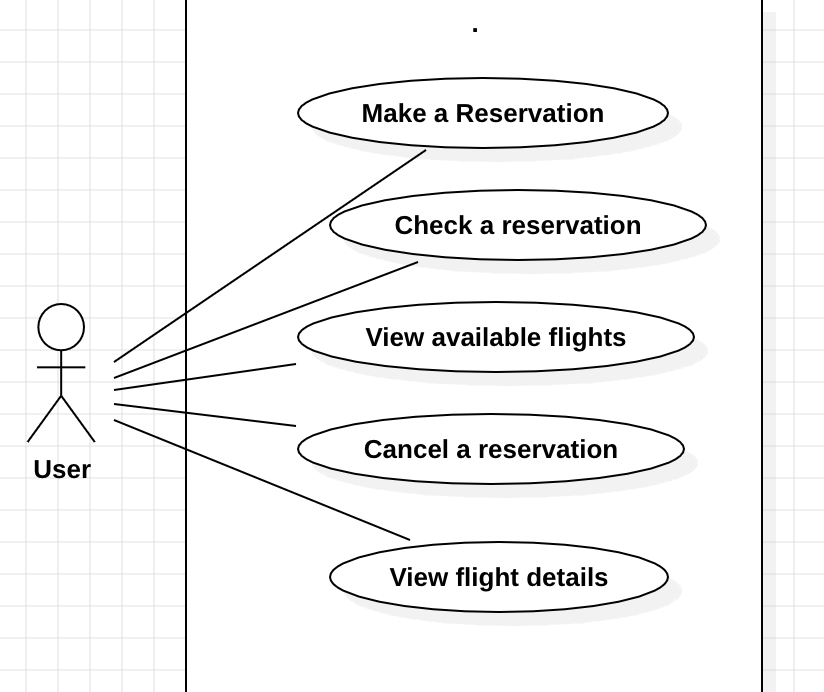
# Conception Phase:

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.

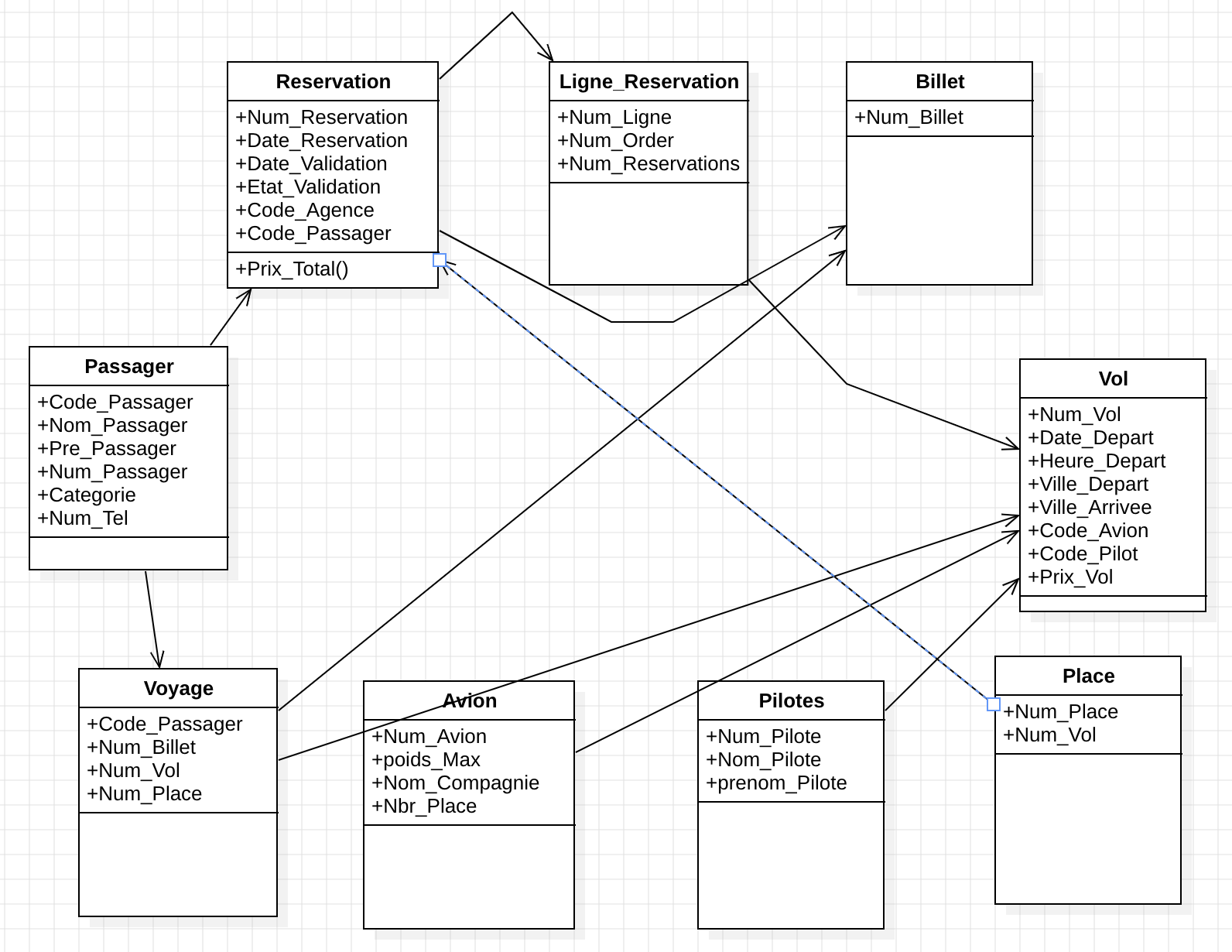
This chapter serves as a bridge between theoretical concepts and practical application, showcasing our ability to transform design blueprints into functional database systems. Through a series of exercises and scenarios, we demonstrate our proficiency in SQL query formulation, database manipulation,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



# 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;



## 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



## 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');



## 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:



## 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');



## 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;



## 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';



## 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;



## 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;

## 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;



## 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;