CSE 581: INTRODUCTION TO DATABASE MANAGEMENT SYSTEM

Project 2- Human Resource Database for the University Medical Center

Title: "Optimizing Staff Management: UMC's Human Resources Database"

Healthcare systems, such as those in New York, are intricate. UMC requires sophisticated databases for tasks such as recruitment. Our project focuses on developing a tailored Human Resources Database to streamline staff management, ensuring operational efficiency and alignment with strategic objectives. This initiative aims to enhance UMC's ability to deliver high-quality patient care while maintaining optimal staffing levels and expertise.

PROJECT 2
RAJ NANDINI
SUID: 949041747
NetId: rajnandi@syr.edu
SUBMISSION DATE: 05-01-2024

Abstract

Healthcare in New York is like a giant puzzle, with many pieces that need to fit together just right for everything to work smoothly. We have access to a lot of information about healthcare in New York, thanks to databases like State of New York | Open Data Health and Health Facility General Information | Socrata API Foundry. These databases help us understand how hospitals operate and how they hire new staff.

Hospitals are really important in healthcare—they're like the heart of the system. They need to have good systems in place to manage everything, including hiring new people. While taking care of patients is the main goal, hospitals also have to make sure they run efficiently and that they have the right people in the right jobs.

Recruiting new staff is a big part of keeping a hospital running smoothly. Hospitals need to make sure they have enough people working and that those people have the right skills for the job. This means working closely with different departments and other people involved in healthcare to make sure they're hiring the right people.

Our project focuses on finding ways to make the hiring process better for hospitals in New York. We want to use data to help hospitals hire the right people more efficiently. By looking at the information available in the databases, we can figure out what hospitals need and how they can find the best people for the job.

Our goal is to make the recruitment process easier for hospitals and to help them find the right staff faster. This will make hospitals run better and, most importantly, it will help them take better care of their patients.

In this study, we'll explore different ways hospitals can use data to improve their recruitment processes. We'll look at how hospitals can use information from the databases to understand what kind of staff they need and where they can find them. We'll also explore how hospitals can work with different departments and stakeholders to make sure they're hiring the right people.

By using a data-driven approach, we believe we can make a real difference in how hospitals hire new staff. This will not only benefit the hospitals themselves but also the people they care for—patients across New York. Our hope is that by improving the recruitment process, we can contribute to making healthcare in New York even better.

Table Of Contents:

INTRODUCTION

Imagine a bustling hub of activity, where every day brings new challenges and opportunities to make a difference in people's lives. That's exactly what a hospital like the University Medical Center (UMC) is—a place where healthcare heroes work tirelessly to ensure patients receive the best care possible. But behind the scenes, there's a lot going on to keep everything running smoothly, especially when it comes to hiring the right people.

Our project focuses on creating something essential for UMC: a Human Resources Database. It's like the brain of the hospital's recruitment process, helping to track and manage every step of hiring new staff. From the moment someone applies for a job to the day they become part of the UMC team, this database keeps everything organized and on track.

Think of it as a roadmap for recruitment. It helps HR teams keep tabs on every candidate, from their initial application to their final interview and beyond. It even helps manage things like scheduling interviews, coordinating travel arrangements, and processing reimbursement requests.

By building this database, we're not just creating a tool for HR—we're making life easier for everyone involved in the hiring process. From interviewers to onboarding specialists, everyone will have the information they need right at their fingertips, making the whole process smoother and more efficient.

But it's not just about making things easier for staff. It's also about ensuring that UMC continues to deliver top-notch care to its patients. By hiring the best people for the job, UMC can maintain its reputation as a world-class healthcare provider, ensuring that everyone who walks through its doors receives the care and attention they deserve.

So, as we embark on this project, we're not just building a database—we're building a foundation for excellence. We're creating a tool that will help UMC continue to thrive and grow, ensuring that it remains at the forefront of healthcare innovation for years to come.

A. DESIGN

"Strategic Database Design for Streamlined Healthcare Recruitment"

In this project, a meticulously crafted database tailored for the recruitment branch of the Human Resources Department at the University Medical Center was designed, implemented, and rigorously tested. Leveraging insights from the complex healthcare systems of New York, key administrative departments were identified to provide a foundational understanding of hospital operations. The project specifications outlined a comprehensive recruitment process, guiding the creation of tables and attributes to track candidate statuses, interview details, evaluations, and more. Through systematic steps including logic development, potential integrity assessment, and normalization, the database design ensured data integrity, minimized redundancy, and facilitated efficient query performance. The final design, exemplified by a descriptive E/R diagram, promises to optimize talent acquisition processes, supporting operational efficiency and effective patient care within healthcare institutions.

ENTITY-RELATIONSHIP MODEL

An ER (Entity-Relationship) diagram is a graphical representation of the entities, attributes, and relationships within a database system. It serves as a visual tool for designing and understanding the structure of a database. The components of an ER diagram include:

Entities: Entities are the objects or concepts represented in the database. They can be tangible, such as a person or a product, or intangible, such as an event or a concept. In a university database, entities might include students, courses, instructors, and departments.

Attributes: Attributes are the properties or characteristics that describe entities. They provide details about the entities and help distinguish one entity from another. For example, attributes of a student entity might include name, ID, address, and major.

Relationships: Relationships define how entities are related to each other within the database. They describe the connections and associations between entities. In the university database example, the relationship between students and courses might indicate that a student can enroll in many courses, and a course can have many students. Relationships can be one-to-one, one-to-many, or many-to-many.

ER diagrams use various symbols and notations to represent entities, attributes, and relationships visually. Entities are typically represented as rectangles, attributes as ovals, and relationships as lines connecting entities.

In this project, an ER diagram plays a crucial role in database design and development:

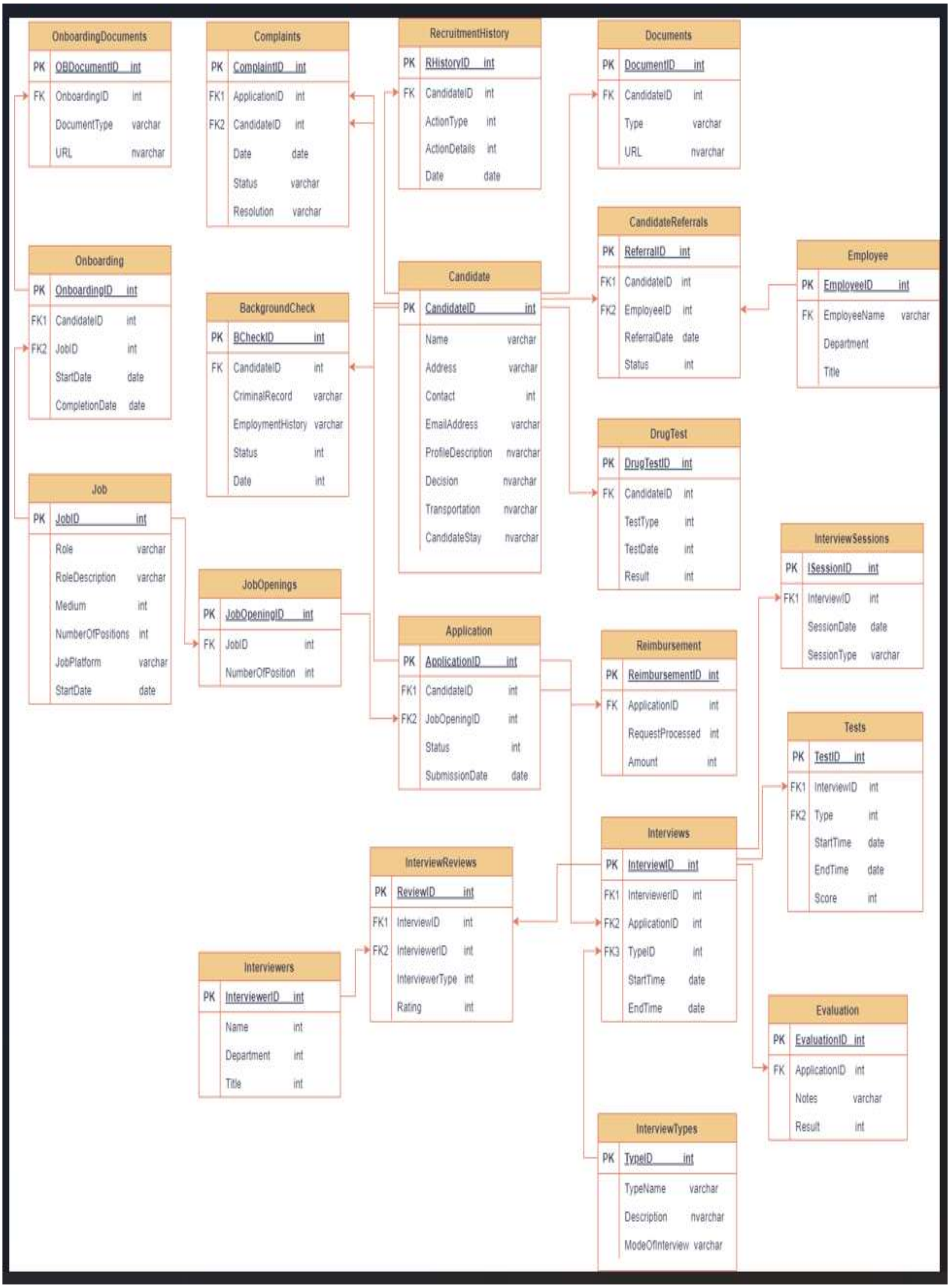
- The ER diagram helps in designing the structure of the database by identifying the entities, their attributes, and the relationships between them. It provides a blueprint for organizing data in a logical and efficient manner.

- ER diagrams serve as a communication tool between stakeholders involved in the project, including developers, designers, and clients. They help convey the database structure and requirements in a clear and understandable way.

- By visualizing the relationships between entities, ER diagrams help ensure data integrity within the database. They assist in defining constraints and rules that govern how data can be stored and manipulated, thereby preventing inconsistencies and errors.

- Understanding the database structure through ER diagrams facilitates writing queries and performing data analysis. It provides insights into how data is related and organized, enabling efficient retrieval and manipulation of information.

In summary, an ER diagram is essential in a project as it serves as a foundational tool for database design, communication, data integrity, and query support. It helps ensure that the database is structured in a way that meets the business requirements and supports efficient data management.


OnboardingDocuments
PK OBDocumentID int
FK OnboardingID int
DocumentType varchar
URL nvarchar
Complaints
PK ComplaintID int
FK1 ApplicationID int
FK2 CandidateID int
Date date
Status varchar
Resolution varchar
RecruitmentHistory
PK RHistoryID int
FK CandidateID int
ActionType int
ActionDetails int
Date date
Documents
PK DocumentID int
FK CandidateID int
Type varchar
URL nvarchar
CandidateReferrals
PK ReferralID int
FK1 CandidateID int
FK2 EmployeeID int
ReferralDate date
Status int
Employee
PK EmployeeID int
FK EmployeeName varchar
Department
Title
Onboarding
PK OnboardingID int
FK1 CandidateID int
FK2 JobID int
StartDate date
CompletionDate date
BackgroundCheck
PK BCheckID int
FK CandidateID int
CriminalRecord varchar
EmploymentHistory varchar
Status int
Date int
Candidate
PK CandidateID int
Name varchar
Address varchar
Contact int
EmailAddress varchar
ProfileDescription nvarchar
Decision nvarchar
Transportation nvarchar
CandidateStay nvarchar
DrugTest
PK DrugTestID int
FK CandidateID int
TestType int
TestDate int
Result int
Job
PK JobID int
Role varchar
RoleDescription varchar
Medium int
NumberOfPositions int
JobPlatform varchar
StartDate date
JobOpenings
PK JobOpeningID int
FK JobID int
NumberOfPosition int
InterviewSessions
PK ISessionID int
FK1 InterviewID int
SessionDate date
SessionType varchar
Application
PK ApplicationID int
FK1 CandidateID int
FK2 JobOpeningID int
Status int
SubmissionDate date
Reimbursement
PK ReimbursementID int
FK ApplicationID int
RequestProcessed int
Amount int
Tests
PK TestID int
FK1 InterviewID int
FK2 Type int
StartTime date
EndTime date
Score int
Interviews
PK InterviewID int
FK1 InterviewerID int
FK2 ApplicationID int
FK3 TypeID int
StartTime date
EndTime date
InterviewReviews
PK ReviewID int
FK1 InterviewID int
FK2 InterviewerID int
InterviewerType int
Rating int
Interviewers
PK InterviewerID int
Name int
Department int
Title int
Evaluation
PK EvaluationID int
FK ApplicationID int
Notes varchar
Result int
InterviewTypes
PK TypeID int
TypeName varchar
Description nvarchar
ModeOfInterview varchar

	Entity	Attributes	Explanation
1.	Job	**JobID (PK),** Title, Description, Type, Medium, StartDate, NumberOfPositions, JobPlatform	- Relationships: **- One-to-Many relationship with JobOpenings** (JobID -> JobOpenings.JobID) **- One-to-Many relationship with Onboarding** (JobID -> Onboarding.JobID)
2.	JobOpenings	**JobOpeningID (PK),** JobID (FK), NumberOfPositions	- Relationships: **- Many-to-One relationship with Job** (JobID -> Job.JobID) **- One-to-Many relationship with Application** (JobOpeningID -> Application.JobOpeningID)
3.	Candidate	**CandidateID (PK),** Name, Address EmailAddress, Contact, ProfileDescription	- Relationships: **- One-to-Many relationship with Documents** (CandidateID -> Documents.CandidateID) **- One-to-Many relationship with Application** (CandidateID -> Application.CandidateID) **- One-to-One relationship with BackgroundCheck** (CandidateID -> BackgroundCheck.CandidateID) **- One-to-One relationship with DrugTest** (CandidateID -> DrugTest.CandidateID) **- One-to-Many relationship with Complaints** (CandidateID -> Complaints.CandidateID) **- One-to-Many relationship with RecruitmentHistory** (CandidateID -> RecruitmentHistory.CandidateID) **- One-to-Many relationship with Onboarding** (CandidateID -> Onboarding.CandidateID)
4.	Documents	**DocumentID (PK),** CandidateID (FK), Type, URL	- Relationships: **- Many-to-One relationship with Candidate** (CandidateID -> Candidate.CandidateID)
5.	Application	**ApplicationID (PK),** CandidateID (FK), JobOpeningID (FK), Status, DateSubmitted	- Relationships: **- Many-to-One relationship with Candidate** (CandidateID -> Candidate.CandidateID) **- Many-to-One relationship with JobOpenings** (JobOpeningID -> JobOpenings.JobOpeningID) **- One-to-Many relationship with Interviews** (ApplicationID -> Interviews.ApplicationID) **- One-to-Many relationship with Evaluation** (ApplicationID -> Evaluation.ApplicationID) **- One-to-Many relationship with Reimbursement** (ApplicationID -> Reimbursement.ApplicationID)

6.	Interviewers	**InterviewerID (PK),** Name, Department, Title	- Relationships: **- One-to-Many relationship with Interviews** (InterviewerID -> Interviews.InterviewerID)
7.	Interviews	**InterviewID (PK),** ApplicationID (FK), InterviewerID (FK), TypeID (FK), StartTime, EndTime	- Relationships: **- Many-to-One relationship with Application** (ApplicationID -> Application.ApplicationID) **- Many-to-One relationship with Interviewers** (InterviewerID -> Interviewers.InterviewerID) **- Many-to-One relationship with InterviewTypes** (TypeID -> InterviewTypes.TypeID) **- One-to-Many relationship with Tests** (InterviewID -> Tests.InterviewID) **- One-to-Many relationship with InterviewReviews** (InterviewID -> InterviewReviews.InterviewID)
8.	InterviewTypes	**TypeID (PK),** TypeName, Description	- Relationships: **- One-to-Many relationship with Interviews** (TypeID -> Interviews.TypeID)
9.	Tests	**TestID (PK),** InterviewID (FK), Type, StartTime, EndTime, Grade	- Relationships: **- Many-to-One relationship with Interviews** (InterviewID -> Interviews.InterviewID)
10.	InterviewReviews	**ReviewID (PK),** InterviewID (FK), ReviewerType, ReviewerID, ReviewText, Rating	- Relationships: **- Many-to-One relationship with Interviews** (InterviewID -> Interviews.InterviewID) - ReviewerID can be a foreign key referencing either Candidate.CandidateID or Interviewers.InterviewerID depending on ReviewerType
11.	BackgroundCheck	**CheckID (PK),** CandidateID (FK), CriminalRecord, EmploymentHistory, Status, Date	- Relationships: **- One-to-One relationship with Candidate** (CandidateID -> Candidate.CandidateID)
12.	DrugTest	**DrugTestID (PK),** CandidateID (FK), TestType, TestDate, Results	- Relationships: **- One-to-One relationship with Candidate** (CandidateID -> Candidate.CandidateID)
13.	Evaluation	**EvaluationID (PK),** ApplicationID (FK), Notes, Result	- Relationships: **- Many-to-One relationship with Application** (ApplicationID -> Application.ApplicationID)

14.	Reimbursement	**ReimbursementID (PK),** ApplicationID (FK), Request, Processed, Amount	- Relationships: **- Many-to-One relationship with Application** (ApplicationID -> Application.ApplicationID)
15.	Complaints	**ComplaintID (PK),** CandidateID (FK), ApplicationID (FK), Date, Status, Resolution	- Relationships: **- Many-to-One relationship with Candidate** (CandidateID -> Candidate.CandidateID) **- Many-to-One relationship with Application** (ApplicationID -> Application.ApplicationID)
16.	RecruitmentHistory	R**HistoryID (PK),** CandidateID (FK), ActionType, ActionDetails, Date	- Relationships: **- Many-to-One relationship with Candidate** (CandidateID -> Candidate.CandidateID)
17.	Onboarding	**OnboardingID (PK),** CandidateID (FK1), JobID (FK2), StartDate, CompletionDate	- Relationships: **- Many-to-One relationship with Candidate** (CandidateID -> Candidate.CandidateID) **- Many-to-One relationship with Job** (JobID -> Job.JobID)
18.	OnboardingDocuments	**DocumentID (PK),** OnboardingID (FK), DocumentType, URL	- Relationships: **- Many-to-One relationship with Onboarding** (OnboardingID -> Onboarding.OnboardingID)
19.	InterviewSessions	**SessionID (PK),** InterviewID (FK), SessionDate, SessionType	- Relationships: **- Many-to-One relationship with Interviews** (InterviewID -> Interviews.InterviewID)
20.	CandidateReferrals	**ReferralID (PK),** CandidateID (FK), ReferrerID, ReferralDate, Status	- Relationships: **- Many-to-One relationship with Candidate** (CandidateID -> Candidate.CandidateID) - ReferrerID can be a foreign key referencing either Candidate.CandidateID or Interviewers.InterviewerID depending on the referral context
21.	Employee	**EmployeeID (PK),** EmployeeName, Department, Title	**- One-to-Many relationship with CandidateReferral(EmployeeID)**

Figure: Entity-Relationship Table with Attributes

B. IMPLEMENTATION

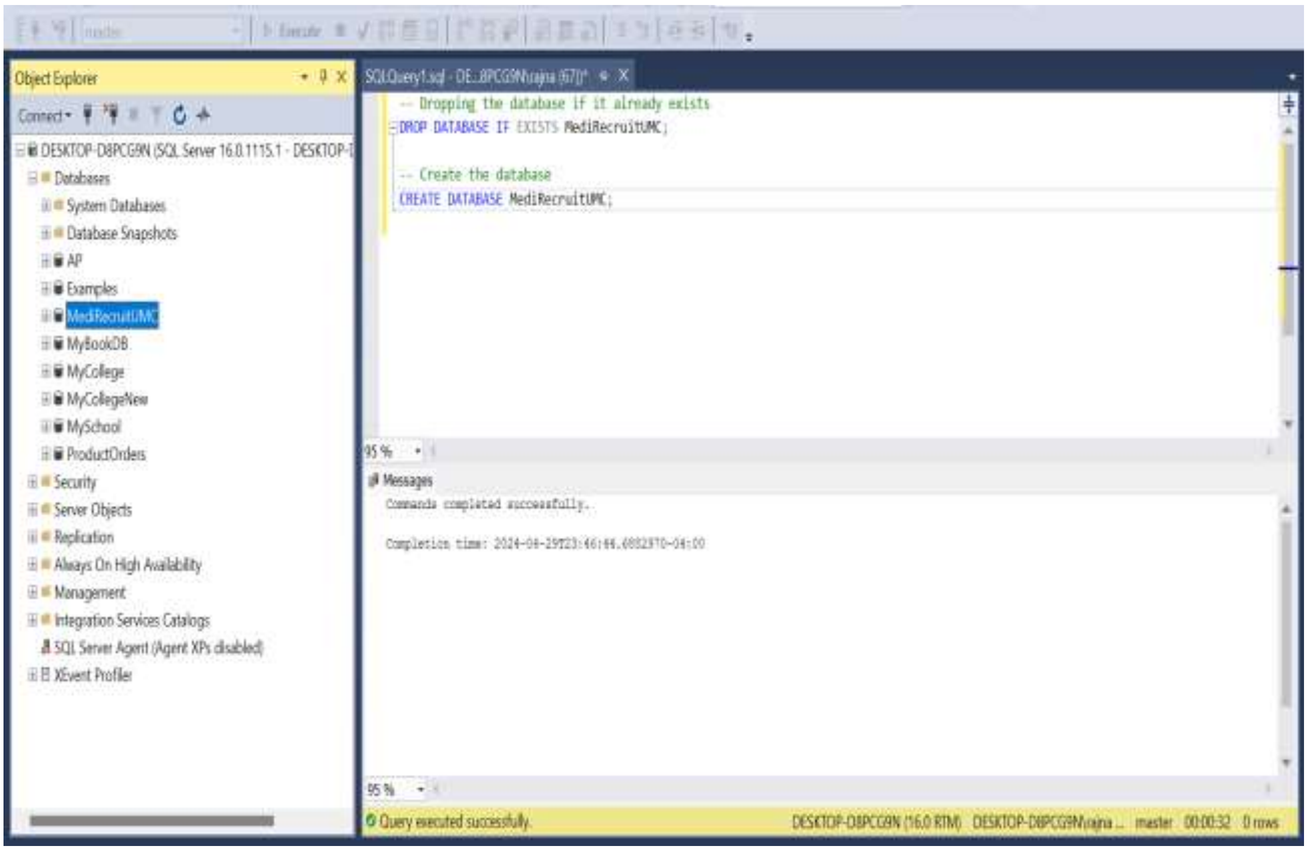
"Streamlining Healthcare Recruitment: A Comprehensive Database Implementation Approach"

This project outlines the design, implementation, and testing of a database system tailored for the recruitment branch of the Human Resources Department at a University Medical Center. In the complex landscape of healthcare systems, efficient recruitment processes are vital for maintaining staffing levels and expertise. The database tracks candidate statuses throughout the recruitment journey, encompassing application submission, multiple interview rounds, offer negotiations, onboarding, and beyond. By adhering to stringent data integrity measures and security protocols, the implemented system ensures accuracy, confidentiality, and accessibility of critical recruitment data. Additionally, the development of comprehensive business reports provides stakeholders with actionable insights into recruitment trends, offer acceptance rates, and process efficiency. This project not only enhances our understanding of healthcare recruitment operations but also underscores the significance of database management in optimizing organizational processes and decision-making.

DATABASE CREATIONS

SQL Query:

```
-- Dropping the database if it already exists
    DROP DATABASE IF EXISTS MediRecruitUMC;

-- Create the database
    CREATE DATABASE MediRecruitUMC;
```



```
-- Dropping the database if it already exists
DROP DATABASE IF EXISTS MediRecruitUMC;

-- Create the database
CREATE DATABASE MediRecruitUMC;
```

```
Messages
Commands completed successfully.

Completion time: 2024-04-29T23:46:44.6882970-04:00
```

Query executed successfully.   DESKTOP-D8PCG9N (16.0 RTM)   DESKTOP-D8PCG9N\rajna ...   master   00:00:32   0 rows

The above SQL code creates a database named '**MediRecruitUMC**' while dropping any existing database with the same name if present. This preventive measure eliminates redundancy and ensures database integrity.

The command 'DROP DATABASE IF EXISTS MediRecruitUMC;' first checks for the existence of the database and drops it if found, thereby avoiding conflicts. Subsequently, 'CREATE DATABASE MediRecruitUMC;' initiates the creation of the database.

This concise sequence guarantees a clean slate for the database, ensuring a fresh start for data management and system operations.

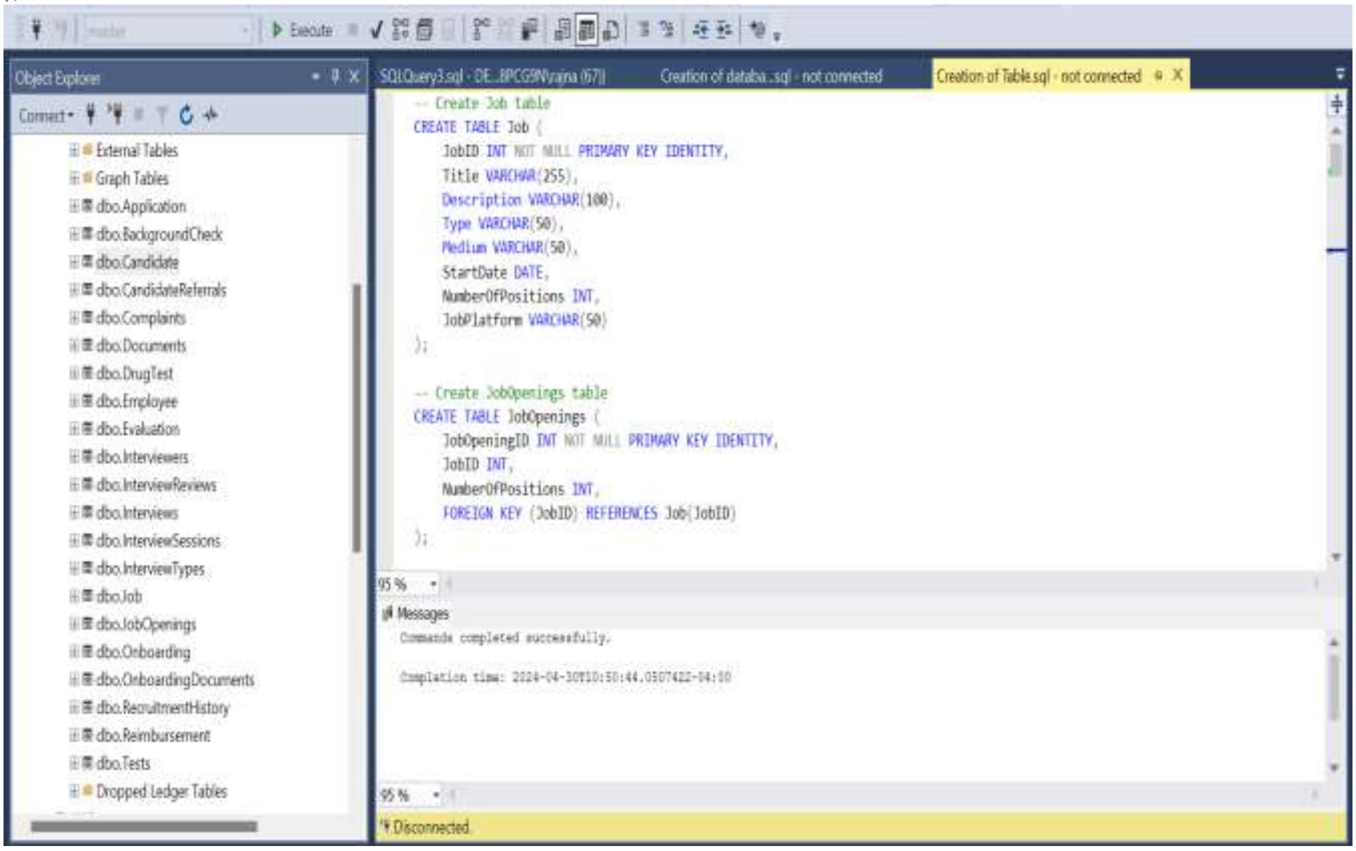
TABLES CREATION

SQL Query:

```
-- Create Job table
CREATE TABLE Job (
    JobID INT NOT NULL PRIMARY KEY IDENTITY,
    Title VARCHAR(255),
    Description VARCHAR(100),
    Type VARCHAR(50),
    Medium VARCHAR(50),
    StartDate DATE,
    NumberOfPositions INT,
    JobPlatform VARCHAR(50)
);

-- Create JobOpenings table
CREATE TABLE JobOpenings (
    JobOpeningID INT NOT NULL PRIMARY KEY IDENTITY,
    JobID INT,
    NumberOfPositions INT,
    FOREIGN KEY (JobID) REFERENCES Job(JobID)
);

-- Create Candidate table
CREATE TABLE Candidate (
    CandidateID INT NOT NULL PRIMARY KEY IDENTITY,
    Name VARCHAR(255),
            Address VARCHAR(250),
    Contact VARCHAR(20),
    ProfileDescription VARCHAR(100)
);

-- Create Documents table
CREATE TABLE Documents (
    DocumentID INT NOT NULL PRIMARY KEY IDENTITY,
    CandidateID INT,
    Type VARCHAR(50),
    URL TEXT,
    FOREIGN KEY (CandidateID) REFERENCES Candidate(CandidateID)
);

-- Create Application table
CREATE TABLE Application (
    ApplicationID INT NOT NULL PRIMARY KEY IDENTITY,
    CandidateID INT,
    JobOpeningID INT,
```

```sql
    Status VARCHAR(50),
    DateSubmitted DATE,
    FOREIGN KEY (CandidateID) REFERENCES Candidate(CandidateID),
    FOREIGN KEY (JobOpeningID) REFERENCES JobOpenings(JobOpeningID)
);

-- Create InterReviewers table
CREATE TABLE Interviewers (
    InterviewerID INT NOT NULL PRIMARY KEY IDENTITY,
    Name VARCHAR(255),
    Department VARCHAR(50),
    Title VARCHAR(50)
);

-- Create InterviewTypes table
CREATE TABLE InterviewTypes (
    TypeID INT NOT NULL PRIMARY KEY IDENTITY,
    TypeName VARCHAR(50),
    Description VARCHAR(255)
);

-- Create Interviews table
CREATE TABLE Interviews (
    InterviewID INT NOT NULL PRIMARY KEY IDENTITY,
    ApplicationID INT,
    InterviewerID INT,
    TypeID INT,
    StartTime DATETIME,
    EndTime DATETIME,
    FOREIGN KEY (ApplicationID) REFERENCES Application(ApplicationID),
    FOREIGN KEY (InterviewerID) REFERENCES Interviewers(InterviewerID),
    FOREIGN KEY (TypeID) REFERENCES InterviewTypes(TypeID)
);

-- Create Tests table
CREATE TABLE Tests (
    TestID INT NOT NULL PRIMARY KEY IDENTITY,
    InterviewID INT,
    Type VARCHAR(50),
    StartTime DATETIME,
    EndTime DATETIME,
    Grade VARCHAR(50),
    FOREIGN KEY (InterviewID) REFERENCES Interviews(InterviewID)
);

-- Create InterviewReviews table
CREATE TABLE InterviewReviews (
    ReviewID INT NOT NULL PRIMARY KEY IDENTITY,
    InterviewID INT,
    ReviewerType VARCHAR(255),
    ReviewerID INT,
    ReviewText TEXT,
    Rating INT,
    FOREIGN KEY (InterviewID) REFERENCES Interviews(InterviewID),
    FOREIGN KEY (ReviewerID) REFERENCES Candidate(CandidateID)
);

-- Create BackgroundCheck table
CREATE TABLE BackgroundCheck (
    CheckID INT NOT NULL PRIMARY KEY IDENTITY,
    CandidateID INT,
    CriminalRecord TEXT,
    EmploymentHistory TEXT,
    Status VARCHAR(50),
    Date DATE,
    FOREIGN KEY (CandidateID) REFERENCES Candidate(CandidateID)
);

-- Create DrugTest table
CREATE TABLE DrugTest (
    DrugTestID INT NOT NULL PRIMARY KEY IDENTITY,
    CandidateID INT,
    TestType VARCHAR(50),
    TestDate DATE,
    Results TEXT,
    FOREIGN KEY (CandidateID) REFERENCES Candidate(CandidateID)
);

-- Create Evaluation table
CREATE TABLE Evaluation (
    EvaluationID INT NOT NULL PRIMARY KEY IDENTITY,
    ApplicationID INT,
    Notes TEXT,
    Result VARCHAR(50),
    FOREIGN KEY (ApplicationID) REFERENCES Application(ApplicationID)
);

-- Create Reimbursement table
CREATE TABLE Reimbursement (
    ReimbursementID INT NOT NULL PRIMARY KEY IDENTITY,
    ApplicationID INT,
    Request TEXT,
    Processed BIT,
    Amount DECIMAL(10, 2),
```

```sql
    FOREIGN KEY (ApplicationID) REFERENCES Application(ApplicationID)
);

-- Create Complaints table
CREATE TABLE Complaints (
    ComplaintID INT NOT NULL PRIMARY KEY IDENTITY,
    CandidateID INT,
    ApplicationID INT,
    Date DATE,
    Status VARCHAR(50),
    Resolution VARCHAR(255),
    FOREIGN KEY (CandidateID) REFERENCES Candidate(CandidateID),
    FOREIGN KEY (ApplicationID) REFERENCES Application(ApplicationID)
);

-- Create RecruitmentHistory table
CREATE TABLE RecruitmentHistory (
    RHistoryID INT NOT NULL PRIMARY KEY IDENTITY,
    CandidateID INT,
    ActionType VARCHAR(50),
    ActionDetails TEXT,
    Date DATE,
    FOREIGN KEY (CandidateID) REFERENCES Candidate(CandidateID)
);

-- Create Onboarding table
CREATE TABLE Onboarding (
    OnboardingID INT NOT NULL PRIMARY KEY IDENTITY,
    CandidateID INT,
    JobID INT,
    StartDate DATE,
    CompletionDate DATE,
    FOREIGN KEY (CandidateID) REFERENCES Candidate(CandidateID),
    FOREIGN KEY (JobID) REFERENCES Job(JobID)
);

-- Create OnboardingDocuments table
CREATE TABLE OnboardingDocuments (
    DocumentID INT NOT NULL PRIMARY KEY IDENTITY,
    OnboardingID INT,
    DocumentType VARCHAR(50),
    URL TEXT,
    FOREIGN KEY (OnboardingID) REFERENCES Onboarding(OnboardingID)
);

-- Create InterviewSessions table
CREATE TABLE InterviewSessions (
    SessionID INT NOT NULL PRIMARY KEY IDENTITY,
    InterviewID INT,
    SessionDate DATE,
    SessionType VARCHAR(50),
    FOREIGN KEY (InterviewID) REFERENCES Interviews(InterviewID)
);

-- Create Employee table
CREATE TABLE Employee (
    EmployeeID INT NOT NULL PRIMARY KEY IDENTITY,
    EmployeeName VARCHAR(255),
    DEPARTMENT VARCHAR(255),
    TITLE VARCHAR(255)
);

-- Create CandidateReferrals table
CREATE TABLE CandidateReferrals (
    ReferralID INT NOT NULL PRIMARY KEY IDENTITY,
    CandidateID INT,
    EmployeeID INT,
    ReferralDate DATE,
    Status VARCHAR(50),
    FOREIGN KEY (CandidateID) REFERENCES Candidate(CandidateID),
    FOREIGN KEY (EmployeeID) REFERENCES Employee(EmployeeID)
);
```



```sql
-- Create Job table
CREATE TABLE Job (
    JobID INT NOT NULL PRIMARY KEY IDENTITY,
    Title VARCHAR(255),
    Description VARCHAR(100),
    Type VARCHAR(50),
    Medium VARCHAR(50),
    StartDate DATE,
    NumberOfPositions INT,
    JobPlatform VARCHAR(50)
);

-- Create JobOpenings table
CREATE TABLE JobOpenings (
    JobOpeningID INT NOT NULL PRIMARY KEY IDENTITY,
    JobID INT,
    NumberOfPositions INT,
    FOREIGN KEY (JobID) REFERENCES Job(JobID)
);
```

```
Commands completed successfully.

Completion time: 2024-04-30T10:50:44.0507422-04:00
```

The collection of tables created here serves to establish a comprehensive database system tailored for managing recruitment and employee onboarding processes. Each table fulfills a specific role in organizing and storing pertinent data, contributing to efficient and structured management of information within the database.

Job-related Tables: These tables (Job, JobOpenings) facilitate the storage of job descriptions, openings, and related details such as job type, start date, and available positions, enabling streamlined management of job postings and applications.

Candidate-related Tables: Tables (Candidate, Documents, BackgroundCheck, DrugTest, Complaints, CandidateReferrals, RecruitmentHistory) are dedicated to storing candidate information, documents, background checks, drug test results, complaints, referrals, and recruitment history. This ensures thorough candidate management and evaluation throughout the recruitment process.

Interview-related Tables: Tables (Interviews, Interviewers, InterviewTypes, Tests, InterviewReviews, InterviewSessions) manage various aspects of the interview process, including interview scheduling, interviewer details, interview types, assessments, and feedback, facilitating organized and effective interview management.

Onboarding-related Tables: Tables (Onboarding, OnboardingDocuments) handle the onboarding process, documenting candidate onboarding progress, required documents, and completion statuses, ensuring a smooth transition for new hires into their roles.

Employee-related Tables: The Employee table manages employee information, providing a centralized repository for employee details such as name, department, and title, supporting efficient employee data management within the organization.

Overall, these tables collectively form a robust database structure tailored to meet the needs of recruitment and onboarding processes, enabling effective organization, storage, and retrieval of relevant data to support seamless operation and decision-making within the organization.

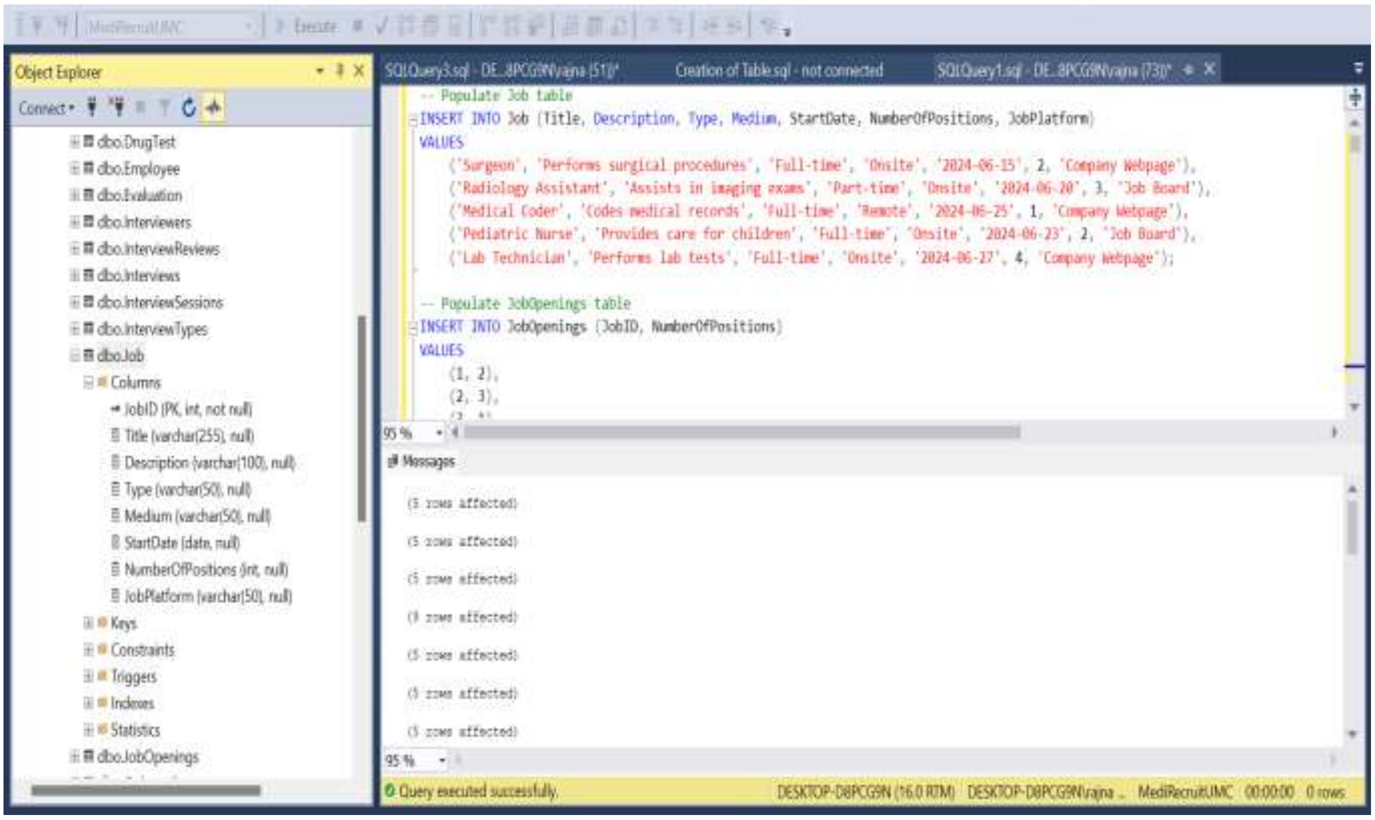
DATA INSERTION

SQL Query:

```
-- Populate Job table
INSERT INTO Job (Title, Description, Type, Medium, StartDate, NumberOfPositions,
JobPlatform)
VALUES
   ('Surgeon', 'Performs surgical procedures', 'Full-time', 'Onsite', '2024-06-15', 2, 'Company
Webpage'),
   ('Radiology Assistant', 'Assists in imaging exams', 'Part-time', 'Onsite', '2024-06-20', 3, 'Job
Board'),
   ('Medical Coder', 'Codes medical records', 'Full-time', 'Remote', '2024-06-25', 1, 'Company
Webpage'),
   ('Pediatric Nurse', 'Provides care for children', 'Full-time', 'Onsite', '2024-06-23', 2, 'Job
Board'),
   ('Lab Technician', 'Performs lab tests', 'Full-time', 'Onsite', '2024-06-27', 4, 'Company
Webpage');

-- Populate JobOpenings table
INSERT INTO JobOpenings (JobID, NumberOfPositions)
VALUES
   (1, 2),
   (2, 3),
   (3, 1),
   (4, 2),
   (5, 4);

-- Populate Candidate table
INSERT INTO Candidate (Name, EmailAddress, Address, Contact, ProfileDescription,
CandidateDecision, TransportationDetails, StayDetails)
VALUES
   ('John Carter', 'john.carter@example.com', '123 Main Street, Anytown, USA', '555-678-
1234', 'Experienced surgeon', 'Accepted', 'Flight & car rental', 'Hotel stay'),
   ('Lucy Peterson', 'lucy.peterson@example.com', '456 Elm Street, Anycity, USA', '999-123-
4567', 'Trained radiology assistant', 'Accepted', 'N/A', 'N/A'),
   ('Michael Thompson', 'michael.thompson@example.com', '789 Oak Avenue, Anyvillage,
USA', '888-234-5678', 'Skilled medical coder', 'Rejected', 'N/A', 'N/A'),
   ('Emily Parker', 'emily.parker@example.com', '101 Pine Road, Anysuburb, USA', '777-345-
6789', 'Pediatric nurse', 'Accepted', 'N/A', 'N/A'),
   ('David Clark', 'david.clark@example.com', '202 Cedar Lane, Anyhamlet, USA', '666-456-
7890', 'Lab technician', 'On-call', 'Flight', 'Hotel');

-- Populate Documents table
INSERT INTO Documents (CandidateID, Type, URL)
VALUES
   (1, 'Resume', 'https://example.com/resume_john.pdf'),
   (1, 'Cover Letter', 'https://example.com/cover_letter_john.pdf'),
   (2, 'Resume', 'https://example.com/resume_lucy.pdf'),
   (2, 'Cover Letter', 'https://example.com/cover_letter_lucy.pdf'),
   (3, 'Resume', 'https://example.com/resume_michael.pdf'),
   (3, 'Cover Letter', 'https://example.com/cover_letter_michael.pdf'),
   (4, 'Resume', 'https://example.com/resume_emily.pdf'),
   (4, 'Cover Letter', 'https://example.com/cover_letter_emily.pdf'),
   (5, 'Resume', 'https://example.com/resume_david.pdf');

-- Populate Application table
INSERT INTO Application (CandidateID, JobOpeningID, Status, DateSubmitted)
VALUES
   (1, 1, 'Accepted', '2024-05-10'),
   (2, 2, 'Accepted', '2024-05-12'),
   (3, 3, 'Rejected', '2024-05-14'),
   (4, 4, 'Accepted', '2024-05-15'),
   (5, 5, 'Under Review', '2024-05-18');

-- Populate Interviewers table
INSERT INTO Interviewers (Name, Department, Title)
VALUES
   ('Dr. Samantha White', 'Surgery', 'Chief Surgeon'),
   ('Dr. James Green', 'Radiology', 'Senior Radiologist'),
   ('Ms. Laura Adams', 'Billing', 'Chief Coder'),
   ('Nurse Helen Smith', 'Pediatrics', 'Senior Nurse'),
   ('Mr. Kevin Johnson', 'Lab', 'Chief Technician');

-- Populate InterviewTypes table
INSERT INTO InterviewTypes (TypeName, Description, ModeOfInstruction)
VALUES
   ('Remote', 'Conducted remotely via video call', 'Online'),
   ('In-Person', 'Conducted in person at the hospital', 'Onsite'),
   ('Technical', 'Tests technical skills', 'Varied'),
   ('Behavioral', 'Assesses soft skills', 'Varied'),
   ('Comprehensive', 'Includes both technical and soft skills tests', 'Varied');

-- Populate Interviews table
INSERT INTO Interviews (ApplicationID, InterviewerID, TypeID, StartTime, EndTime)
VALUES
   (1, 1, 3, '2024-05-15 10:00:00', '2024-05-15 11:00:00'),
   (2, 2, 1, '2024-05-16 09:00:00', '2024-05-16 10:00:00'),
   (3, 3, 4, '2024-05-17 14:00:00', '2024-05-17 15:00:00'),
   (4, 4, 2, '2024-05-18 13:00:00', '2024-05-18 14:00:00'),
   (5, 5, 5, '2024-05-19 12:00:00', '2024-05-19 13:00:00');

-- Populate Tests table
INSERT INTO Tests (InterviewID, Type, StartTime, EndTime, Grade)
```

```
VALUES
  (1, 'Surgical Skills', '2024-05-15 11:00:00', '2024-05-15 12:00:00', 'Passed'),
  (2, 'Radiology Skills', '2024-05-16 10:00:00', '2024-05-16 11:00:00', 'Passed'),
  (3, 'Coding Skills', '2024-05-17 15:00:00', '2024-05-17 16:00:00', 'Failed'),
  (4, 'Nursing Skills', '2024-05-18 14:00:00', '2024-05-18 15:00:00', 'Passed'),
  (5, 'Lab Skills', '2024-05-19 13:00:00', '2024-05-19 14:00:00', 'Passed');

-- Populate InterviewReviews table
INSERT INTO InterviewReviews (InterviewID, ReviewerID, ReviewerType, ReviewText,
Rating)
VALUES
  (1, 1, 'Candidate', 'Dr. White was precise.', 4),
  (2, 2, 'Interviewer', 'Lucy showed promise.', 3),
  (3, 3, 'Interviewer', 'Michael needs improvement.', 2),
  (4, 4, 'Candidate', 'Nurse Smith was supportive.', 4),
  (5, 5, 'Interviewer', 'David is skilled.', 4);

-- Populate BackgroundCheck table
INSERT INTO BackgroundCheck (CandidateID, CriminalRecord, EmploymentHistory, Status,
Date)
VALUES
  (1, 'None', 'Previous jobs verified.', 'Clear', '2024-05-18'),
  (2, 'None', 'Previous jobs verified.', 'Clear', '2024-05-19'),
  (3, 'None', 'Previous jobs verified.', 'Clear', '2024-05-20'),
  (4, 'None', 'Previous jobs verified.', 'Clear', '2024-05-21'),
  (5, 'None', 'Previous jobs verified.', 'Clear', '2024-05-22');

-- Populate DrugTest table
INSERT INTO DrugTest (CandidateID, TestType, TestDate, Results)
VALUES
  (1, 'Urine Test', '2024-05-19', 'Negative'),
  (2, 'Blood Test', '2024-05-20', 'Negative'),
  (3, 'Urine Test', '2024-05-21', 'Negative'),
  (4, 'Blood Test', '2024-05-22', 'Negative'),
  (5, 'Urine Test', '2024-05-23', 'Negative');

-- Populate Evaluation table
INSERT INTO Evaluation (ApplicationID, Notes, Result)
VALUES
  (1, 'Capable candidate.', 'Accepted'),
  (2, 'Needs more experience.', 'Accepted'),
  (3, 'Requires training.', 'Rejected'),
  (4, 'Well-qualified nurse.', 'Accepted'),
  (5, 'Potential technician.', 'Accepted');

-- Populate Reimbursement table
INSERT INTO Reimbursement (ApplicationID, Request, Processed, Amount)
```

```sql
VALUES
   (1, 'Travel expenses', 1, 150.00),
   (2, 'Hotel stay', 1, 100.00),
   (3, 'Travel expenses', 1, 80.00),
   (4, 'Meals', 1, 50.00),
   (5, 'Lab costs', 1, 60.00);

-- Populate Complaints table
INSERT INTO Complaints (CandidateID, ApplicationID, Date, Status, Resolution)
VALUES
   (1, 1, '2024-05-24', 'Pending', 'In Progress'),
   (2, 2, '2024-05-25', 'Resolved', 'Valid complaint'),
   (3, 3, '2024-05-26', 'Pending', 'Under Review'),
   (4, 4, '2024-05-27', 'Pending', 'In Progress'),
   (5, 5, '2024-05-28', 'Pending', 'Under Review');

-- Populate RecruitmentHistory table
INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
VALUES
   (1, 'Application Submitted', 'Applied for Surgeon position', '2024-05-10'),
   (2, 'Interview Passed', 'First interview successful.', '2024-05-16'),
   (3, 'Application Submitted', 'Applied for Medical Coder position.', '2024-05-14'),
   (4, 'Offer Extended', 'Job offer made.', '2024-05-15'),
   (5, 'Application Submitted', 'Applied for Lab Technician position.', '2024-05-18');

-- Populate Onboarding table
INSERT INTO Onboarding (CandidateID, JobID, StartDate, CompletionDate)
VALUES
   (1, 1, '2024-06-16', '2024-06-21'),
   (3, 3, '2024-06-26', '2024-07-01'),
   (4, 4, '2024-07-02', '2024-07-06'),
   (2, 2, '2024-06-21', '2024-06-26'),
   (5, 5, '2024-07-12', '2024-07-16');

-- Populate OnboardingDocuments table
INSERT INTO OnboardingDocuments (OnboardingID, DocumentType, URL)
VALUES
   (1, 'ID Proof', 'url_to_id_john.pdf'),
   (3, 'Contract', 'url_to_contract_michael.pdf'),
   (4, 'ID Proof', 'url_to_id_emily.pdf'),
   (2, 'Contract', 'url_to_contract_lucy.pdf'),
   (5, 'Contract', 'url_to_contract_david.pdf');

-- Populate InterviewSessions table
INSERT INTO InterviewSessions (InterviewID, SessionDate, SessionType)
VALUES
   (1, '2024-05-15', 'Technical Round'),
   (2, '2024-05-16', 'Radiology Skills'),
   (3, '2024-05-17', 'Behavioral Round'),
   (4, '2024-05-18', 'Nursing Skills'),
   (5, '2024-05-19', 'Lab Skills');

-- Populate Employees table
INSERT INTO Employee (EmployeeName, DEPARTMENT, TITLE)
VALUES
   ('Dr. White', 'Surgery', 'Chief Surgeon'),
   ('Dr. Green', 'Radiology', 'Senior Radiologist'),
   ('Ms. Adams', 'Billing', 'Chief Coder'),
   ('Nurse Smith', 'Pediatrics', 'Senior Nurse'),
   ('Mr. Johnson', 'Lab', 'Chief Technician');

-- Populate CandidateReferrals table
INSERT INTO CandidateReferrals (CandidateID, EmployeeID, ReferralDate, Status)
VALUES
   (1, 5, '2024-05-10', 'Active'),
   (2, 1, '2024-05-11', 'Inactive'),
   (3, 4, '2024-05-12', 'Active'),
   (4, 2, '2024-05-13', 'Inactive'),
   (5, 3, '2024-05-14', 'Active');
```



```
-- Populate Job table
INSERT INTO Job (Title, Description, Type, Medium, StartDate, NumberOfPositions, JobPlatform)
VALUES
    ('Surgeon', 'Performs surgical procedures', 'Full-time', 'Onsite', '2024-06-15', 2, 'Company Webpage'),
    ('Radiology Assistant', 'Assists in imaging exams', 'Part-time', 'Onsite', '2024-06-20', 3, 'Job Board'),
    ('Medical Coder', 'Codes medical records', 'Full-time', 'Remote', '2024-06-25', 1, 'Company Webpage'),
    ('Pediatric Nurse', 'Provides care for children', 'Full-time', 'Onsite', '2024-06-23', 2, 'Job Board'),
    ('Lab Technician', 'Performs lab tests', 'Full-time', 'Onsite', '2024-06-27', 4, 'Company Webpage');

-- Populate JobOpenings table
INSERT INTO JobOpenings (JobID, NumberOfPositions)
VALUES
    (1, 2),
    (2, 3),
```

```
(5 rows affected)

(5 rows affected)

(5 rows affected)

(5 rows affected)

(5 rows affected)

(5 rows affected)

(5 rows affected)
```

Object Explorer: dbo.DrugTest, dbo.Employee, dbo.Evaluation, dbo.Interviewers, dbo.InterviewReviews, dbo.Interviews, dbo.InterviewSessions, dbo.InterviewTypes, dbo.Job, Columns, JobID (PK, int, not null), Title (varchar(255), null), Description (varchar(100), null), Type (varchar(50), null), Medium (varchar(50), null), StartDate (date, null), NumberOfPositions (int, null), JobPlatform (varchar(50), null), Keys, Constraints, Triggers, Indexes, Statistics, dbo.JobOpenings

Query executed successfully. DESKTOP-D8PCG9N (16.0 RTM) DESKTOP-D8PCG9N\vajna ... MediRecruitUMC 00:00:00 0 rows

The SQL queries provided populate the tables created in the database with relevant data. Let's break down what each query does:

Populate Job table: Inserts job listings with their titles, descriptions, types, mediums, start dates, number of positions, and platforms where they are posted.
Populate JobOpenings table: Associates job openings with their respective Job IDs and specifies the number of available positions for each opening.
Populate Candidate table: Inserts candidate details including their names, email addresses, addresses, contact numbers, profile descriptions, decisions (e.g., accepted, rejected), transportation details, and stay details.
Populate Documents table: Adds documents related to candidates such as resumes and cover letters along with their URLs.
Populate Application table: Associates candidates with job openings they applied for, including the application status and submission dates.
Populate Interviewers table: Inserts information about interviewers including their names, departments, and titles.
Populate InterviewTypes table: Adds types of interviews with descriptions and modes of instruction.
Populate Interviews table: Associates interviews with specific applications, interviewers, interview types, start times, and end times.
Populate Tests table: Inserts details of tests conducted during interviews, including test types, start times, end times, and grades.
Populate InterviewReviews table: Adds reviews of interviews by reviewers, including reviewer types, review texts, and ratings.
Populate BackgroundCheck table: Inserts background check results for candidates, including criminal records, employment history, statuses, and dates.
Populate DrugTest table: Adds drug test results for candidates, including test types, test dates, and results.
Populate Evaluation table: Associates evaluations with specific applications, including notes and results.
Populate Reimbursement table: Adds reimbursement requests related to applications, including request details, processing statuses, and amounts.
Populate Complaints table: Inserts complaints filed by candidates, including complaint dates, statuses, and resolutions.
Populate RecruitmentHistory table: Records the history of recruitment actions for candidates, including action types, action details, and dates.
Populate Onboarding table: Associates candidates with their onboarding details, including job IDs, start dates, and completion dates.
Populate OnboardingDocuments table: Inserts documents required for onboarding, including document types and URLs.
Populate InterviewSessions table: Associates interview sessions with interviews, including session dates and types.
Populate Employee table: Inserts employee details such as names, departments, and titles.
Populate CandidateReferrals table: Records candidate referrals made by employees, including referral dates and statuses.
These SQL queries populate the database tables with data related to job listings, candidates, applications, interviews, evaluations, background checks, drug tests, reimbursements, complaints, recruitment history, onboarding, interview sessions, employees, and candidate referrals. This data will facilitate the management and tracking of recruitment and onboarding processes within the organization.

"Optimizing Database Functionality and Business Logic: Implementing Views, Procedures, Functions, Triggers, and Transactions"

VIEWS: Enhancing Data Accessibility and Security

Views in a database serve as virtual tables generated from the outcome of a SQL query, offering a versatile toolset for simplifying data management, enforcing security protocols, and enhancing data usability. Here's an in-depth exploration of their capabilities:

I. Views streamline data access by providing users with tailored subsets of data without requiring them to comprehend the intricate database schema. Users can interact with views through simplified queries, abstracting away the complexity of underlying tables. This simplification not only enhances user experience but also optimizes query performance by retrieving only the necessary data, thereby improving overall system efficiency.

II. Security is paramount in database management, and views play a pivotal role in enforcing access controls. Database administrators can craft views that expose only the essential columns and rows to specific user roles, effectively limiting access to sensitive data. By granting access to views instead of underlying tables, administrators maintain granular control over data exposure, ensuring compliance with security regulations and safeguarding confidential information from unauthorized access.

III. Views offer dynamic capabilities for data aggregation and transformation in real-time. Through the use of aggregate functions and JOIN operations, users can construct views that consolidate data from multiple tables, facilitating seamless analysis and reporting. Furthermore, views enable on-the-fly data manipulation, empowering users to derive actionable insights without altering the underlying dataset. This flexibility fosters agility in decision-making processes and supports iterative data exploration, driving informed business decisions.

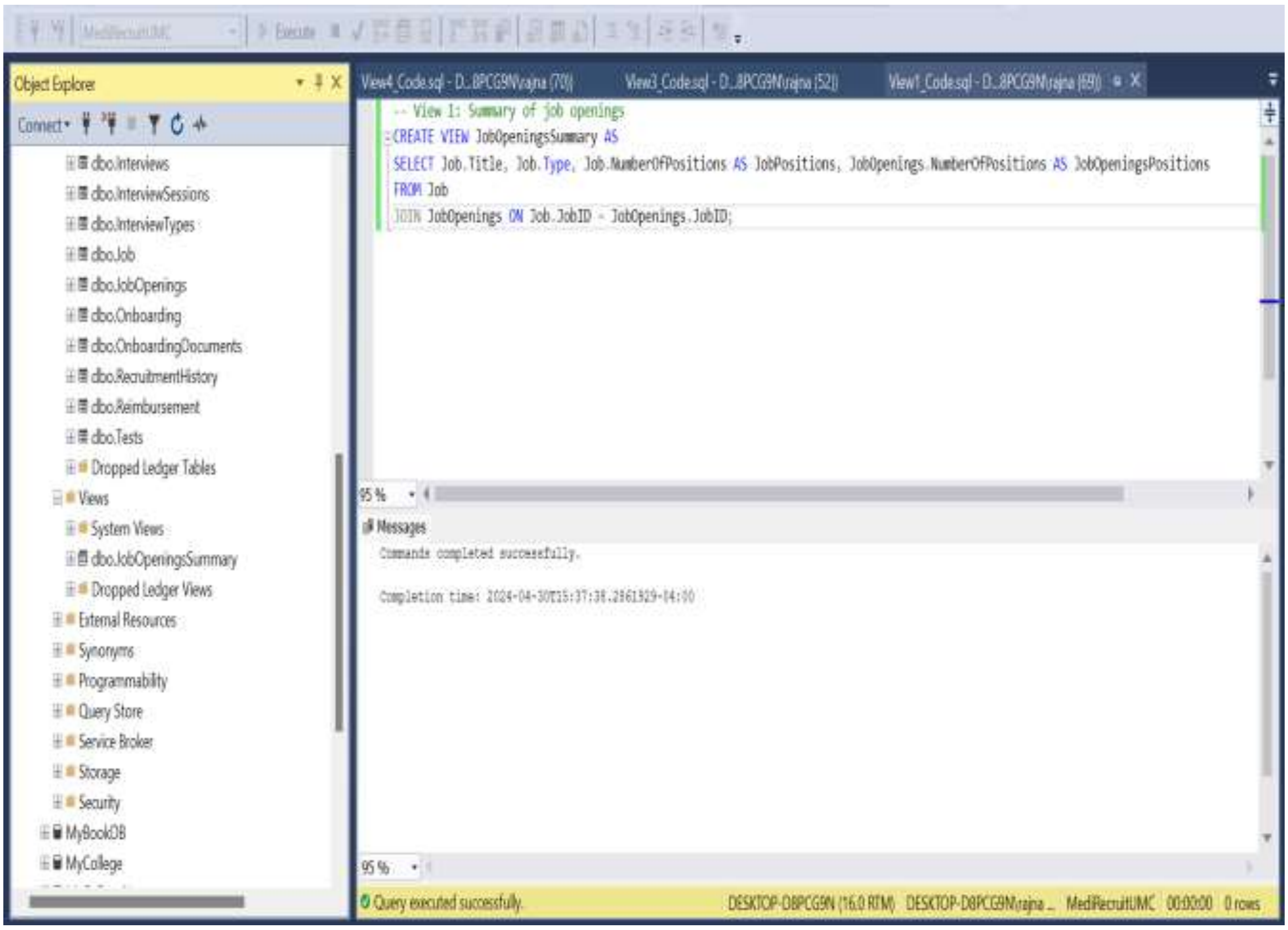
IV. Views serve as a vehicle for enforcing business rules and logic within the database environment. By encapsulating complex calculations, data validations, and filtering criteria, views ensure data consistency and accuracy across applications. Additionally, views enhance code maintainability by centralizing business logic within the database layer, promoting scalability and reducing redundancy in application code.

In summary, views are assets in modern database management, offering a comprehensive suite of functionalities to enhance data accessibility, security, and usability. By providing a simplified and controlled interface for interacting with data, views empower organizations to effectively manage complex data structures, enforce security protocols, and enforce business rules, ultimately driving efficiency and integrity within the database ecosystem.

Below is the query designed for Four different view of Database:

1). View 1: Summary of job openings

```
CREATE VIEW JobOpeningsSummary AS
SELECT Job.Title, Job.Type, Job.NumberOfPositions AS JobPositions,
JobOpenings.NumberOfPositions AS JobOpeningsPositions
FROM Job
JOIN JobOpenings ON Job.JobID = JobOpenings.JobID;
```



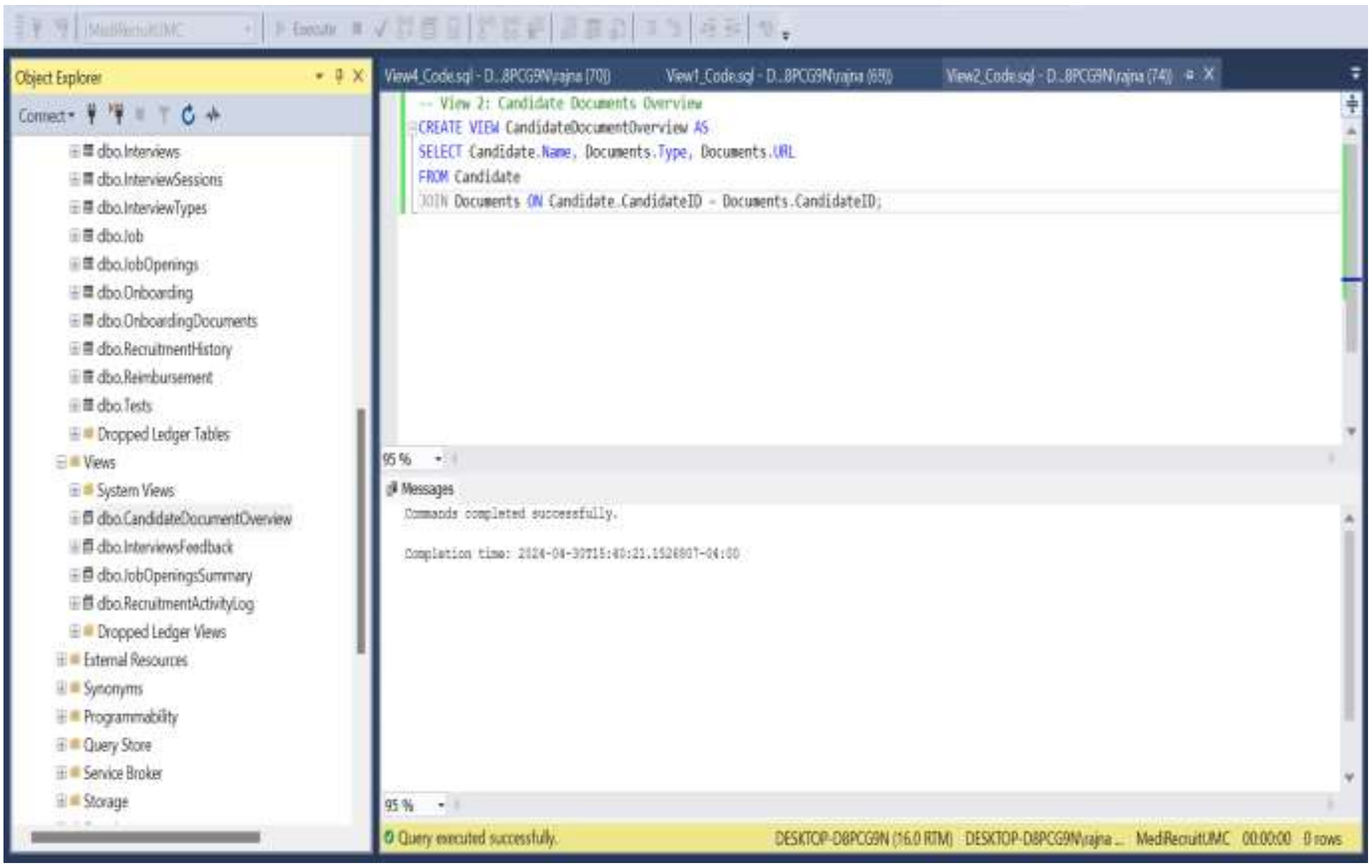
```
-- View 1: Summary of job openings
CREATE VIEW JobOpeningsSummary AS
SELECT Job.Title, Job.Type, Job.NumberOfPositions AS JobPositions, JobOpenings.NumberOfPositions AS JobOpeningsPositions
FROM Job
JOIN JobOpenings ON Job.JobID = JobOpenings.JobID;
```

```
Commands completed successfully.

Completion time: 2024-04-30T15:37:38.2861329-04:00
```

The "JobOpeningsSummary" view consolidates job information from the "Job" and "JobOpenings" tables. It selects job titles and types from "Job" alongside the respective numbers of positions from both tables. The view employs a join operation based on JobID columns to merge relevant data. This yields a streamlined overview of available job openings, displaying titles, types, and the total number of positions. Stakeholders, such as HR managers or recruiters, benefit from this concise summary, facilitating quick access to essential job details for informed decision-making. Users can query the view to retrieve comprehensive insights without navigating the underlying table structures.

2). View 2: Candidate Documents Overview

```
CREATE VIEW CandidateDocumentOverview AS
SELECT Candidate.Name, Documents.Type, Documents.URL
FROM Candidate
JOIN Documents ON Candidate.CandidateID = Documents.CandidateID;
```



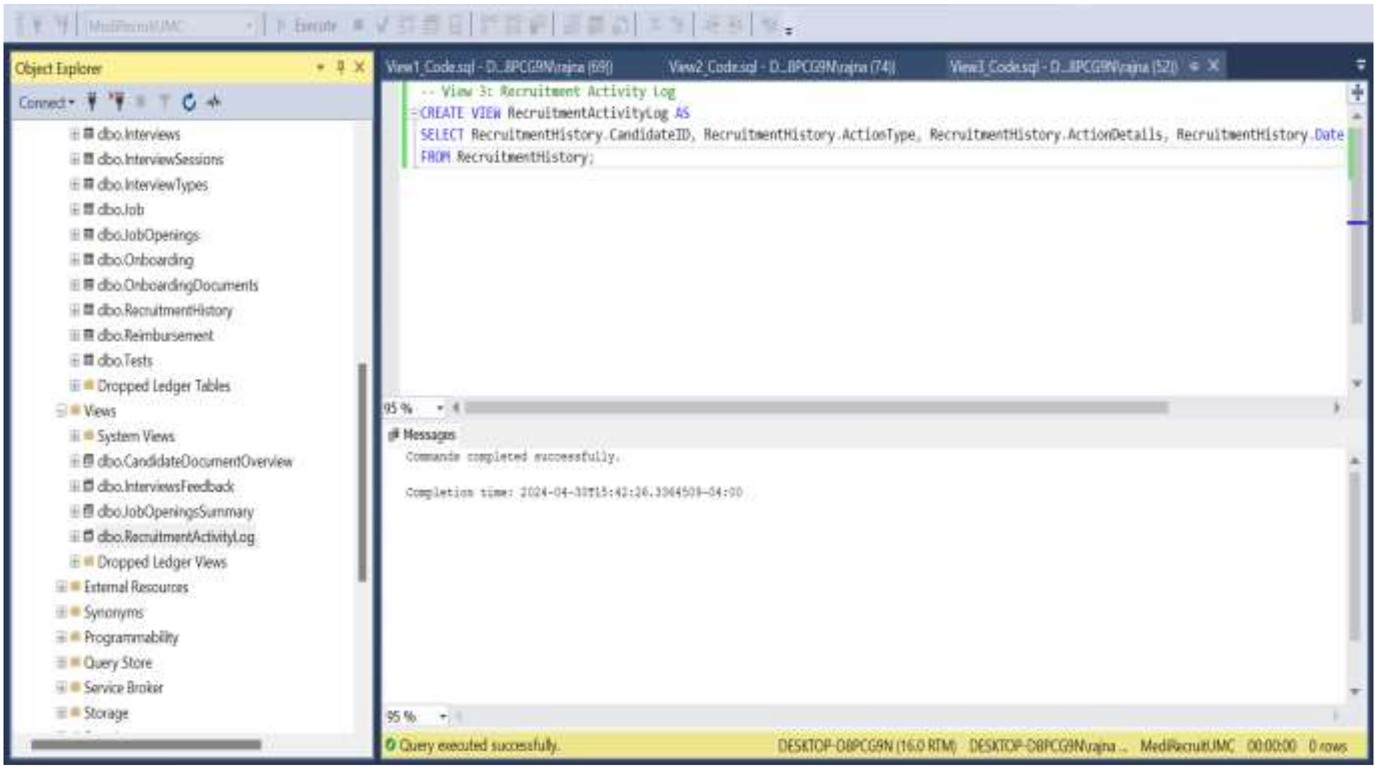
```
-- View 2: Candidate Documents Overview
CREATE VIEW CandidateDocumentOverview AS
SELECT Candidate.Name, Documents.Type, Documents.URL
FROM Candidate
JOIN Documents ON Candidate.CandidateID = Documents.CandidateID;
```

```
Commands completed successfully.

Completion time: 2024-04-30T15:40:21.1526807-04:00
```

The "CandidateDocumentOverview" view amalgamates data from the "Candidate" and "Documents" tables to offer a comprehensive snapshot of candidate documents. By selecting the candidate's name, document type, and URL, the view provides stakeholders with easy access to crucial information regarding candidate profiles. The view achieves this by performing a join operation based on the CandidateID columns of both tables, effectively linking each candidate with their respective documents. This streamlined overview facilitates efficient document management and enables recruiters or HR managers to swiftly review candidate documents without navigating through multiple tables. With its simplified interface, the view enhances user experience and supports informed decision-making in candidate evaluation processes. Users can query the view to retrieve essential details about candidate documents, empowering them to make data-driven decisions and streamline recruitment workflows effectively.

3). View 3: Recruitment Activity Log

```
CREATE VIEW RecruitmentActivityLog AS
SELECT RecruitmentHistory.CandidateID, RecruitmentHistory.ActionType,
RecruitmentHistory.ActionDetails, RecruitmentHistory.Date
FROM RecruitmentHistory;
```



```
-- View 3: Recruitment Activity Log
CREATE VIEW RecruitmentActivityLog AS
SELECT RecruitmentHistory.CandidateID, RecruitmentHistory.ActionType, RecruitmentHistory.ActionDetails, RecruitmentHistory.Date
FROM RecruitmentHistory;
```

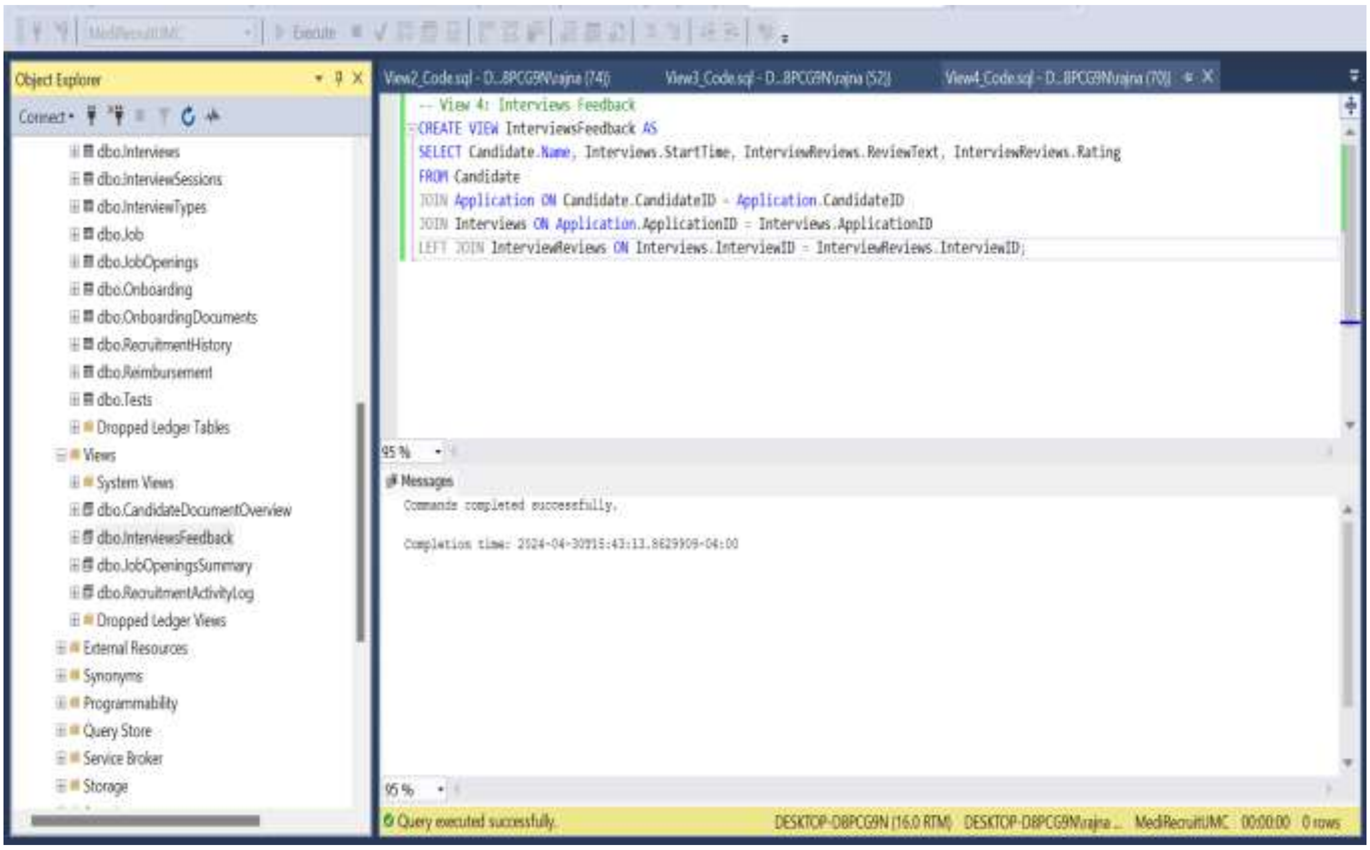
```
Commands completed successfully.

Completion time: 2024-04-30T15:42:26.3364509-04:00
```

Query executed successfully. DESKTOP-O8PCG9N (16.0 RTM) DESKTOP-O8PCG9N\rajna ... MediRecruitUMC 00:00:00 0 rows

The "RecruitmentActivityLog" view compiles a chronological record of recruitment activities from the "RecruitmentHistory" table. The view selects columns representing the CandidateID, ActionType, ActionDetails, and Date from the "RecruitmentHistory" table, providing a concise summary of recruitment-related actions over time. Each entry in the view corresponds to a specific action taken during the recruitment process, such as candidate interviews, application reviews, or status updates. By encapsulating this information in a single view, stakeholders gain insights into the progression of recruitment activities and can track the history of interactions with candidates. This facilitates transparency and accountability in recruitment processes, enabling recruiters, HR personnel, and management to monitor and analyze recruitment activities efficiently. Users can query the view to retrieve detailed information about past recruitment actions, supporting data-driven decision-making and optimizing recruitment strategies for enhanced efficiency and effectiveness.

4). View 4: Interviews Feedback

```
CREATE VIEW InterviewsFeedback AS
SELECT Candidate.Name, Interviews.StartTime, InterviewReviews.ReviewText,
InterviewReviews.Rating
FROM Candidate
JOIN Application ON Candidate.CandidateID = Application.CandidateID
JOIN Interviews ON Application.ApplicationID = Interviews.ApplicationID
LEFT JOIN InterviewReviews ON Interviews.InterviewID = InterviewReviews.InterviewID;
```



```
-- View 4: Interviews Feedback
CREATE VIEW InterviewsFeedback AS
SELECT Candidate.Name, Interviews.StartTime, InterviewReviews.ReviewText, InterviewReviews.Rating
FROM Candidate
JOIN Application ON Candidate.CandidateID = Application.CandidateID
JOIN Interviews ON Application.ApplicationID = Interviews.ApplicationID
LEFT JOIN InterviewReviews ON Interviews.InterviewID = InterviewReviews.InterviewID;
```

```
Commands completed successfully.

Completion time: 2024-04-30T15:43:13.8629909-04:00
```

```
Query executed successfully.    DESKTOP-D8PCG9N (16.0 RTM)    DESKTOP-D8PCG9N\rajna ...    MediRecruitUMC    00:00:00    0 rows
```

The "InterviewsFeedback" view consolidates feedback from candidate interviews by combining data from multiple tables. It selects the candidate's name, interview start time, review text, and rating from the respective tables: "Candidate," "Application," "Interviews," and "InterviewReviews." By joining these tables based on their related keys, the view links candidates with their interview details, including any associated feedback. The use of a LEFT JOIN ensures that all interviews are included in the view, regardless of whether feedback exists. This view provides stakeholders, such as hiring managers and recruiters, with valuable insights into candidate performance and interviewer assessments. It enables efficient tracking of interview feedback and ratings, facilitating data-driven decision-making in the recruitment process. Users can query the view to retrieve comprehensive information about interview feedback, aiding in candidate evaluation and selection for optimal hiring outcomes.

Stored Procedures: Streamlining Database Operations

Stored procedures stand as strong and liable components in database management, offering a multitude of advantages that elevate the functionality and integrity of database systems. Here's a detailed exploration of their benefits:

I.Stored procedures encapsulate SQL statements, transforming them into reusable units of logic. This modular approach promotes code organization and reduces redundancy by centralizing common database operations. By abstracting complex SQL logic into stored procedures, developers can easily reuse code across multiple applications and minimize the risk of errors caused by duplicated code segments.

II. The precompiled nature of stored procedures results in optimized query execution speed. By compiling SQL code once and storing it in the database, stored procedures eliminate the need for repetitive compilation during execution, thereby reducing overhead and enhancing overall performance. This efficiency improvement translates to faster data retrieval and manipulation, contributing to a responsive and scalable database environment, particularly in scenarios with high query frequency or large datasets.

III. Stored procedures serve as gatekeepers for database access, enabling administrators to enforce stringent security policies. By encapsulating sensitive operations within stored procedures, direct access to underlying tables can be restricted, ensuring that only authorized users can execute predefined procedures. This granular control over data access helps prevent unauthorized data manipulation or disclosure, bolstering data security and compliance with regulatory requirements.

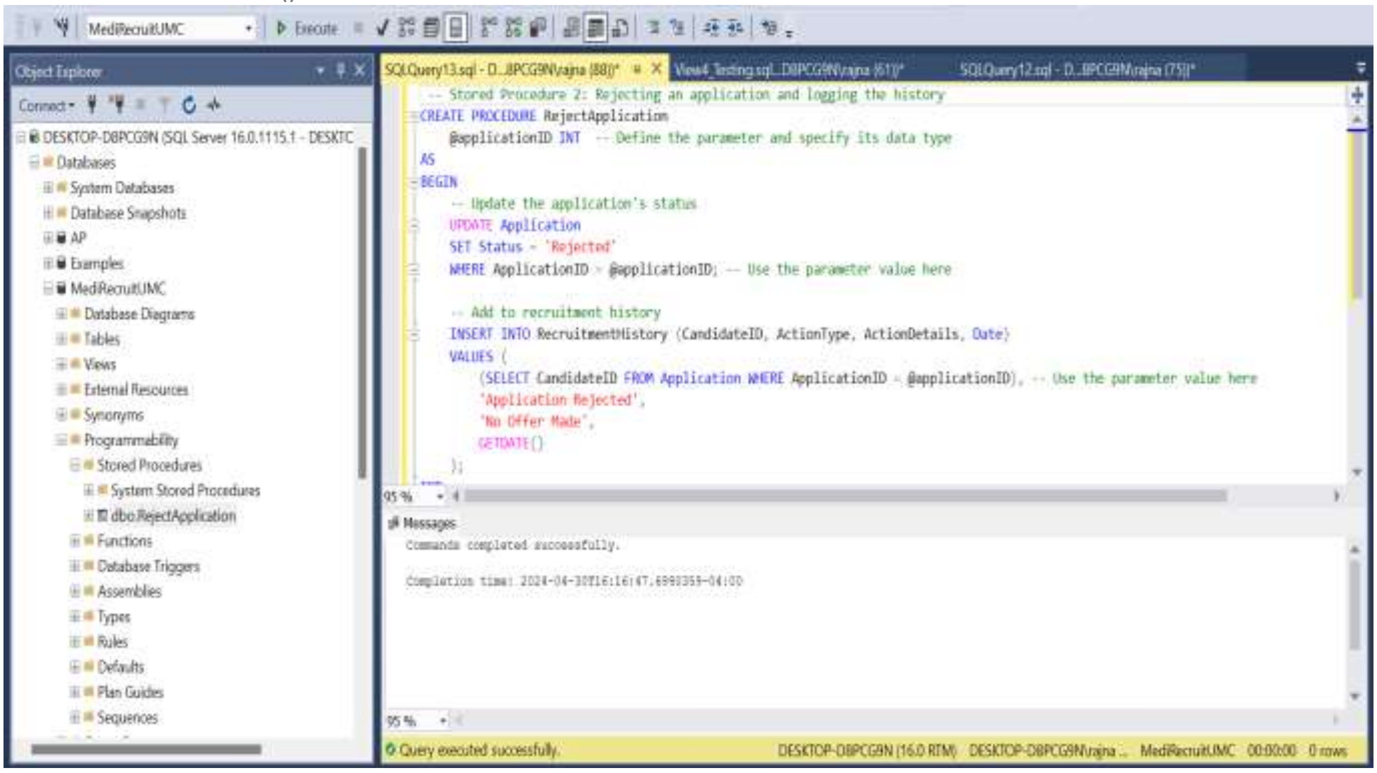
IV. Stored procedures play a pivotal role in transaction management, enabling the execution of multiple SQL statements as a single atomic operation. This atomicity ensures that either all operations within a transaction are successfully completed or none of them are, thereby preserving data consistency and integrity. With stored procedures, complex transactions involving multiple database modifications can be orchestrated seamlessly, mitigating the risk of data anomalies or corruption.

V. Stored procedures empower organizations to implement and enforce complex business rules directly within the database layer. By embedding business logic within stored procedures, critical data consistency checks, validation rules, and application-specific workflows can be enforced consistently across various database interactions. This centralized enforcement of business rules ensures that data integrity is maintained and application logic is adhered to, regardless of the client application or access method employed.

In summary, stored procedures serve as indispensable assets in database management, offering unparalleled benefits in code organization, performance optimization, security enforcement, transaction management, and business rule implementation. Their multifaceted capabilities contribute to enhanced efficiency, reliability, and scalability in database operations, making them essential components in modern database architectures.

1). Stored Procedure 1: Rejecting an application and logging the history

```
    CREATE PROCEDURE RejectApplication
@applicationID INT  -- Define the parameter and specify its data type
    AS
    BEGIN
          -- Updating the application's status
          UPDATE Application
          SET Status = 'Rejected'
         WHERE ApplicationID = @applicationID;

-- Add to recruitment history
INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
VALUES (
   (SELECT CandidateID FROM Application WHERE ApplicationID = @applicationID),
   'Application Rejected',
   'No Offer Made',
          GETDATE()
```



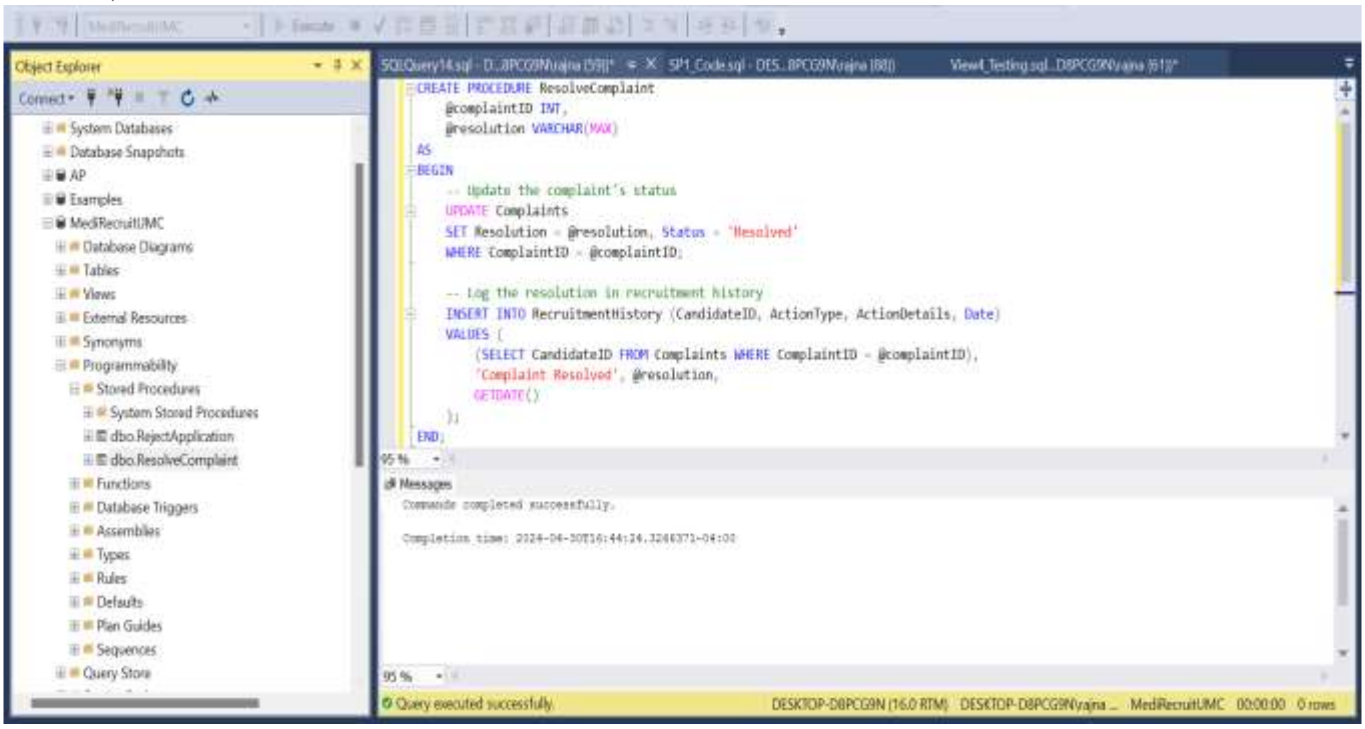
```
-- Stored Procedure 2: Rejecting an application and logging the history
CREATE PROCEDURE RejectApplication
    @applicationID INT  -- Define the parameter and specify its data type
AS
BEGIN
    -- Update the application's status
    UPDATE Application
    SET Status = 'Rejected'
    WHERE ApplicationID = @applicationID; -- Use the parameter value here

    -- Add to recruitment history
    INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
    VALUES (
        (SELECT CandidateID FROM Application WHERE ApplicationID = @applicationID), -- Use the parameter value here
        'Application Rejected',
        'No Offer Made',
        GETDATE()
    );
```

```
Commands completed successfully.

Completion time: 2024-04-30T16:16:47.6980359-04:00
```

Query executed successfully. DESKTOP-D8PCG9N (16.0 RTM) DESKTOP-D8PCG9N\rajna ... MediRecruitUMC 00:00:00 0 rows

The "RejectApplication" stored procedure facilitates the rejection of a job application and logs this action for historical reference. Upon execution, it updates the status of the specified application to "Rejected" in the "Application" table. Additionally, it records relevant details, such as the candidate ID, action type ("Application Rejected"), action details ("No Offer Made"), and the current date, into the "RecruitmentHistory" table. This logging ensures transparency and accountability in the recruitment process, allowing stakeholders to track the history of application rejections. By encapsulating these operations within a stored procedure, code modularity is promoted, reducing redundancy, and facilitating maintenance. Overall, this stored procedure streamlines the application rejection process while providing a comprehensive audit trail of recruitment activities.

```
2). -- Stored Procedure 2: CREATE PROCEDURE for Handling complaint resolution,
        CREATE PROCEDURE ResolveComplaint
            @complaintID INT,
            @resolution VARCHAR(MAX) -- specify maximum length for VARCHAR(MAX)
        AS
        BEGIN
               -- Update the complaint's status
                UPDATE Complaints
                SET Resolution = @resolution, Status = 'Resolved'
                WHERE ComplaintID = @complaintID;
               -- Log the resolution in recruitment history
       INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
       VALUES (
                 (SELECT CandidateID FROM Complaints WHERE ComplaintID = @complaintID),
                  'Complaint Resolved', @resolution,
                 GETDATE()
              );
      END;
```



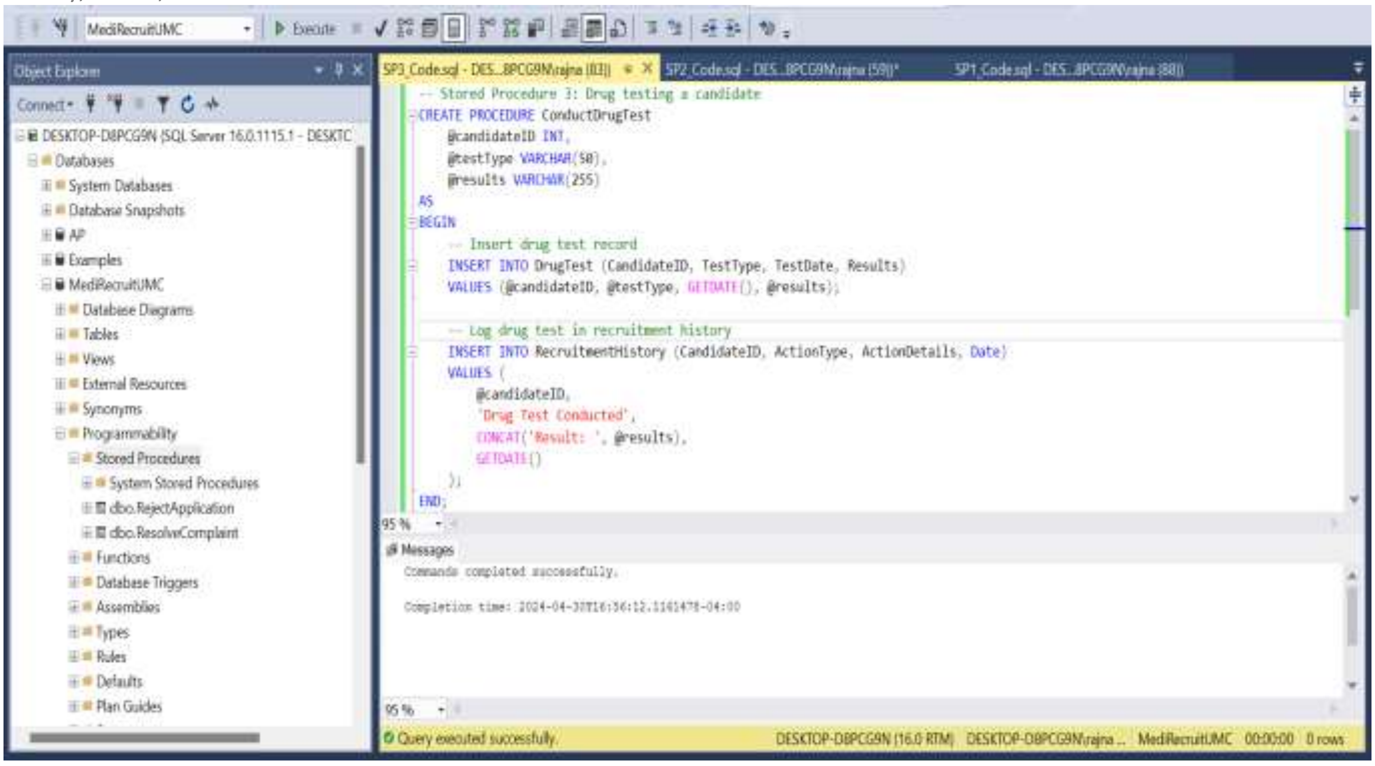
```
CREATE PROCEDURE ResolveComplaint
    @complaintID INT,
    @resolution VARCHAR(MAX)
AS
BEGIN
    -- Update the complaint's status
    UPDATE Complaints
    SET Resolution = @resolution, Status = 'Resolved'
    WHERE ComplaintID = @complaintID;

    -- Log the resolution in recruitment history
    INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
    VALUES (
        (SELECT CandidateID FROM Complaints WHERE ComplaintID = @complaintID),
        'Complaint Resolved', @resolution,
        GETDATE()
    );
END;
```

```
Commands completed successfully.

Completion time: 2024-04-30T16:44:24.3266371-04:00
```

Query executed successfully. DESKTOP-D8PCG9N (16.0 RTM) DESKTOP-D8PCG9N\vajna ... MediRecruitUMC 00:00:00 0 rows

The "ResolveComplaint" stored procedure facilitates the resolution of a complaint by updating its status and logging the resolution. Upon execution, it updates the "Resolution" and "Status" columns of the specified complaint in the "Complaints" table, marking it as "Resolved" and storing the resolution details provided as input. Subsequently, it records relevant information, including the candidate ID associated with the complaint, the action type ("Complaint Resolved"), the resolution details, and the current date, into the "RecruitmentHistory" table. This logging ensures a comprehensive audit trail of complaint resolutions within the recruitment process, aiding in accountability and transparency. By encapsulating these operations within a stored procedure, code modularity is enhanced, promoting maintainability, and reducing redundancy.

```
3). -- Stored Procedure 3: Drug testing a candidate
        CREATE PROCEDURE ConductDrugTest
             @candidateID INT,
             @testType VARCHAR(50),
            @results VARCHAR(255)
       AS
      BEGIN
          -- Insert drug test record
          INSERT INTO DrugTest (CandidateID, TestType, TestDate, Results)
          VALUES (@candidateID, @testType, GETDATE(), @results);
          -- Log drug test in recruitment history
          INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
          VALUES (
              @candidateID,
              'Drug Test Conducted',
              CONCAT('Result: ', @results),
              GETDATE()
         );  END;
```



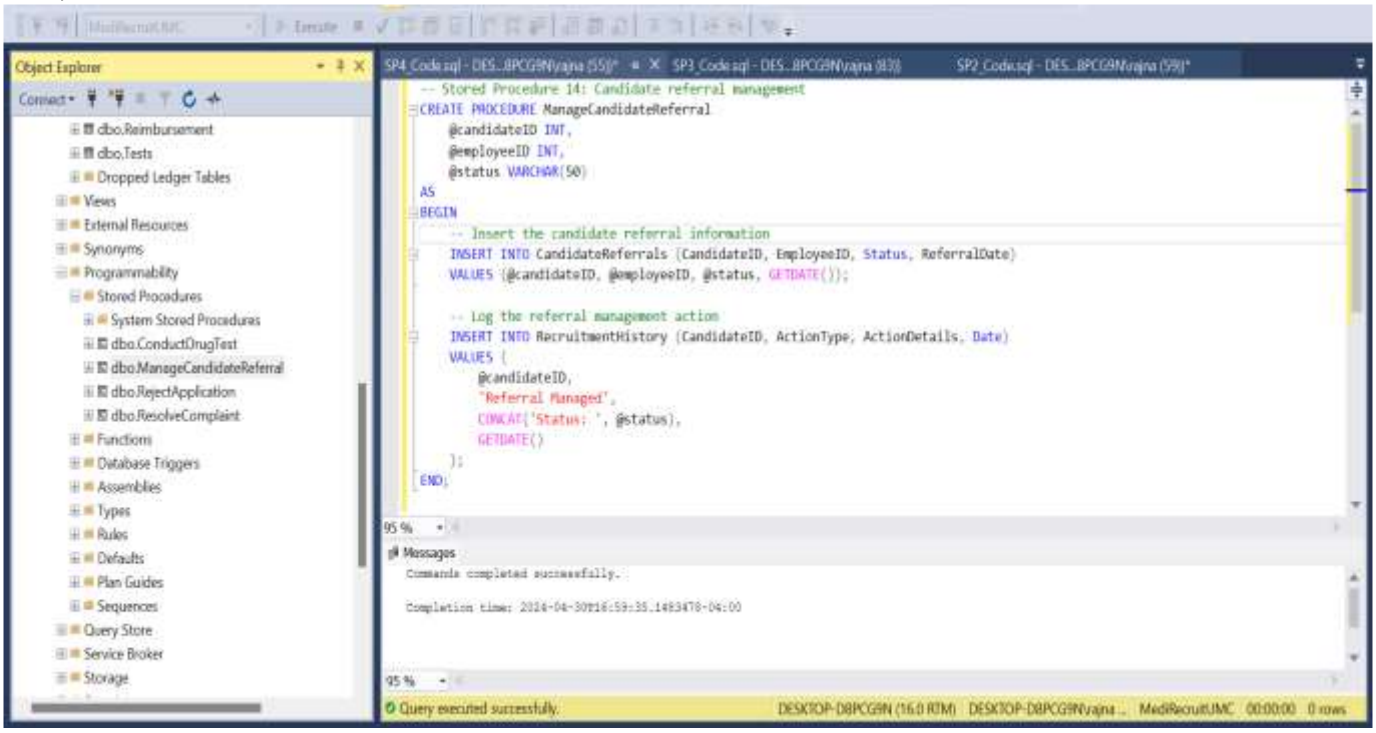
```
-- Stored Procedure 3: Drug testing a candidate
CREATE PROCEDURE ConductDrugTest
    @candidateID INT,
    @testType VARCHAR(50),
    @results VARCHAR(255)
AS
BEGIN
    -- Insert drug test record
    INSERT INTO DrugTest (CandidateID, TestType, TestDate, Results)
    VALUES (@candidateID, @testType, GETDATE(), @results);

    -- Log drug test in recruitment history
    INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
    VALUES (
        @candidateID,
        'Drug Test Conducted',
        CONCAT('Result: ', @results),
        GETDATE()
    );
END;
```

```
Commands completed successfully.

Completion time: 2024-04-30T16:56:12.1161478-04:00
```

```
Query executed successfully.    DESKTOP-D8PCG9N (16.0 RTM)  DESKTOP-D8PCG9N\rajna ...  MediRecruitUMC  00:00:00  0 rows
```

The "ConductDrugTest" stored procedure conducts a drug test for a specified candidate and logs the test activity. Upon execution, it inserts a record into the "DrugTest" table, capturing details such as the candidate's ID, the type of test conducted, the test date (obtained using the GETDATE() function), and the test results provided as input parameters. Subsequently, it logs relevant information into the "RecruitmentHistory" table, including the candidate ID, the action type ("Drug Test Conducted"), a concatenated string indicating the test result, and the current date. This logging ensures a comprehensive record of drug testing activities within the recruitment process, aiding in tracking and accountability. By encapsulating these operations within a stored procedure, code modularity is promoted, facilitating maintenance, and reducing redundancy.

```sql
4). -- Stored Procedure 4: Candidate referral management
CREATE PROCEDURE ManageCandidateReferral
   @candidateID INT,
   @employeeID INT,
   @status VARCHAR(50)
AS
BEGIN
   -- Insert the candidate referral information
   INSERT INTO CandidateReferrals (CandidateID, EmployeeID, Status, ReferralDate)
   VALUES (@candidateID, @employeeID, @status, GETDATE());
   -- Log the referral management action
   INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
   VALUES (
      @candidateID,
      'Referral Managed',
      CONCAT('Status: ', @status),
      GETDATE()
   );
END;
```



```sql
-- Stored Procedure 14: Candidate referral management
CREATE PROCEDURE ManageCandidateReferral
    @candidateID INT,
    @employeeID INT,
    @status VARCHAR(50)
AS
BEGIN
    -- Insert the candidate referral information
    INSERT INTO CandidateReferrals (CandidateID, EmployeeID, Status, ReferralDate)
    VALUES (@candidateID, @employeeID, @status, GETDATE());

    -- Log the referral management action
    INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
    VALUES (
        @candidateID,
        'Referral Managed',
        CONCAT('Status: ', @status),
        GETDATE()
    );
END;
```

```
Commands completed successfully.

Completion time: 2024-04-30T16:59:35.1483478-04:00
```

Query executed successfully. DESKTOP-D8PCG9N (16.0 RTM) DESKTOP-D8PCG9N\vajna ... MediRecruitUMC 00:00:00 0 rows

The "ManageCandidateReferral" stored procedure facilitates the management of candidate referrals by inserting referral information into the "CandidateReferrals" table and logging the action. It first inserts a record into "CandidateReferrals," capturing the candidate's ID, referring employee's ID, referral status, and the current date. Subsequently, it logs relevant details into the "RecruitmentHistory" table, including the candidate ID, action type ("Referral Managed"), the referral status, and the current date. This procedure ensures systematic tracking of candidate referrals and actions taken, promoting transparency and accountability in the recruitment process. By encapsulating these operations within a stored procedure, code modularity is enhanced, aiding in maintenance, and reducing redundancy.

User-Defined Functions (UDFs): Enhancing Database Functionality

User-Defined Functions (UDFs) are custom functions created by users to perform specific tasks within a database. They offer several benefits for enhancing database functionality and enforcing business rules:

I. One of the primary advantages of UDFs is their ability to promote code modularity and reduce redundancy in database operations. By encapsulating reusable logic within custom functions, users can simplify code maintenance and streamline complex operations. Instead of rewriting the same code segments across multiple queries or stored procedures, developers can define UDFs tailored to specific tasks, promoting code reuse, and improving code organization.

II. UDFs also play a crucial role in data transformation within a database environment. They enable users to manipulate data according to specific requirements, such as converting data types, formatting values, or performing calculations. Whether it's transforming raw data into meaningful insights or preparing data for reporting purposes, UDFs provide a flexible mechanism for data manipulation, facilitating data-driven decision-making and analysis.

III. UDFs facilitate the enforcement of business rules within the database layer. By embedding business logic within custom functions, users can ensure consistent application of rules such as data validation, constraints, and calculations, maintaining data integrity and enforcing compliance with business requirements.

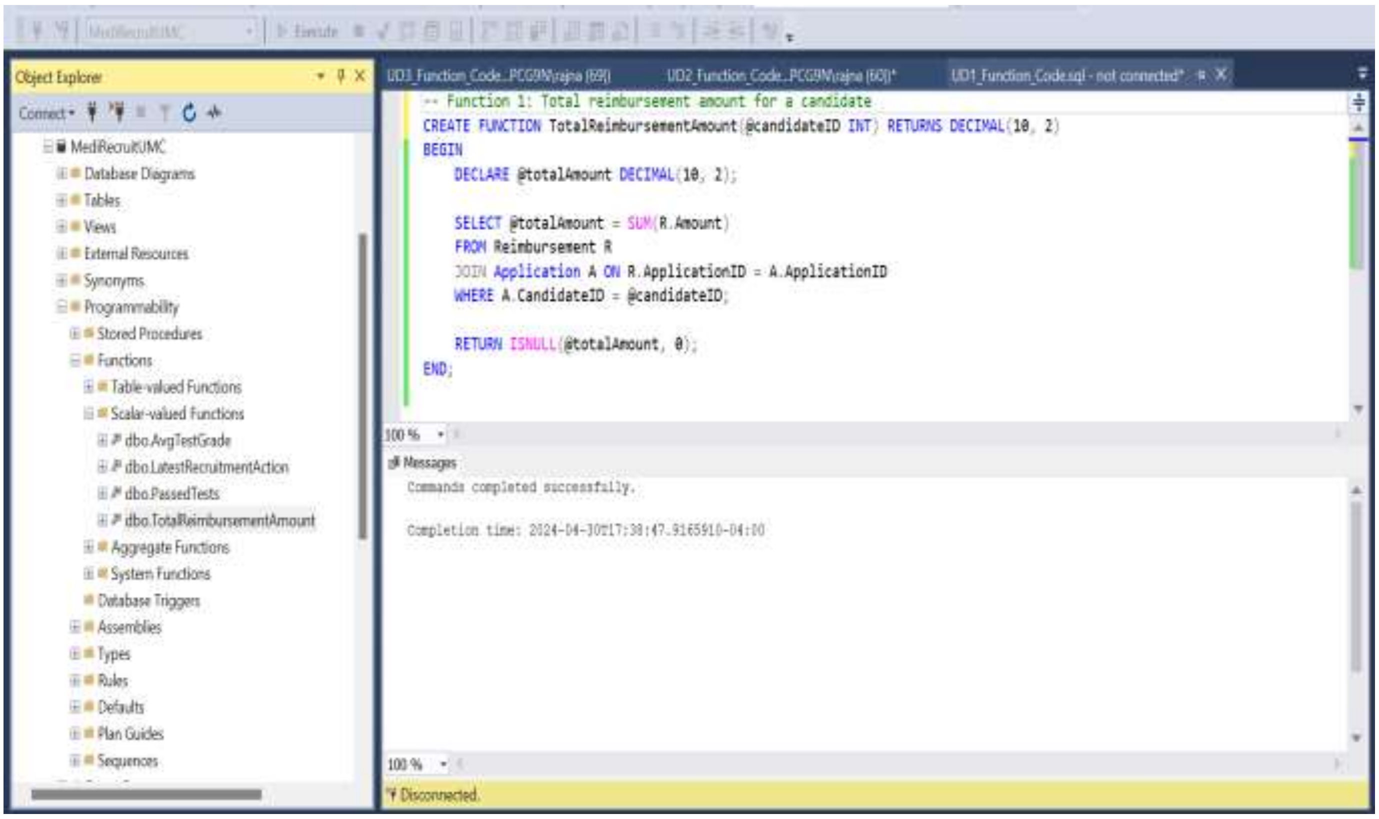
IV. Additionally, UDFs can contribute to performance optimization in database systems. By predefining computations or data transformations within UDFs, users can reduce the need for repetitive code execution and improve query efficiency. For instance, a UDF that performs complex calculations can be invoked multiple times within queries without the need to rewrite the underlying logic. This optimization streamlines database operations, reduces computational overhead, and improves overall system performance, particularly in scenarios with high query frequency or large datasets.

In summary, User-Defined Functions (UDFs) are powerful tools for enhancing database functionality and enforcing business rules. By encapsulating reusable logic, facilitating data transformation, enforcing business rules, and optimizing performance, UDFs contribute to the efficiency, reliability, and integrity of database systems. Their versatility and flexibility make them indispensable components in modern database architectures, empowering users to streamline operations, ensure data consistency, and drive informed decision-making.

Below are Four queries on User-defined Functions:

1). -- Function 1: Total reimbursement amount for a candidate

```
    CREATE FUNCTION TotalReimbursementAmount(@candidateID INT) RETURNS
DECIMAL(10, 2)
    BEGIN
        DECLARE @totalAmount DECIMAL(10, 2);
         SELECT @totalAmount = SUM(R.Amount)
        FROM Reimbursement R
       JOIN Application A ON R.ApplicationID = A.ApplicationID
      WHERE A.CandidateID = @candidateID;

    RETURN ISNULL(@totalAmount, 0);
 END;
```

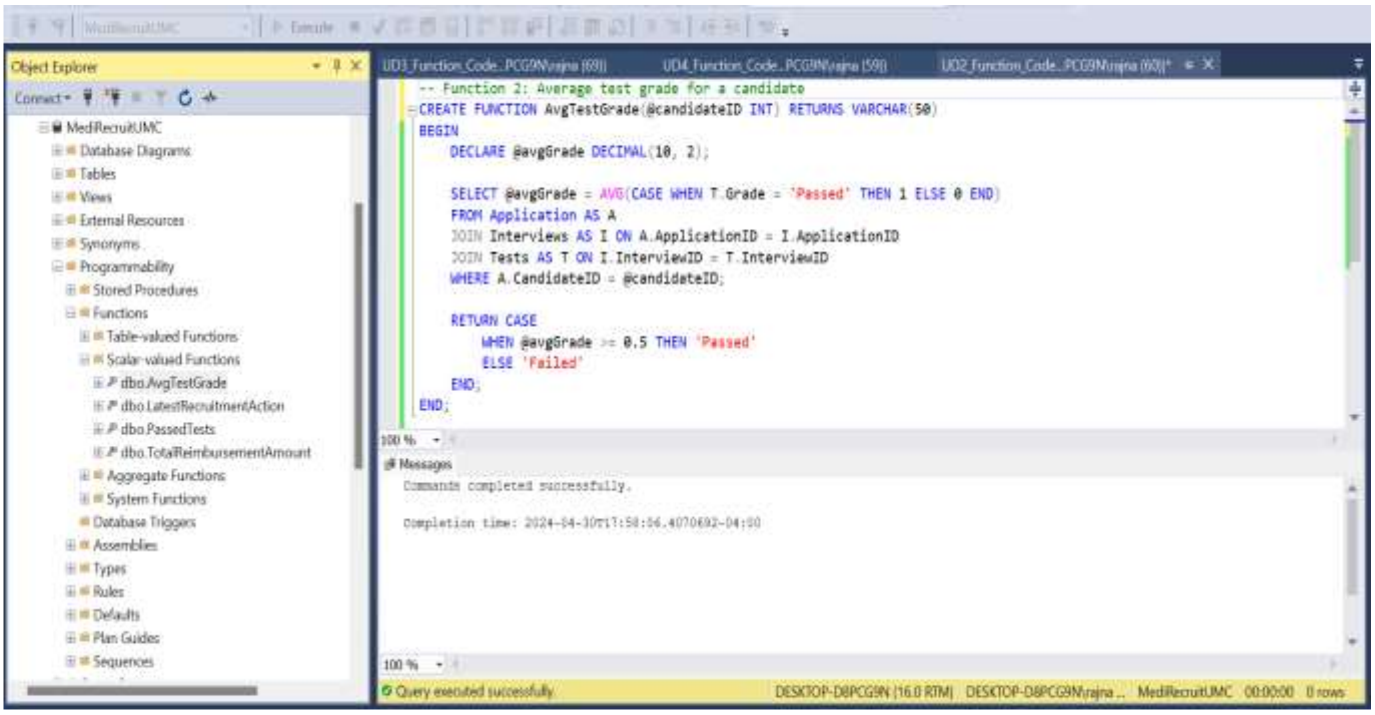


```
-- Function 1: Total reimbursement amount for a candidate
CREATE FUNCTION TotalReimbursementAmount(@candidateID INT) RETURNS DECIMAL(10, 2)
BEGIN
    DECLARE @totalAmount DECIMAL(10, 2);

    SELECT @totalAmount = SUM(R.Amount)
    FROM Reimbursement R
    JOIN Application A ON R.ApplicationID = A.ApplicationID
    WHERE A.CandidateID = @candidateID;

    RETURN ISNULL(@totalAmount, 0);
END;
```

```
Commands completed successfully.

Completion time: 2024-04-30T17:38:47.9165910-04:00
```

The "TotalReimbursementAmount" function calculates the total reimbursement amount for a specified candidate. It takes the candidate's ID as input and returns a decimal value representing the total reimbursement amount. The function first declares a variable to store the total amount. Then, it retrieves reimbursement amounts from the "Reimbursement" table, joining it with the "Application" table based on the ApplicationID. The amounts corresponding to the candidate's applications are summed up. If no reimbursements are found, it returns 0. In summary, this function efficiently computes the total reimbursement amount for a candidate based on their applications.

```
2). -- Function 2: Average test grade for a candidate
CREATE FUNCTION AvgTestGrade(@candidateID INT) RETURNS VARCHAR(50)
BEGIN
  DECLARE @avgGrade DECIMAL(10, 2);

  SELECT @avgGrade = AVG(CASE WHEN T.Grade = 'Passed' THEN 1 ELSE 0 END)
  FROM Application AS A
  JOIN Interviews AS I ON A.ApplicationID = I.ApplicationID
  JOIN Tests AS T ON I.InterviewID = T.InterviewID
  WHERE A.CandidateID = @candidateID;

  RETURN CASE
    WHEN @avgGrade >= 0.5 THEN 'Passed'
    ELSE 'Failed'
  END;
END;
```



```
-- Function 2: Average test grade for a candidate
CREATE FUNCTION AvgTestGrade(@candidateID INT) RETURNS VARCHAR(50)
BEGIN
    DECLARE @avgGrade DECIMAL(10, 2);

    SELECT @avgGrade = AVG(CASE WHEN T.Grade = 'Passed' THEN 1 ELSE 0 END)
    FROM Application AS A
    JOIN Interviews AS I ON A.ApplicationID = I.ApplicationID
    JOIN Tests AS T ON I.InterviewID = T.InterviewID
    WHERE A.CandidateID = @candidateID;

    RETURN CASE
        WHEN @avgGrade >= 0.5 THEN 'Passed'
        ELSE 'Failed'
    END;
END;
```

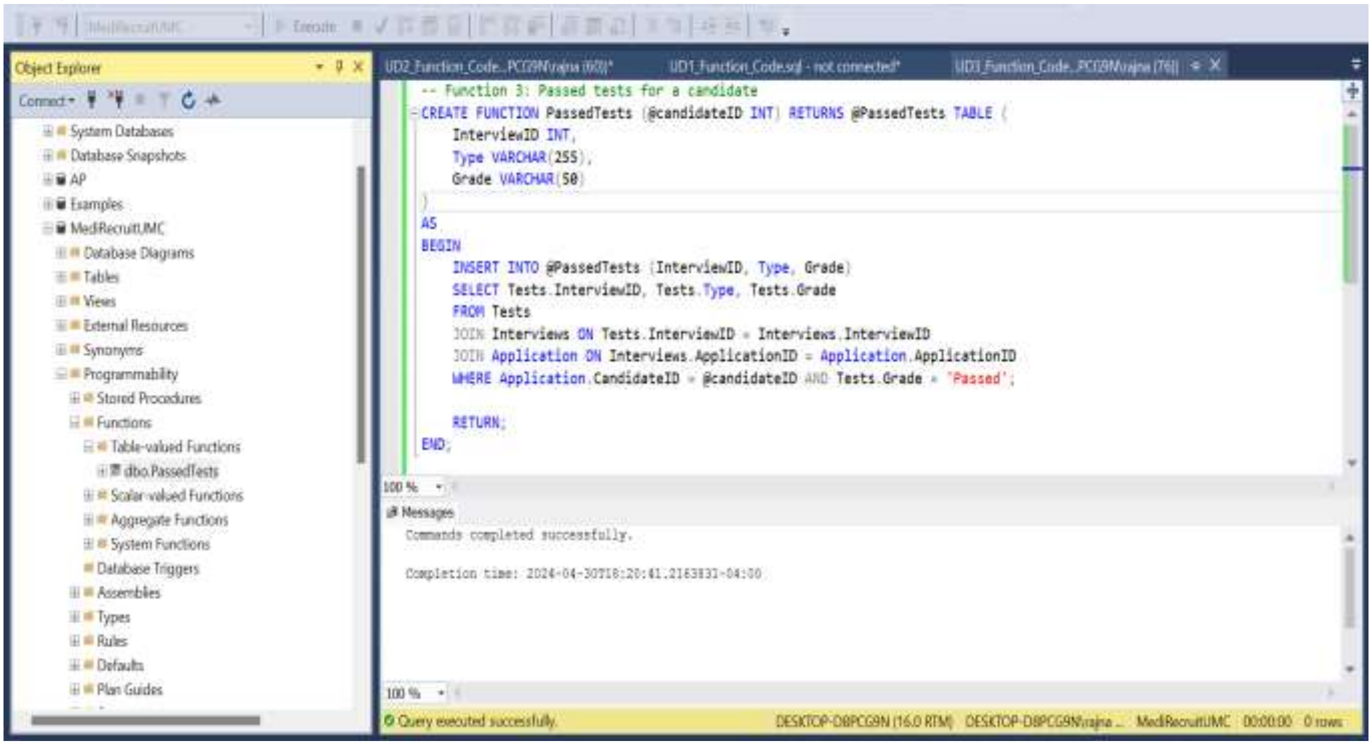
```
Commands completed successfully.

Completion time: 2024-04-30T17:58:06.4070692-04:00
```

```
Query executed successfully.    DESKTOP-D8PCG9N (16.0 RTM)  DESKTOP-D8PCG9N\rajna ...  MediRecruitUMC  00:00:00  0 rows
```

The "AvgTestGrade" function calculates the average test grade for a candidate by evaluating their performance across interviews and associated tests. It computes the average grade by assigning a value of 1 for "Passed" grades and 0 for other grades, then calculates the average based on these values. If the average grade is equal to or greater than 0.5, the function returns "Passed"; otherwise, it returns "Failed". This function effectively assesses the overall test performance of a candidate, providing a concise indication of their proficiency. By encapsulating this logic within a user-defined function, code modularity is promoted, enabling easy reuse and maintenance. Overall, the function facilitates streamlined evaluation of candidate test performance, aiding in decision-making processes within recruitment or assessment contexts.

3). -- Function 3: Passed tests for a candidate

```
 CREATE FUNCTION PassedTests (@candidateID INT) RETURNS @PassedTests TABLE (
InterviewID INT,
     Type VARCHAR(255),
     Grade VARCHAR(50)
 )
AS
BEGIN
    INSERT INTO @PassedTests (InterviewID, Type, Grade)
    SELECT Tests.InterviewID, Tests.Type, Tests.Grade
    FROM Tests
    JOIN Interviews ON Tests.InterviewID = Interviews.InterviewID
   JOIN Application ON Interviews.ApplicationID = Application.ApplicationID
  WHERE Application.CandidateID = @candidateID AND Tests.Grade = 'Passed';

RETURN;
END;
```



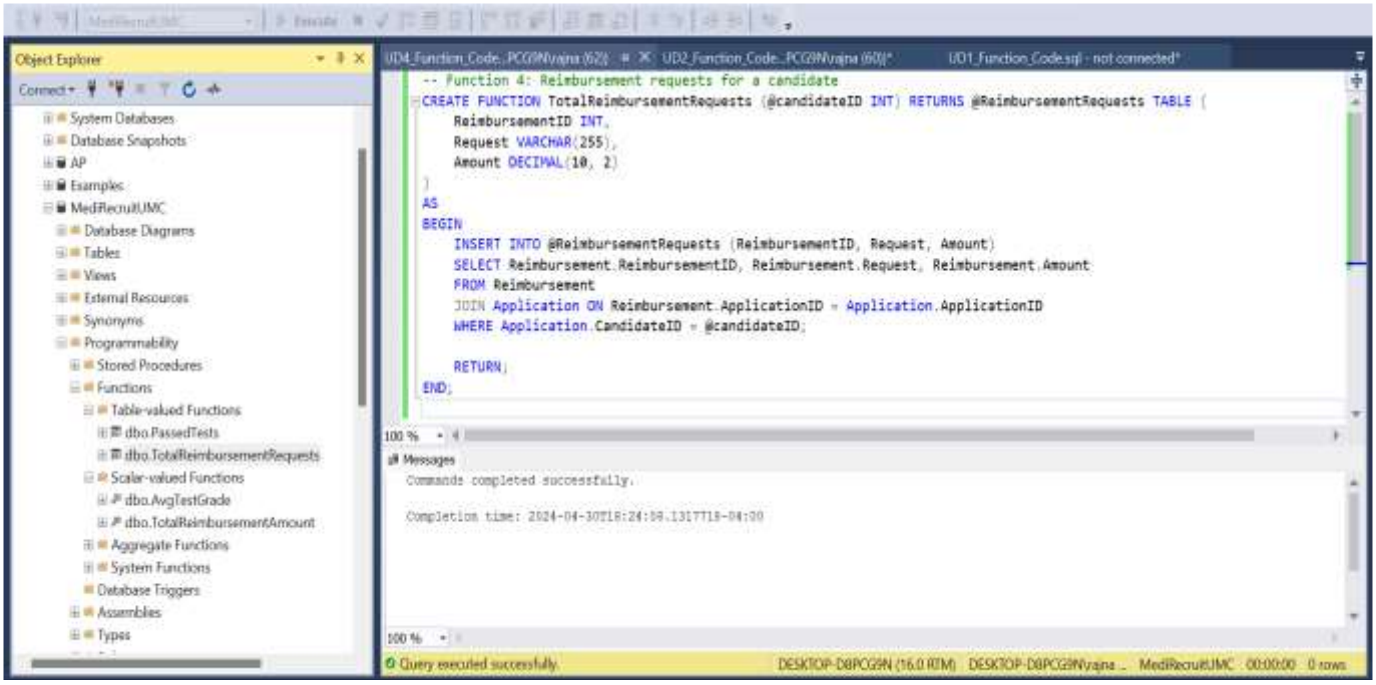
```
-- Function 3: Passed tests for a candidate
CREATE FUNCTION PassedTests (@candidateID INT) RETURNS @PassedTests TABLE (
    InterviewID INT,
    Type VARCHAR(255),
    Grade VARCHAR(50)
)
AS
BEGIN
    INSERT INTO @PassedTests (InterviewID, Type, Grade)
    SELECT Tests.InterviewID, Tests.Type, Tests.Grade
    FROM Tests
    JOIN Interviews ON Tests.InterviewID = Interviews.InterviewID
    JOIN Application ON Interviews.ApplicationID = Application.ApplicationID
    WHERE Application.CandidateID = @candidateID AND Tests.Grade = 'Passed';

    RETURN;
END;
```

```
Commands completed successfully.

Completion time: 2024-04-30T18:20:41.2163831-04:00
```

The "PassedTests" function retrieves details of tests that a candidate has passed. It returns a table containing information about the tests, including the interview ID, test type, and grade. The function selects data from the "Tests" table, joining it with the "Interviews" and "Application" tables based on interview and application IDs, respectively. It filters the results to include only tests with a grade of 'Passed' for the specified candidate ID. The selected data is then inserted into a table variable, which is returned as the output of the function. By encapsulating this logic within a user-defined function, it promotes code reuse and simplifies querying for passed tests. This function aids in evaluating a candidate's performance by providing a consolidated view of their successful test outcomes, facilitating decision-making in recruitment or assessment processes.

4). -- Function 4: Reimbursement requests for a candidate

```
    CREATE FUNCTION TotalReimbursementRequests (@candidateID INT) RETURNS
@ReimbursementRequests TABLE (
    ReimbursementID INT,
    Request VARCHAR(255),
    Amount DECIMAL(10, 2)
    )
  AS
  BEGIN
      INSERT INTO @ReimbursementRequests (ReimbursementID, Request, Amount)
      SELECT Reimbursement.ReimbursementID, Reimbursement.Request,
Reimbursement.Amount
      FROM Reimbursement
      JOIN Application ON Reimbursement.ApplicationID = Application.ApplicationID
     WHERE Application.CandidateID = @candidateID;

     RETURN;
  END;
```



```
-- Function 4: Reimbursement requests for a candidate
CREATE FUNCTION TotalReimbursementRequests (@candidateID INT) RETURNS @ReimbursementRequests TABLE (
    ReimbursementID INT,
    Request VARCHAR(255),
    Amount DECIMAL(10, 2)
)
AS
BEGIN
    INSERT INTO @ReimbursementRequests (ReimbursementID, Request, Amount)
    SELECT Reimbursement.ReimbursementID, Reimbursement.Request, Reimbursement.Amount
    FROM Reimbursement
    JOIN Application ON Reimbursement.ApplicationID = Application.ApplicationID
    WHERE Application.CandidateID = @candidateID;

    RETURN;
END;
```

```
Commands completed successfully.

Completion time: 2024-04-30T18:24:58.1317718-04:00
```

Query executed successfully. DESKTOP-D8PCG9N (16.0 RTM) DESKTOP-D8PCG9N\vajna ... MediRecruitUMC 00:00:00 0 rows

The "TotalReimbursementRequests" function retrieves reimbursement requests made by a specific candidate. It returns a table containing details such as the reimbursement ID, request description, and requested amount. The function selects data from the "Reimbursement" table, joining it with the "Application" table based on the application ID to identify requests associated with the candidate specified by the input parameter. The selected reimbursement requests, along with their respective details, are inserted into a table variable. This table variable serves as the output of the function, containing a consolidated view of all reimbursement requests made by the candidate. By encapsulating this logic within a user-defined function, it promotes code modularity and simplifies the process of retrieving reimbursement information for a specific candidate, facilitating efficient management of reimbursement requests within the database.

Triggers: Enforcing Business Rules and Automating Actions

Triggers in a database are special types of stored procedures that automatically execute in response to specified data manipulation events, such as INSERT, UPDATE, or DELETE operations on tables. They enable database administrators to enforce business rules, maintain data integrity, and automate actions without requiring explicit user intervention.

I. Triggers play a pivotal role in ensuring data integrity by enforcing constraints and rules before data is inserted, updated, or deleted. For example, a trigger can be designed to prevent the deletion of critical records or to ensure that specific fields always contain valid values. By intercepting these data manipulation events, triggers act as gatekeepers, validating data changes against predefined criteria and ensuring that only permissible modifications are allowed. This proactive enforcement of data integrity constraints helps to maintain the accuracy and reliability of the database.

II. Triggers are commonly used to capture and log information about data modifications. By responding to INSERT, UPDATE, and DELETE operations, triggers can record details such as the old and new values of modified records, the user who initiated the change, and the timestamp of the modification. This audit trail of changes provides valuable insight into the history of data modifications, facilitating data analysis, troubleshooting, and compliance auditing. Additionally, it enhances transparency and accountability by enabling administrators to track and review the evolution of data over time.

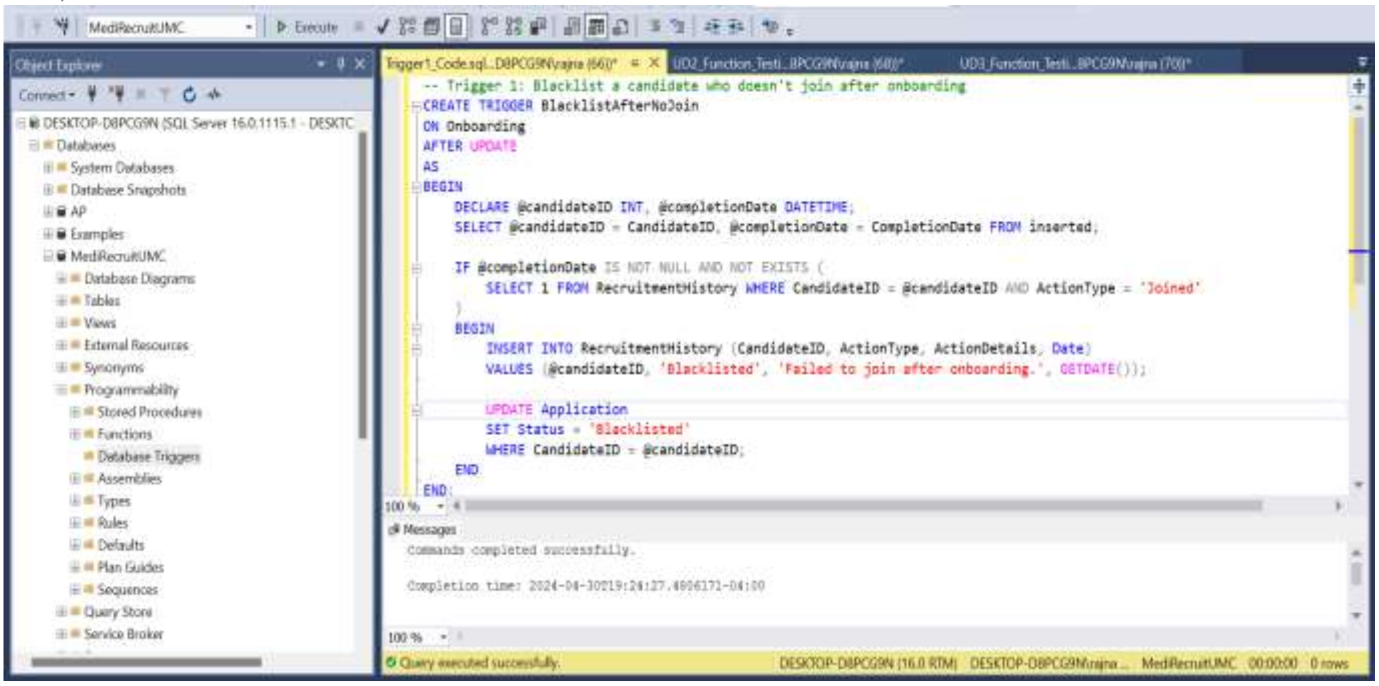
III. Triggers can automate tasks or execute additional actions in response to data manipulation events. For instance, a trigger can be configured to send notifications to relevant stakeholders when certain conditions are met, such as when a high-priority record is updated or when a specific threshold is exceeded. Triggers can also perform complex calculations, update related records, or invoke external processes based on predefined criteria. By automating routine tasks and workflows, triggers streamline business processes, reduce manual intervention, and improve overall efficiency.

IV. Triggers enable the enforcement of complex business rules within the database layer. These rules can include constraints related to referential integrity, cascading updates or deletes, and custom validation logic specific to the organization's requirements. Triggers respond to data manipulation events in real-time, ensuring that business rules are consistently applied and enforced across all database interactions.

In summary, triggers serve as powerful mechanisms for enhancing database functionality and enforcing business rules. By automatically responding to data manipulation events, triggers enable administrators to maintain data integrity, automate tasks, and enforce compliance with business requirements, thereby improving the overall efficiency and reliability of database operations.

Below are Four queries on Triggers:

```sql
1). -- Trigger 1: Select a candidate who has passed the interview
CREATE TRIGGER BlacklistAfterNoJoin
ON Onboarding
AFTER UPDATE
AS
BEGIN
   DECLARE @candidateID INT, @completionDate DATETIME;
   SELECT @candidateID = CandidateID, @completionDate = CompletionDate FROM
inserted;
   IF @completionDate IS NOT NULL AND NOT EXISTS (
      SELECT 1 FROM RecruitmentHistory WHERE CandidateID = @candidateID AND
ActionType = 'Joined'
   )
   BEGIN
      INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
      VALUES (@candidateID, 'Blacklisted', 'Failed to join after onboarding.', GETDATE());
      UPDATE Application
      SET Status = 'Blacklisted'
      WHERE CandidateID = @candidateID;
   END
END;
```



```sql
-- Trigger 1: Blacklist a candidate who doesn't join after onboarding
CREATE TRIGGER BlacklistAfterNoJoin
ON Onboarding
AFTER UPDATE
AS
BEGIN
    DECLARE @candidateID INT, @completionDate DATETIME;
    SELECT @candidateID = CandidateID, @completionDate = CompletionDate FROM inserted;

    IF @completionDate IS NOT NULL AND NOT EXISTS (
        SELECT 1 FROM RecruitmentHistory WHERE CandidateID = @candidateID AND ActionType = 'Joined'
    )
    BEGIN
        INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
        VALUES (@candidateID, 'Blacklisted', 'Failed to join after onboarding.', GETDATE());

        UPDATE Application
        SET Status = 'Blacklisted'
        WHERE CandidateID = @candidateID;
    END
END;
```

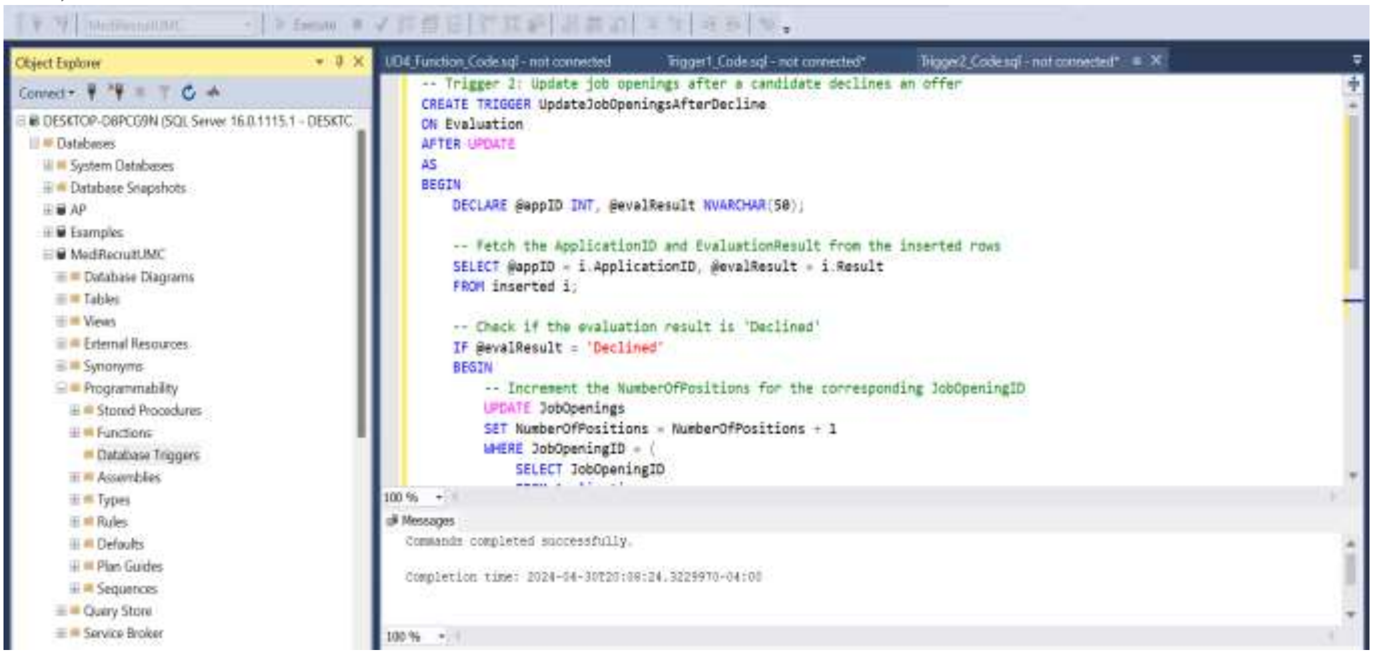
```
Commands completed successfully.

Completion time: 2024-04-30T19:24:27.4806171-04:00
```

Query executed successfully. DESKTOP-D8PCG9N (16.0 RTM) DESKTOP-D8PCG9N\rajna ... MediRecruitUMC 00:00:00 0 rows

This trigger, "BlacklistAfterNoJoin," fires after an update operation on the "Onboarding" table. It selects the candidate ID and completion date from the "inserted" table. If the completion date is not null and the candidate hasn't joined (based on absence of 'Joined' action in RecruitmentHistory), it logs the candidate as "Blacklisted" in the RecruitmentHistory table with a corresponding action detail and the current date. Additionally, it updates the status of the candidate's application to 'Blacklisted' in the Application table. This trigger automates the blacklisting process for candidates who fail to join after the onboarding process, enhancing recruitment management and enforcing business rules.

2). -- Trigger 2: Update job openings after a candidate declines an offer

```
CREATE TRIGGER UpdateJobOpeningsAfterDecline
ON Evaluation
AFTER UPDATE
AS
BEGIN
   DECLARE @appID INT, @evalResult NVARCHAR(50);
   -- Fetch the ApplicationID and EvaluationResult from the inserted rows
   SELECT @appID = i.ApplicationID, @evalResult = i.Result
   FROM inserted i;
   -- Check if the evaluation result is 'Declined'
   IF @evalResult = 'Declined'
   BEGIN
      -- Increment the NumberOfPositions for the corresponding JobOpeningID
      UPDATE JobOpenings
      SET NumberOfPositions = NumberOfPositions + 1
      WHERE JobOpeningID = (
         SELECT JobOpeningID
         FROM Application
         WHERE ApplicationID = @appID
      );
   END
END;
```

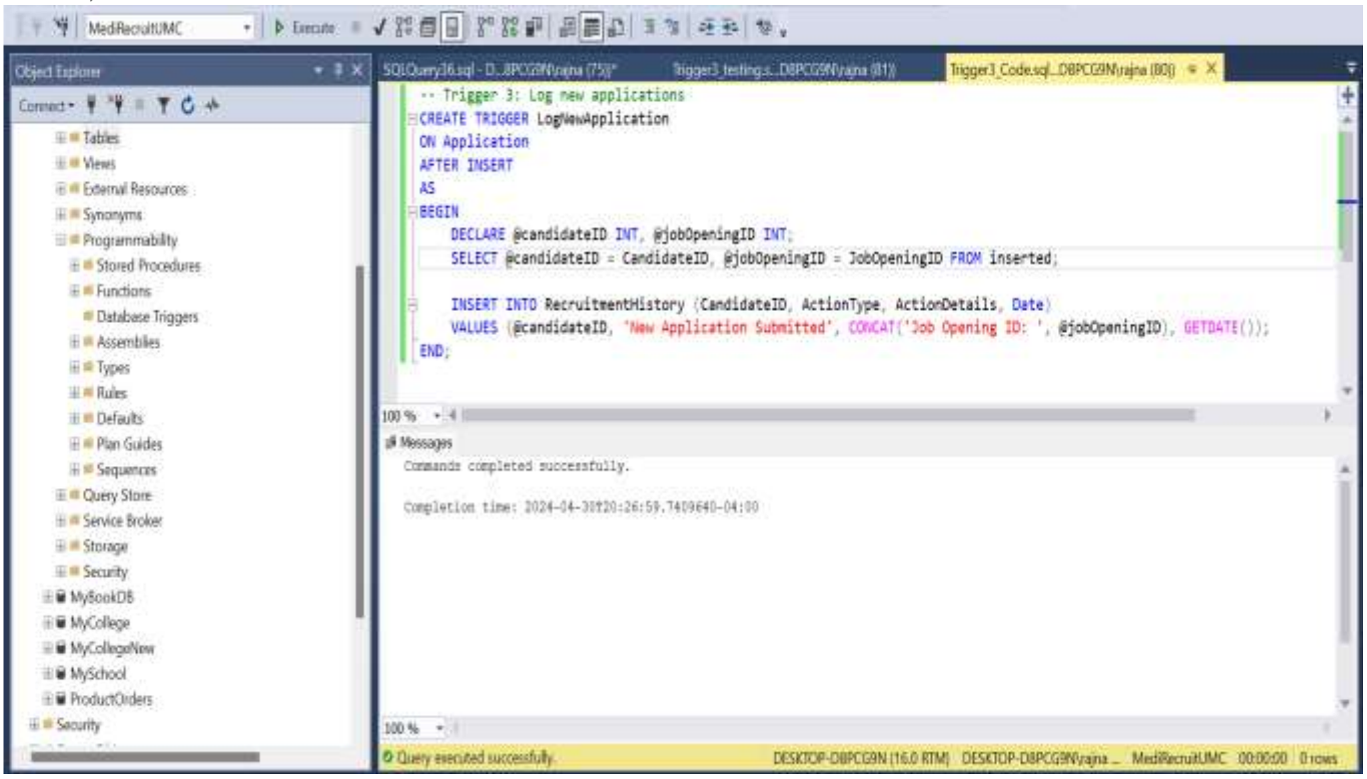


```
-- Trigger 2: Update job openings after a candidate declines an offer
CREATE TRIGGER UpdateJobOpeningsAfterDecline
ON Evaluation
AFTER UPDATE
AS
BEGIN
    DECLARE @appID INT, @evalResult NVARCHAR(50);

    -- Fetch the ApplicationID and EvaluationResult from the inserted rows
    SELECT @appID = i.ApplicationID, @evalResult = i.Result
    FROM inserted i;

    -- Check if the evaluation result is 'Declined'
    IF @evalResult = 'Declined'
    BEGIN
        -- Increment the NumberOfPositions for the corresponding JobOpeningID
        UPDATE JobOpenings
        SET NumberOfPositions = NumberOfPositions + 1
        WHERE JobOpeningID = (
            SELECT JobOpeningID
```

```
Commands completed successfully.

Completion time: 2024-04-30T20:08:24.3229970-04:00
```

This trigger, "UpdateJobOpeningsAfterDecline," executes after an update operation on the "Evaluation" table. It fetches the ApplicationID and EvaluationResult from the "inserted" rows. If the evaluation result is 'Declined,' it increments the NumberOfPositions for the corresponding JobOpeningID by one in the "JobOpenings" table. This trigger automates the adjustment of job openings availability when a candidate declines an offer, ensuring that the number of available positions is accurately reflected.

```
3). -- Trigger 3: Log new applications
     CREATE TRIGGER LogNewApplication
     ON Application
    AFTER INSERT
    AS
   BEGIN
       DECLARE @candidateID INT, @jobOpeningID INT;
       SELECT @candidateID = CandidateID, @jobOpeningID = JobOpeningID FROM inserted;

       INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
      VALUES (@candidateID, 'New Application Submitted', CONCAT('Job Opening ID: ',
@jobOpeningID), GETDATE());
  END;
```



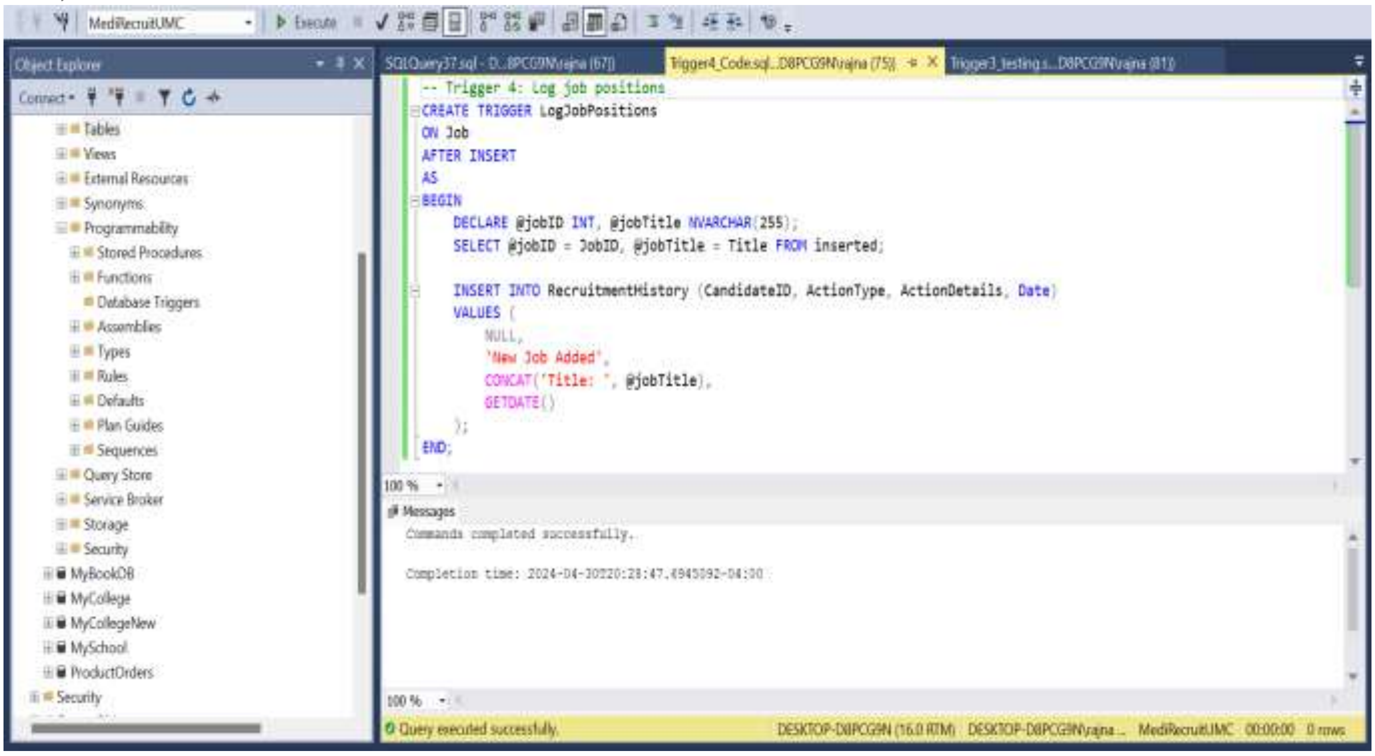
```
-- Trigger 3: Log new applications
CREATE TRIGGER LogNewApplication
ON Application
AFTER INSERT
AS
BEGIN
    DECLARE @candidateID INT, @jobOpeningID INT;
    SELECT @candidateID = CandidateID, @jobOpeningID = JobOpeningID FROM inserted;

    INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
    VALUES (@candidateID, 'New Application Submitted', CONCAT('Job Opening ID: ', @jobOpeningID), GETDATE());
END;
```

```
Commands completed successfully.

Completion time: 2024-04-30T20:26:59.7409640-04:00
```

Query executed successfully. DESKTOP-D8PCG9N (16.0 RTM) DESKTOP-D8PCG9N\rajna ... MediRecruitUMC 00:00:00 0 rows

The "LogNewApplication" trigger fires after an insertion operation on the "Application" table. It retrieves the CandidateID and JobOpeningID from the "inserted" table, representing the newly inserted application. Then, it inserts a record into the "RecruitmentHistory" table, documenting the event of a new application submission. This log entry includes details such as the CandidateID, the action type ('New Application Submitted'), the specific Job Opening ID associated with the application, and the timestamp of the event. Essentially, this trigger automates the logging of new application submissions, providing a comprehensive record of recruitment activities. It enhances transparency and accountability in the recruitment process, enabling stakeholders to track the flow of applications and monitor recruitment trends over time. Additionally, it aids in auditing and analysis by providing a historical record of application submissions.

```
4). -- Trigger 4: Log job positions
CREATE TRIGGER LogJobPositions
ON Job
AFTER INSERT
AS
BEGIN
  DECLARE @jobID INT, @jobTitle NVARCHAR(255);
  SELECT @jobID = JobID, @jobTitle = Title FROM inserted;

  INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
  VALUES (
    NULL,
    'New Job Added',
    CONCAT('Title: ', @jobTitle),
    GETDATE()
  );
END;
```



```
-- Trigger 4: Log job positions
CREATE TRIGGER LogJobPositions
ON Job
AFTER INSERT
AS
BEGIN
    DECLARE @jobID INT, @jobTitle NVARCHAR(255);
    SELECT @jobID = JobID, @jobTitle = Title FROM inserted;

    INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
    VALUES (
        NULL,
        'New Job Added',
        CONCAT('Title: ', @jobTitle),
        GETDATE()
    );
END;
```

```
Commands completed successfully.

Completion time: 2024-04-30T20:28:47.4945092-04:00
```

Query executed successfully. DESKTOP-D8PCG9N (16.0 RTM) DESKTOP-D8PCG9N\rajna... MediRecruitUMC 00:00:00 0 rows

The "LogJobPositions" trigger executes after an insertion operation on the "Job" table. It retrieves the JobID and Title of the newly inserted job from the "inserted" table. Subsequently, it inserts a record into the "RecruitmentHistory" table to log the addition of a new job position. The log entry includes details such as the action type ('New Job Added'), the job title, and the timestamp of the event. As the trigger is executed after inserting a new job, it does not associate a specific candidate with the action. Therefore, it sets the CandidateID field to NULL. Essentially, this trigger automates the logging of new job positions, providing a historical record of job additions within the organization. It enhances transparency and facilitates tracking of organizational growth and changes in staffing needs over time.

Transactions: Safeguarding Data Consistency and Integrity

Transactions are a foundational concept in database management systems, serving as a fundamental mechanism for ensuring data consistency, integrity, and reliability. A transaction represents a logical unit of work that consists of one or more operations, such as INSERT, UPDATE, DELETE, or SELECT, which are treated as a single indivisible unit. By adhering to the principles of **ACID** (Atomicity, Consistency, Isolation, Durability), transactions provide a robust framework for managing data manipulation operations within a database environment.

Atomicity:
Atomicity ensures that transactions are either fully completed or fully rolled back in case of failure. This means that if any part of a transaction fails, all changes made by the transaction are undone, leaving the database in its original state. For example, if a transaction involves transferring funds from one account to another and the transfer fails midway due to a system crash, atomicity guarantees that neither account is affected, and the transaction is rolled back to maintain data integrity.

Consistency:
Consistency ensures that the database remains in a valid state before and after the execution of a transaction. Transactions must adhere to integrity constraints, business rules, and other validation criteria to prevent the database from entering an inconsistent state. For instance, if a transaction attempts to insert data that violates a unique key constraint or a foreign key constraint, the transaction is aborted to maintain the overall consistency of the database.

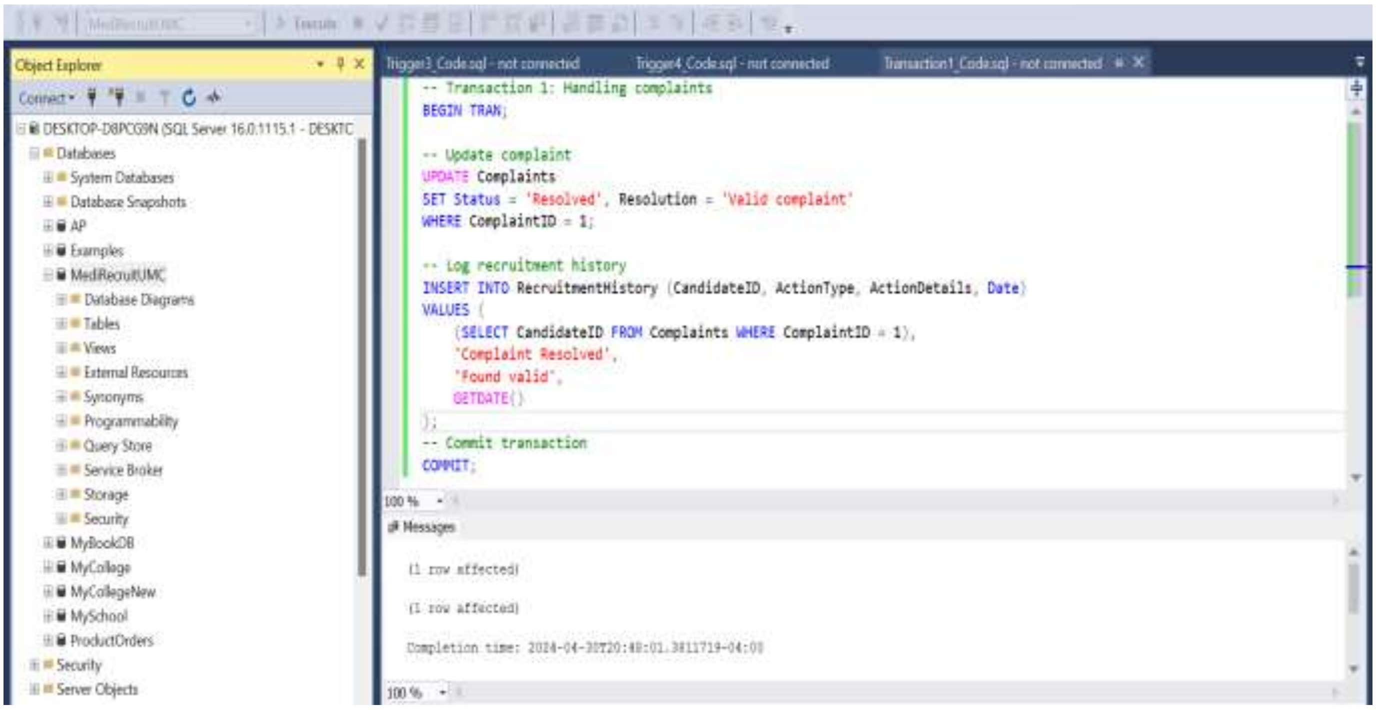
Isolation:
Isolation ensures that transactions operate independently of one another and are isolated from concurrent transactions. This prevents interference between transactions and maintains data integrity by ensuring that each transaction sees a consistent snapshot of the database. Isolation levels such as Read Uncommitted, Read Committed, Repeatable Read, and Serializable control the degree of isolation between transactions, with higher isolation levels providing stronger guarantees at the expense of performance.

Durability:
Durability guarantees that once a transaction is committed, its changes are permanently saved in the database and are not lost in the event of a system failure or crash. This is achieved through mechanisms such as transaction logging and write-ahead logging, which ensure that committed transactions are durably stored on disk before being acknowledged to the user. As a result, even in the face of hardware failures, power outages, or other catastrophic events, the database can be recovered to a consistent state by replaying logged transactions.

In summary, transactions are essential for maintaining data consistency, integrity, and reliability in database systems. By providing atomicity, consistency, isolation, and durability guarantees, transactions ensure that database operations are performed reliably and that the database remains in a consistent state, even in the presence of failures or concurrent access. As a fundamental building block of database management, transactions enable organizations to manage complex data manipulation tasks with confidence, knowing that their data remains accurate, reliable, and secure.

Below are the four queries on transaction:

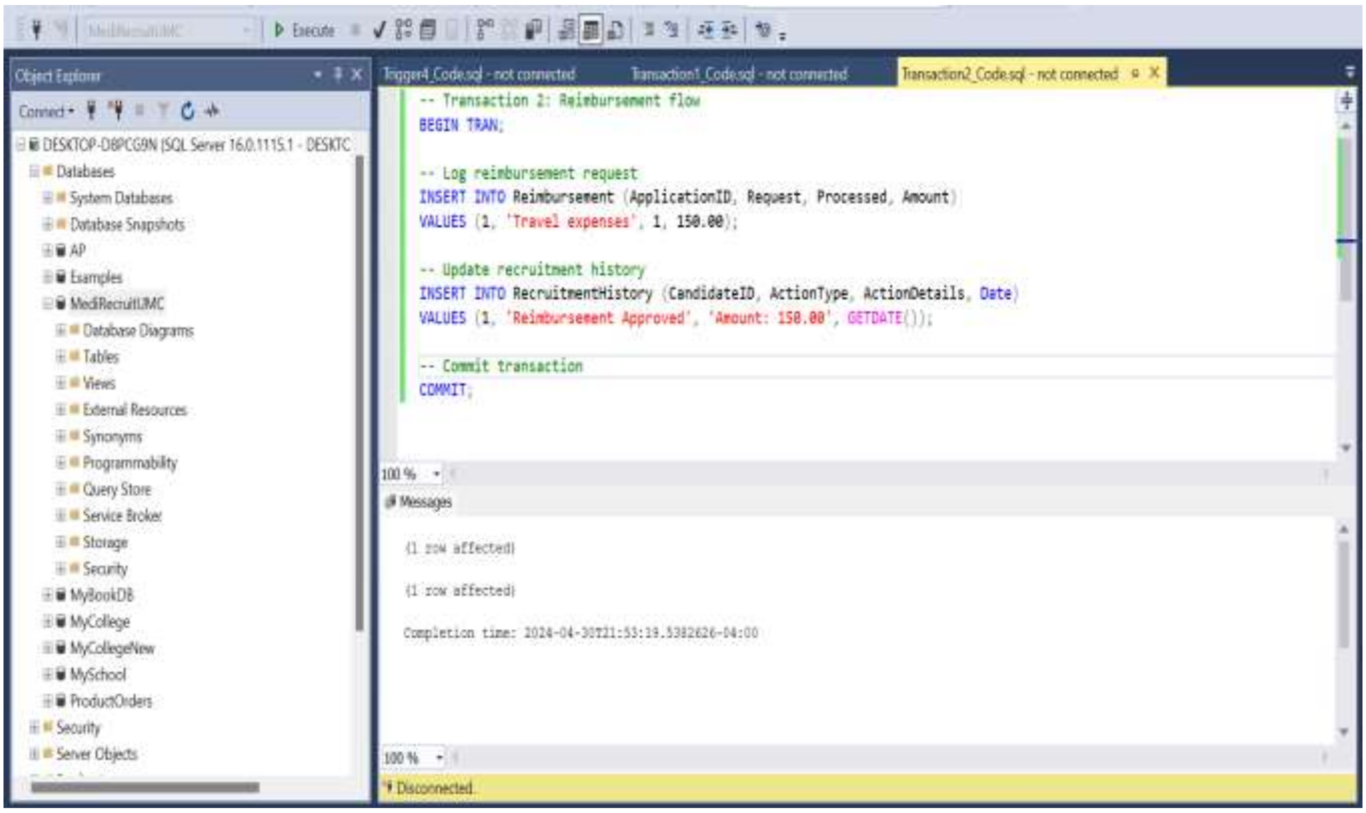
1).
```
-- Transaction 1: Handling complaints
    BEGIN TRAN;
 -- Update complaint
   UPDATE Complaints
   SET Status = 'Resolved', Resolution = 'Valid complaint'
  WHERE ComplaintID = 1;
-- Log recruitment history
  INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
  VALUES (
     (SELECT CandidateID FROM Complaints WHERE ComplaintID = 1),
     'Complaint Resolved',
     'Found valid',
     GETDATE()
     );
-- Commit transaction
  COMMIT;
```


```
-- Transaction 1: Handling complaints
BEGIN TRAN;

-- Update complaint
UPDATE Complaints
SET Status = 'Resolved', Resolution = 'Valid complaint'
WHERE ComplaintID = 1;

-- Log recruitment history
INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
VALUES (
    (SELECT CandidateID FROM Complaints WHERE ComplaintID = 1),
    'Complaint Resolved',
    'Found valid',
    GETDATE()
);
-- Commit transaction
COMMIT;
```

```
(1 row affected)

(1 row affected)

Completion time: 2024-04-30T20:48:01.3811719-04:00
```

This transaction begins with a sequence of database operations aimed at handling complaints. First, it updates the status and resolution of a specific complaint identified by ComplaintID=1 in the "Complaints" table, marking it as resolved with a valid resolution. Subsequently, it logs this resolution action in the "RecruitmentHistory" table, detailing the resolution of the complaint and associating it with the corresponding candidate. Finally, the transaction is committed, ensuring that all changes made within the transaction are permanently saved in the database. By encapsulating these operations within a transaction, data consistency and integrity are maintained, ensuring that all changes are either fully completed or rolled back in case of failure, thus safeguarding the reliability of the database.

2).
```
-- Transaction 2: Reimbursement flow
 BEGIN TRAN;

-- Log reimbursement request
 INSERT INTO Reimbursement (ApplicationID, Request, Processed, Amount)
 VALUES (1, 'Travel expenses', 1, 150.00);

-- Update recruitment history
INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
VALUES (1, 'Reimbursement Approved', 'Amount: 150.00', GETDATE());

-- Commit transaction
COMMIT;
```



```
-- Transaction 2: Reimbursement flow
BEGIN TRAN;

-- Log reimbursement request
INSERT INTO Reimbursement (ApplicationID, Request, Processed, Amount)
VALUES (1, 'Travel expenses', 1, 150.00);

-- Update recruitment history
INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
VALUES (1, 'Reimbursement Approved', 'Amount: 150.00', GETDATE());

-- Commit transaction
COMMIT;
```

Messages

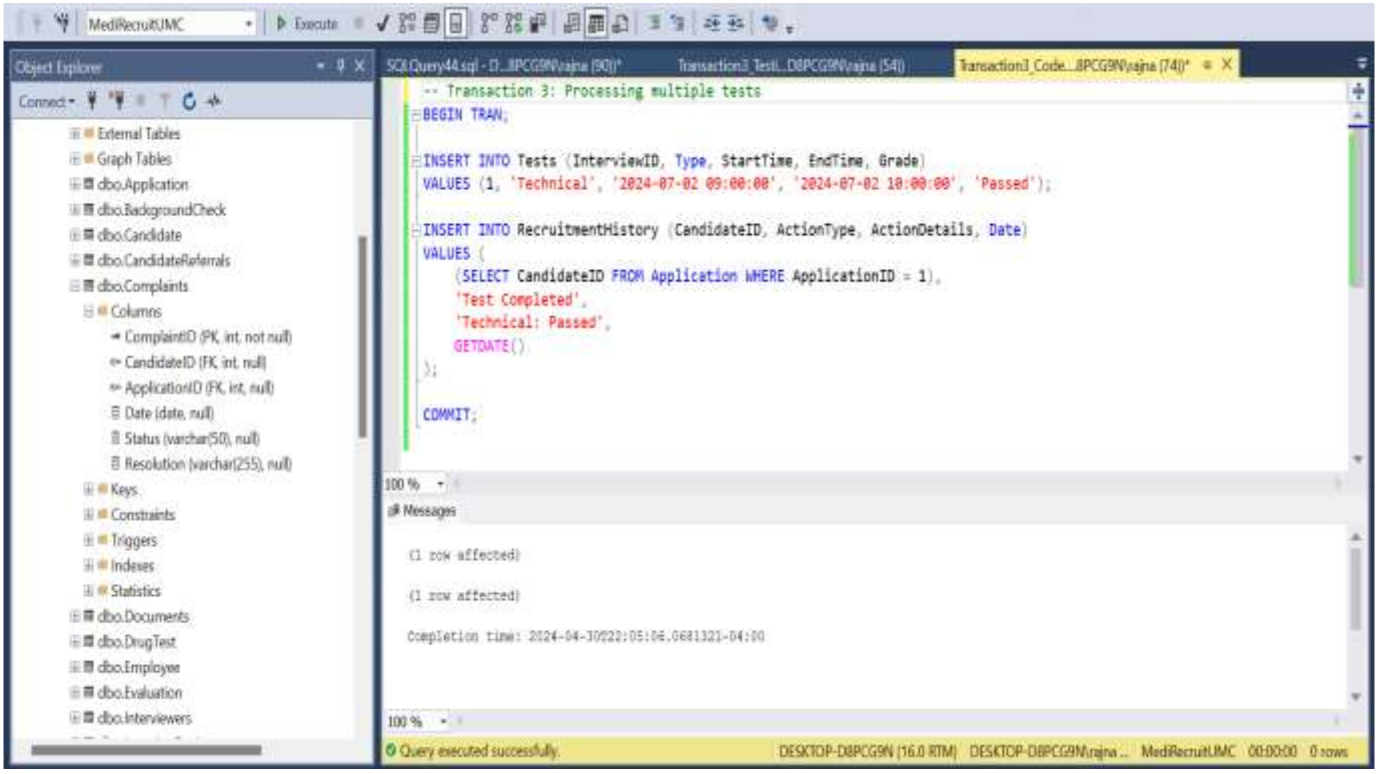
```
(1 row affected)

(1 row affected)

Completion time: 2024-04-30T21:53:19.5382626-04:00
```

Disconnected.

This transaction initiates the reimbursement process by logging a reimbursement request and updating the recruitment history accordingly. Firstly, it inserts a record into the "Reimbursement" table, specifying details such as the ApplicationID (identifying the associated application), the nature of the request (travel expenses), whether it's been processed (1 for processed), and the reimbursement amount ($150.00). Next, it logs this reimbursement approval action in the "RecruitmentHistory" table, associating it with the relevant candidate (CandidateID=1), and detailing the approved amount. Finally, the transaction is committed, ensuring that all changes made within the transaction are permanently saved in the database. This transaction streamlines the reimbursement flow, ensuring that requests are accurately logged and documented in the recruitment history, thereby enhancing transparency and accountability in the reimbursement process.

3). -- Transaction 3: Processing multiple tests

```
  BEGIN TRAN;

  INSERT INTO Tests (InterviewID, Type, StartTime, EndTime, Grade)
  VALUES (1, 'Technical', '2024-07-02 09:00:00', '2024-07-02 10:00:00', 'Passed');

 INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
 VALUES (
     (SELECT CandidateID FROM Application WHERE ApplicationID = 1),
      'Test Completed',
      'Technical: Passed',
      GETDATE()
 );
COMMIT;
```



```
-- Transaction 3: Processing multiple tests
BEGIN TRAN;

INSERT INTO Tests (InterviewID, Type, StartTime, EndTime, Grade)
VALUES (1, 'Technical', '2024-07-02 09:00:00', '2024-07-02 10:00:00', 'Passed');

INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
VALUES (
    (SELECT CandidateID FROM Application WHERE ApplicationID = 1),
    'Test Completed',
    'Technical: Passed',
    GETDATE()
);

COMMIT;
```

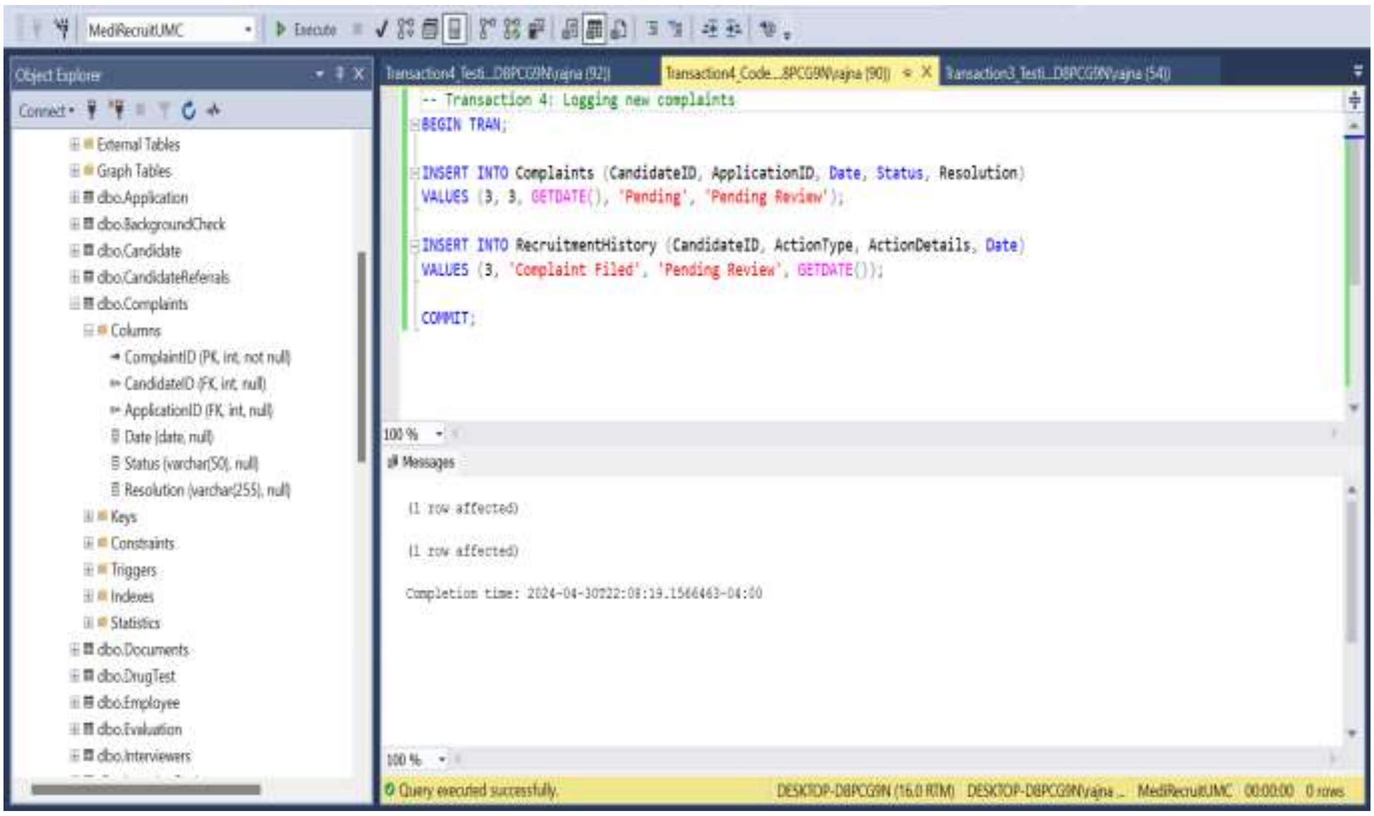
```
(1 row affected)

(1 row affected)

Completion time: 2024-04-30T22:05:06.0681321-04:00
```

Query executed successfully. DESKTOP-D8PCG9N (16.0 RTM) DESKTOP-D8PCG9N\rajna ... MediRecruitUMC 00:00:00 0 rows

This transaction focuses on processing multiple tests associated with a candidate. Firstly, it inserts a record into the "Tests" table, specifying details such as the InterviewID, type of test (Technical), start and end times, and the test grade (Passed). Subsequently, it logs this test completion action in the "RecruitmentHistory" table, associating it with the corresponding candidate identified by ApplicationID=1. The action details include the type of test completed (Technical) and the outcome (Passed), providing a comprehensive record of the candidate's test performance. Finally, the transaction is committed, ensuring that all changes made within the transaction are permanently saved in the database. This transaction streamlines the process of recording test completions and outcomes, facilitating efficient tracking and analysis of candidate performance during the recruitment process.

4). -- Transaction 4: Logging new complaints

```
BEGIN TRAN;

INSERT INTO Complaints (CandidateID, ApplicationID, Date, Status, Resolution)
VALUES (3, 3, GETDATE(), 'Pending', 'Pending Review');

INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
VALUES (3, 'Complaint Filed', 'Pending Review', GETDATE());

COMMIT;
```



```
-- Transaction 4: Logging new complaints
BEGIN TRAN;

INSERT INTO Complaints (CandidateID, ApplicationID, Date, Status, Resolution)
VALUES (3, 3, GETDATE(), 'Pending', 'Pending Review');

INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
VALUES (3, 'Complaint Filed', 'Pending Review', GETDATE());

COMMIT;
```

```
(1 row affected)

(1 row affected)

Completion time: 2024-04-30T22:08:19.1566463-04:00
```

```
Query executed successfully.    DESKTOP-D8PCG9N (16.0 RTM)  DESKTOP-D8PCG9N\rajna ...  MediRecruitUMC  00:00:00  0 rows
```

This transaction is designed to handle the logging of new complaints within the database. It begins by inserting a new complaint record into the "Complaints" table, specifying details such as the CandidateID, ApplicationID, current date, status (Pending), and resolution (Pending Review). This action effectively records the complaint submission, capturing essential information about the complainant, the associated application, and the complaint status. Subsequently, the transaction logs this complaint filing action in the "RecruitmentHistory" table, associating it with the candidate identified by CandidateID=3. The action details include the type of action (Complaint Filed) and the status of the complaint (Pending Review), providing a comprehensive record of complaint activities. Finally, the transaction is committed, ensuring that all changes made within the transaction are permanently saved in the database. This transactional approach ensures data consistency and integrity while facilitating the efficient handling of complaints within the recruitment process.

User Management: Efficient Access Privilege Control

User management in a database system is essential for controlling access privileges efficiently. By creating users with varying security levels, assigning passwords, and defining roles, administrators can effectively manage data security and access control within the database environment.

I. Creating Users with Different Security Levels and Passwords:

Administrators can create users with different security levels by specifying the complexity of their passwords. High-security users may require strong passwords containing a mix of uppercase and lowercase letters, numbers, and special characters, while low-security users may have simpler passwords. Creating users with appropriate password complexity helps mitigate the risk of unauthorized access.

II. Assigning Roles to Users:

Roles define sets of permissions that determine the actions users can perform within the database. By assigning users to roles based on their responsibilities and access requirements, administrators can streamline access control and ensure compliance with security policies. For example, roles such as 'Admin', 'Manager', and 'Employee' may have different levels of access privileges, allowing users to perform tasks according to their roles.

Efficient user management involves considering the following:

I. **Granular Access Control:** Define roles and permissions at a granular level to ensure users only have access to the data and functionalities necessary for their roles.

II. **Regular Password Updates:** Implement policies for regular password updates to enhance security and minimize the risk of password-related vulnerabilities.

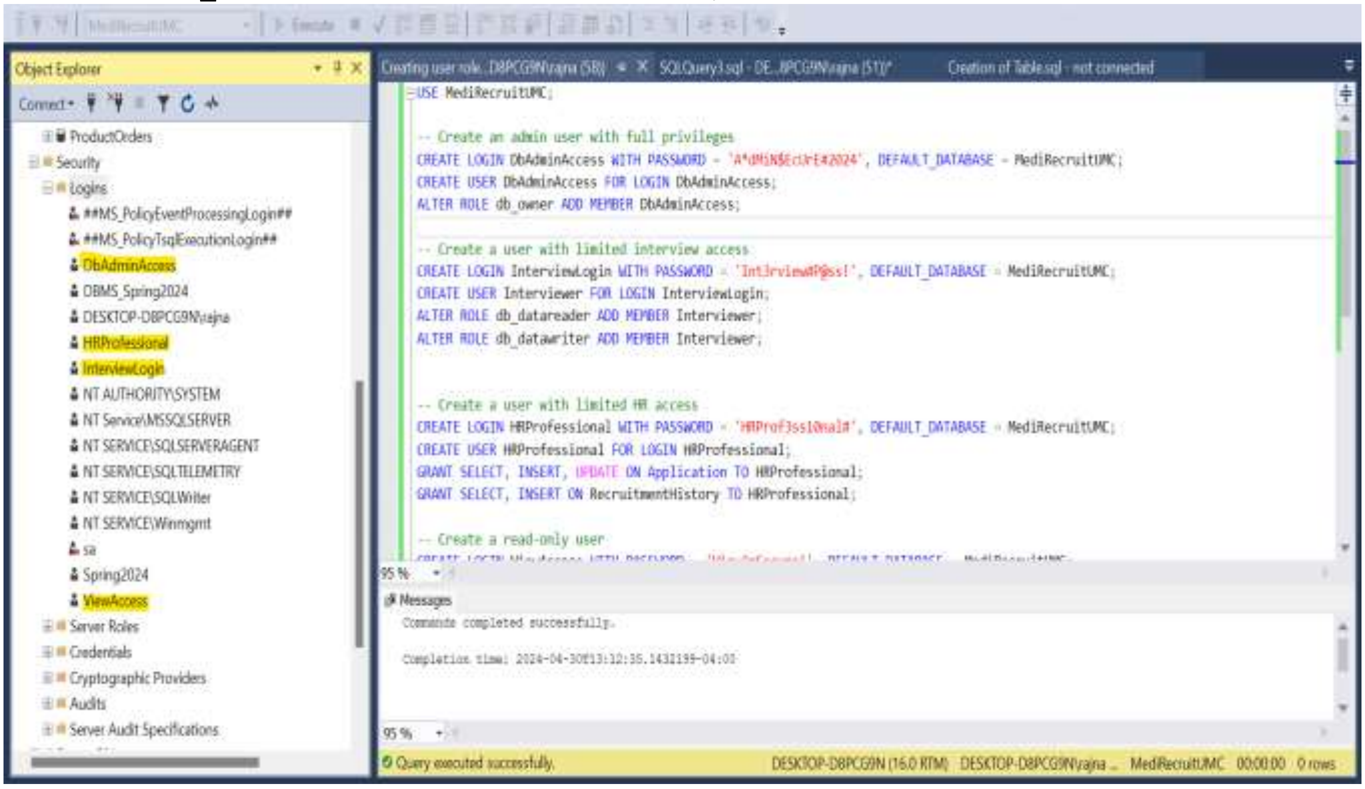
III. **Role-Based Access Control (RBAC):** Implement RBAC strategies to assign users to roles based on their job functions, simplifying user management, and ensuring consistent access control.

IV. **Audit Trails:** Maintain audit trails to track user activities and changes to access privileges, enabling administrators to monitor user behaviour and detect potential security breaches.

By implementing robust user management practices, organizations can effectively control access privileges, protect sensitive data, and maintain regulatory compliance within their database systems. Efficient user management is crucial for safeguarding data integrity and confidentiality while facilitating secure and seamless access for authorized users.

SQL Query:

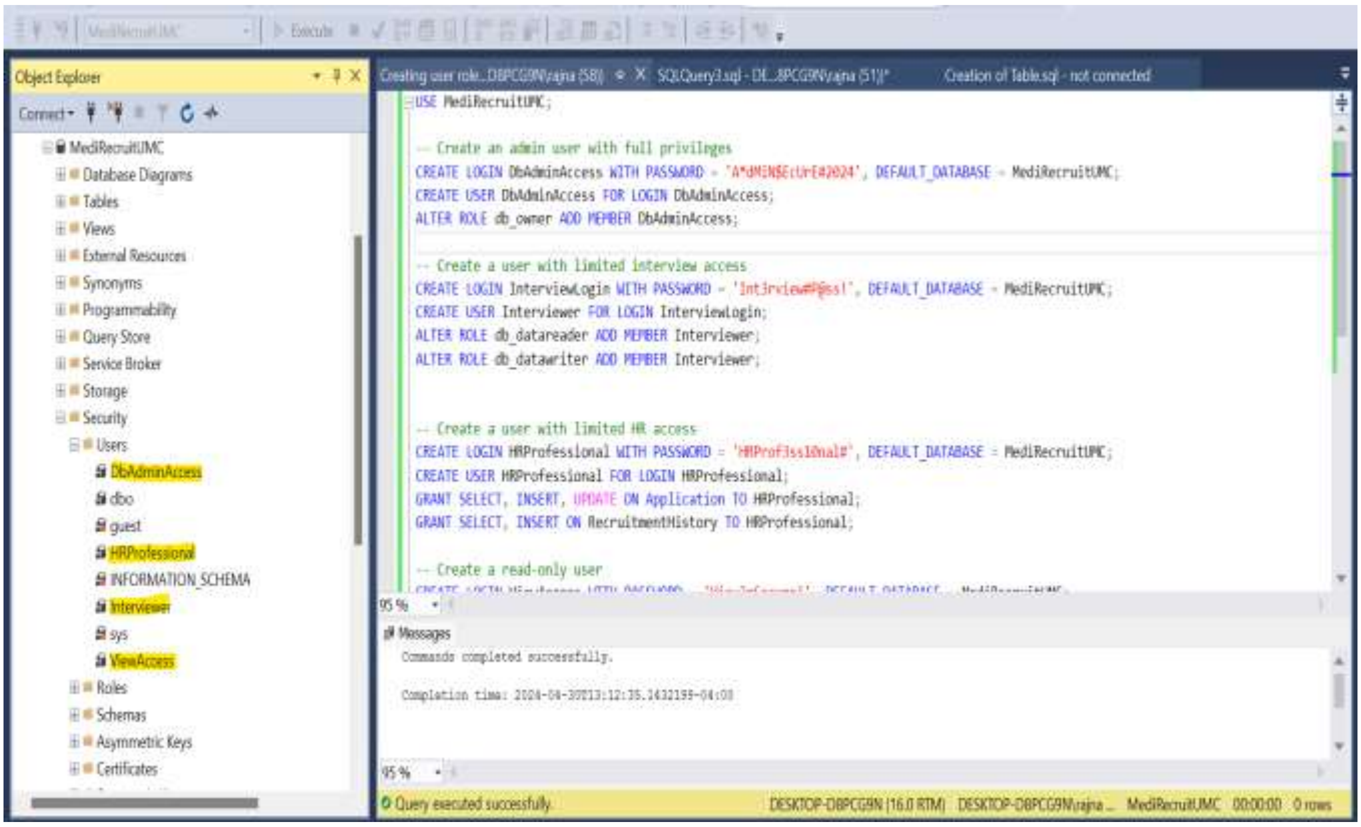
```
USE MediRecruitUMC;
-- Create an admin user with full privileges
CREATE LOGIN DbAdminAccess WITH PASSWORD = 'A*dMiN$EcUrE#2024',
DEFAULT_DATABASE = MediRecruitUMC;
CREATE USER DbAdminAccess FOR LOGIN DbAdminAccess;
ALTER ROLE db_owner ADD MEMBER DbAdminAccess;

-- Create a user with limited interview access
CREATE LOGIN InterviewLogin WITH PASSWORD = 'Int3rview#P@ss!', DEFAULT_DATABASE
= MediRecruitUMC;
CREATE USER Interviewer FOR LOGIN InterviewLogin;
ALTER ROLE db_datareader ADD MEMBER Interviewer;
ALTER ROLE db_datawriter ADD MEMBER Interviewer;

-- Create a user with limited HR access
CREATE LOGIN HRProfessional WITH PASSWORD = 'HRProf3ss10nal#', DEFAULT_DATABASE
= MediRecruitUMC;
CREATE USER HRProfessional FOR LOGIN HRProfessional;
GRANT SELECT, INSERT, UPDATE ON Application TO HRProfessional;
GRANT SELECT, INSERT ON RecruitmentHistory TO HRProfessional;

-- Create a read-only user
CREATE LOGIN ViewAccess WITH PASSWORD = 'View3rSecure!', DEFAULT_DATABASE =
MediRecruitUMC;
CREATE USER ViewAccess FOR LOGIN ViewAccess;
ALTER ROLE db_datareader ADD MEMBER ViewAccess;
```



```
USE MediRecruitUMC;

-- Create an admin user with full privileges
CREATE LOGIN DbAdminAccess WITH PASSWORD = 'A*dMiN$EcUrE#2024', DEFAULT_DATABASE = MediRecruitUMC;
CREATE USER DbAdminAccess FOR LOGIN DbAdminAccess;
ALTER ROLE db_owner ADD MEMBER DbAdminAccess;

-- Create a user with limited interview access
CREATE LOGIN InterviewLogin WITH PASSWORD = 'Int3rview#P@ss!', DEFAULT_DATABASE = MediRecruitUMC;
CREATE USER Interviewer FOR LOGIN InterviewLogin;
ALTER ROLE db_datareader ADD MEMBER Interviewer;
ALTER ROLE db_datawriter ADD MEMBER Interviewer;


-- Create a user with limited HR access
CREATE LOGIN HRProfessional WITH PASSWORD = 'HRProf3ss10nal#', DEFAULT_DATABASE = MediRecruitUMC;
CREATE USER HRProfessional FOR LOGIN HRProfessional;
GRANT SELECT, INSERT, UPDATE ON Application TO HRProfessional;
GRANT SELECT, INSERT ON RecruitmentHistory TO HRProfessional;

-- Create a read-only user
```

```
Commands completed successfully.

Completion time: 2024-04-30T13:12:35.1432195-04:00
```

Query executed successfully. DESKTOP-DBPCG9N (16.0 RTM) DESKTOP-DBPCG9N\rajna ... MediRecruitUMC 00:00:00 0 rows



```
USE MediRecruitUMC;

-- Create an admin user with full privileges
CREATE LOGIN DbAdminAccess WITH PASSWORD = 'A*dMIN$EcUrE#2024', DEFAULT_DATABASE = MediRecruitUMC;
CREATE USER DbAdminAccess FOR LOGIN DbAdminAccess;
ALTER ROLE db_owner ADD MEMBER DbAdminAccess;

-- Create a user with limited interview access
CREATE LOGIN InterviewLogin WITH PASSWORD = 'Int3rview#P@ss!', DEFAULT_DATABASE = MediRecruitUMC;
CREATE USER Interviewer FOR LOGIN InterviewLogin;
ALTER ROLE db_datareader ADD MEMBER Interviewer;
ALTER ROLE db_datawriter ADD MEMBER Interviewer;


-- Create a user with limited HR access
CREATE LOGIN HRProfessional WITH PASSWORD = 'HRProf3ss10nal#', DEFAULT_DATABASE = MediRecruitUMC;
CREATE USER HRProfessional FOR LOGIN HRProfessional;
GRANT SELECT, INSERT, UPDATE ON Application TO HRProfessional;
GRANT SELECT, INSERT ON RecruitmentHistory TO HRProfessional;

-- Create a read-only user
```

```
Commands completed successfully.

Completion time: 2024-04-30T13:12:35.1432195-04:00
```

```
Query executed successfully.    DESKTOP-DBPCG9N (16.0 RTM)  DESKTOP-DBPCG9N\rajna ...  MediRecruitUMC  00:00:00  0 rows
```

The above SQL code sets up user accounts with different levels of access and privileges within the database named "**MediRecruitUMC**". Let's summarize the actions taken:

Admin User Creation: An admin user named "DbAdminAccess" is created with full privileges. This user is granted ownership of the database.

Limited Interview Access: A user named "Interviewer" is created with limited access specifically for conducting interviews. This user is granted read and write permissions to all tables in the database.

Limited HR Access: An HR user named "HRProfessional" is created with restricted access tailored for HR tasks. This user is granted select, insert, and update permissions on the "Application" table and select, insert permissions on the "RecruitmentHistory" table.

Read-Only User Creation: A read-only user named "ViewAccess" is created for viewing data within the database. This user is granted read-only access to all tables in the database.

In summary, the SQL code establishes user accounts with varying levels of access and permissions to facilitate different roles and responsibilities within the MediRecruitUMC database. This approach ensures that users have appropriate access rights to perform their respective tasks while maintaining security and data integrity.

C. TESTING

"Ensuring Data Integrity and Performance: Testing SQL Queries, Views, Triggers, Stored Procedures, and Business Report Generation"

This project focuses on testing the robustness and effectiveness of SQL queries, views, triggers, stored procedures, and business report generation in a healthcare recruitment database system. Through the insertion of meaningful test data and the execution of complex scenarios, the database's logic, integrity, and performance are rigorously evaluated. Various test cases are designed to assess the system's ability to handle integrity constraints, security vulnerabilities, and transactional operations seamlessly. Additionally, the generation of comprehensive business reports provides stakeholders with actionable insights into recruitment trends and process efficiency, further validating the database's functionality and usability. This testing phase underscores the importance of thorough evaluation in ensuring the reliability and effectiveness of database systems in real-world applications.

Testing Database Logic and Integrity

1. Testing Views:

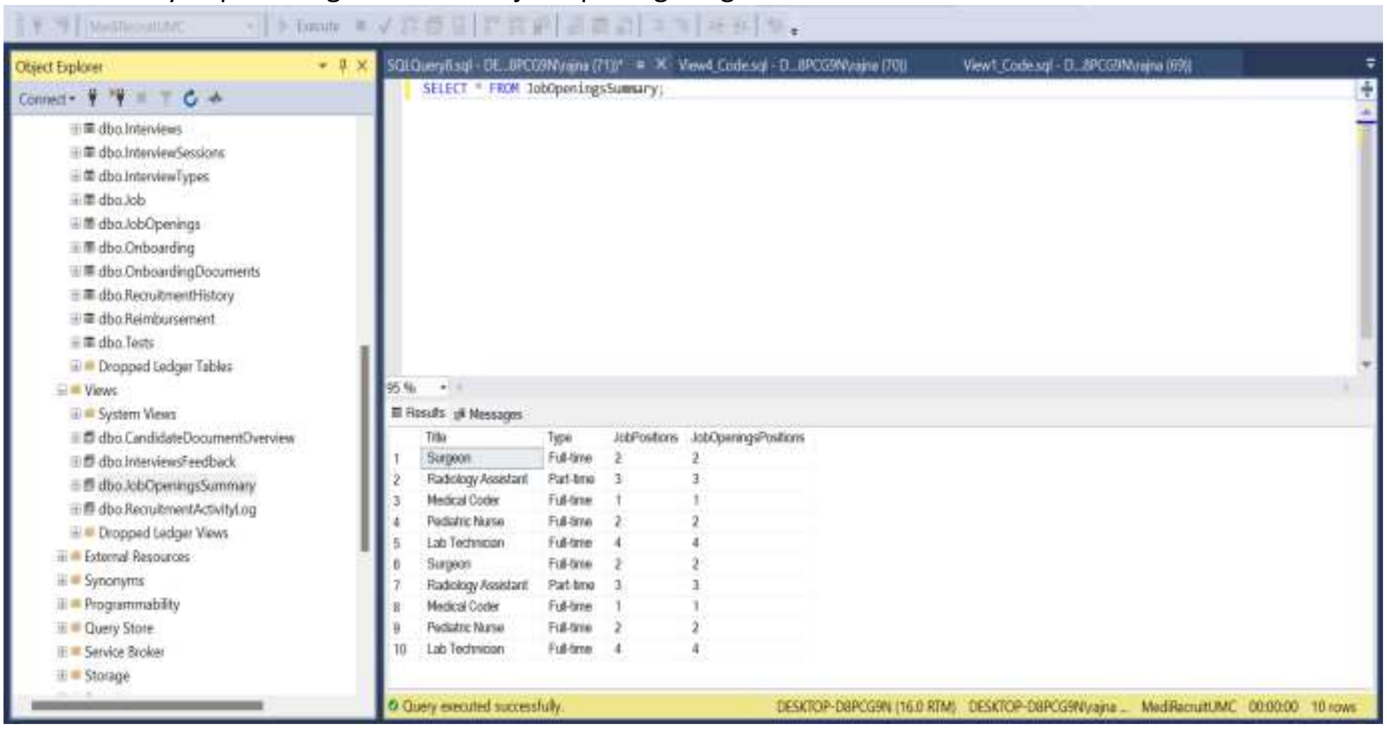
```
 -- View 1: Summary of job openings
CREATE VIEW JobOpeningsSummary AS
SELECT Job.Title, Job.Type, Job.NumberOfPositions AS JobPositions,
JobOpenings.NumberOfPositions AS JobOpeningsPositions
FROM Job
        JOIN JobOpenings ON Job.JobID = JobOpenings.JobID;
```

Testing: Job Openings Summary View
During the testing phase, the "JobOpeningsSummary" view undergoes scrutiny to validate its functionality and adherence to intended specifications.

SQL Code:

```
SELECT * FROM JobOpeningsSummary;
```

Purpose of the View:
The "JobOpeningsSummary" view serves to offer a consolidated overview of job openings by amalgamating pertinent data from the "Job" and "JobOpenings" tables. It selects essential information like job title, type, total positions defined in the "Job" table, and current available positions from the "JobOpenings" table.
Expected Output:
Upon executing the testing query, the output showcases job openings' summary data, effectively consolidating information from both the "Job" and "JobOpenings" tables. The displayed rows offer a comprehensive overview of each job opening, encompassing key details essential for decision-making processes. Throughout the testing phase, any deviations from expected results are scrutinized and rectified to ensure the view's accuracy and reliability in providing summarized job opening insights.



```
SELECT * FROM JobOpeningsSummary;
```

	Title	Type	JobPositions	JobOpeningsPositions
1	Surgeon	Full-time	2	2
2	Radiology Assistant	Part-time	3	3
3	Medical Coder	Full-time	1	1
4	Pediatric Nurse	Full-time	2	2
5	Lab Technician	Full-time	4	4
6	Surgeon	Full-time	2	2
7	Radiology Assistant	Part-time	3	3
8	Medical Coder	Full-time	1	1
9	Pediatric Nurse	Full-time	2	2
10	Lab Technician	Full-time	4	4

Query executed successfully. DESKTOP-D8PCG9N (16.0 RTM) DESKTOP-D8PCG9N\rajna ... MediRecruitUMC 00:00:00 10 rows

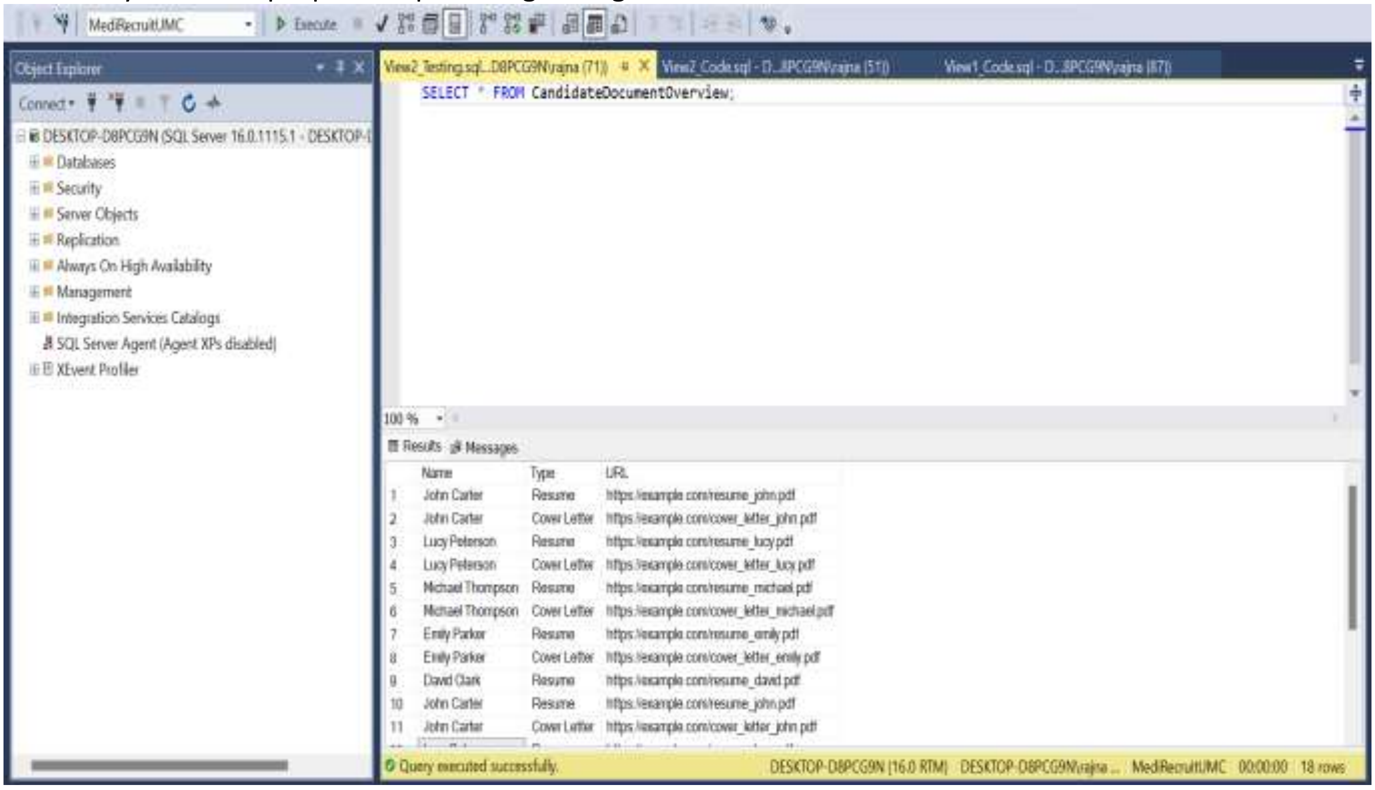
```
-- View 2: Candidate Documents Overview
        CREATE VIEW CandidateDocumentOverview AS
        SELECT Candidate.Name, Documents.Type, Documents.URL
        FROM Candidate
        JOIN Documents ON Candidate.CandidateID = Documents.CandidateID;
```

Testing: Candidate Documents Overview View
In the testing phase, the "CandidateDocumentOverview" view is evaluated to verify its functionality and alignment with intended objectives.

SQL Code:
```
SELECT * FROM CandidateDocumentOverview;
```

Purpose of the View:
The "CandidateDocumentOverview" view aims to present a consolidated overview of candidate documents by integrating relevant data from the "Candidate" and "Documents" tables. It selects vital information such as candidate names, document types, and URLs for easy access and reference.

Expected Output:
Upon executing the testing query, the output should reveal a comprehensive summary of candidate documents. Each row should provide essential details, including the candidate's name, document type, and URL, facilitating efficient document management and retrieval. During the testing phase, any inconsistencies or discrepancies between the expected and actual results are identified and addressed to ensure the accuracy and reliability of the view. This meticulous testing process ensures that the "CandidateDocumentOverview" view effectively fulfills its purpose of providing an organized overview of candidate documents.


MediRecruitUMC
Execute
Object Explorer
Connect
DESKTOP-D8PCG9N (SQL Server 16.0.1115.1 - DESKTOP-
Databases
Security
Server Objects
Replication
Always On High Availability
Management
Integration Services Catalogs
SQL Server Agent (Agent XPs disabled)
XEvent Profiler
View2_Testing.sql...D8PCG9N\rajna (71))
View2_Code.sql - D...8PCG9N\rajna (51))
View1_Code.sql - D...8PCG9N\rajna (87))
100 %
Results
Messages

```
SELECT * FROM CandidateDocumentOverview;
```

	Name	Type	URL
1	John Carter	Resume	https://example.com/resume_john.pdf
2	John Carter	Cover Letter	https://example.com/cover_letter_john.pdf
3	Lucy Peterson	Resume	https://example.com/resume_lucy.pdf
4	Lucy Peterson	Cover Letter	https://example.com/cover_letter_lucy.pdf
5	Michael Thompson	Resume	https://example.com/resume_michael.pdf
6	Michael Thompson	Cover Letter	https://example.com/cover_letter_michael.pdf
7	Emily Parker	Resume	https://example.com/resume_emily.pdf
8	Emily Parker	Cover Letter	https://example.com/cover_letter_emily.pdf
9	David Clark	Resume	https://example.com/resume_david.pdf
10	John Carter	Resume	https://example.com/resume_john.pdf
11	John Carter	Cover Letter	https://example.com/cover_letter_john.pdf

Query executed successfully. DESKTOP-D8PCG9N (16.0 RTM) DESKTOP-D8PCG9N\rajna ... MediRecruitUMC 00:00:00 18 rows

```
-- View 3: Recruitment Activity Log
CREATE VIEW RecruitmentActivityLog AS
SELECT RecruitmentHistory.CandidateID, RecruitmentHistory.ActionType,
RecruitmentHistory.ActionDetails, RecruitmentHistory.Date
FROM RecruitmentHistory;
```

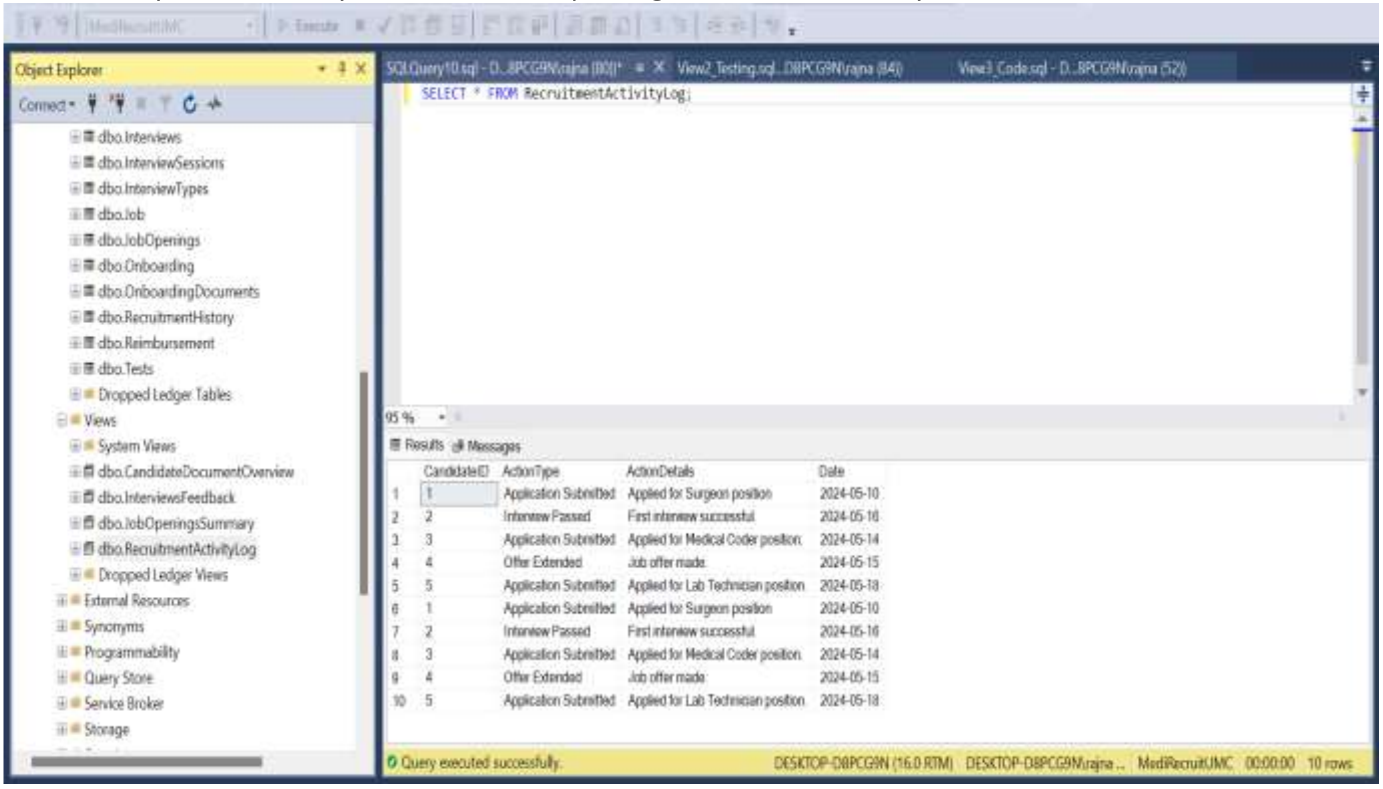
Testing: Recruitment Activity Log View
During the testing phase, the "RecruitmentActivityLog" view undergoes examination to validate its functionality and conformity with the intended objectives.

SQL Query:

```
SELECT * FROM RecruitmentActivityLog;
```

Purpose of the View:
The "RecruitmentActivityLog" view is designed to present a comprehensive log of recruitment activities by consolidating pertinent data from the "RecruitmentHistory" table. It selects essential information such as candidate IDs, action types, action details, and dates to provide a chronological record of recruitment events.

Expected Output:
Upon executing the testing query, the output should display a detailed log of recruitment activities. Each row should encapsulate crucial details, including candidate ID, action type, action details, and the date of the activity. This comprehensive overview facilitates tracking and analysis of recruitment processes, enabling stakeholders to monitor the progression of candidate interactions and recruitment milestones. Throughout the testing phase, any disparities between the expected and actual results are identified and resolved to ensure the accuracy and reliability of the view in capturing recruitment activity data.



```
SELECT * FROM RecruitmentActivityLog;
```

	CandidateID	ActionType	ActionDetails	Date
1	1	Application Submitted	Applied for Surgeon position	2024-05-10
2	2	Interview Passed	First interview successful	2024-05-16
3	3	Application Submitted	Applied for Medical Coder position	2024-05-14
4	4	Offer Extended	Job offer made	2024-05-15
5	5	Application Submitted	Applied for Lab Technician position	2024-05-18
6	1	Application Submitted	Applied for Surgeon position	2024-05-10
7	2	Interview Passed	First interview successful	2024-05-16
8	3	Application Submitted	Applied for Medical Coder position	2024-05-14
9	4	Offer Extended	Job offer made	2024-05-15
10	5	Application Submitted	Applied for Lab Technician position	2024-05-18

Query executed successfully. DESKTOP-D8PCG9N (16.0 RTM) DESKTOP-D8PCG9N\rajna ... MediRecruitUMC 00:00:00 10 rows

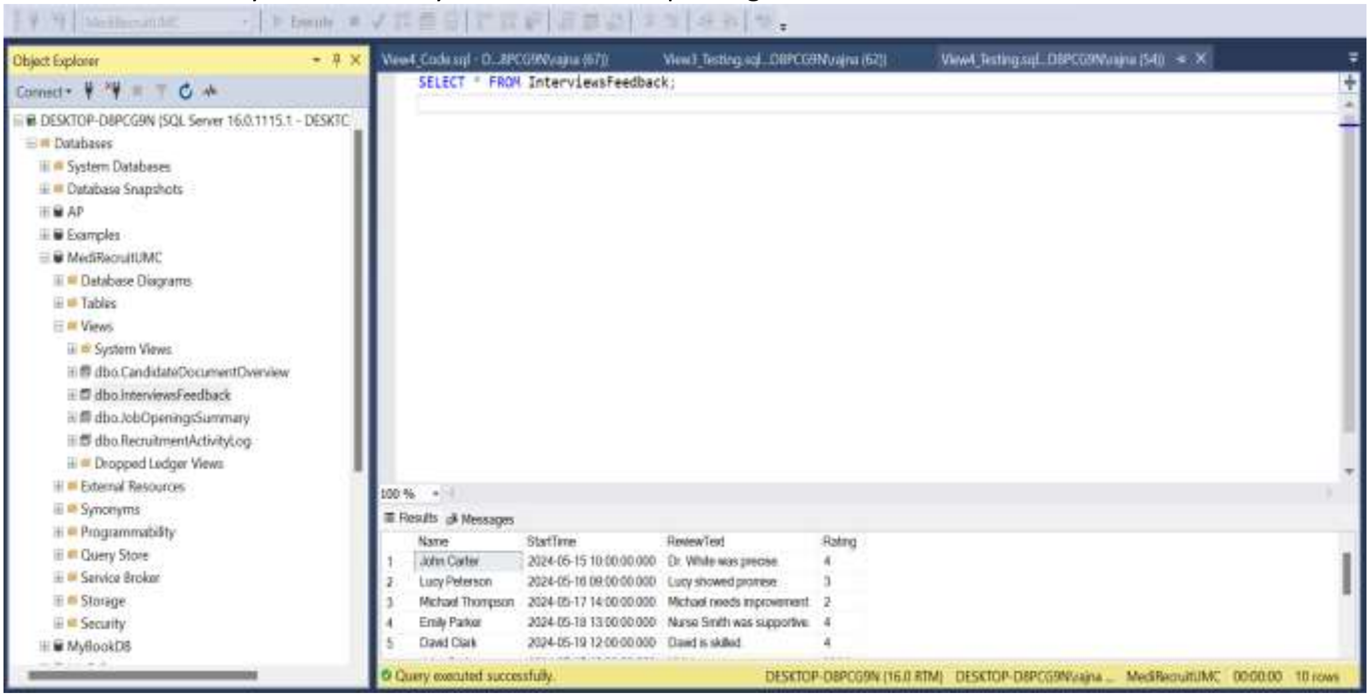
```
-- View 4: Interviews Feedback
        CREATE VIEW InterviewsFeedback AS
        SELECT Candidate.Name, Interviews.StartTime, InterviewReviews.ReviewText,
        InterviewReviews.Rating
        FROM Candidate
        JOIN Application ON Candidate.CandidateID = Application.CandidateID
        JOIN Interviews ON Application.ApplicationID = Interviews.ApplicationID
        LEFT JOIN InterviewReviews ON Interviews.InterviewID =
        InterviewReviews.InterviewID;
```

Testing: Interviews Feedback View
The "InterviewsFeedback" view is subjected to testing to validate its functionality and ensure alignment with the intended objectives.

SQL Query:
```
SELECT * FROM InterviewsFeedback;
```

Purpose of the View:
The "InterviewsFeedback" view aims to provide a consolidated overview of interview feedback by integrating relevant data from the "Candidate," "Application," "Interviews," and "InterviewReviews" tables. It selects essential information such as candidate names, interview start times, review text, and ratings to facilitate comprehensive feedback analysis.

Expected Output:
Upon executing the testing query, the output should present a comprehensive summary of interview feedback. Each row should contain pertinent details, including candidate name, interview start time, review text, and rating. This consolidated overview facilitates analysis of candidate performance and interview effectiveness. During the testing phase, any discrepancies between the expected and actual results are identified and addressed to ensure the accuracy and reliability of the view in capturing interview feedback data.



```
SELECT * FROM InterviewsFeedback;
```

	Name	StartTime	ReviewText	Rating
1	John Carter	2024-05-15 10:00:00.000	Dr. White was precise.	4
2	Lucy Peterson	2024-05-16 09:00:00.000	Lucy showed promise.	3
3	Michael Thompson	2024-05-17 14:00:00.000	Michael needs improvement.	2
4	Emily Parker	2024-05-18 13:00:00.000	Nurse Smith was supportive.	4
5	David Clark	2024-05-19 12:00:00.000	David is skilled.	4

Query executed successfully. DESKTOP-D8PCG9N (16.0 RTM) DESKTOP-D8PCG9N\rajna ... MediRecruitUMC 00:00:00 10 rows

2. Testing Stored Procedures:

```
-- Stored Procedure 1: Rejecting an application and logging the history
CREATE PROCEDURE RejectApplication
  @applicationID INT  -- Define the parameter and specify its data type
AS
BEGIN
  -- Update the application's status
  UPDATE Application
  SET Status = 'Rejected'
  WHERE ApplicationID = @applicationID; -- Use the parameter value here

  -- Add to recruitment history
  INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
  VALUES (
    (SELECT CandidateID FROM Application WHERE ApplicationID = @applicationID),
-- Use the parameter value here
    'Application Rejected',
    'No Offer Made',
    GETDATE()
  );
END;
```

Testing Stored Procedure: Rejecting an Application
In this scenario, we have a stored procedure named "RejectApplication" designed to reject a job application and log the action in the "RecruitmentHistory" table.

Before Testing:
Before executing the stored procedure, we examine the "Application" and "RecruitmentHistory" tables to observe their initial state.

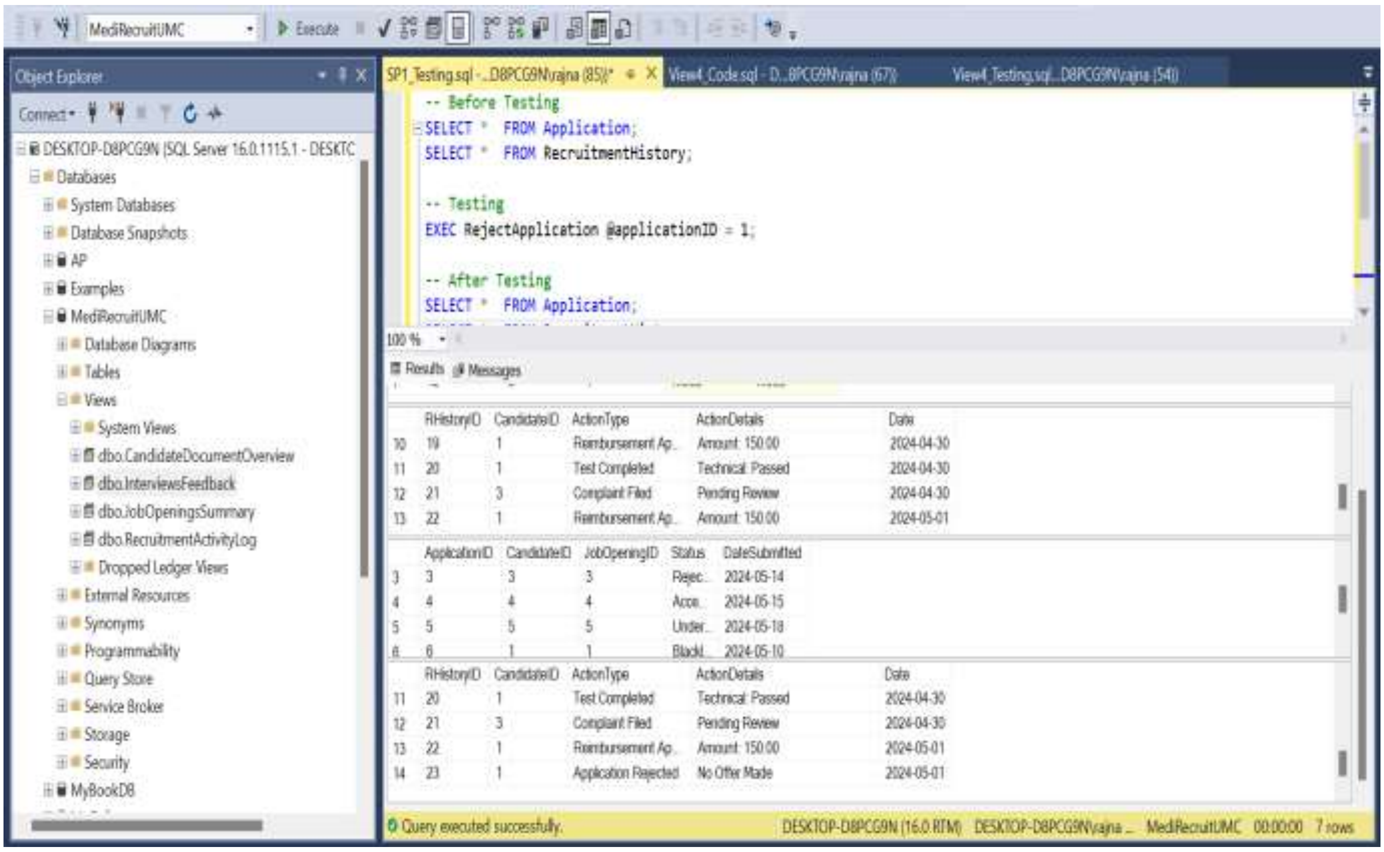
Testing:
We execute the "RejectApplication" stored procedure, passing the application ID as a parameter. This action updates the status of the specified application to "Rejected" in the "Application" table and logs the rejection action in the "RecruitmentHistory" table.

After Testing:
Following the execution of the stored procedure, we inspect the "Application" and "RecruitmentHistory" tables again to verify that the application status has been updated to "Rejected" and that a corresponding entry reflecting the rejection action has been added to the recruitment history.

This testing scenario ensures that the "RejectApplication" stored procedure functions as intended, accurately rejecting job applications, and maintaining a comprehensive record of recruitment actions.

SQL Query:

```
-- Before Testing
SELECT * FROM Application;
SELECT * FROM RecruitmentHistory;

-- Testing
EXEC RejectApplication @applicationID = 1;

-- After Testing
SELECT * FROM Application;
SELECT * FROM RecruitmentHistory;
```



```
-- Before Testing
SELECT * FROM Application;
SELECT * FROM RecruitmentHistory;

-- Testing
EXEC RejectApplication @applicationID = 1;

-- After Testing
SELECT * FROM Application;
```

	RHistoryID	CandidateID	ActionType	ActionDetails	Date
10	19	1	Reimbursement Ap...	Amount: 150.00	2024-04-30
11	20	1	Test Completed	Technical: Passed	2024-04-30
12	21	3	Complaint Filed	Pending Review	2024-04-30
13	22	1	Reimbursement Ap...	Amount: 150.00	2024-05-01

	ApplicationID	CandidateID	JobOpeningID	Status	DateSubmitted
3	3	3	3	Rejec...	2024-05-14
4	4	4	4	Acce...	2024-05-15
5	5	5	5	Under...	2024-05-18
6	6	1	1	Blackl...	2024-05-10

	RHistoryID	CandidateID	ActionType	ActionDetails	Date
11	20	1	Test Completed	Technical: Passed	2024-04-30
12	21	3	Complaint Filed	Pending Review	2024-04-30
13	22	1	Reimbursement Ap...	Amount: 150.00	2024-05-01
14	23	1	Application Rejected	No Offer Made	2024-05-01

Query executed successfully. DESKTOP-D8PCG9N (16.0 RTM) DESKTOP-D8PCG9N\rajna ... MediRecruitUMC 00:00:00 7 rows

```
-- Stored Procedure 2: Handling complaint resolutionCREATE PROCEDURE ResolveComplaint
    CREATE PROCEDURE ResolveComplaint
      @complaintID INT,
      @resolution VARCHAR(MAX) -- specify maximum length for VARCHAR(MAX)
    AS
    BEGIN
      -- Update the complaint's status
      UPDATE Complaints
      SET Resolution = @resolution, Status = 'Resolved'
      WHERE ComplaintID = @complaintID;

      -- Log the resolution in recruitment history
      INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
      VALUES (
        (SELECT CandidateID FROM Complaints WHERE ComplaintID = @complaintID),
        'Complaint Resolved', @resolution,
        GETDATE()
      );
    END;
```

Testing Stored Procedure: Handling Complaint Resolution
In this scenario, we assess the functionality of the "ResolveComplaint" stored procedure, designed to handle complaint resolution by updating the status and logging the resolution in the "RecruitmentHistory" table.

Before Testing:
Prior to executing the stored procedure, we examine the initial state of the "Complaints" and "RecruitmentHistory" tables to ensure a clear understanding of their contents.

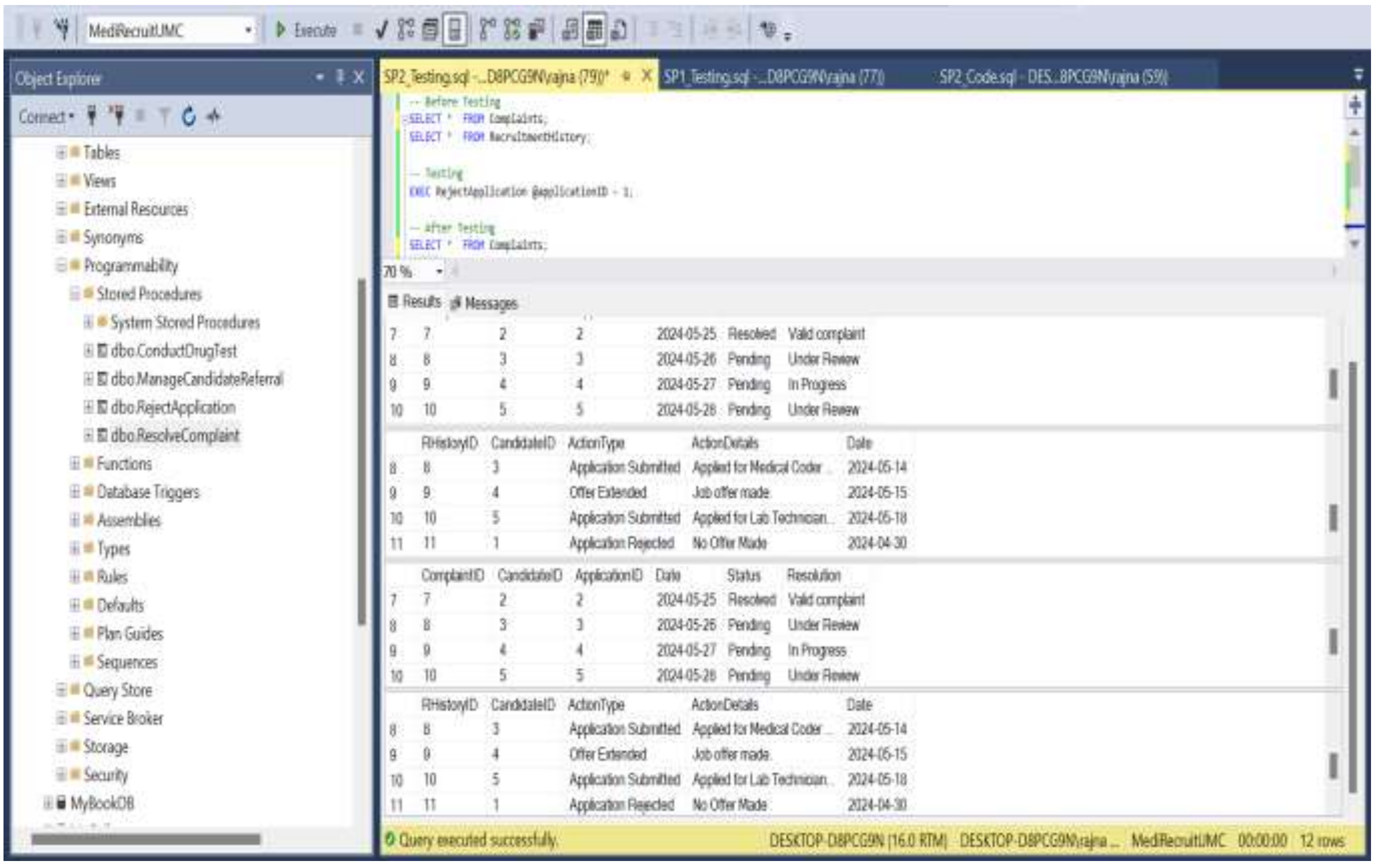
Testing:
We execute the "ResolveComplaint" stored procedure, providing the complaint ID and resolution details as parameters. This action updates the status of the specified complaint to "Resolved" in the "Complaints" table and records the resolution action in the "RecruitmentHistory" table.

After Testing:
Following the execution of the stored procedure, we review the "Complaints" and "RecruitmentHistory" tables again to confirm that the complaint status has been updated to "Resolved" and that a corresponding entry reflecting the resolution action has been appended to the recruitment history.

This testing scenario ensures that the "ResolveComplaint" stored procedure effectively manages complaint resolution, accurately updating complaint statuses and maintaining a comprehensive record of resolution actions in the recruitment history.

SQL Query:

```
-- Before Testing
SELECT * FROM Complaints;
SELECT * FROM RecruitmentHistory;

-- Testing
EXEC RejectApplication @applicationID = 1;

-- After Testing
SELECT * FROM Complaints;
SELECT * FROM RecruitmentHistory;
```



```
-- Before Testing
SELECT * FROM Complaints;
SELECT * FROM RecruitmentHistory;

-- Testing
EXEC RejectApplication @applicationID = 1;

-- After Testing
SELECT * FROM Complaints;
```

7	7	2	2	2024-05-25	Resolved	Valid complaint
8	8	3	3	2024-05-26	Pending	Under Review
9	9	4	4	2024-05-27	Pending	In Progress
10	10	5	5	2024-05-28	Pending	Under Review

	RHistoryID	CandidateID	ActionType	ActionDetails	Date
8	8	3	Application Submitted	Applied for Medical Coder ...	2024-05-14
9	9	4	Offer Extended	Job offer made	2024-05-15
10	10	5	Application Submitted	Applied for Lab Technician...	2024-05-18
11	11	1	Application Rejected	No Offer Made	2024-04-30

	ComplaintID	CandidateID	ApplicationID	Date	Status	Resolution
7	7	2	2	2024-05-25	Resolved	Valid complaint
8	8	3	3	2024-05-26	Pending	Under Review
9	9	4	4	2024-05-27	Pending	In Progress
10	10	5	5	2024-05-28	Pending	Under Review

	RHistoryID	CandidateID	ActionType	ActionDetails	Date
8	8	3	Application Submitted	Applied for Medical Coder ...	2024-05-14
9	9	4	Offer Extended	Job offer made	2024-05-15
10	10	5	Application Submitted	Applied for Lab Technician...	2024-05-18
11	11	1	Application Rejected	No Offer Made	2024-04-30

Query executed successfully. DESKTOP-D8PCG9N (16.0 RTM) DESKTOP-D8PCG9N\rajna ... MediRecruitUMC 00:00:00 12 rows

```
-- Stored Procedure 3: Drug testing a candidate
        CREATE PROCEDURE ConductDrugTest
            @candidateID INT,
            @testType VARCHAR(50),
            @results VARCHAR(255)
        AS
        BEGIN
            -- Insert drug test record
            INSERT INTO DrugTest (CandidateID, TestType, TestDate, Results)
            VALUES (@candidateID, @testType, GETDATE(), @results);

            -- Log drug test in recruitment history
            INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
            VALUES (
                @candidateID,
                'Drug Test Conducted',
                CONCAT('Result: ', @results),
                GETDATE()
            );
        END;
```

Testing Stored Procedure: Conducting Drug Test
In this scenario, we evaluate the functionality of the "ConductDrugTest" stored procedure, which facilitates the drug testing process for candidates by recording test results in the "DrugTest" table and logging the test action in the "RecruitmentHistory" table.

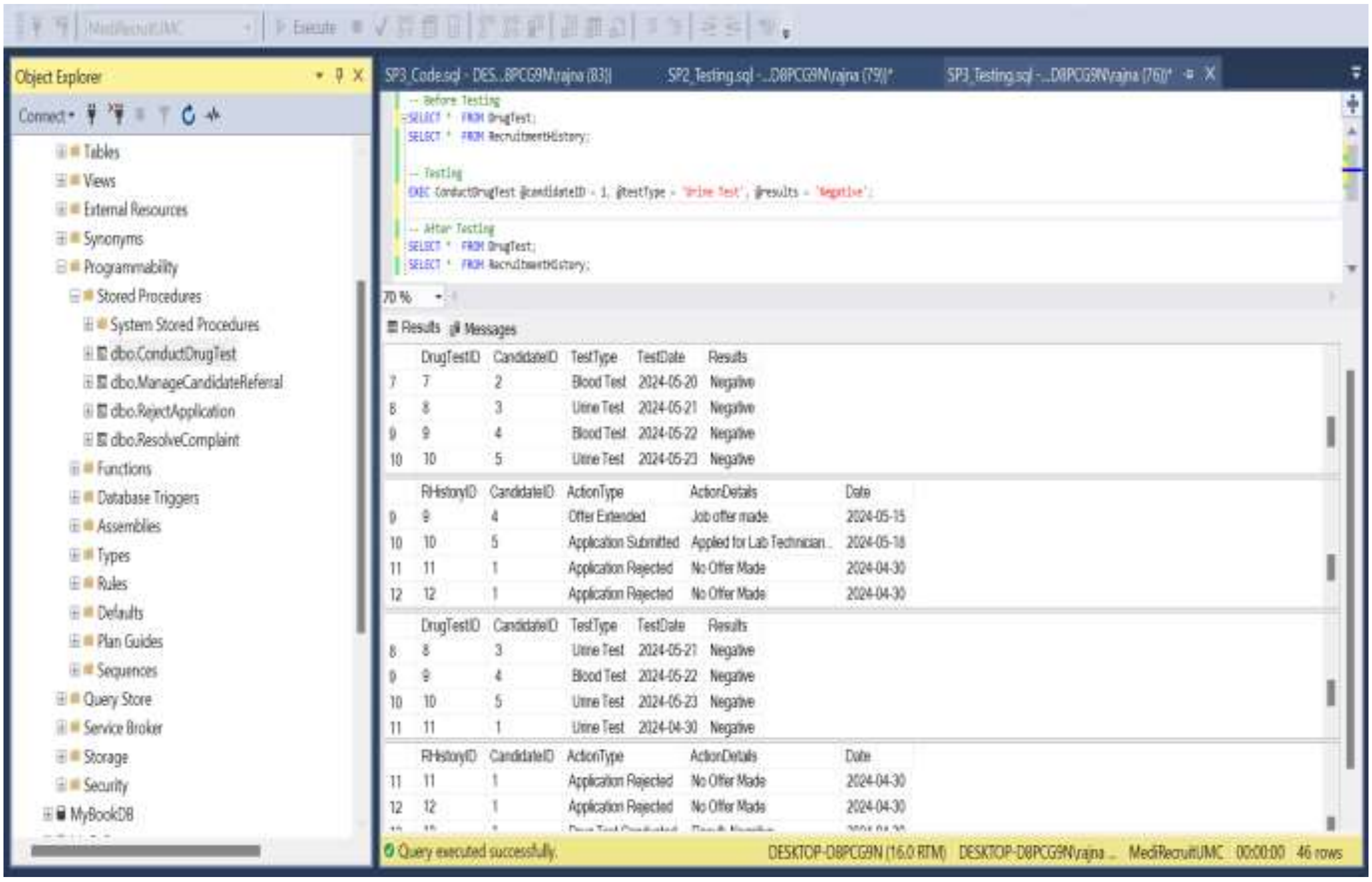
Before Testing:
We examine the initial contents of the "DrugTest" and "RecruitmentHistory" tables to establish a baseline before executing the stored procedure.

Testing:
The "ConductDrugTest" stored procedure is executed with parameters specifying the candidate ID, test type, and results. This action inserts a record into the "DrugTest" table containing details of the conducted drug test. Additionally, it logs the drug test action, along with the test results, in the "RecruitmentHistory" table.

After Testing:
Following the execution of the stored procedure, we review the contents of the "DrugTest" and "RecruitmentHistory" tables to confirm that the drug test record and corresponding test action have been accurately recorded. This validation ensures that the "ConductDrugTest" stored procedure effectively manages the drug testing process and maintains comprehensive records of conducted tests in the recruitment history.

SQL Query:

```
-- Before Testing
SELECT * FROM DrugTest;
SELECT * FROM RecruitmentHistory;

-- Testing
EXEC ConductDrugTest @candidateID = 1, @testType = 'Urine Test', @results = 'Negative';

-- After Testing
SELECT * FROM DrugTest;
SELECT * FROM RecruitmentHistory;
```



	DrugTestID	CandidateID	TestType	TestDate	Results
7	7	2	Blood Test	2024-05-20	Negative
8	8	3	Urine Test	2024-05-21	Negative
9	9	4	Blood Test	2024-05-22	Negative
10	10	5	Urine Test	2024-05-23	Negative

	RHistoryID	CandidateID	ActionType	ActionDetails	Date
9	9	4	Offer Extended	Job offer made	2024-05-15
10	10	5	Application Submitted	Applied for Lab Technician...	2024-05-18
11	11	1	Application Rejected	No Offer Made	2024-04-30
12	12	1	Application Rejected	No Offer Made	2024-04-30

	DrugTestID	CandidateID	TestType	TestDate	Results
8	8	3	Urine Test	2024-05-21	Negative
9	9	4	Blood Test	2024-05-22	Negative
10	10	5	Urine Test	2024-05-23	Negative
11	11	1	Urine Test	2024-04-30	Negative

	RHistoryID	CandidateID	ActionType	ActionDetails	Date
11	11	1	Application Rejected	No Offer Made	2024-04-30
12	12	1	Application Rejected	No Offer Made	2024-04-30

Query executed successfully. DESKTOP-D8PCG9N (16.0 RTM) DESKTOP-D8PCG9N\rajna ... MediRecruitUMC 00:00:00 46 rows

```
-- Stored Procedure 4: Candidate referral management
CREATE PROCEDURE ManageCandidateReferral
    @candidateID INT,
    @employeeID INT,
    @status VARCHAR(50)
AS
BEGIN
    -- Insert the candidate referral information
    INSERT INTO CandidateReferrals (CandidateID, EmployeeID, Status, ReferralDate)
    VALUES (@candidateID, @employeeID, @status, GETDATE());

    -- Log the referral management action
    INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
    VALUES (
        @candidateID,
        'Referral Managed',
        CONCAT('Status: ', @status),
        GETDATE()
    );
END;
```

Testing Stored Procedure: Managing Candidate Referral
This scenario involves testing the "ManageCandidateReferral" stored procedure, which handles the management of candidate referrals by inserting referral information into the "CandidateReferrals" table and logging the action in the "RecruitmentHistory" table.

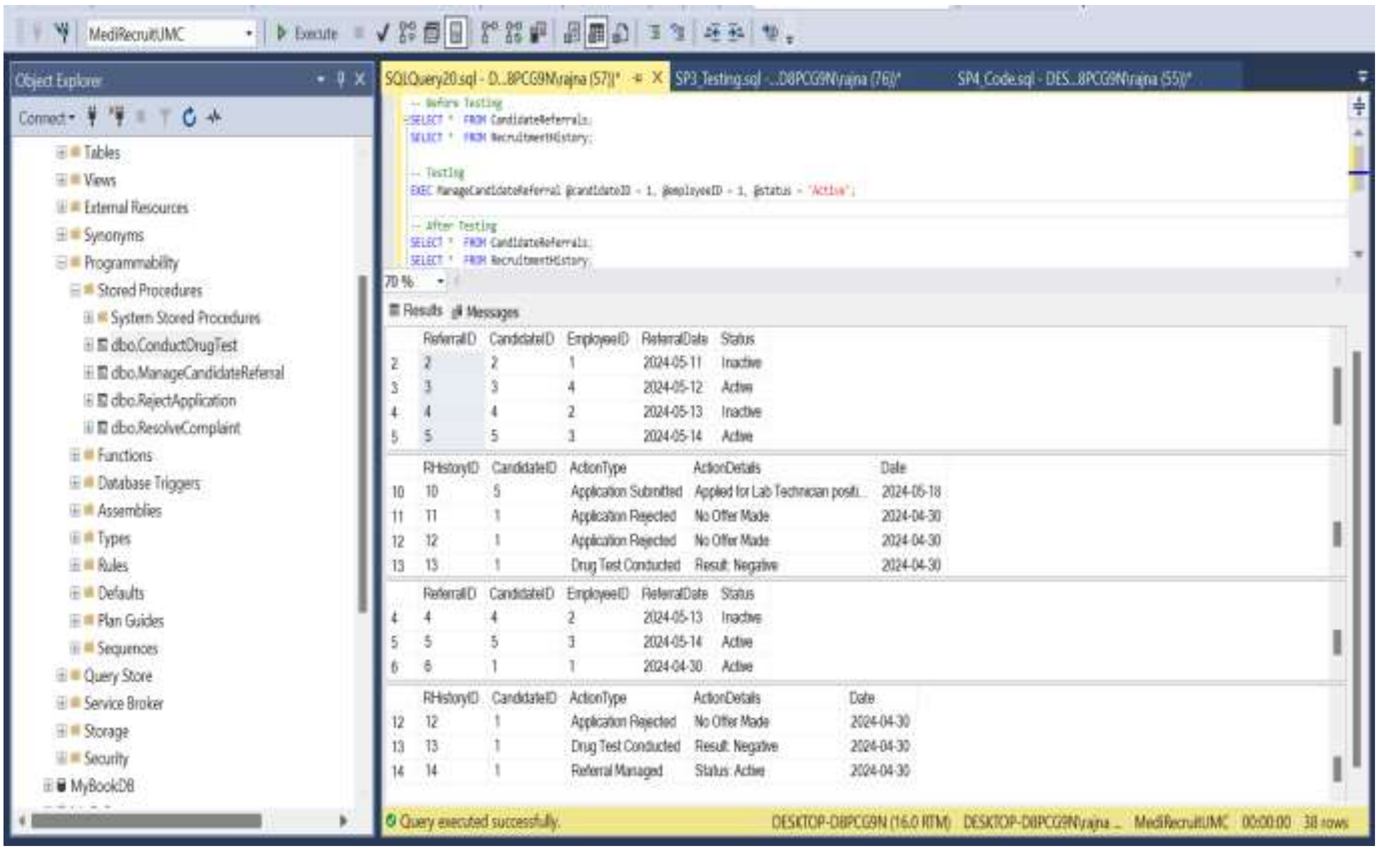
Before Testing:
We examine the initial contents of the "CandidateReferrals" and "RecruitmentHistory" tables to establish a baseline before executing the stored procedure.

Testing:
The "ManageCandidateReferral" stored procedure is executed with parameters specifying the candidate ID, employee ID, and referral status. This action inserts a record into the "CandidateReferrals" table containing details of the candidate referral. Additionally, it logs the referral management action, including the status, in the "RecruitmentHistory" table.

After Testing:
Following the execution of the stored procedure, we review the contents of the "CandidateReferrals" and "RecruitmentHistory" tables to verify that the candidate referral information and corresponding management action have been accurately recorded. This validation ensures that the "ManageCandidateReferral" stored procedure effectively manages candidate referrals and maintains comprehensive records of referral management actions in the recruitment history.

SQL Query:

```
-- Before Testing
SELECT * FROM CandidateReferrals;
SELECT * FROM RecruitmentHistory;

-- Testing
EXEC ManageCandidateReferral @candidateID = 1, @employeeID = 1, @status = 'Active';

-- After Testing
SELECT * FROM CandidateReferrals;
SELECT * FROM RecruitmentHistory;
```



	ReferralID	CandidateID	EmployeeID	ReferralDate	Status
2	2	2	1	2024-05-11	Inactive
3	3	3	4	2024-05-12	Active
4	4	4	2	2024-05-13	Inactive
5	5	5	3	2024-05-14	Active

	RHistoryID	CandidateID	ActionType	ActionDetails	Date
10	10	5	Application Submitted	Applied for Lab Technician positi...	2024-05-18
11	11	1	Application Rejected	No Offer Made	2024-04-30
12	12	1	Application Rejected	No Offer Made	2024-04-30
13	13	1	Drug Test Conducted	Result: Negative	2024-04-30

	ReferralID	CandidateID	EmployeeID	ReferralDate	Status
4	4	4	2	2024-05-13	Inactive
5	5	5	3	2024-05-14	Active
6	6	1	1	2024-04-30	Active

	RHistoryID	CandidateID	ActionType	ActionDetails	Date
12	12	1	Application Rejected	No Offer Made	2024-04-30
13	13	1	Drug Test Conducted	Result: Negative	2024-04-30
14	14	1	Referral Managed	Status: Active	2024-04-30

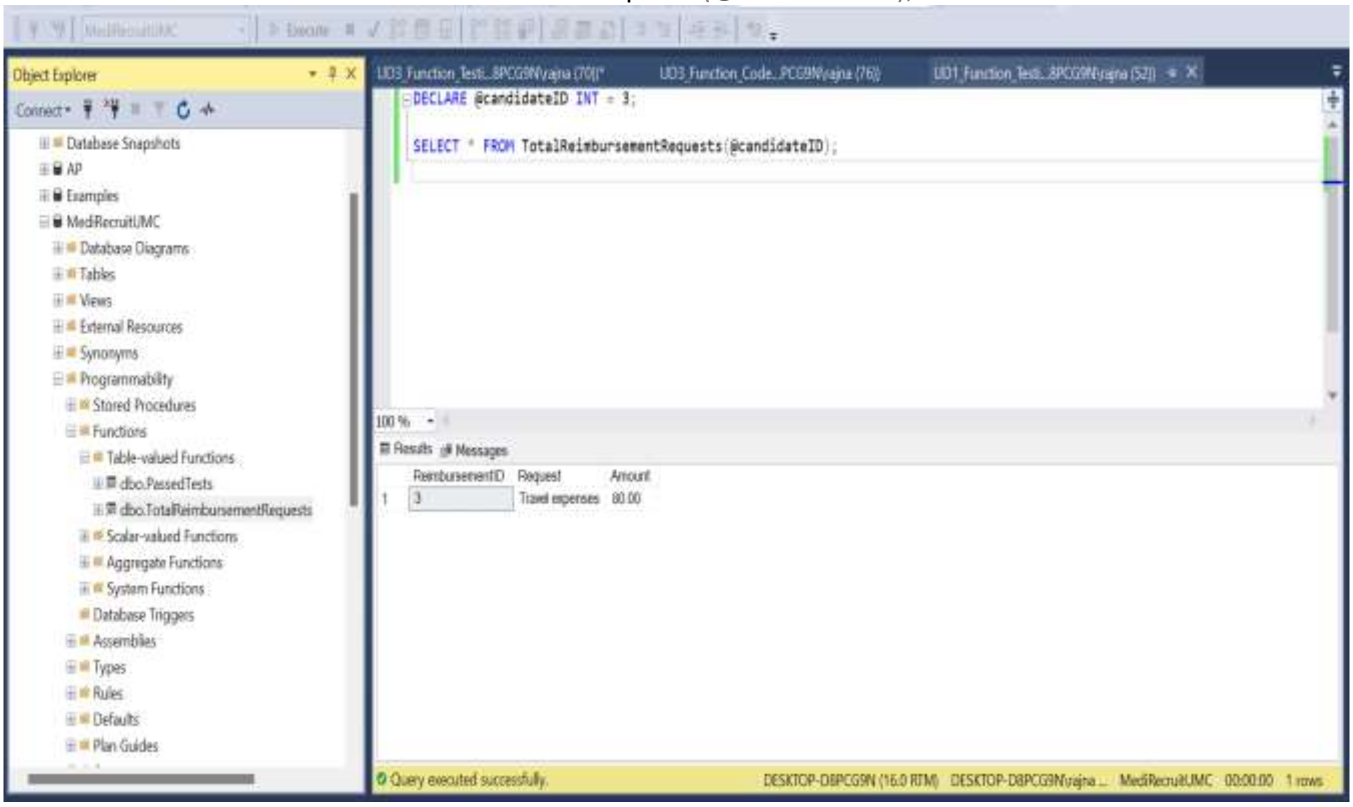
Query executed successfully. DESKTOP-D8PCG9N (16.0 RTM) DESKTOP-D8PCG9N\rajna ... MediRecruitUMC 00:00:00 38 rows

3. Testing User-defined Functions:

```
-- Function 1: Total reimbursement amount for a candidate
CREATE FUNCTION TotalReimbursementAmount(@candidateID INT) RETURNS
DECIMAL(10, 2)
BEGIN
    DECLARE @totalAmount DECIMAL(10, 2);
    SELECT @totalAmount = SUM(R.Amount)
    FROM Reimbursement R
    JOIN Application A ON R.ApplicationID = A.ApplicationID
    WHERE A.CandidateID = @candidateID;
    RETURN ISNULL(@totalAmount, 0);
        END;
```

User-Defined Function: Total Reimbursement Amount for a Candidate
This user-defined function calculates the total reimbursement amount for a specified candidate by summing up the reimbursement amounts associated with their applications.
Function Logic:
The function takes a candidate ID as input, calculates the total reimbursement amount by summing the reimbursement amounts associated with the candidate's applications, and returns the result. If no reimbursement is found, it returns 0.
Testing Query:
A candidate ID variable is declared and set to 3. The "TotalReimbursementRequests" function is then executed with this candidate ID, returning the total reimbursement amount. This query enables testing and validation of the function's accuracy.
SQL Query:

```
DECLARE @candidateID INT = 3;
SELECT * FROM TotalReimbursementRequests(@candidateID);
```



```
DECLARE @candidateID INT = 3;

SELECT * FROM TotalReimbursementRequests(@candidateID);
```

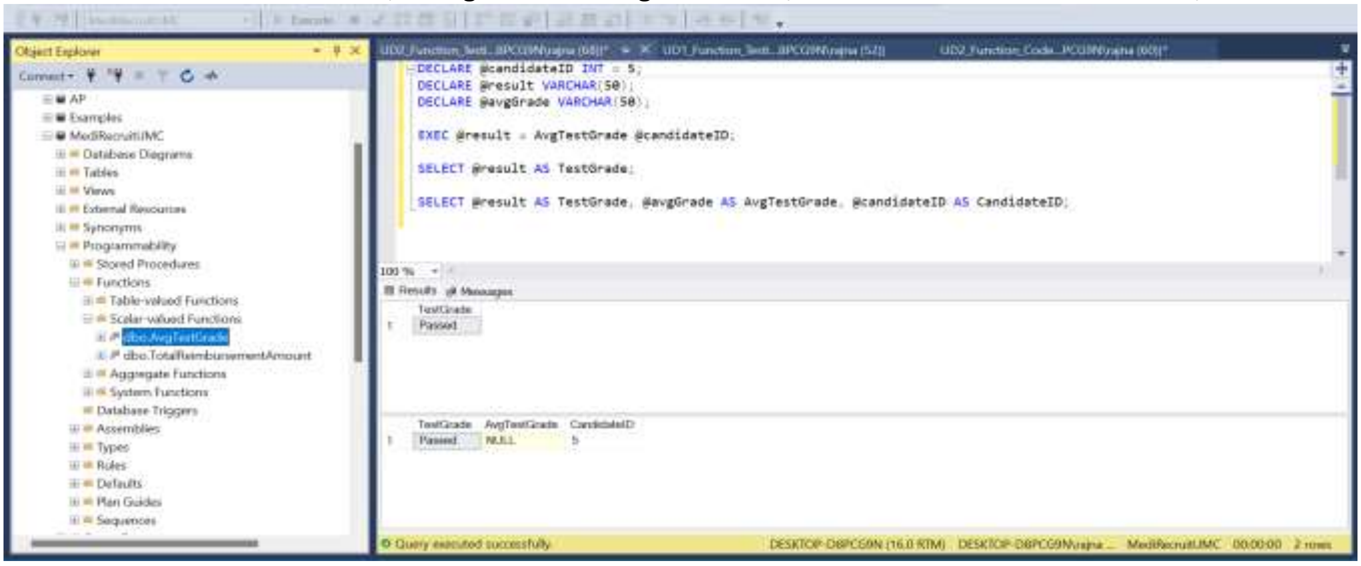
```
    ReimbursementID  Request          Amount
1   3                Travel expenses  80.00
```

```
Query executed successfully.    DESKTOP-D8PCG9N (16.0 RTM)  DESKTOP-D8PCG9N\rajna ...  MediRecruitUMC  00:00:00  1 rows
```

```
-- Function 2: Average test grade for a candidate
        CREATE FUNCTION AvgTestGrade(@candidateID INT) RETURNS VARCHAR(50)
        BEGIN
          DECLARE @avgGrade DECIMAL(10, 2);
          SELECT @avgGrade = AVG(CASE WHEN T.Grade = 'Passed' THEN 1 ELSE 0 END)
          FROM Application AS A
          JOIN Interviews AS I ON A.ApplicationID = I.ApplicationID
          JOIN Tests AS T ON I.InterviewID = T.InterviewID
          WHERE A.CandidateID = @candidateID;
          RETURN CASE
            WHEN @avgGrade >= 0.5 THEN 'Passed'
            ELSE 'Failed'
          END;
        END;
```

Function Logic:
This function calculates the average test grade for a given candidate. It first calculates the average grade for all tests associated with the candidate's interviews. If the average grade is equal to or above 0.5, it returns 'Passed'; otherwise, it returns 'Failed'.
Testing Function:
In the testing script, a candidate ID is declared and set to 5. The "AvgTestGrade" function is executed with this candidate ID. The result, indicating whether the candidate passed or failed based on their average test grade, is stored in the variable @result. This variable is then selected to display the test grade.
Output:
The output displays the test grade for the candidate, indicating whether they passed or failed. Additionally, it includes the candidate's ID for reference.
SQL Query:

```
DECLARE @candidateID INT = 5;
DECLARE @result VARCHAR(50);
DECLARE @avgGrade VARCHAR(50);
EXEC @result = AvgTestGrade @candidateID;
SELECT @result AS TestGrade;
SELECT @result AS TestGrade, @avgGrade AS AvgTestGrade, @candidateID AS CandidateID;
```



```
DECLARE @candidateID INT = 5;
DECLARE @result VARCHAR(50);
DECLARE @avgGrade VARCHAR(50);

EXEC @result = AvgTestGrade @candidateID;

SELECT @result AS TestGrade;

SELECT @result AS TestGrade, @avgGrade AS AvgTestGrade, @candidateID AS CandidateID;
```

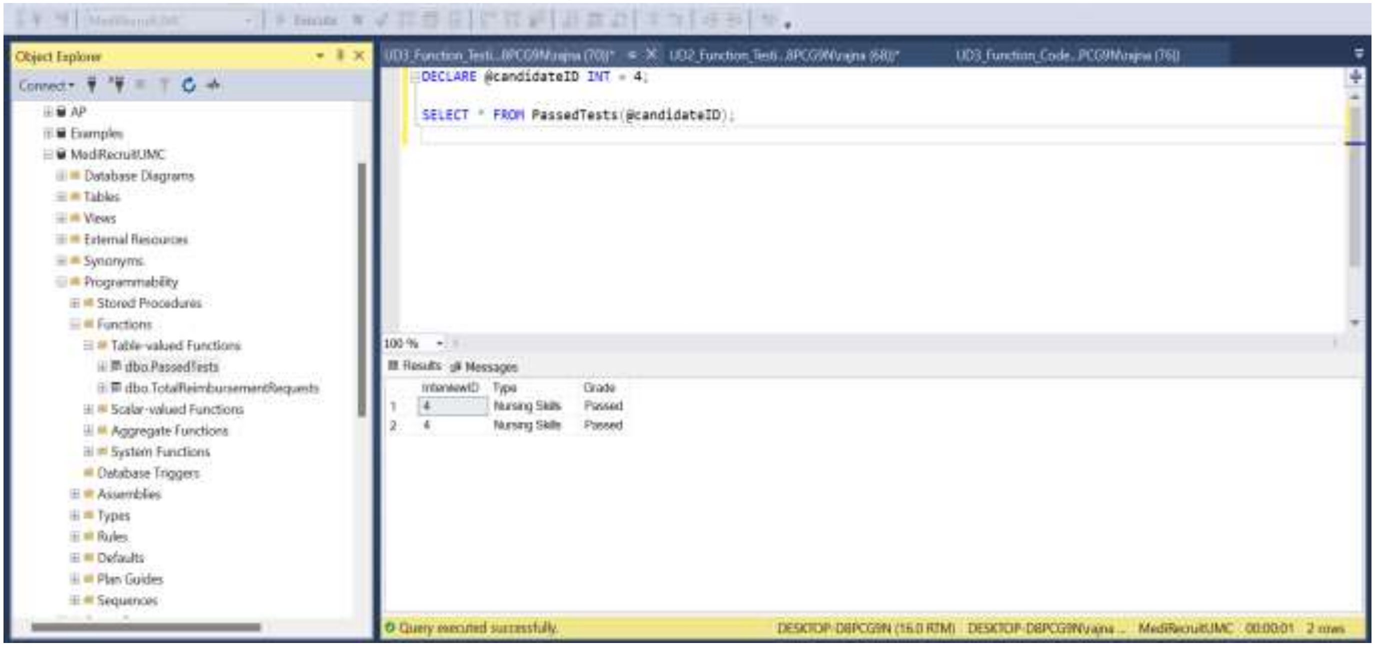
```
   TestGrade
1  Passed

   TestGrade  AvgTestGrade  CandidateID
1  Passed     NULL          5
```

Query executed successfully. DESKTOP-D8PCG9N (16.0 RTM) DESKTOP-D8PCG9N\rajna ... MediRecruitUMC 00:00:00 2 rows

```
-- Function 3: Passed tests for a candidate
CREATE FUNCTION PassedTests (@candidateID INT) RETURNS @PassedTests TABLE (
    InterviewID INT,
    Type VARCHAR(255),
    Grade VARCHAR(50)
)
AS
BEGIN
    INSERT INTO @PassedTests (InterviewID, Type, Grade)
    SELECT Tests.InterviewID, Tests.Type, Tests.Grade
    FROM Tests
    JOIN Interviews ON Tests.InterviewID = Interviews.InterviewID
    JOIN Application ON Interviews.ApplicationID = Application.ApplicationID
    WHERE Application.CandidateID = @candidateID AND Tests.Grade = 'Passed';
    RETURN;
END;
```

Function Logic:
The "PassedTests" function retrieves the details of all tests passed by a given candidate. It returns a table containing the interview ID, type of test, and grade for each passed test associated with the specified candidate.
Testing Query:
In the testing SQL code, a candidate ID variable is declared and set to 4. The "PassedTests" function is then invoked with this candidate ID. The function retrieves and returns the details of all tests passed by the candidate.
Output:
The output displays a table containing the interview ID, type of test, and grade for each test passed by the candidate. This information helps evaluate the candidate's performance in different types of tests.
SQL Code:

```
DECLARE @candidateID INT = 4;
SELECT * FROM PassedTests(@candidateID);
```



```
DECLARE @candidateID INT = 4;

SELECT * FROM PassedTests(@candidateID);
```

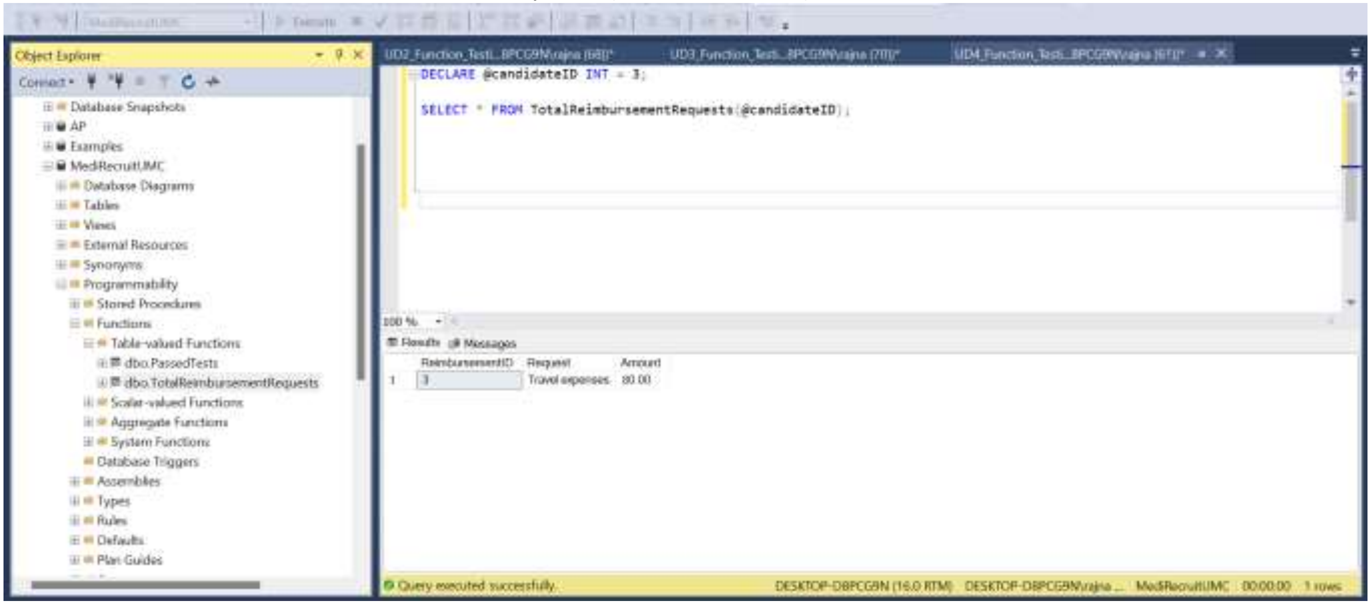
```
   InterviewID  Type           Grade
1  4            Nursing Skills Passed
2  4            Nursing Skills Passed
```

Query executed successfully.

```
-- Function 4: Reimbursement requests for a candidate
CREATE FUNCTION TotalReimbursementRequests (@candidateID INT) RETURNS
@ReimbursementRequests TABLE (
   ReimbursementID INT,
   Request VARCHAR(255),
   Amount DECIMAL(10, 2)
)
AS
BEGIN
   INSERT INTO @ReimbursementRequests (ReimbursementID, Request, Amount)
   SELECT Reimbursement.ReimbursementID, Reimbursement.Request,
Reimbursement.Amount
   FROM Reimbursement
   JOIN Application ON Reimbursement.ApplicationID = Application.ApplicationID
   WHERE Application.CandidateID = @candidateID;
   RETURN;
END;
```

Function Logic:
The "TotalReimbursementRequests" function retrieves reimbursement requests made by a specified candidate. It returns a table containing the reimbursement ID, request details, and amount for each request associated with the candidate.
Testing Query:
In the testing script, a candidate ID variable is declared and set to 3. The "TotalReimbursementRequests" function is then invoked with this candidate ID. The function retrieves and returns the details of all reimbursement requests made by the candidate.
Output:
The output displays a table containing the reimbursement ID, request details, and amount for each reimbursement request made by the candidate. This information assists in monitoring the candidate's reimbursement requests and managing financial transactions.
SQL Query:

```
DECLARE @candidateID INT = 3;
SELECT * FROM TotalReimbursementRequests(@candidateID);
```



```
DECLARE @candidateID INT = 3;

SELECT * FROM TotalReimbursementRequests(@candidateID);
```

```
   ReimbursementID  Request          Amount
1  3                Travel expenses  80.00
```

```
Query executed successfully.   DESKTOP-DBPCG9N (16.0 RTM)  DESKTOP-DBPCG9N\rajna ...  MediRecruitUMC  00:00:00  1 rows
```

4. Testing Triggers:

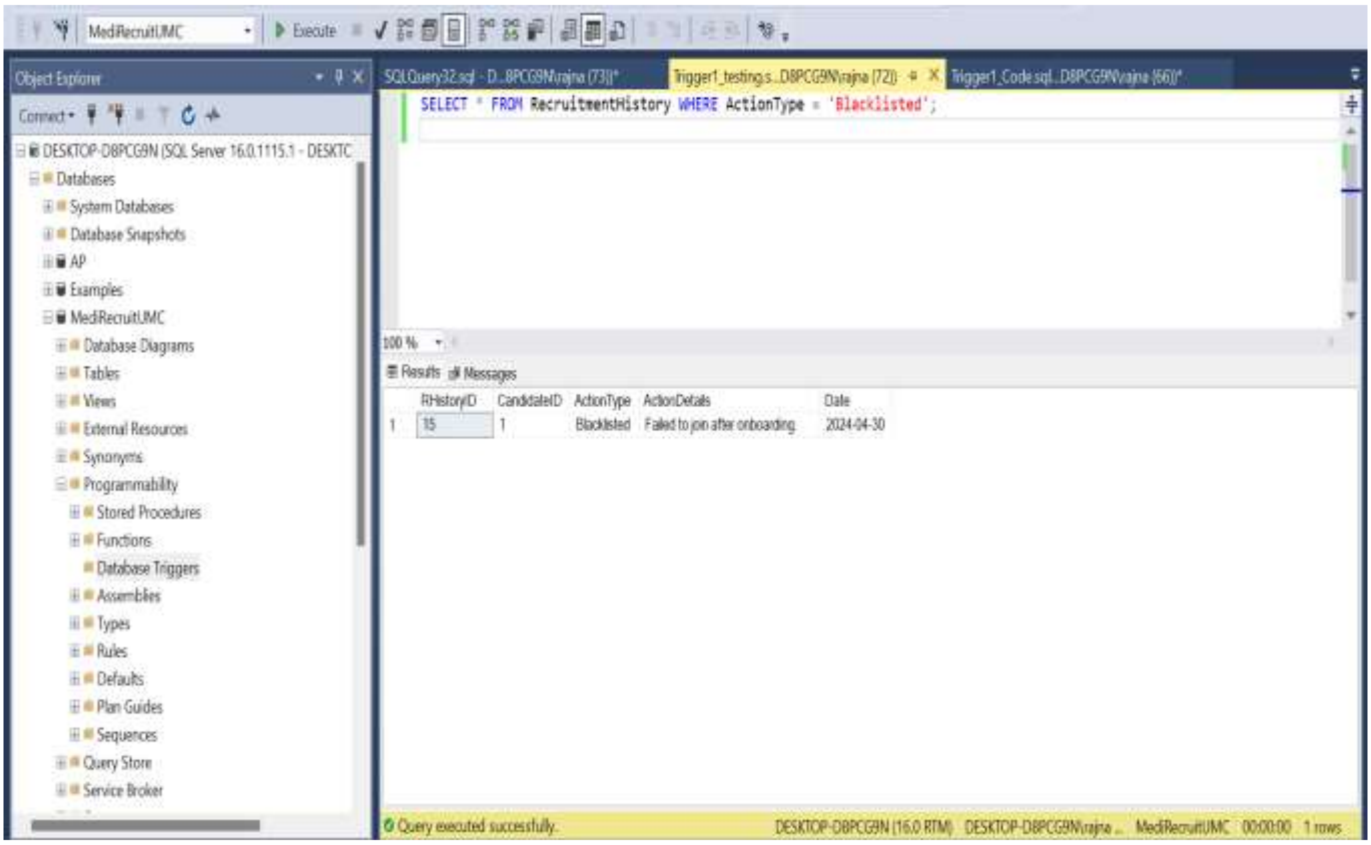
```
-- Trigger 1: Select a candidate who has passed the interview
CREATE TRIGGER BlacklistAfterNoJoin
ON Onboarding
AFTER UPDATE
AS
BEGIN
   DECLARE @candidateID INT, @completionDate DATETIME;
   SELECT @candidateID = CandidateID, @completionDate = CompletionDate FROM
inserted;

   IF @completionDate IS NOT NULL AND NOT EXISTS (
      SELECT 1 FROM RecruitmentHistory WHERE CandidateID = @candidateID AND
ActionType = 'Joined'
   )
   BEGIN
      INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
      VALUES (@candidateID, 'Blacklisted', 'Failed to join after onboarding.', GETDATE());

      UPDATE Application
      SET Status = 'Blacklisted'
      WHERE CandidateID = @candidateID;
   END
         END;
```

Trigger Logic:
The "BlacklistAfterNoJoin" trigger is executed after an update operation on the "Onboarding" table. It retrieves the candidate ID and completion date from the inserted data. If the completion date is not null and there is no record indicating that the candidate has joined in the recruitment history, the trigger inserts a new entry into the recruitment history table, blacklisting the candidate for failing to join after onboarding. Additionally, it updates the status of the candidate's application to "Blacklisted".

Testing Query:
The testing query selects all records from the recruitment history table where the action type is "Blacklisted". This query is used to verify whether the trigger successfully executed and added entries to the recruitment history for blacklisted candidates.
This testing scenario validates the functionality of the "BlacklistAfterNoJoin" trigger in automatically blacklisting candidates who fail to join after completing the onboarding process.

SQL Query:

```
SELECT * FROM RecruitmentHistory WHERE ActionType = 'Blacklisted';
```



```
SELECT * FROM RecruitmentHistory WHERE ActionType = 'Blacklisted';
```

	RHistoryID	CandidateID	ActionType	ActionDetails	Date
1	15	1	Blacklisted	Failed to join after onboarding	2024-04-30

Query executed successfully.   DESKTOP-D8PCG9N (16.0 RTM) DESKTOP-D8PCG9N\rajna ... MediRecruitUMC 00:00:00 1 rows

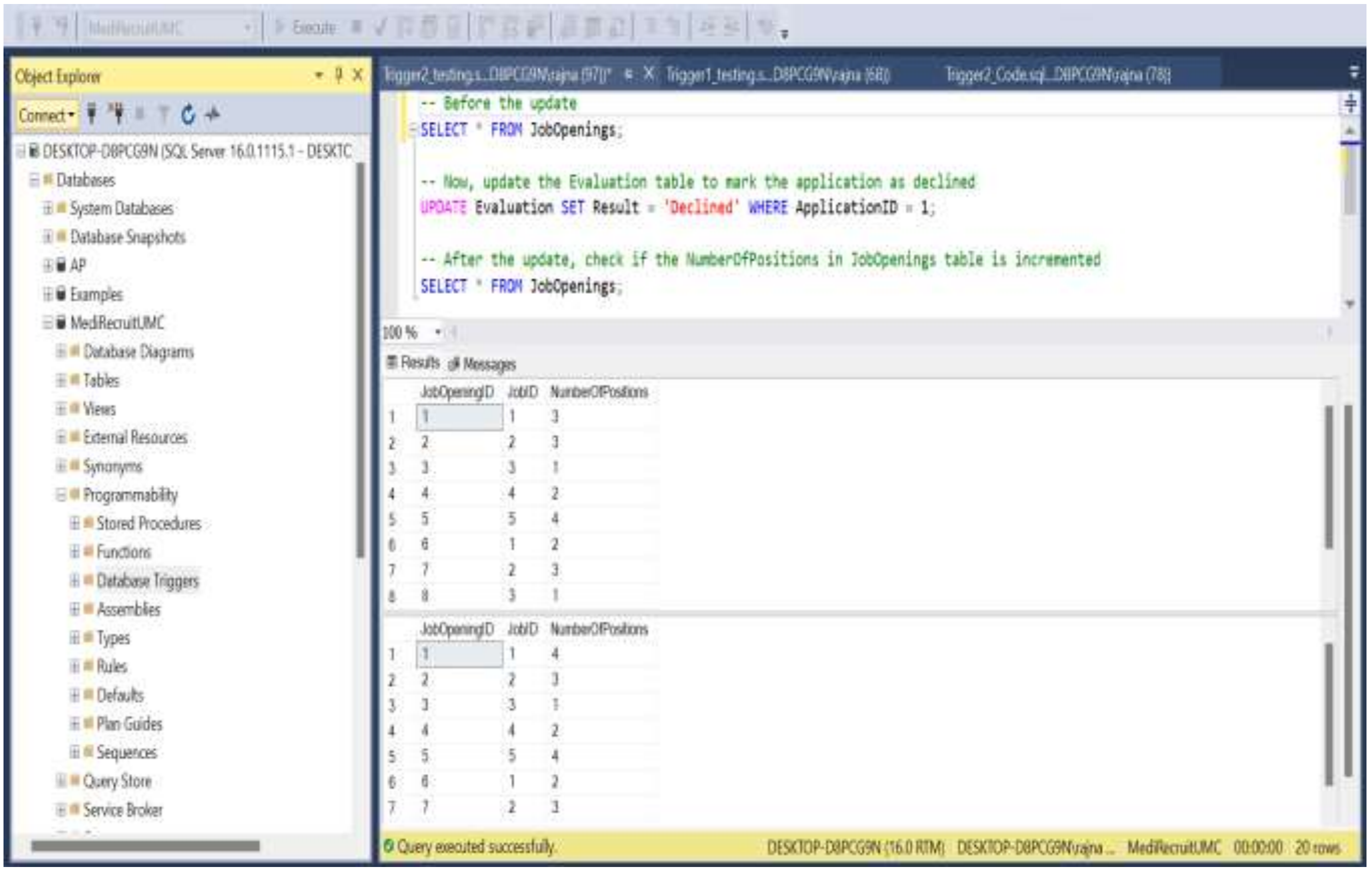
```
-- Trigger 2: Update job openings after a candidate declines an offer
CREATE TRIGGER UpdateJobOpeningsAfterDecline
ON Evaluation
AFTER UPDATE
AS
BEGIN
   DECLARE @appID INT, @evalResult NVARCHAR(50);

   -- Fetch the ApplicationID and EvaluationResult from the inserted rows
   SELECT @appID = i.ApplicationID, @evalResult = i.Result
   FROM inserted i;

   -- Check if the evaluation result is 'Declined'
   IF @evalResult = 'Declined'
   BEGIN
      -- Increment the NumberOfPositions for the corresponding JobOpeningID
      UPDATE JobOpenings
      SET NumberOfPositions = NumberOfPositions + 1
      WHERE JobOpeningID = (
         SELECT JobOpeningID
         FROM Application
         WHERE ApplicationID = @appID
      );
   END
END;
```

Trigger Logic:
The "UpdateJobOpeningsAfterDecline" trigger is executed after an update operation on the "Evaluation" table. It retrieves the application ID and evaluation result from the inserted data. If the evaluation result is "Declined", it increments the "NumberOfPositions" for the corresponding job opening ID in the "JobOpenings" table.

Testing Trigger:
The testing trigger query initially selects all records from the "JobOpenings" table to capture the current state. Then, it updates the "Evaluation" table to mark an application as declined. After the update, it verifies if the "NumberOfPositions" in the "JobOpenings" table is incremented, indicating that a job opening is now available due to the declined application. This testing scenario ensures that the "UpdateJobOpeningsAfterDecline" trigger correctly adjusts the number of positions available in the "JobOpenings" table when an application is declined.

SQL Query:

```
-- Before the update
SELECT * FROM JobOpenings;

-- Now, update the Evaluation table to mark the application as declined
UPDATE Evaluation SET Result = 'Declined' WHERE ApplicationID = 1;

-- After the update, check if the NumberOfPositions in JobOpenings table is incremented
SELECT * FROM JobOpenings;
```



```
-- Before the update
SELECT * FROM JobOpenings;

-- Now, update the Evaluation table to mark the application as declined
UPDATE Evaluation SET Result = 'Declined' WHERE ApplicationID = 1;

-- After the update, check if the NumberOfPositions in JobOpenings table is incremented
SELECT * FROM JobOpenings;
```

	JobOpeningID	JobID	NumberOfPositions
1	1	1	3
2	2	2	3
3	3	3	1
4	4	4	2
5	5	5	4
6	6	1	2
7	7	2	3
8	8	3	1

	JobOpeningID	JobID	NumberOfPositions
1	1	1	4
2	2	2	3
3	3	3	1
4	4	4	2
5	5	5	4
6	6	1	2
7	7	2	3

Query executed successfully. DESKTOP-D8PCG9N (16.0 RTM) DESKTOP-D8PCG9N\rajna ... MediRecruitUMC 00:00:00 20 rows

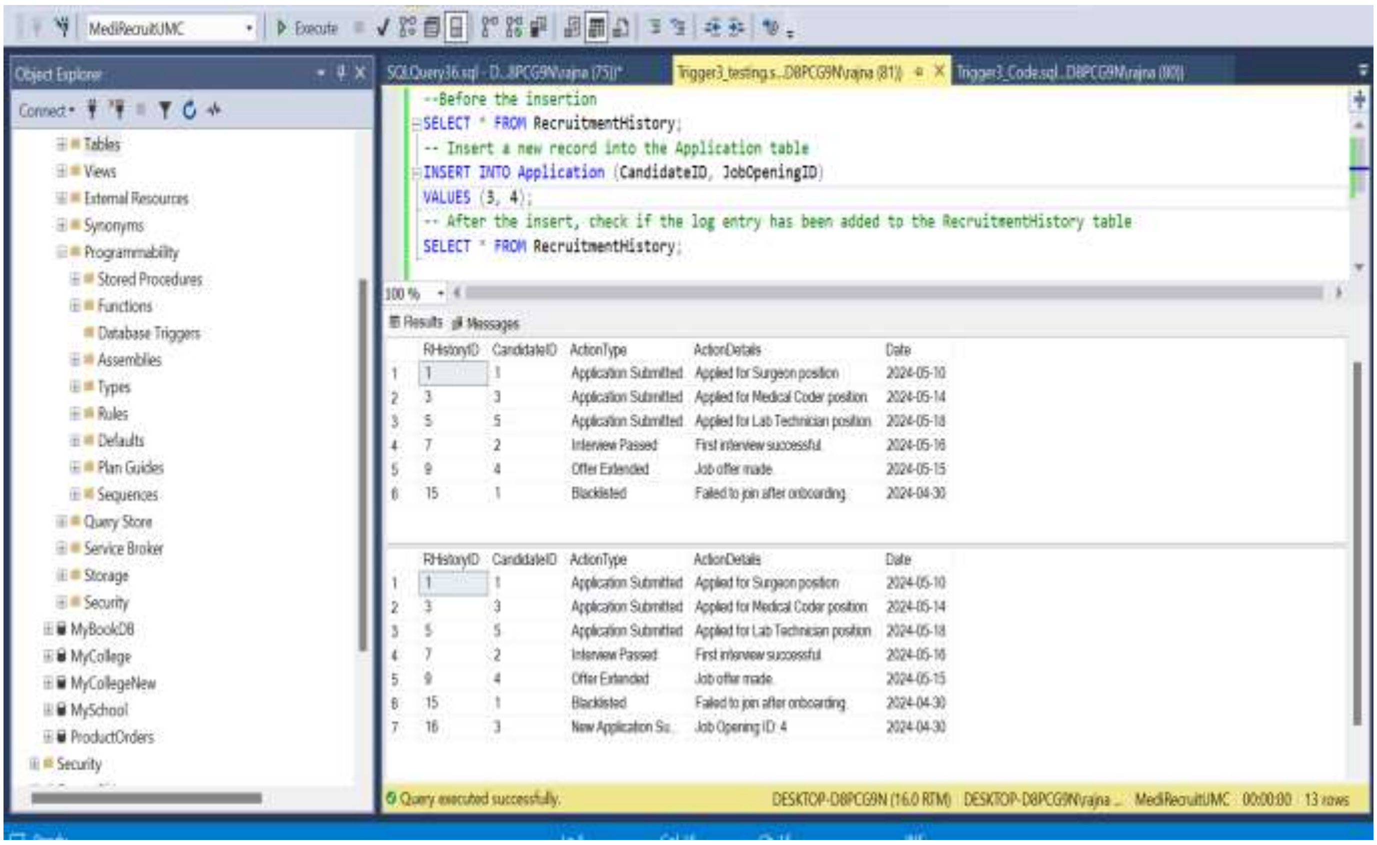
```
-- Trigger 3: Log new applications
CREATE TRIGGER LogNewApplication
ON Application
AFTER INSERT
AS
BEGIN
  DECLARE @candidateID INT, @jobOpeningID INT;
  SELECT @candidateID = CandidateID, @jobOpeningID = JobOpeningID FROM inserted;

  INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
  VALUES (@candidateID, 'New Application Submitted', CONCAT('Job Opening ID: ',
@jobOpeningID), GETDATE());
END;
```

Trigger Logic:
The "LogNewApplication" trigger fires after an insertion operation on the "Application" table. It extracts the candidate ID and job opening ID from the inserted data. Subsequently, it records a log entry in the "RecruitmentHistory" table, indicating that a new application has been submitted, along with details such as the candidate ID and job opening ID, along with the timestamp.

Testing Trigger:
The testing query initially retrieves all records from the "RecruitmentHistory" table to capture the current state. Then, it inserts a new record into the "Application" table to simulate a new application submission. After the insertion, it verifies if the log entry has been successfully added to the "RecruitmentHistory" table, reflecting the new application submission.
This testing scenario ensures that the "LogNewApplication" trigger correctly logs new application submissions in the "RecruitmentHistory" table.

SQL Query:

```
--Before the insertion
SELECT * FROM RecruitmentHistory;
-- Insert a new record into the Application table
INSERT INTO Application (CandidateID, JobOpeningID)
VALUES (3, 4);
-- After the insert, check if the log entry has been added to the RecruitmentHistory table
SELECT * FROM RecruitmentHistory;
```



```
--Before the insertion
SELECT * FROM RecruitmentHistory;
-- Insert a new record into the Application table
INSERT INTO Application (CandidateID, JobOpeningID)
VALUES (3, 4);
-- After the insert, check if the log entry has been added to the RecruitmentHistory table
SELECT * FROM RecruitmentHistory;
```

	RHistoryID	CandidateID	ActionType	ActionDetails	Date
1	1	1	Application Submitted	Applied for Surgeon position	2024-05-10
2	3	3	Application Submitted	Applied for Medical Coder position	2024-05-14
3	5	5	Application Submitted	Applied for Lab Technician position	2024-05-18
4	7	2	Interview Passed	First interview successful	2024-05-16
5	9	4	Offer Extended	Job offer made	2024-05-15
6	15	1	Blacklisted	Failed to join after onboarding	2024-04-30

	RHistoryID	CandidateID	ActionType	ActionDetails	Date
1	1	1	Application Submitted	Applied for Surgeon position	2024-05-10
2	3	3	Application Submitted	Applied for Medical Coder position	2024-05-14
3	5	5	Application Submitted	Applied for Lab Technician position	2024-05-18
4	7	2	Interview Passed	First interview successful	2024-05-16
5	9	4	Offer Extended	Job offer made	2024-05-15
6	15	1	Blacklisted	Failed to join after onboarding	2024-04-30
7	16	3	New Application Su...	Job Opening ID: 4	2024-04-30

Query executed successfully.   DESKTOP-D8PCG9N (16.0 RTM)   DESKTOP-D8PCG9N\rajna ...   MediRecruitUMC   00:00:00   13 rows

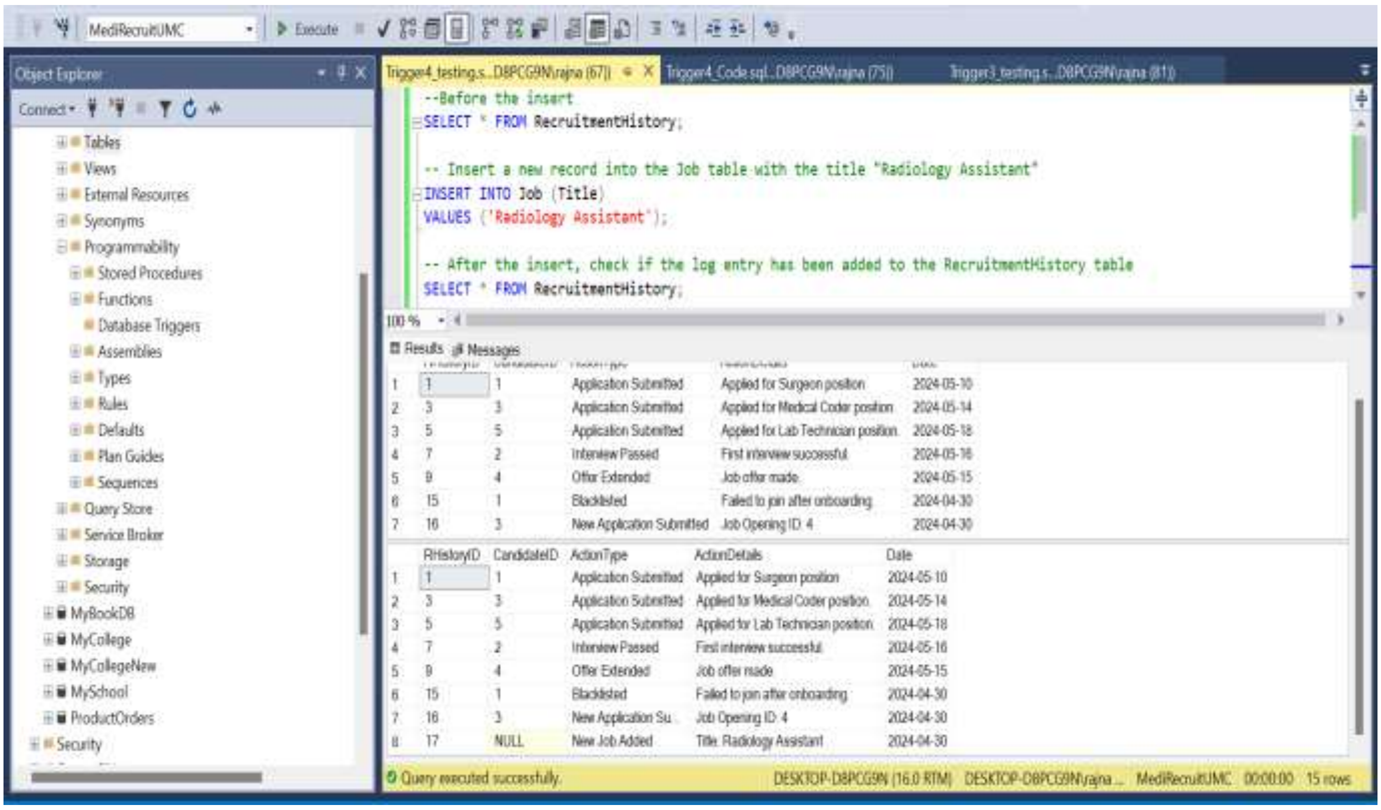
```
-- Trigger 4: Log job positions
CREATE TRIGGER LogJobPositions
ON Job
AFTER INSERT
AS
BEGIN
  DECLARE @jobID INT, @jobTitle NVARCHAR(255);
  SELECT @jobID = JobID, @jobTitle = Title FROM inserted;

  INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
  VALUES (
    NULL,
    'New Job Added',
    CONCAT('Title: ', @jobTitle),
    GETDATE()
  );
END;
```

Trigger Logic:
The "LogJobPositions" trigger activates after an insertion operation on the "Job" table. It retrieves the newly inserted job's ID and title from the "inserted" pseudo-table. Subsequently, it records a log entry in the "RecruitmentHistory" table, indicating the addition of a new job position. The log entry includes details such as the job title and a timestamp.

Testing Trigger:
The testing query initially retrieves all records from the "RecruitmentHistory" table to capture the current state. Then, it inserts a new record into the "Job" table, creating a new job position titled "Radiology Assistant." After the insertion, it verifies if the log entry has been successfully added to the "RecruitmentHistory" table, reflecting the addition of the new job position. This testing scenario ensures that the "LogJobPositions" trigger correctly logs the addition of new job positions in the "RecruitmentHistory" table.

SQL Query:

```sql
--Before the insert
SELECT * FROM RecruitmentHistory;

-- Insert a new record into the Job table with the title "Radiology Assistant"
INSERT INTO Job (Title)
VALUES ('Radiology Assistant');

-- After the insert, check if the log entry has been added to the RecruitmentHistory table
SELECT * FROM RecruitmentHistory;
```



1	1	1	Application Submitted	Applied for Surgeon position	2024-05-10
2	3	3	Application Submitted	Applied for Medical Coder position	2024-05-14
3	5	5	Application Submitted	Applied for Lab Technician position	2024-05-18
4	7	2	Interview Passed	First interview successful	2024-05-16
5	9	4	Offer Extended	Job offer made	2024-05-15
6	15	1	Blacklisted	Failed to join after onboarding	2024-04-30
7	16	3	New Application Submitted	Job Opening ID: 4	2024-04-30

	RHistoryID	CandidateID	ActionType	ActionDetails	Date
1	1	1	Application Submitted	Applied for Surgeon position	2024-05-10
2	3	3	Application Submitted	Applied for Medical Coder position	2024-05-14
3	5	5	Application Submitted	Applied for Lab Technician position	2024-05-18
4	7	2	Interview Passed	First interview successful	2024-05-16
5	9	4	Offer Extended	Job offer made	2024-05-15
6	15	1	Blacklisted	Failed to join after onboarding	2024-04-30
7	16	3	New Application Su...	Job Opening ID: 4	2024-04-30
8	17	NULL	New Job Added	Title: Radiology Assistant	2024-04-30

Query executed successfully. DESKTOP-D8PCG9N (16.0 RTM) DESKTOP-D8PCG9N\rajna... MediRecruitUMC 00:00:00 15 rows

5. Testing Transactions:

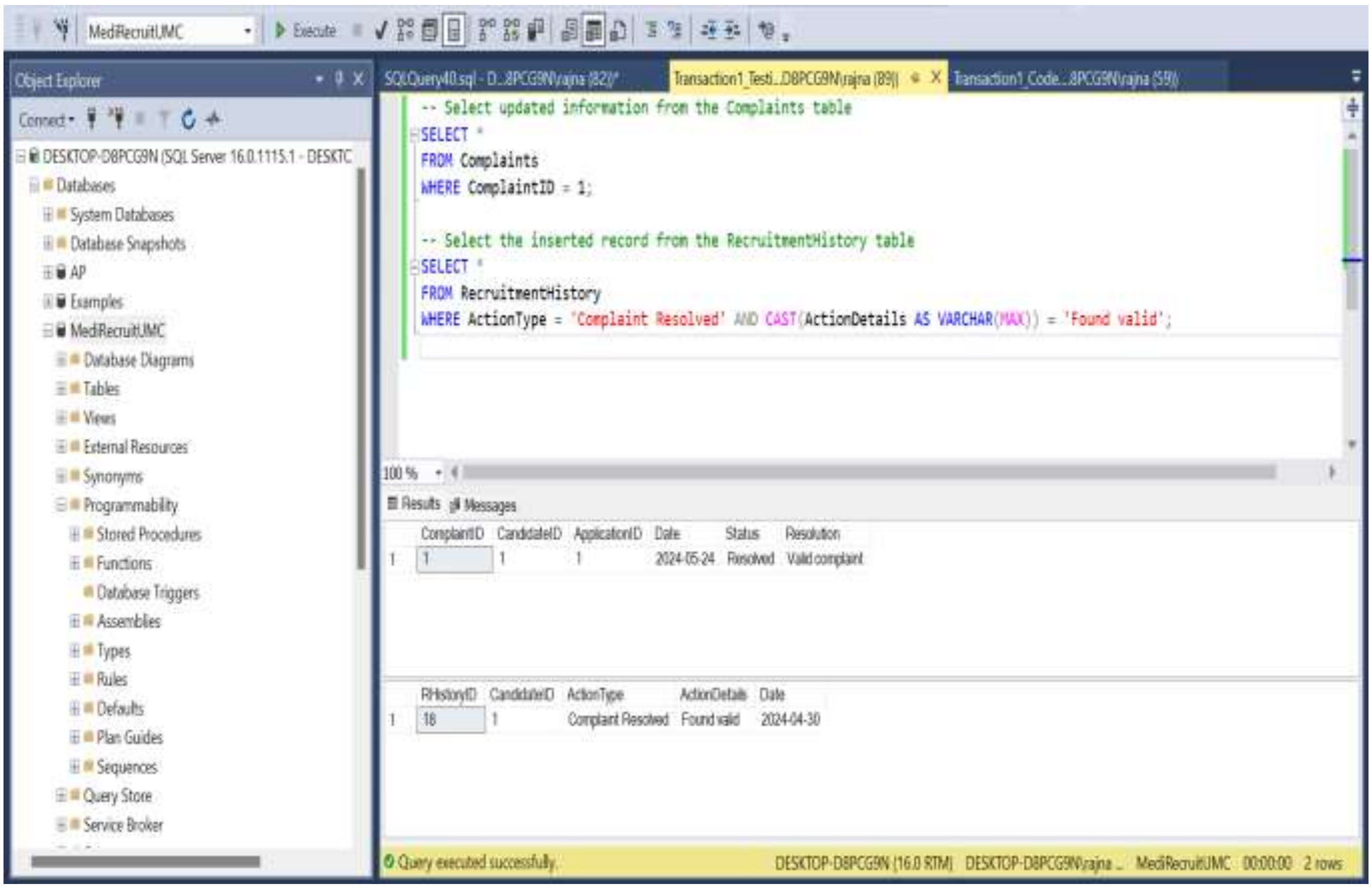
```
-- Transaction 1: Handling complaints
BEGIN TRAN;
-- Update complaint
UPDATE Complaints
SET Status = 'Resolved', Resolution = 'Valid complaint'
WHERE ComplaintID = 1;
-- Log recruitment history
INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
VALUES (
   (SELECT CandidateID FROM Complaints WHERE ComplaintID = 1),
   'Complaint Resolved',
   'Found valid',
   GETDATE()
);
-- Commit transaction
         COMMIT;
```

Transaction Logic:
The transaction begins by updating a complaint's status and resolution based on its unique ID in the "Complaints" table. It sets the status to 'Resolved' and assigns a resolution of 'Valid complaint'. Subsequently, it records this resolution action by inserting a log entry into the "RecruitmentHistory" table, detailing the resolution as 'Valid'.

Testing Transaction:
Verification begins by retrieving the updated information of the complaint with the specified ID from the "Complaints" table, ensuring the status and resolution have been appropriately updated. Additionally, the query selects the inserted log entry from the "RecruitmentHistory" table, filtering for the log entry representing the resolution of the complaint and confirming its validity.
This testing scenario confirms the transaction's effectiveness in managing complaints by accurately updating their status and resolution and logging the resolution details, thus validating its integrity and functionality within the system.

SQL Query:

```
-- Select updated information from the Complaints table
SELECT *
FROM Complaints
WHERE ComplaintID = 1;

-- Select the inserted record from the RecruitmentHistory table
SELECT *
FROM RecruitmentHistory
        WHERE ActionType = 'Complaint Resolved' AND CAST(ActionDetails AS
        VARCHAR(MAX)) = 'Found valid';
```



```
-- Select updated information from the Complaints table
SELECT *
FROM Complaints
WHERE ComplaintID = 1;

-- Select the inserted record from the RecruitmentHistory table
SELECT *
FROM RecruitmentHistory
WHERE ActionType = 'Complaint Resolved' AND CAST(ActionDetails AS VARCHAR(MAX)) = 'Found valid';
```

	ComplaintID	CandidateID	ApplicationID	Date	Status	Resolution
1	1	1	1	2024-05-24	Resolved	Valid complaint

	RHistoryID	CandidateID	ActionType	ActionDetails	Date
1	18	1	Complaint Resolved	Found valid	2024-04-30

Query executed successfully. DESKTOP-D8PCG9N (16.0 RTM) DESKTOP-D8PCG9N\rajna ... MediRecruitUMC 00:00:00 2 rows

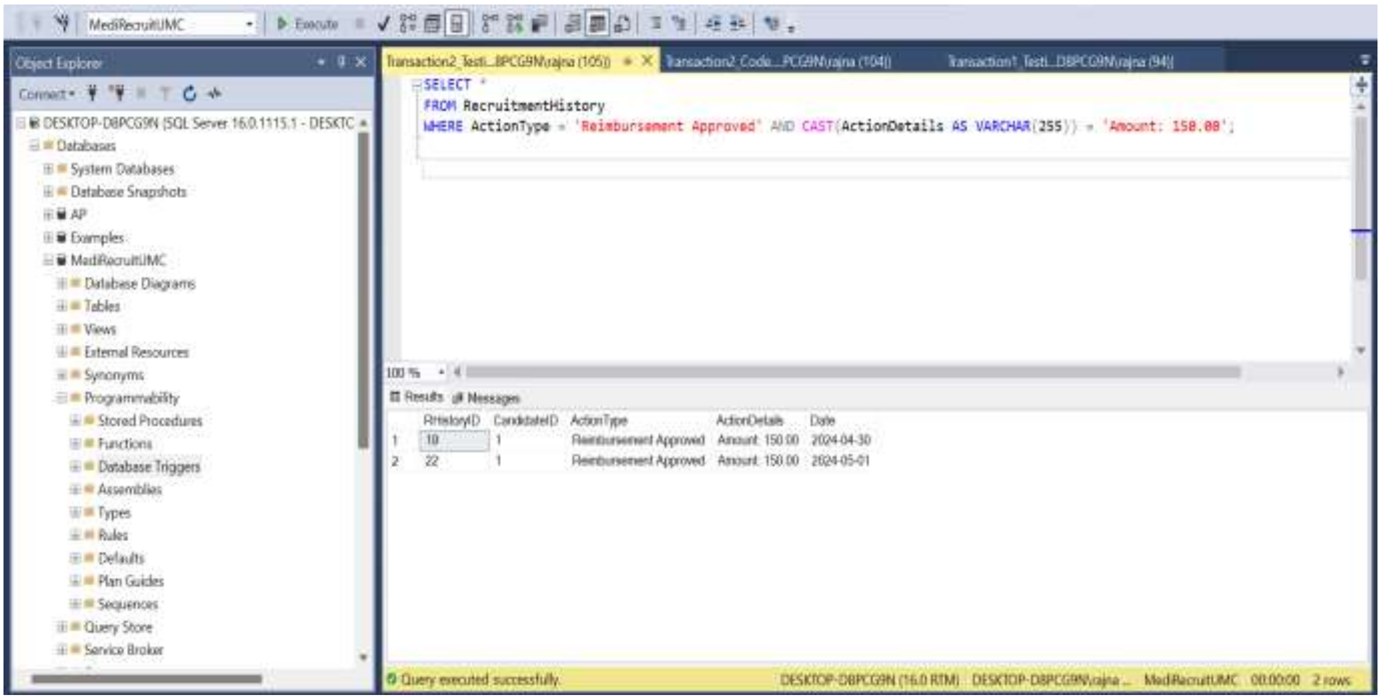
```sql
-- Transaction 2: Reimbursement flow
BEGIN TRAN;
-- Log reimbursement request
INSERT INTO Reimbursement (ApplicationID, Request, Processed, Amount)
VALUES (1, 'Travel expenses', 1, 150.00);
-- Update recruitment history
INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
VALUES (1, 'Reimbursement Approved', 'Amount: 150.00', GETDATE());
-- Commit transaction
COMMIT;
```

Transaction Logic:
The transaction begins by inserting a reimbursement request into the "Reimbursement" table, specifying the application ID, request details, processed status, and amount. Subsequently, it records this reimbursement approval action in the "RecruitmentHistory" table, detailing the approval of a reimbursement request with a specific amount. Finally, the transaction commits these changes, confirming the successful completion of the reimbursement flow process.

Testing Query:
The query verifies the transaction's effectiveness by selecting records from the "RecruitmentHistory" table, specifically filtering for entries indicating the approval of a reimbursement request with the exact amount of $150.00. This ensures that the reimbursement flow transaction accurately logs the reimbursement approval action with the specified amount in the system.

SQL Query:

```sql
SELECT *
FROM RecruitmentHistory
WHERE ActionType = 'Reimbursement Approved' AND CAST(ActionDetails AS
VARCHAR(255)) = 'Amount: 150.00';
```



```sql
SELECT *
FROM RecruitmentHistory
WHERE ActionType = 'Reimbursement Approved' AND CAST(ActionDetails AS VARCHAR(255)) = 'Amount: 150.00';
```

```
   RHistoryID  CandidateID  ActionType              ActionDetails   Date
1  10          1            Reimbursement Approved  Amount: 150.00  2024-04-30
2  22          1            Reimbursement Approved  Amount: 150.00  2024-05-01
```

```
Query executed successfully.   DESKTOP-DBPCG9N (16.0 RTM)   DESKTOP-DBPCG9N\rajna ...   MediRecruitUMC   00:00:00   2 rows
```

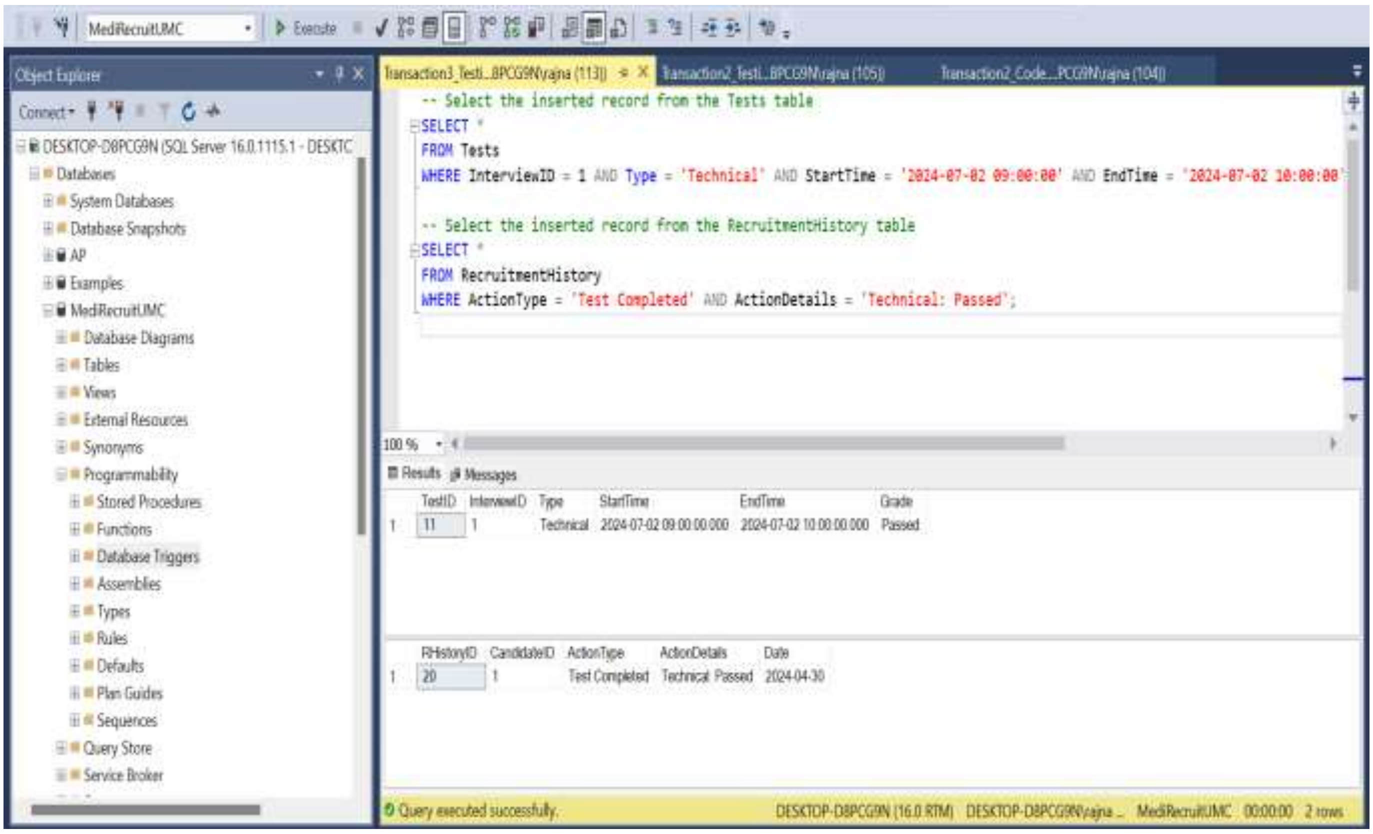
```sql
-- Transaction 3: Processing multiple tests
        BEGIN TRAN;

        INSERT INTO Tests (InterviewID, Type, StartTime, EndTime, Grade)
        VALUES (1, 'Technical', '2024-07-02 09:00:00', '2024-07-02 10:00:00', 'Passed');

        INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
        VALUES (
          (SELECT CandidateID FROM Application WHERE ApplicationID = 1),
          'Test Completed',
          'Technical: Passed',
          GETDATE()
        );
        COMMIT;
```

Transaction Logic:
This transaction facilitates the processing of multiple tests for a candidate. Firstly, it inserts a new record into the "Tests" table, specifying details such as the interview ID, test type, start and end times, and the test grade. Subsequently, it records the completion of the test in the "RecruitmentHistory" table, capturing relevant details like the test type and its outcome. Finally, the transaction commits these changes, ensuring the successful completion of the test processing.

Testing Query:
To validate the transaction, two queries are executed. The first query selects the inserted record from the "Tests" table, filtering by the specified test details. The second query retrieves the corresponding log entry from the "RecruitmentHistory" table, specifically filtering for entries indicating the completion of the test with the expected outcome. These queries confirm that the transaction accurately processes multiple tests and logs the completion details appropriately.

SQL Query:

```sql
-- Select the inserted record from the Tests table
SELECT *
FROM Tests
WHERE InterviewID = 1 AND Type = 'Technical' AND StartTime = '2024-07-02 09:00:00' AND
EndTime = '2024-07-02 10:00:00' AND Grade = 'Passed';
-- Select the inserted record from the RecruitmentHistory table
SELECT *
FROM RecruitmentHistory
WHERE ActionType = 'Test Completed' AND ActionDetails = 'Technical: Passed';
```



```
-- Select the inserted record from the Tests table
SELECT *
FROM Tests
WHERE InterviewID = 1 AND Type = 'Technical' AND StartTime = '2024-07-02 09:00:00' AND EndTime = '2024-07-02 10:00:00'

-- Select the inserted record from the RecruitmentHistory table
SELECT *
FROM RecruitmentHistory
WHERE ActionType = 'Test Completed' AND ActionDetails = 'Technical: Passed';
```

	TestID	InterviewID	Type	StartTime	EndTime	Grade
1	11	1	Technical	2024-07-02 09:00:00.000	2024-07-02 10:00:00.000	Passed

	RHistoryID	CandidateID	ActionType	ActionDetails	Date
1	20	1	Test Completed	Technical: Passed	2024-04-30

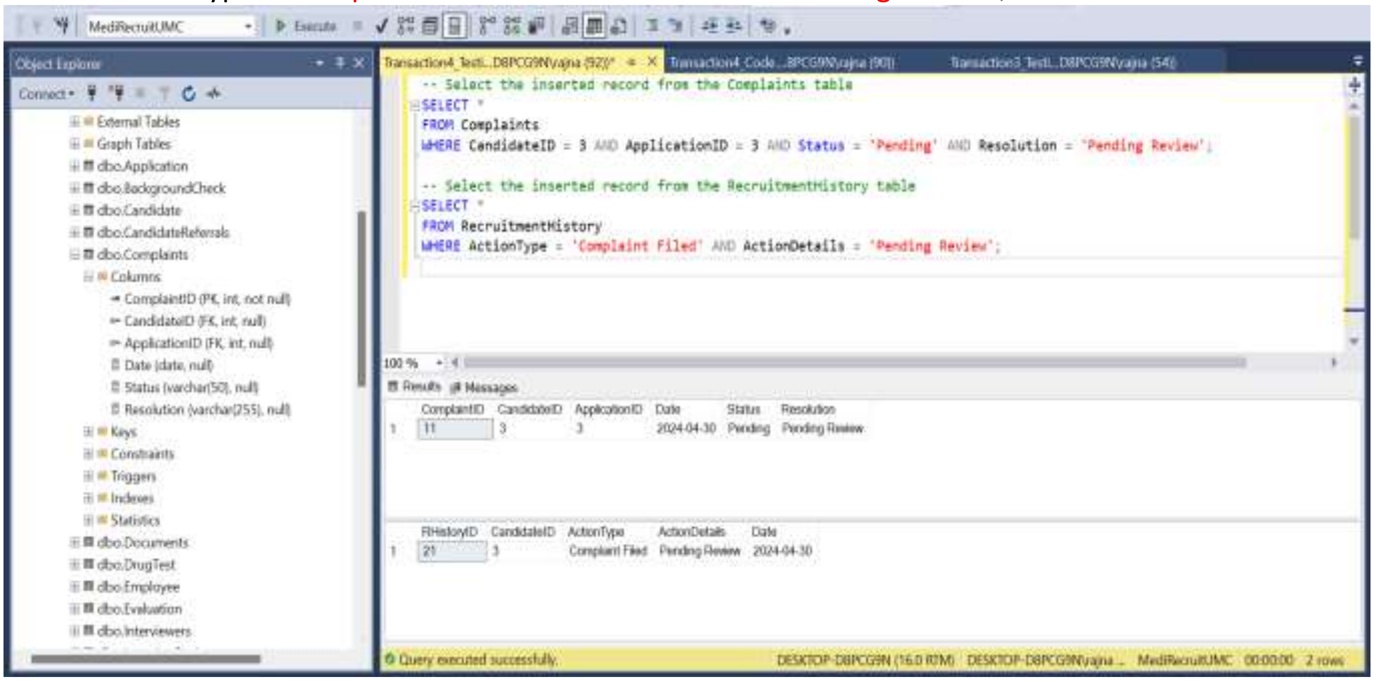
Query executed successfully. DESKTOP-D8PCG9N (16.0 RTM) DESKTOP-D8PCG9N\rajna ... MediRecruitUMC 00:00:00 2 rows

```
-- Transaction 4: Logging new complaints
BEGIN TRAN;
INSERT INTO Complaints (CandidateID, ApplicationID, Date, Status, Resolution)
VALUES (3, 3, GETDATE(), 'Pending', 'Pending Review');
INSERT INTO RecruitmentHistory (CandidateID, ActionType, ActionDetails, Date)
VALUES (3, 'Complaint Filed', 'Pending Review', GETDATE());
COMMIT;
```

Transaction Logic:
This transaction involves logging new complaints filed by candidates. It begins by inserting a new complaint record into the "Complaints" table. Subsequently, it adds a corresponding log entry to the "RecruitmentHistory" table, documenting the filing of the complaint with the status set to 'Pending' and the resolution as 'Pending Review'. Finally, the transaction commits these changes, ensuring the successful recording of the new complaint and its details.

Testing Query:
To verify the transaction's functionality, two queries are executed. The first query selects the inserted complaint record from the "Complaints" table, filtering by the specified complaint details. The second query retrieves the corresponding log entry from the "RecruitmentHistory" table, specifically filtering for entries indicating the filing of a new complaint with the expected status and resolution.
SQL Query:

```
-- Select the inserted record from the Complaints table
SELECT *
FROM Complaints
WHERE CandidateID = 3 AND ApplicationID = 3 AND Status = 'Pending' AND Resolution =
'Pending Review';
-- Select the inserted record from the RecruitmentHistory table
SELECT *
FROM RecruitmentHistory
WHERE ActionType = 'Complaint Filed' AND ActionDetails = 'Pending Review';
```



```
-- Select the inserted record from the Complaints table
SELECT *
FROM Complaints
WHERE CandidateID = 3 AND ApplicationID = 3 AND Status = 'Pending' AND Resolution = 'Pending Review';

-- Select the inserted record from the RecruitmentHistory table
SELECT *
FROM RecruitmentHistory
WHERE ActionType = 'Complaint Filed' AND ActionDetails = 'Pending Review';
```

	ComplaintID	CandidateID	ApplicationID	Date	Status	Resolution
1	11	3	3	2024-04-30	Pending	Pending Review

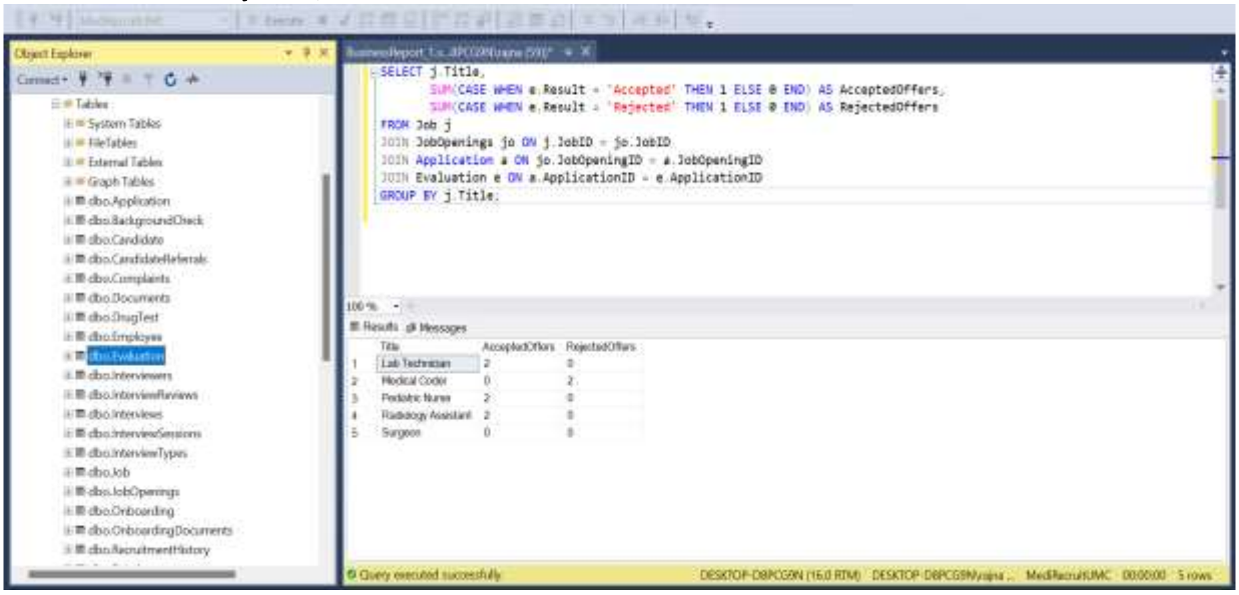
	RHistoryID	CandidateID	ActionType	ActionDetails	Date
1	21	3	Complaint Filed	Pending Review	2024-04-30

```
Query executed successfully.    DESKTOP-D8PCG9N (16.0 RTM)   DESKTOP-D8PCG9N\vajna ...   MediRecruitUMC   00:00:00   2 rows
```

Business Reports Development

Query 1: “Job Offer Acceptance Analysis”

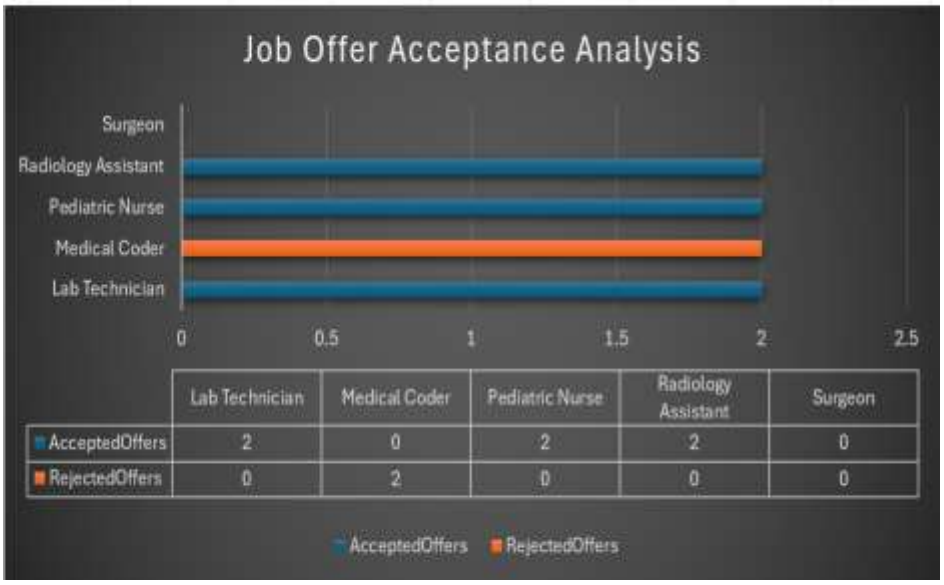
```
SELECT j.Title,
    SUM(CASE WHEN e.Result = 'Accepted' THEN 1 ELSE 0 END) AS AcceptedOffers,
    SUM(CASE WHEN e.Result = 'Rejected' THEN 1 ELSE 0 END) AS RejectedOffers
FROM Job j
JOIN JobOpenings jo ON j.JobID = jo.JobID
JOIN Application a ON jo.JobOpeningID = a.JobOpeningID
JOIN Evaluation e ON a.ApplicationID = e.ApplicationID
GROUP BY j.Title;
```



```
SELECT j.Title,
    SUM(CASE WHEN e.Result = 'Accepted' THEN 1 ELSE 0 END) AS AcceptedOffers,
    SUM(CASE WHEN e.Result = 'Rejected' THEN 1 ELSE 0 END) AS RejectedOffers
FROM Job j
JOIN JobOpenings jo ON j.JobID = jo.JobID
JOIN Application a ON jo.JobOpeningID = a.JobOpeningID
JOIN Evaluation e ON a.ApplicationID = e.ApplicationID
GROUP BY j.Title;
```

	Title	AcceptedOffers	RejectedOffers
1	Lab Technician	2	0
2	Medical Coder	0	2
3	Pediatric Nurse	2	0
4	Radiology Assistant	2	0
5	Surgeon	0	0

```
Query executed successfully    DESKTOP-D8PCG9N (16.0 RTM)   DESKTOP-D8PCG9N\yajna ...   MediRecruitUMC   00:00:00   5 rows
```

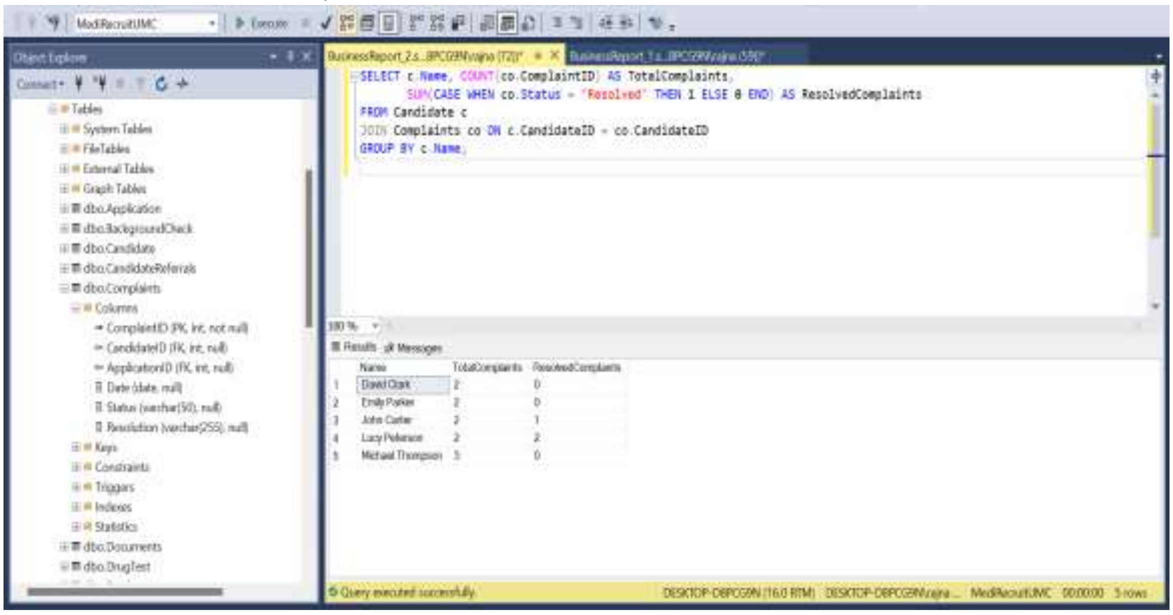


	Lab Technician	Medical Coder	Pediatric Nurse	Radiology Assistant	Surgeon
AcceptedOffers	2	0	2	2	0
RejectedOffers	0	2	0	0	0

The above report analyses job offer acceptance rates for different job titles. It selects the title of each job, alongside two calculated columns: "AcceptedOffers" and "RejectedOffers". These columns count the number of jobs offers accepted and rejected, respectively, based on evaluation results stored in the "Evaluation" table. It groups the data by job title to provide a summary of offer acceptance and rejection rates for each job. This analysis helps to assess the effectiveness of job offers and understand candidate preferences.

Query 2: "Candidate Complaint Resolution Analysis"

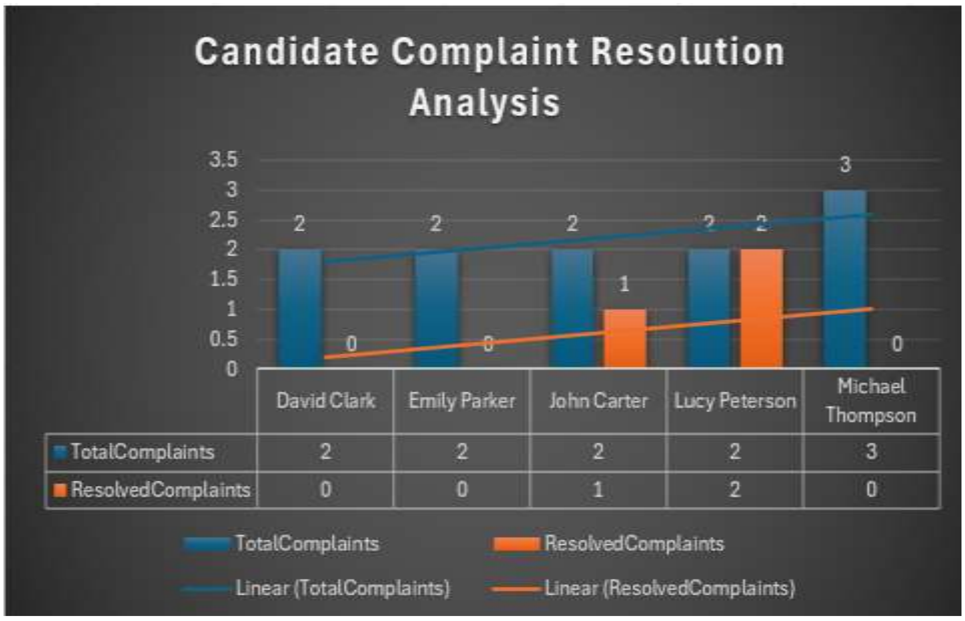
```
SELECT c.Name, COUNT(co.ComplaintID) AS TotalComplaints,
    SUM(CASE WHEN co.Status = 'Resolved' THEN 1 ELSE 0 END) AS
ResolvedComplaints
FROM Candidate c
JOIN Complaints co ON c.CandidateID = co.CandidateID
GROUP BY c.Name;
```



```
SELECT c.Name, COUNT(co.ComplaintID) AS TotalComplaints,
       SUM(CASE WHEN co.Status = 'Resolved' THEN 1 ELSE 0 END) AS ResolvedComplaints
FROM Candidate c
JOIN Complaints co ON c.CandidateID = co.CandidateID
GROUP BY c.Name;
```

```
   Name              TotalComplaints  ResolvedComplaints
1  David Clark       2                0
2  Emily Parker      2                0
3  John Carter       2                1
4  Lucy Peterson     2                2
5  Michael Thompson  3                0
```

Query executed successfully.   DESKTOP-DBPCG9N (16.0 RTM)   DESKTOP-DBPCG9N\vajna ...   MediRecruitUMC   00:00:00   5 rows

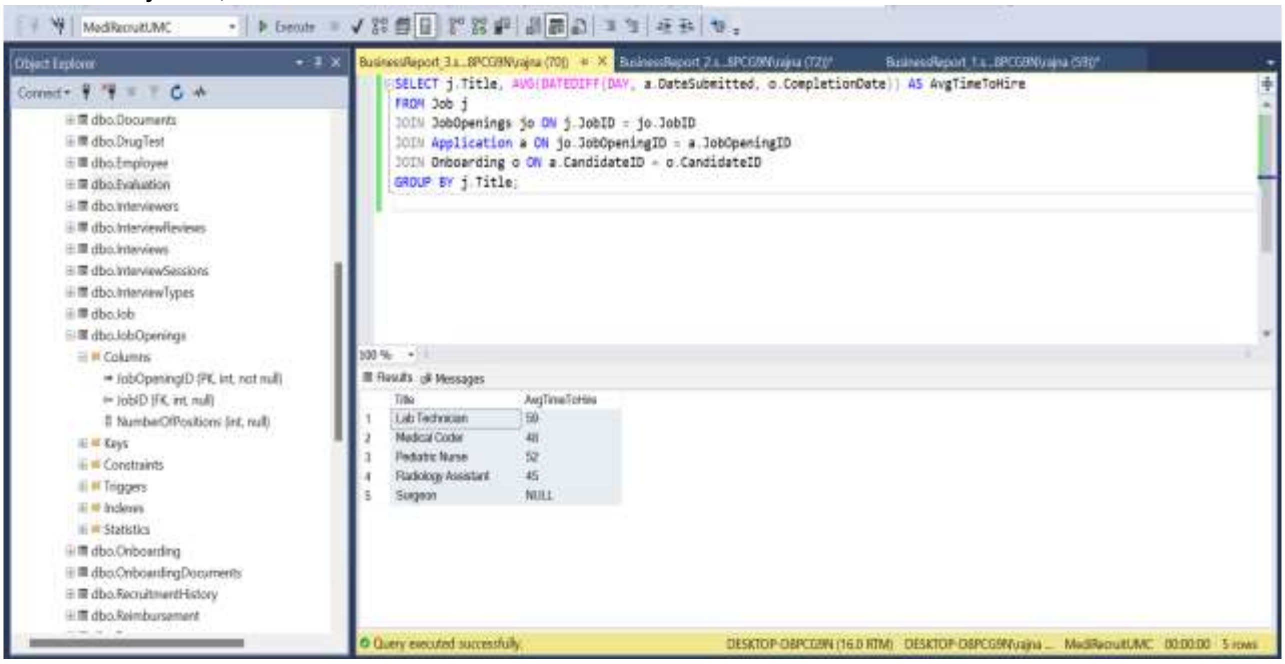


	David Clark	Emily Parker	John Carter	Lucy Peterson	Michael Thompson
TotalComplaints	2	2	2	2	3
ResolvedComplaints	0	0	1	2	0

This report assesses the complaint resolution status for each candidate. It selects the name of each candidate from the "Candidate" table and calculates two metrics: "TotalComplaints" and "ResolvedComplaints". "TotalComplaints" counts the total number of complaints associated with each candidate, while "ResolvedComplaints" counts the number of complaints that have been resolved (status = 'Resolved'). It then groups the data by candidate name to provide a summary of the total complaints and resolved complaints for each candidate. This analysis helps in evaluating candidates' satisfaction levels and the effectiveness of the complaint resolution process.

Query 3: "Average Time to Hire by Job Title"

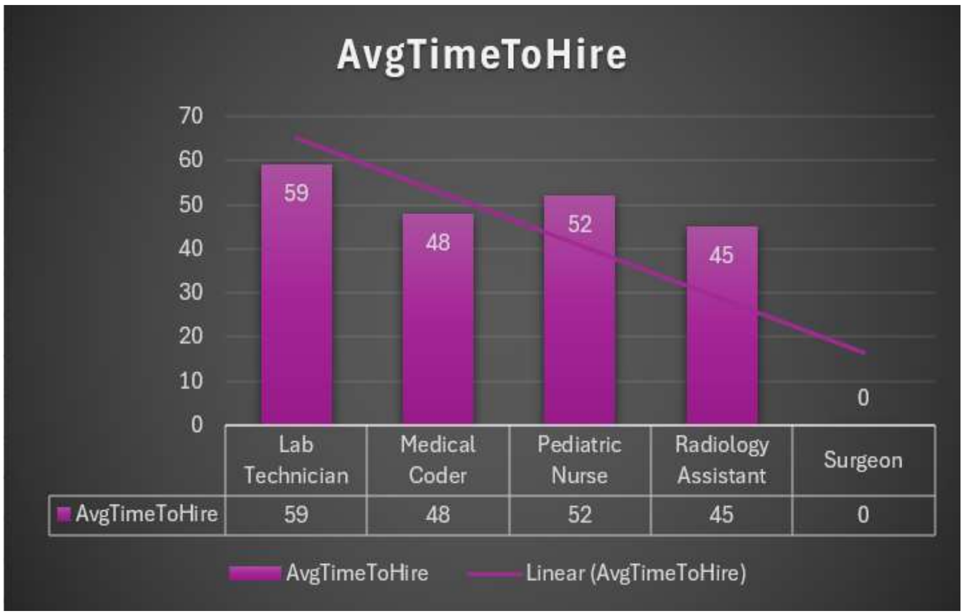
```
SELECT j.Title, AVG(DATEDIFF(DAY, a.DateSubmitted, o.CompletionDate)) AS AvgTimeToHire
FROM Job j
JOIN JobOpenings jo ON j.JobID = jo.JobID
JOIN Application a ON jo.JobOpeningID = a.JobOpeningID
JOIN Onboarding o ON a.CandidateID = o.CandidateID
GROUP BY j.Title;
```



```
SELECT j.Title, AVG(DATEDIFF(DAY, a.DateSubmitted, o.CompletionDate)) AS AvgTimeToHire
FROM Job j
JOIN JobOpenings jo ON j.JobID = jo.JobID
JOIN Application a ON jo.JobOpeningID = a.JobOpeningID
JOIN Onboarding o ON a.CandidateID = o.CandidateID
GROUP BY j.Title;
```

	Title	AvgTimeToHire
1	Lab Technician	59
2	Medical Coder	48
3	Pediatric Nurse	52
4	Radiology Assistant	45
5	Surgeon	NULL

Query executed successfully. DESKTOP-D8PCG9N (16.0 RTM) DESKTOP-D8PCG9N\rajna ... MediRecruitUMC 00:00:00 5 rows

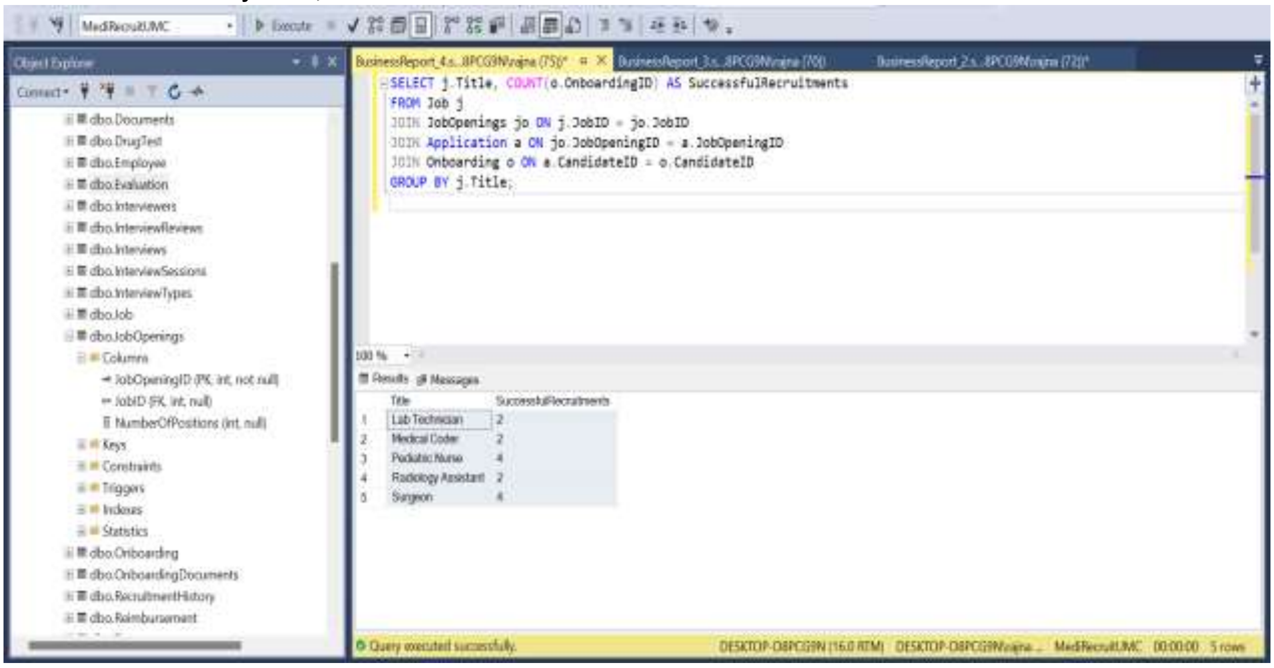


	Lab Technician	Medical Coder	Pediatric Nurse	Radiology Assistant	Surgeon
AvgTimeToHire	59	48	52	45	0

This report calculates the average time taken to hire candidates for each job title. It selects the job title from the "Job" table and computes the average number of days between the date of application submission ("DateSubmitted" in the "Application" table) and the completion date of onboarding ("CompletionDate" in the "Onboarding" table). By joining "Job", "JobOpenings", "Application", and "Onboarding" tables and grouping by job title, it provides insights into the efficiency of the hiring process for different job roles.

<u>Query 4: "Successful Recruitments by Job Title"</u>

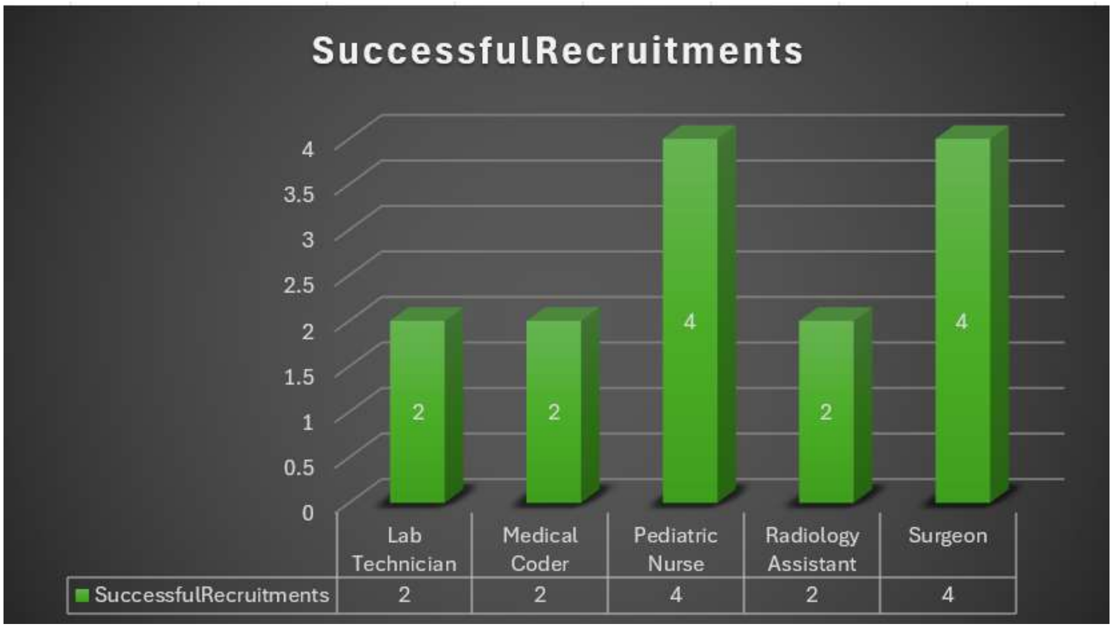
```
SELECT j.Title, COUNT(o.OnboardingID) AS SuccessfulRecruitments
FROM Job j
JOIN JobOpenings jo ON j.JobID = jo.JobID
JOIN Application a ON jo.JobOpeningID = a.JobOpeningID
JOIN Onboarding o ON a.CandidateID = o.CandidateID
GROUP BY j.Title;
```



```
SELECT j.Title, COUNT(o.OnboardingID) AS SuccessfulRecruitments
FROM Job j
JOIN JobOpenings jo ON j.JobID = jo.JobID
JOIN Application a ON jo.JobOpeningID = a.JobOpeningID
JOIN Onboarding o ON a.CandidateID = o.CandidateID
GROUP BY j.Title;
```

	Title	SuccessfulRecruitments
1	Lab Technician	2
2	Medical Coder	2
3	Pediatric Nurse	4
4	Radiology Assistant	2
5	Surgeon	4

Query executed successfully.



	Lab Technician	Medical Coder	Pediatric Nurse	Radiology Assistant	Surgeon
🟩 SuccessfulRecruitments	2	2	4	2	4

This report counts the number of successful recruitments for each job title. It selects the job title from the "Job" table and counts the occurrences of successful recruitments, identified by the "OnboardingID" in the "Onboarding" table. By joining the "Job", "JobOpenings", "Applications", and "Onboarding" tables and grouping by job title, it provides insights into the effectiveness of recruitment efforts for different job roles.

CONCLUSION AND KEY INSIGHTS

Through the completion of this project, I have acquired invaluable knowledge and enhanced my skills in database design, implementation, and management. The intricate process of conceptualizing, planning, and executing a comprehensive recruitment database for the University Medical Center's HR Department has been both challenging and rewarding.

Learning and Skill Enhancement:

- I have deepened my understanding of database normalization, schema design, and entity-relationship modelling. By adhering to best practices, I ensured the efficiency, scalability, and maintainability of the database structure.

- Working extensively with SQL queries, stored procedures, functions, triggers, and transactions has honed my SQL proficiency. I have learned to write complex queries, optimize database performance, and enforce data integrity constraints effectively.

- The development of business reports has sharpened my analytical skills and data visualization techniques. I gained insights into interpreting data trends, generating actionable insights, and presenting findings in a compelling manner.

- Managing the project lifecycle from inception to completion has bolstered my project management skills. I learned to prioritize tasks, allocate resources efficiently, and adapt to evolving requirements while maintaining project timelines and deliverables.

Usefulness and Impact:
This project holds immense significance as it addresses real-world challenges faced by healthcare institutions in talent acquisition and management. By providing a robust recruitment database solution, we contribute to streamlining HR operations, enhancing organizational efficiency, and ultimately improving patient care outcomes. The insights derived from business reports empower stakeholders to make informed decisions, optimize recruitment strategies, and drive organizational success.

Final Conclusion:
The successful completion of this project not only underscores my proficiency in database management but also reflects my commitment to leveraging technology for tangible organizational impact. As I reflect on this journey, I am confident that the knowledge gained, and skills honed will serve as a solid foundation for tackling future challenges and driving innovation in the healthcare industry. This project exemplifies my dedication to continuous learning, professional growth, and making a meaningful difference in the world.

BIBLIOGRAPHY:

The material for this project, including definitions and understanding of design principles, is derived from various online sources and search engines, including Google. These sources provide insights into database design, SQL queries, views, triggers, stored procedures, and business report generation, among other related topics.