DATABASE 281 PROJECT

DR KNOETZE'S PHARMACEUTICAL PRACTICE

Table of Contents

Introduction

Background of Organization

Dr. Johannalie Knoetze is esteemed in social work, dedicating herself to children's well-being. Her private practice offers a nurturing space for children to heal and grow, leveraging play therapy as a transformative tool.She takes a holistic approach, addressing root causes while nurturing strengths. Collaborating closely with families and professionals, Dr. Knoetze tailors treatment plans to each child's needs. Her practice promotes inclusivity and cultural sensitivity, ensuring all feel valued.With unwavering dedication, Dr. Knoetze empowers children to overcome challenges and embrace their potential, leaving a profound impact on their lives.

Problem Statement:

The reliance on fragmented and paper-based documentation methods hinders Dr. Knoetze's practice, scattering client information across various files and impeding efficient access and updates. This fragmentation not only slows administrative tasks but also heightens the risk of errors due to outdated or misplaced data.Moreover, the absence of a centralized database inhibits the practice's ability to track client progress and analyze trends effectively. Social work interventions necessitate longitudinal care monitoring, yet without a systematic approach to track therapy outcomes, the practice struggles to provide evidence-based care and measure service impact.Implementing an SQL database server presents a solution to these challenges. By consolidating client data, session details, and therapy outcomes into a single, secure database, administrative processes can be streamlined, reducing errors and enhancing data integrity.Furthermore, an SQL database enables advanced analytics, allowing practitioners to track client progress, analyze trends, and make data-driven decisions. With insights into intervention effectiveness and areas for improvement, service delivery can be tailored to meet client needs more effectively.Additionally, an SQL database provides a scalable platform for future growth and innovation, ensuring adaptability to changing needs and emerging technologies.In conclusion, implementing an SQL database server empowers Dr. Knoetze's practice to overcome the limitations of fragmented data management methods. By centralizing data, improving analytics capabilities, and ensuring scalability, the practice can enhance service delivery, improve outcomes, and demonstrate commitment to delivering high-quality social work services.

User Needs

Administrative support in receiving and documenting new intakes:

- Develop an online intake form accessible through the database management system for new clients to submit their information.
- Design automated workflows within the database system to route intake forms to appropriate administrative staff for review and processing.
- Enable data validation features to ensure accuracy and completeness of intake information entered the system.
- Appointments for the different types of services rendered to be categorized and booked accordingly:
- Utilize the database system to create appointment slots categorized by service type, therapist availability, and location.
- Implement a booking interface that integrates with the database to allow staff to schedule appointments directly within the system.
- Enable real-time updates to appointment availability to prevent double bookings and ensure efficient allocation of resources.

Reminders before appointments:

- Set up automated reminder functionalities within the database system to send appointment reminders to clients based on their preferred communication method.
- Integrate appointment confirmation features to track client responses and update appointment statuses in real-time.
- Utilize data analytics tools within the database to analyze appointment reminder effectiveness and optimize communication strategies.

Monitoring of payments to ensure that payment is made in advance:

- Implement payment tracking modules within the database system to record payment status and history for each client.
- Create alerts or notifications to flag clients with outstanding balances or upcoming payment deadlines.
- Utilize reporting capabilities to generate payment status reports and monitor trends in payment behavior over time.

Method of easily monitoring outstanding payments and clients that need to be invoiced or followed up:

- Develop dashboard widgets or reports within the database system to display a summary of outstanding payments and overdue invoices.
- Implement filters and sorting options to prioritize follow-up actions based on payment urgency and client priority.

- Integrate communication features to enable staff to send automated reminders or follow-up messages directly from the database system.
- Easily accessible profile of each client with parent's initial concern, therapeutic needs, identifying details, and other descriptive details (age, school, family, etc.):
- Design comprehensive client profiles within the database system to capture all relevant information, including parent details, therapeutic needs, and demographic data.
- Implement search and filtering functionalities to quickly locate and access client profiles based on various criteria such as name, age, or presenting concern.
- Utilize secure access controls to ensure that only authorized staff can view and edit client information, maintaining confidentiality and compliance with privacy regulations.

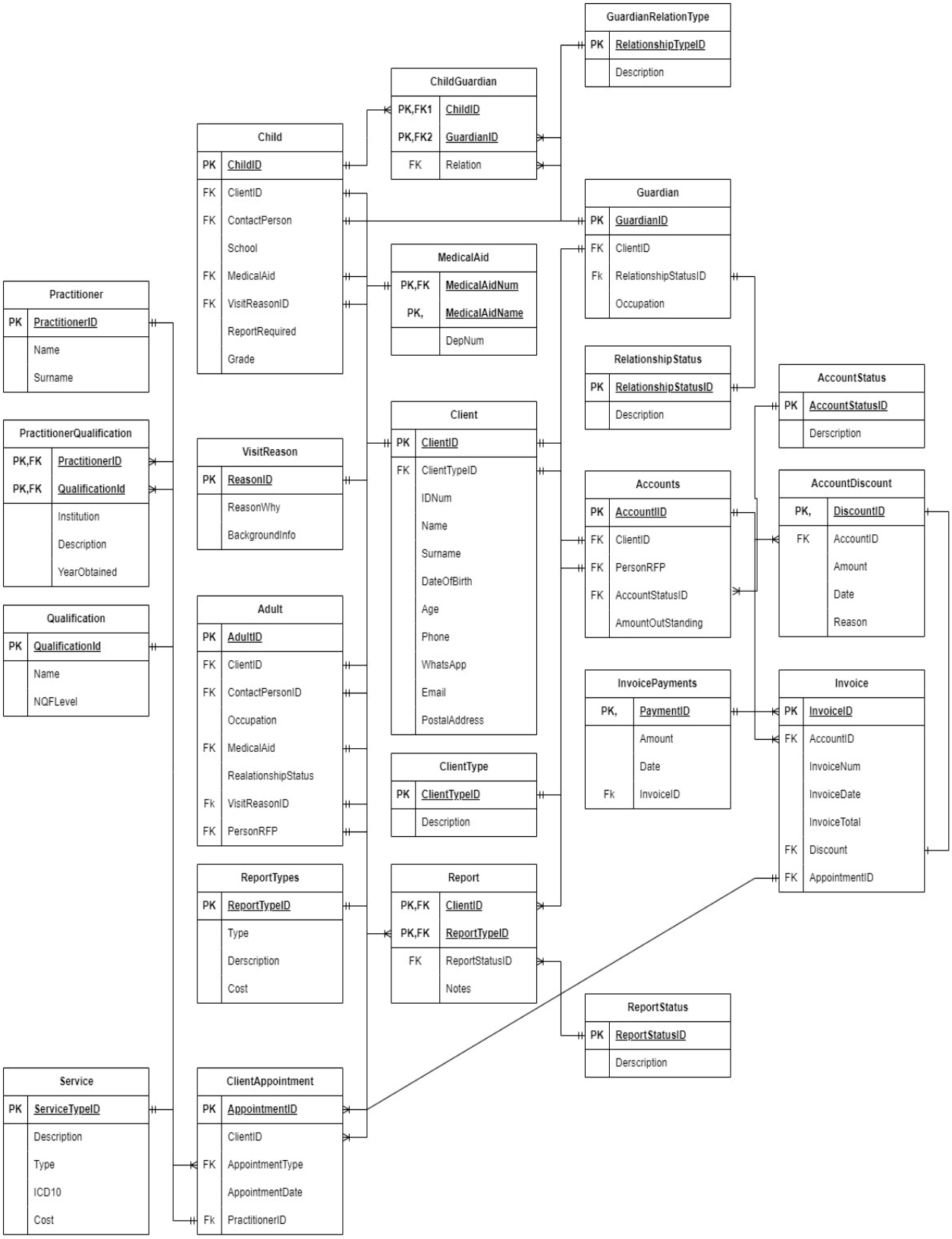
Numbering system for filing and referencing purposes:

- Establish a standardized numbering system within the database system to assign unique identifiers or codes to client records, appointments, and documents.
- Implement metadata tagging capabilities to categorize and classify files for easy retrieval and referencing.
- Utilize hyperlinking features to create cross-references between related records and documents within the database system.

Way to link Parent to multiple children as well as multiple guardians to one child:

- Utilize relational database structures to establish parent-child and guardian-child relationships within the system.
- Create parent and guardian profiles linked to respective child records, enabling seamless navigation between family members' information.
- Implement cascading updates to ensure consistency and accuracy of family relationship data across all linked records within the database system.

Database ERD



Normalization Process

1. First Normal Form (1NF)

Ensuring that each entity **(Client, Practitioner, Invoice, etc.)** has its own table with a primary key is foundational to achieving 1NF. This means that each table must have a unique identifier (primary key) and that each attribute within the table must hold atomic values (no repeating groups or arrays).

For example, in the **Client table**, each client would have a unique client ID (primary key), and attributes such as client name, address, and contact information would be stored in separate columns, ensuring atomicity.

2. Second Normal Form (2NF)

Splitting tables with multi-valued dependencies into separate tables is crucial for achieving 2NF. This means identifying attributes that are functionally dependent on only part of the primary key and moving them to separate tables.

For instance, in the case of **PractitionerQualification,** if qualifications are specific to practitioners and not to the entire practice, it's appropriate to create a separate table for qualifications. This new table would have a composite primary key consisting of both the **practitioner ID and the qualification ID**, ensuring that each qualification is uniquely identified in the context of a particular practitioner.

3. Third Normal Form (3NF)

Eliminating attributes that could potentially cause transitive dependencies involves removing them from tables and placing them in their own tables with appropriate relationships. This ensures 3NF by eliminating transitive dependencies.

For example, separating Guardian from Client and Child eliminates transitive dependencies. If a Guardian's information is stored directly in the Client or Child table, it could lead to redundancy and potential anomalies. Instead, creating a separate table for Guardian and linking them via **ChildGuardian** ensures that the relationship between clients/children and their guardians is properly represented without redundancy.

Similarly, separating **ReportTypes** from Report ensures that each report type is uniquely identified and avoids the potential for transitive dependencies. Storing **MedicalAid** details in their own structure instead of directly in the Client table maintains data integrity and avoids redundancy, thereby adhering to 3NF principles.

By following these normalization principles, the database design achieves a higher level of data integrity, reduces redundancy, and ensures that data anomalies are minimized, thereby promoting efficient data management and maintenance

Proposed Physical Database Environment

Database Server Setup:

1. SQL Server Standard Edition 2019:

SQL Server Standard Edition 2019 is a robust and versatile database management system developed by Microsoft. It offers a comprehensive set of features designed to support various data management tasks, including storage, retrieval, manipulation, and analysis of data.

Some key features of SQL Server Standard Edition 2019 include:

The core component of SQL Server responsible for storing, processing, and securing data. It provides support for relational database management, transaction processing, and data integrity enforcement.

SQL Server includes tools and services for business intelligence (BI) and analytics, such as SQL Server Analysis Services (SSAS) for multidimensional data analysis, SQL Server Reporting Services (SSRS) for creating and distributing reports, and SQL Server Integration Services (SSIS) for data integration and transformation.

SQL Server is designed to scale efficiently to accommodate growing data volumes and increasing workloads. It includes features such as partitioning, indexing, and query optimization to enhance performance and ensure responsiveness.

2. Windows Server 2019 Datacenter Edition:

Windows Server 2019 Datacenter Edition is a server operating system developed by Microsoft that offers advanced capabilities for hosting and managing mission-critical applications and services.

Key features and capabilities of Windows Server 2019 Datacenter Edition include:

Windows Server 2019 includes enhanced security features to protect against cybersecurity threats and safeguard sensitive data. This includes features such as Windows Defender Advanced Threat Protection (ATP), Credential Guard, and Windows Defender Exploit Guard.

Windows Server 2019 is designed to deliver high performance and scalability to meet the demands of modern enterprise workloads. It supports large-scale virtualization, software-defined storage, and software-defined networking capabilities to optimize resource utilization and enhance scalability

Hardware Requirements:

1. CPU: 2 x Intel Xeon Gold 6248R (2.9 GHz, 24 cores each):

- The Intel Xeon Gold 6248R processors are high-performance CPUs designed for demanding workloads, such as database management systems.
- With a total of 48 cores (24 cores per CPU), these processors provide significant processing power, enabling efficient execution of database queries, transactions, and analytics tasks.
- The 2.9 GHz base clock speed ensures fast data processing, while the high number of cores allows for parallel processing of multiple database operations simultaneously.
- For a pharmaceutical practice, which may deal with large volumes of data related to drug development, clinical trials, regulatory compliance, and inventory management, powerful CPUs like the Intel Xeon Gold 6248R are essential for handling complex queries and data analysis tasks efficiently.

2. RAM: 128 GB DDR4 ECC RAM

- The 128 GB of DDR4 ECC (Error-Correcting Code) RAM provides ample memory capacity to support the database server's operations.
- ECC RAM is crucial for ensuring data integrity and reliability by detecting and correcting memory errors that could lead to data corruption or system instability.
- With a large memory capacity, the database server can efficiently cache frequently accessed data, reducing disk I/O operations and improving overall system performance.
- In a pharmaceutical practice, where databases may contain extensive datasets, including patient records, drug formulations, research data, and regulatory documentation, sufficient RAM is essential for maintaining optimal database performance during data retrieval, processing, and analysis.

3. Storage:

1 TB NVMe SSD for OS and SQL Server installation:

- NVMe SSDs offer high-speed data transfer rates and low latency, making them ideal for hosting the operating system (OS) and SQL Server installation.
- The fast read/write speeds of NVMe SSDs ensure quick boot times, rapid application loading, and efficient SQL Server performance, especially for tasks involving database indexing, logging, and temporary file storage.
- Storing the OS and SQL Server installation on a separate SSD drive helps isolate system files from database data, reducing contention and improving overall system reliability and performance.

4 TB SAS HDD for database storage:

- SAS (Serial Attached SCSI) HDDs provide ample storage capacity for storing pharmaceutical practice databases, which may include large volumes of transactional and analytical data.

- Although SAS HDDs offer lower performance compared to SSDs, they provide cost-effective storage solutions for storing less frequently accessed data, historical records, and backups.
- By segregating database storage onto a separate HDD drive, organizations can optimize storage resources, balance performance and cost considerations, and ensure scalability and data protection for pharmaceutical-related data.

Security:

Authentication: Utilize Windows Authentication for enhanced security:

Leveraging Windows Authentication, which is integrated with Active Directory, enhances security by allowing users to log in to the SQL Server instance using their Windows credentials.

Since the database server is running on Windows Server 2019 Datacenter Edition, Windows Authentication seamlessly integrates with the operating system's security infrastructure, simplifying user management and authentication processes.

With Windows Authentication, administrators can easily manage user accounts, enforce password policies, and control access to the database server based on Windows user groups and permissions.

Additionally, Windows Authentication provides Single Sign-On (SSO) capabilities, streamlining the login process for users and reducing the risk of password-related security vulnerabilities.

Authorization: Define roles such as Therapists, Administrators, and Receptionists with appropriate permissions:

Utilizing role-based authorization ensures that users are granted access to database resources based on their specific roles and responsibilities within the pharmaceutical practice.

By defining roles such as Therapists, Administrators, and Receptionists, administrators can assign granular permissions to each role, controlling access to database objects such as tables, views, stored procedures, and functions.

For example, Therapists may be granted permissions to access and modify patient records, while Administrators have full control over database management tasks such as backups, restores, and user management.

Receptionists, on the other hand, may only have permissions to view patient appointment schedules and update basic demographic information.

Role-based authorization ensures that users have the necessary access rights to perform their job duties while restricting unauthorized access to sensitive data and database functionality.

Encryption: Implement Transparent Data Encryption (TDE) to encrypt sensitive data at rest:

Transparent Data Encryption (TDE) provides encryption at the database level, encrypting data files, log files, and backups to protect sensitive information stored on disk.

With the database server configured with a 1 TB NVMe SSD for OS and SQL Server installation, implementing TDE ensures that sensitive pharmaceutical data stored on the SSD is encrypted, mitigating the risk of unauthorized access or data breaches in case of physical theft or unauthorized access to the storage device.

TDE encrypts data using a database encryption key (DEK) stored in the database boot record, which is itself encrypted with a certificate stored in the master database. This ensures that data remains encrypted even if the underlying storage media is compromised.

By implementing TDE, the pharmaceutical practice can maintain compliance with regulatory requirements such as HIPAA (Health Insurance Portability and Accountability Act) and GDPR (General Data Protection Regulation) and demonstrate a commitment to protecting patient privacy and confidentiality.

Monitoring and Maintenance:

Performance Monitoring using SSMS and SQL Server Profiler:

SQL Server Management Studio (SSMS):

- SSMS provides a comprehensive set of tools for monitoring and managing SQL Server instances, databases, and performance metrics.
- Administrators can use SSMS to monitor key performance indicators such as CPU usage, memory consumption, disk I/O, and query execution times through built-in performance monitoring dashboards and reports.
- SSMS allows for real-time monitoring of active sessions, locks, and database activity, enabling administrators to identify and troubleshoot performance issues quickly.
- Additionally, SSMS provides tools for configuring and analyzing execution plans, identifying query bottlenecks, and optimizing query performance.

SQL Server Profiler:

- SQL Server Profiler is a powerful tool for capturing and analyzing database events and queries in real-time.
- Administrators can use SQL Server Profiler to create trace sessions that capture events such as SQL statements, stored procedure executions, and database errors.

- By analyzing captured traces, administrators can identify poorly performing queries, diagnose performance bottlenecks, and optimize query execution plans.
- SQL Server Profiler also allows for the creation of custom trace templates and filters to focus on specific events or activities relevant to performance monitoring and troubleshooting.

Maintenance Tasks using SQL Server Agent Jobs:

Index Maintenance:

- Schedule regular index maintenance tasks, such as rebuilding or reorganizing indexes, to optimize query performance and reduce fragmentation.
- SQL Server Agent jobs can be created to automate index maintenance tasks based on predefined schedules or performance thresholds.
- By regularly maintaining indexes, administrators can ensure optimal query execution times and improve overall database performance.

Statistics Updates:

- Schedule statistics updates to ensure accurate query optimization and execution plans.
- SQL Server Agent jobs can be configured to update statistics on tables and indexes based on predefined schedules or database usage patterns.
- Regular statistics updates help the query optimizer make informed decisions about query execution plans, leading to improved query performance and resource utilization.

Database Integrity Checks:

- Schedule regular database integrity checks to identify and repair any inconsistencies or corruption in database objects.
- SQL Server Agent jobs can automate the execution of DBCC CHECKDB commands to verify database integrity and repair any detected issues.
- Performing regular database integrity checks helps maintain data consistency, reliability, and data integrity, reducing the risk of data loss or corruption.
- By leveraging SQL Server Management Studio, SQL Server Profiler, and SQL Server Agent jobs for performance monitoring and maintenance tasks, administrators can effectively manage and optimize the database environment, ensuring optimal performance, reliability, and data integrity for the pharmaceutical practice's critical operations

Data File Organization:

Data Files Organization Across Multiple Filegroups:

- Utilizing multiple filegroups allows for better management and optimization of storage resources within the database.
- The PRIMARY filegroup typically contains system tables and frequently accessed data, such as core tables and indexes essential for the database's operation.
- By placing system tables and commonly accessed data in the PRIMARY filegroup, database operations that involve these objects can benefit from faster access times and improved performance.
- The SECONDARY filegroup, on the other hand, can be designated for storing less frequently accessed data, historical records, or large data sets that do not require frequent updates or queries.
- Organizing data across multiple filegroups enables administrators to prioritize storage resources based on access patterns, workload requirements, and performance considerations.
- For example, tables containing transactional data or real-time analytics may be placed in the PRIMARY filegroup for optimized performance, while archival data or historical logs may be stored in the SECONDARY filegroup to free up space and streamline database maintenance tasks.

Transaction Log Files Separated on Dedicated NVMe SSD:

- Separating transaction log files (.ldf) onto a dedicated NVMe SSD offers several benefits for performance and reliability.
- The transaction log records all changes made to the database, including inserts, updates, and deletes, providing a crucial mechanism for ensuring data integrity and enabling point-in-time recovery.
- Storing transaction log files on a high-speed NVMe SSD enhances write performance and reduces latency, as NVMe SSDs offer faster data transfer rates and lower access times compared to traditional HDDs or SATA SSDs.
- Placing transaction log files on a dedicated NVMe SSD helps prevent contention with data files and other database operations, ensuring consistent and reliable performance for transaction processing.
- Additionally, separating transaction log files onto a dedicated SSD enhances fault tolerance and resilience, as it reduces the risk of log file corruption or performance degradation caused by storage failures or contention.
- By optimizing transaction log storage with a dedicated NVMe SSD, the database environment can achieve higher throughput, lower latency, and improved reliability for critical transactional workloads

Defining and Describing Database Entities

CREATION OF TABLES

CREATE DATABASE DrJKnoetze: This query creates a new database named "DrJKnoetze" with specified data and log file details.

```
-- Create the DrJKnoetze database
CREATE DATABASE DrJKnoetze
ON
(NAME = DrJKnoetzeDataFile1,
FILENAME = 'C:\Users\erick\Documents\ProjectDBD\DrJKnoetzeDataFile1.mdf',
SIZE = 5MB,
MAXSIZE = UNLIMITED,
FILEGROWTH = 10%)
LOG ON
(NAME = DrJKnoetzeLogFile1,
FILENAME = 'C:\Users\erick\Documents\ProjectDBD\DrJKnoetzeDataLogFile1.ldf',
SIZE = 5MB,
MAXSIZE = UNLIMITED,
FILEGROWTH = 10MB
)
```

USE DrJKnoetze: This command switches the current database context to "DrJKnoetze" so that subsequent queries will be executed within this database.

```
-- Switch to the DrJKnoetze database context
USE DrJKnoetze;
```

BEGIN TRANSACTION; BEGIN TRY... END TRY: This block of code starts a transaction and wraps the subsequent code within a TRY block, which allows for error handling.

```
GO

BEGIN TRANSACTION;
BEGIN TRY
```

CREATE TABLE Practitioner: This query creates a table named "Practitioner" with columns for PractitionerID, Name, and Surname. PractitionerID is the primary key of the table.

```
-- Create Practitioner table
CREATE TABLE Practitioner (
    PractitionerID INT PRIMARY KEY IDENTITY,
    Name NVARCHAR(30) NOT NULL,
    Surname NVARCHAR(30) NOT NULL
);
```

CREATE **TABLE Qualification:** This query creates a table named "Qualification" with columns for QualificationID, Name, and NQFLevel. QualificationID is the primary key, and NQFLevel has a constraint to ensure it falls within a specified range.

```
-- Create Qualification table
CREATE TABLE Qualification (
    QualificationID INT PRIMARY KEY IDENTITY,
    Name NVARCHAR(30) NOT NULL,
    NQFLevel INT NOT NULL CONSTRAINT CHK_Qualification_NQFLevel CHECK (NQFLevel BETWEEN 1 AND 10)
);
```

CREATE TABLE PractitionerQualification: This query creates a junction table to establish a many-to-many relationship between Practitioner and Qualification tables. It includes columns for PractitionerID, QualificationID, Institution, Description, and YearObtained. PractitionerID and QualificationID together form the composite primary key.

```
-- Create PractitionerQualification table
CREATE TABLE PractitionerQualification (
    PractitionerID INT FOREIGN KEY REFERENCES Practitioner(PractitionerID),
    QualificationID INT FOREIGN KEY REFERENCES Qualification(QualificationID),
    Institution  NVARCHAR(80) NOT NULL,
    Description NVARCHAR(255),
    YearObtained SMALLINT NOT NULL,
    PRIMARY KEY (PractitionerID, QualificationID)
);
```

CREATE TABLE MedicalAid: This query creates a table named "MedicalAid" with columns for MedicalAidID, MedicalAidNum, and MedicalAidName. MedicalAidID is the primary key.

```
-- Create MedicalAid table
CREATE TABLE MedicalAid (
    MedicalAidID INT PRIMARY KEY IDENTITY,
    MedicalAidNum NVARCHAR(100) NOT NULL,
    MedicalAidName NVARCHAR(150) NOT NULL,
);
```

CREATE TABLE Client: This query creates a table named "Client" with columns for ClientID, IDNum, Name, Surname, DateOfBirth, MedicalAidID, Phone, WhatsAppNumber, PostalAddress, and Email. ClientID is the primary key, and several columns have constraints to ensure data integrity.

```
-- Create Client table
CREATE TABLE Client (
    ClientID INT PRIMARY KEY IDENTITY,
    IDNum VARCHAR(13) NOT NULL, CONSTRAINT CHK_SA_ID_Length CHECK (LEN(IDNum) = 13),
    Name NVARCHAR(30) NOT NULL,
    Surname NVARCHAR(30) NOT NULL,
    DateOfBirth DATE NOT NULL CONSTRAINT CHK_Client_DateOfBirth_Reasonable CHECK (DateOfBirth <=
GETDATE() AND DateOfBirth > '1900-01-01'),
    MedicalAidID INT FOREIGN KEY REFERENCES MedicalAid(MedicalAidID),
    Phone NVARCHAR(20) CONSTRAINT CHK_Client_Phone_Format CHECK (LEN(Phone) BETWEEN 10 AND 20),
    WhatsAppNumber  NVARCHAR(20) CONSTRAINT CHK_Client_WhatsAppNumber_Format CHECK (LEN
(WhatsAppNumber) BETWEEN 10 AND 20),
    PostalAddress NVARCHAR(255),
    Email NVARCHAR(80)
);
```

CREATE TABLE MedicalAidClient: This query creates a junction table to establish a many-to-many relationship between Client and MedicalAid tables.

```
CREATE TABLE MedicalAidClient(
MedicalAidID INT FOREIGN KEY REFERENCES MedicalAid(MedicalAidID),
ClientID INT FOREIGN KEY REFERENCES Client(ClientID),
DepNum INT NOT NULL,
);
```

CREATE TABLE VisitReason: This query creates a table named "VisitReason" with columns for ReasonID, ReasonWhy, and BackgroundInfo. ReasonID is the primary key.

```
-- Create VisitReason table
CREATE TABLE VisitReason (
    ReasonID INT PRIMARY KEY IDENTITY,
    ReasonWhy NVARCHAR(255) NOT NULL,
    BackgroundInfo NVARCHAR(1000)
);
```

CREATE TABLE Adult: This query creates a table named "Adult" to store information about adult clients. It includes columns for AdultID, ClientID, ContactPersonID, Occupation, RelationshipStatus, and VisitReasonID.

```
-- Create Adult table
CREATE TABLE Adult (
    AdultID INT PRIMARY KEY IDENTITY,
    ClientID INT NOT NULL FOREIGN KEY REFERENCES Client(ClientID),
    ContactPersonID INT NOT NULL FOREIGN KEY REFERENCES Client(ClientID),
    Occupation NVARCHAR(255) NOT NULL CONSTRAINT CHK_Adult_Occupation_Length CHECK (LEN(Occupation)
<= 255),
    RelationshipStatus  NVARCHAR(50) NOT NULL,
    VisitReasonID INT FOREIGN KEY REFERENCES VisitReason(ReasonID),
    PersonRFP INT NOT NULL FOREIGN KEY REFERENCES Client(ClientID)
);
```

CREATE TABLE Child: This query creates a table named "Child" to store information about child clients. It includes columns for ChildID, ClientID, ContactPersonID, School, VisitReasonID, and ReportRequired.

```
-- Create Child table
CREATE TABLE Child (
    ChildID INT PRIMARY KEY IDENTITY,
    ClientID INT NOT NULL FOREIGN KEY REFERENCES Client(ClientID),
    ContactPersonID INT NOT NULL FOREIGN KEY REFERENCES Client(ClientID),
    School NVARCHAR(100) CONSTRAINT CHK_Child_School_Length CHECK (LEN(School) <= 100),
    VisitReasonID INT FOREIGN KEY REFERENCES VisitReason(ReasonID),
    ReportRequired BIT NOT NULL,
    Grade INT NOT NULL
);
```

CREATE TABLE RelationshipStatus: This query creates a table named "RelationshipStatus" to store various relationship statuses.

```
-- Create RelationshipStatus table
CREATE TABLE RelationshipStatus (
    RelationshipStatusID INT PRIMARY KEY IDENTITY,
    Description NVARCHAR(50) NOT NULL
);
```

CREATE TABLE Guardian: This query creates a table named "Guardian" to store information about guardians of clients.

```
-- Create Guardian table
CREATE TABLE Guardian (
    GuardianID INT PRIMARY KEY IDENTITY,
    ClientID INT NOT NULL FOREIGN KEY REFERENCES Client(ClientID),
    RelationshipStatusID INT NOT NULL FOREIGN KEY REFERENCES RelationshipStatus(RelationshipStatusID)
,
    Occupation NVARCHAR(150)
);
```

CREATE TABLE GuardianRelationType: This query creates a table named "GuardianRelationType" to store different types of relationships between guardians and children.

```
-- Create GuardianRelationType table
CREATE TABLE GuardianRelationType (
    RelationshipTypeID INT PRIMARY KEY IDENTITY,
    Description NVARCHAR(50) NOT NULL
);
```

CREATE TABLE ChildGuardian: This query creates a junction table to establish a many-to-many relationship between Child and Guardian tables.

```
-- Create ChildGuardian table
CREATE TABLE ChildGuardian (
    ChildID INT FOREIGN KEY REFERENCES Child(ChildID),
    GuardianID INT FOREIGN KEY REFERENCES Guardian(GuardianID),
    GuardianRelationType INT NOT NULL FOREIGN KEY REFERENCES GuardianRelationType(RelationshipTypeID)
,
    PRIMARY KEY (ChildID, GuardianID)
);
```

CREATE TABLE ReportTypes: This query creates a table named "ReportTypes" to store different types of reports.

```
-- Create ReportTypes table
CREATE TABLE ReportTypes (
    ReportTypeID INT PRIMARY KEY IDENTITY,
    Type NVARCHAR(255) NOT NULL,
    Description NVARCHAR(1000),
    Cost MONEY NOT NULL CONSTRAINT CHK_ReportTypes_Cost_NonNegative CHECK (Cost >= 0)
);
```

CREATE TABLE ReportStatus: This query creates a table named "ReportStatus" to store statuses of reports.

CREATE TABLE Report: This query creates a table named "Report" to store reports related to clients. It includes columns for ClientID, ReportTypeID, ReportStatusID, and Notes.

```
-- Create Report table
CREATE TABLE Report (
    ClientID INT FOREIGN KEY REFERENCES Client(ClientID),
    ReportTypeID INT FOREIGN KEY REFERENCES ReportTypes(ReportTypeID),
    ReportStatusID INT NOT NULL FOREIGN KEY REFERENCES ReportStatus(ReportStatusID),
    Notes NVARCHAR(MAX),
    PRIMARY KEY (ClientID,ReportTypeID)
);
```

CREATE TABLE AccountStatus: This query creates a table named "AccountStatus" to store various statuses of accounts.

```
-- Create AccountStatus table
CREATE TABLE AccountStatus (
    AccountStatusID INT PRIMARY KEY IDENTITY,
    Description NVARCHAR(50) NOT NULL
);
```

CREATE TABLE Accounts: This query creates a table named "Accounts" to store account information related to clients.

```
-- Create Accounts table
CREATE TABLE Accounts (
    AccountID INT PRIMARY KEY IDENTITY,
    ClientID INT NOT NULL FOREIGN KEY REFERENCES Client(ClientID),
    PersonRFP INT NOT NULL FOREIGN KEY REFERENCES Client(ClientID),
    AccountStatusID INT FOREIGN KEY REFERENCES AccountStatus(AccountStatusID),
    AmountOutStanding MONEY
);
```

CREATE TABLE AccountDiscount: This query creates a table named "AccountDiscount" to store discounts applied to accounts.

```
-- Create AccountDiscount table
CREATE TABLE AccountDiscount (
    DiscountID INT PRIMARY KEY IDENTITY,
    AccountID INT NOT NULL FOREIGN KEY REFERENCES Accounts(AccountID),
    Amount MONEY CONSTRAINT CHK_AccountDiscount_Amount_NonNegative CHECK (Amount >= 0),
    Date DATE,
    Reason NVARCHAR(255)
);
```

CREATE TABLE ServiceType: This query creates a table named "ServiceType" to store different types of services offered.

CREATE TABLE ClientAppointment: This query creates a table named "ClientAppointment" to store appointments made by clients. It includes columns for AppointmentID, ClientID, AppointmentType, AppointmentDate, and PractitionerID.

```
-- Create ClientAppointment table
CREATE TABLE ClientAppointment  (
    AppointmentID INT PRIMARY KEY IDENTITY,
    ClientID INT  NOT NULL FOREIGN KEY REFERENCES Client(ClientID),
    AppointmentType INT  NOT NULL FOREIGN KEY REFERENCES ServiceType(ServiceTypeID),
    AppointmentDate  DateTime  NOT NULL,
    PractitionerID INT  NOT NULL FOREIGN KEY REFERENCES Practitioner(PractitionerID)
);
```

CREATE TABLE Invoice: This query creates a table named "Invoice" to store invoice information. It includes columns for InvoiceID, AccountID, InvoiceNum, InvoiceDate, InvoiceTotal, DiscountID, and AppointmentID.

```
-- Create Invoice table
CREATE TABLE Invoice (
    InvoiceID INT PRIMARY KEY IDENTITY,
    AccountID INT NOT NULL FOREIGN KEY REFERENCES Accounts(AccountID),
    InvoiceNum NVARCHAR(80) NOT NULL,
    InvoiceDate DATE  NOT NULL,
    InvoiceTotal MONEY  NOT NULL,
    DiscountID INT FOREIGN KEY REFERENCES AccountDiscount(DiscountID),
    AppointmentID INT  NOT NULL FOREIGN KEY REFERENCES ClientAppointment(AppointmentID)
);
```

CREATE TABLE InvoicePayments: This query creates a table named "InvoicePayments" to store payments made against invoices.

```
-- Create InvoicePayments table
CREATE TABLE InvoicePayments (
    PaymentID INT PRIMARY KEY IDENTITY,
    InvoiceID INT  NOT NULL FOREIGN KEY REFERENCES Invoice(InvoiceID),
    Amount MONEY CONSTRAINT CHK_InvoicePayments_Amount_NonNegative CHECK (Amount >= 0),
    Date DATE
);
```

COMMIT TRANSACTION: If everything is successful, this command commits the transaction, making all changes permanent.

```
COMMIT TRANSACTION; -- If everything is successful, commit the transaction
```

BEGIN CATCH... END CATCH: This block of code handles errors that may occur during the execution of the preceding queries. If an error occurs, it rolls back the transaction and raises an error message.

```
BEGIN CATCH
    -- If there's an error, roll back the transaction
    ROLLBACK TRANSACTION;

    -- Error handling: capture and rethrow the error
    DECLARE @ErrorMessage NVARCHAR(4000) = ERROR_MESSAGE();
    DECLARE @ErrorSeverity INT = ERROR_SEVERITY();
    DECLARE @ErrorState INT = ERROR_STATE();
    RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState);
END CATCH
```

DATA DICTIONARY FOR EACH CREATED TABLE

PRACTIONER TABLE

FIELD NAME	DATA TYPE	FIELD LENGTH	CONSTRAINT	DESCRIPTION
PractionerID	INT	30	PRIMARY KEY, IDENTITY,	
Name	NVARCHAR	30	NOT NULL	
Surname	NVARCHAR	30	NOT NULL	

QUALIFICATION TABLE

FIELD NAME	DATA TYPE	FIELD LENGTH	CONSTRAINT	DESCRIPTION
QualificationID	INT		PRIMARYKEY, IDENTITY	
Name	NVARCHAR	30	NOT NULL	
NQF Level	INT		NOT NULL, CONSTRAINT CHK_Qualification_NQFLevel CHECK (NQFLevel BETWEEN 1 AND 10)	

PRACTIONER QUALIFICATION TABLE

FIELD NAME	DATA TYPE	FIELD LENGTH	CONSTRAINT	DESCRIPTION
PractitionerID	INT	30	FOREIGN KEY REFERENCES Practitioner(PractitionerID)	
QualificationID	INT	30	FOREIGN KEY REFERENCES Qualification(QualificationID)	
Institution	NVARCHAR	80	NOT NULL	
Description	NVARCHAR	255		
YearObtained	SMALLINT		NOT NULL, PRIMARY KEY (PractitionerID, QualificationID)	

MEDICAL AID TABLE

FIELD NAME	DATA TYPE	FIELD LENGTH	CONSTRAINT	DESCRIPTION
MedicalAidID	INT	30	INT PRIMARY KEY IDENTITY	
MedicalAidNum	NVARCHAR	100	NOT NULL	
MedicalAidName	NVARCHAR	150	NOT NULL	

CLIENT TABLE

FIELD NAME	DATA TYPE	FIELD LENGTH	CONSTRAINT	DESCRIPTION
ClientID	INT	30	PRIMARY KEY IDENTITY,	
IDNum	VARCHAR	13	NOT NULL, CONSTRAINT CHK_SA_ID_Length CHECK (LEN(IDNum) = 13)	
Name	NVARCHAR	30	NOT NULL	
Surname	NVARCHAR	30	NOT NULL	
DateOfBirth	DATE		NOT NULL CONSTRAINT CHK_Client_DateOfBirth_Reasonable CHECK (DateOfBirth <= GETDATE() AND DateOfBirth > '1900-01-01'),	
MedicalAidID	INT	30	FOREIGN KEY REFERENCES MedicalAid(MedicalAidID)	
Phone	NVARCHAR	20	CHK_Client_Phone_Format CHECK (LEN(Phone) BETWEEN 10 AND 20)	
WhatsAppNumber	NVARCHAR	20	CONSTRAINT CHK_Client_WhatsAppNumber_Format CHECK (LEN(WhatsAppNumber) BETWEEN 10 AND 20)	
PostalAddress	NVARCHAR	255	NOT NULL	
Email	NVARCHAR	80	NOT NULL	

MEDICAL AID CLIENT TABLE

FIELD NAME	DATA TYPE	FIELD LENGTH	CONSTRAINT	DESCRIPTION
MedicalAidID	INT	30	NOT NULL FOREIGN KEY REFERENCES MedicalAid(MedicalAidID)	
ClientID	INT	30	NOT NULL FOREIGN KEY REFERENCES Client(ClientID)	
DepNum	INT	30	NOT NULL	

VISIT REASON TABLE

FIELD NAME	DATA TYPE	FIELD LENGTH	CONSTRAINT	DESCRIPTION
ReasonID	INT	30	PRIMARY KEY IDENTITY	
ReasonWhy	NVARCHAR	255	NOT NULL	
BackgroundInfo	NVARCHAR	1000	NOT NULL	

ADULT TABLE

FIELD NAME	DATA TYPE	FIELD LENGTH	CONSTRAINT	DESCRIPTION
AdultID	INT	30	PRIMARY KEY IDENTITY	
ClientID	INT	30	NOT NULL, FOREIGN KEY REFERENCES Client(ClientID)	
ContactPersonID	INT	30	NOT NULL, FOREIGN KEY REFERENCES Client(ClientID)	
Occupation	NVARCHAR	255	NOT NULL CONSTRAINT, CHK_Adult_Occupation_Length CHECK (LEN(Occupation) <= 255)	
RelationshipStatus	NVARCHAR	50	NOT NULL	
VisitReasonID	INT	30	FOREIGN KEY REFERENCES VisitReason(ReasonID)	
PersonRFP	INT	30	NOT NULL,FOREIGN KEY REFERENCES Client(ClientID)	

CHILD TABLE

FIELD NAME	DATA TYPE	FIELD LENGTH	CONSTRAINT	DESCRIPTION
ChildID	INT	30	NOT NULL	
ClientID	INT	30	FOREIGN KEY REFERENCES Client(ClientID)	
ContactPersonID	INT	30	NOT NULL FOREIGN KEY REFERENCES Client(ClientID)	
School	NVARCHAR	100	CONSTRAINT CHK_Child_School_Length CHECK (LEN(School) <= 100)	
VisitReasonID	INT	30	FOREIGN KEY REFERENCES VisitReason(ReasonID)	
ReportRequired	BIT	30	NOT NULL	
Grade	INT	30	NOT NULL	

RELATIONSHIP STATUS TABLE

FIELD NAME	DATA TYPE	FIELD LENGTH	CONSTRAINT	DESCRIPTION
RelationshipStatusID	INT	30	PRIMARY KEY IDENTITY	
Description	NVARCHAR	50	NOT NULL	

GUARDIAN TABLE

FIELD NAME	DATA TYPE	FIELD LENGTH	CONSTRAINT	DESCRIPTION
GuardianID	INT	30	PRIMARY KEY IDENTITY	
ClientID	INT	30	NOT NULL FOREIGN KEY REFERENCES Client(ClientID)	
RelationshipStatusID	INT	30	NOT NULL, FOREIGN KEY REFERENCES RelationshipStatus	
Occupation	NVARCHAR	30	NOT NULL	

GUARDIAN RELATION TYPE TABLE

FIELD NAME	DATA TYPE	FIELD LENGTH	CONSTRAINT	DESCRIPTION
RelationshipTypeID	INT	30	INT PRIMARY KEY IDENTITY	
Description	NVARCHAR	50	NOT NULL	

CHILD GUARDIAN TABLE

FIELD NAME	DATA TYPE	FIELD LENGTH	CONSTRAINT	DESCRIPTION
ChildID	INT	30	FOREIGN KEY REFERENCES Child(ChildID)	
GuardianID	INT	30	FOREIGN KEY REFERENCES Guardian(GuardianID)	
GuardianRelationType	INT	30	FOREIGN KEY REFERENCES GuardianRelationType(RelationshipTypeID)	

REPORT STATUS TABLE

FIELD NAME	DATA TYPE	FIELD LENGTH	CONSTRAINT	DESCRIPTION
ReportStatusID	INT	30	PRIMARY KEY IDENTITY	
Description	NVARCHAR	50	NOT NULL	

REPORT TABLE

FIELD NAME	DATA TYPE	FIELD LENGTH	CONSTRAINT	DESCRIPTION
ClientID	INT	30	Client(ClientID)	
ReportTypeID	INT	30	FOREIGN KEY REFERENCES ReportTypes(ReportTypeID	
ReportStatusID	INT	30	FOREIGN KEY REFERENCES ReportStatus(ReportStatusID)	
Notes	NVARCHAR	MAX	NOT NULL, FOREIGN KEY REFERENCES ReportStatus(ReportStatusID), Notes NVARCHAR(MAX)	

ACCOUNT STATUS TABLE

FIELD NAME	DATA TYPE	FIELD LENGTH	CONSTRAINT	DESCRIPTION
AccountStatusID	INT	30	PRIMARY KEY IDENTITY	
Description	NVARCHAR	50	NOT NULL	

ACCOUNTS TABLE

FIELD NAME	DATA TYPE	FIELD LENGTH	CONSTRAINT	DESCRIPTION
AccountID	INT	30	NOT NULL, PRIMARY KEY IDENTITY	
ClientID	INT	30	NOT NULL, FOREIGN KEY REFERENCES Client(ClientID)	
PersonRFP	INT	30	NOT NULL, FOREIGN KEY REFERENCES Client(ClientID)	
AccountStatusID	INT	30	NOT NULL, FOREIGN KEY REFERENCES AccountStatus(AccountStatusID)	
AmountOutStanding	MONEY	30	NOT NULL	

ACCOUNT DISCOUNT TABLE

FIELD NAME	DATA TYPE	FIELD LENGTH	CONSTRAINT	DESCRIPTION
DiscountID	INT	30	PRIMARY KEY IDENTITY	
AccountID	INT	30	NOT NULL FOREIGN KEY REFERENCES Accounts(AccountID)	
Amount	MONEY	30	CONSTRAINT CHK_AccountDiscount_Amount_NonNegative CHECK (Amount >= 0)	
Date	DATE	30	DATE	
Reason	NVARCHAR	255	NVARCHAR(255)	

SERVICE TABLE

FIELD NAME	DATA TYPE	FIELD LENGTH	CONSTRAINT	DESCRIPTION
ServiceTypeID	INT	30	PRIMARY KEY IDENTITY	
ServiceType	NVARCHAR	80	NOT NULL	
Description	NVARCHAR	150	NOT NULL	
ICD10	NVARCHAR	30	NOT NULL	
Cost	MONEY	30	NOT NULL CONSTRAINT CHK_Service_Cost_NonNegative CHECK (Cost >= 0)	

CLIENT APPOINTMENT TABLE

FIELD NAME	DATA TYPE	FIELD LENGTH	CONSTRAINT	DESCRIPTION
AppointmentID	INT	30	PRIMARY KEY IDENTITY	
ClientID	INT	30	NOT NULL FOREIGN KEY REFERENCES Client(ClientID)	
AppointmentType	INT	30	NOT NULL FOREIGN KEY REFERENCES ServiceType(ServiceTypeID)	
AppointmentDate	DateTime	30	NOT NULL	
PractitionerID	INT	30	NOT NULL FOREIGN KEY REFERENCES Practitioner(PractitionerID)	

INVOICE TABLE

FIELD NAME	DATA TYPE	FIELD LENGTH	CONSTRAINT	DESCRIPTION
InvoiceID	INT	30	PRIMARY KEY IDENTITY	
AccountID	INT	30	NOT NULL FOREIGN KEY REFERENCES Accounts(AccountID)	
InvoiceNum	NVARCHAR	30	NOT NULL	
InvoiceDate	DATE	30	NOT NULL	
InvoiceTotal	MONEY	30	NOT NULL	
DiscountID	INT	30	FOREIGN KEY REFERENCES AccountDiscount(DiscountID)	
AppointmentID	INT	30	NOT NULL FOREIGN KEY REFERENCES ClientAppointment(AppointmentID)	

INVOICE PAYMENTS TABLE

FIELD NAME	DATA TYPE	FIELD LENGTH	CONSTRAINT	DESCRIPTION
PaymentID	INT	30	PRIMARY KEY IDENTITY	
InvoiceID	INT	30	NOT NULL FOREIGN KEY REFERENCES Invoice(InvoiceID)	
Amount	MONEY	30	CONSTRAINT CHK_InvoicePayments_Amount_NonNegative CHECK (Amount >= 0)	
Date	DATE	30	NOT NULL	

USE OF TRIGGERS EXPLAINED

TRIGGER 1

```
CREATE TRIGGER trg_add_new_client
ON Client
AFTER INSERT
AS
BEGIN
    SET NOCOUNT ON;


    INSERT INTO Client (IDNum, Name, Surname, DateOfBirth, MedicalAidID, Phone, WhatsAppNumber,
PostalAddress, Email)
    SELECT
        i.IDNum,
        i.Name,
        i.Surname,
        i.DateOfBirth,
        i.MedicalAidID,
        i.Phone,
        i.WhatsAppNumber,
        i.PostalAddress,
        i.Email
    FROM
        inserted i;
END;
```

This SQL query creates a trigger named trg_add_new_client on the Client table.

- **ON Client AFTER INSERT:** This specifies that the trigger will activate after new rows are inserted into the Client table.
- **AS BEGIN:** This marks the beginning of the trigger's definition.
- **SET NOCOUNT ON:** This statement is used to suppress the "xx rows affected" message that SQL Server sends as a result of DML (Data Manipulation Language) statements such as INSERT, **UPDATE, or DELETE**. It's a best practice to include this statement in triggers to avoid confusion when executing batches of SQL statements.
- **INSERT INTO Client:** This is an INSERT statement that inserts data into the Client table.
- **SELECT FROM inserted**: The inserted table is a special table in SQL Server that holds copies of the rows that were affected by the DML operation (in this case, INSERT) that fired the trigger. Here, the trigger is selecting data from the inserted table to insert into the Client table.
- i.IDNum, i.Name, i.Surname, i.DateOfBirth, i.MedicalAidID, i.Phone, i.WhatsAppNumber, i.PostalAddress, i.Email: These are the columns selected from the inserted table to be inserted into the Client table.

FROM inserted i: This specifies that data should be selected from the inserted table and assigns it an alias i.

END: This marks the end of the trigger's definition.

This trigger ensures that whenever a new row is inserted into the Client table, a copy of that row's data is also inserted back into the Client table.

TRIGGER 2

```
CREATE TRIGGER trgValidateSAID
ON Person
AFTER INSERT, UPDATE
AS
BEGIN
    DECLARE @SA_ID VARCHAR(13);
    DECLARE @CheckDigit INT;
    DECLARE @i INT;
    DECLARE @Multiplier INT;
    DECLARE @Sum INT = 0;
    DECLARE @NextDigit INT;

    -- Assuming the ID is in the inserted pseudo table
    SELECT @SA_ID = SA_ID FROM inserted;

    -- Loop through each digit except the last one
    SET @i = 1;
    WHILE @i <= 12
    BEGIN
        -- Alternate between multiplying by 1 and 2
        SET @Multiplier = 2 - (@i % 2);
        SET @NextDigit = CAST(SUBSTRING(@SA_ID, @i, 1) AS INT) * @Multiplier;

        -- If the product is greater than 9, subtract 9 from it
        IF @NextDigit > 9 SET @NextDigit = @NextDigit - 9;

        -- Add the result to the sum
        SET @Sum = @Sum + @NextDigit;
        SET @i = @i + 1;
    END
```

```
    -- Calculate the check digit
    SET @CheckDigit = 10 - (@Sum % 10);
    IF @CheckDigit = 10 SET @CheckDigit = 0;

    -- Validate the check digit
    IF @CheckDigit != CAST(SUBSTRING(@SA_ID, 13, 1) AS INT)
    BEGIN
        RAISERROR ('Invalid SA ID number.', 16, 1);
        ROLLBACK TRANSACTION;
    END
END;
GO

INSERT INTO GuardianRelationType (Description) VALUES
('Parent'),
('Step-Parent'),
('Grandparent'),
('Sibling'),
('Aunt/Uncle'),
('Cousin'),
('Legal Guardian'),
('Foster Parent'),
('Godparent'),
('Family Friend'),
('Other');
Go
```

Trigger Definition (trgValidateSAID):

- **ON Person AFTER INSERT, UPDATE:** This trigger fires after rows are inserted into or updated in the Person table.
- **BEGIN:** This marks the beginning of the trigger's logic.
- Variable Declarations: Several variables are declared to store intermediate results during the validation process.
- **SELECT @SA_ID = SA_ID FROM inserted:** This statement selects the value of the SA_ID column from the inserted pseudo table. It assumes that the South African ID number (SA_ID) is being inserted or updated.
- **Loop through each digit:** This loop iterates through each digit of the SA ID number except the last one.
- **Multiplication and Summation:** Each digit is multiplied by 1 or 2 alternately, based on its position. The resulting values are summed up.
- **Calculate the check digit:** The script calculates the check digit using a formula based on the sum obtained from the previous step.
- **Validation:** The calculated check digit is compared to the last digit of the SA ID number. If they do not match, an error is raised, indicating an invalid SA ID number.
- **ROLLBACK TRANSACTION:** If the SA ID number is invalid, the transaction is rolled back to ensure that the invalid data is not committed to the database.
- **END:** This marks the end of the trigger's logic.
- **INSERT INTO GuardianRelationType:** This INSERT statement adds records to the GuardianRelationType table, specifying various descriptions of guardian relationships.

USE AND EXPLANATION OF STORED PROCEDURES

```
CREATE PROCEDURE UPDATECLIENT @ClientPhone NVARCHAR(20),@WApp NVARCHAR(20),@PostalAddress NVARCHAR(255), @ClientID INT
AS
Update Client
SET Phone = @ClientPhone , WhatsAppNumber = @WApp, PostalAddress = @PostalAddress
WHERE ClientID = @ClientID
GO

CREATE PROCEDURE ValidateEmail
    @Email NVARCHAR(320), -- The maximum length of an email address is 320 characters
    @IsValid BIT OUTPUT
AS
BEGIN
    SET @IsValid = 0 -- Default to invalid

    IF @Email LIKE '%@%.%' AND -- Contains @ and .
       @Email NOT LIKE '%@%@%' AND -- Only one @ character
       @Email NOT LIKE '%.@%' AND -- Dot not directly after @
       @Email NOT LIKE '%@.%' AND -- Dot not directly before @
       @Email NOT LIKE '%..%' AND -- No consecutive dots
       CHARINDEX('.', REVERSE(@Email)) > 2 AND -- Domain part has at least two characters after the last dot
       LEFT(@Email, 1) NOT LIKE '[.@]' AND -- Does not start with @ or .
       RIGHT(@Email, 1) NOT LIKE '[.@]' -- Does not end with @ or .
    BEGIN
        SET @IsValid = 1 -- Email is valid based on the pattern match
    END
END
GO
```

UPDATECLIENT Procedure:

This procedure is responsible for updating the phone number (Phone), WhatsApp number (WhatsAppNumber), and postal address (PostalAddress) of a client in the Client table based on the provided ClientID.

Parameters:

- @ClientPhone: New phone number of the client.
- @WApp: New WhatsApp number of the client.
- @PostalAddress: New postal address of the client.
- @ClientID: ID of the client whose information is being updated.

Logic:

The procedure updates the Phone, WhatsAppNumber, and PostalAddress fields of the Client table for the client identified by @ClientID.

It sets the values of these fields to the corresponding parameters passed to the procedure

Validate Email Procedure:

This procedure validates an email address provided as input against a set of conditions and outputs a bit flag indicating whether the email is valid or not.

Parameters:

- @Email: Email address to be validated.
- @IsValid: Output parameter that indicates whether the email is valid (1) or not (0).

Logic:

- The procedure sets @IsValid to 0 by default, indicating that the email is invalid.
- It then checks the provided email address against a series of conditions:
- Ensure the email contains an "@" character followed by a dot "." (%@%.%).
- Make sure there is only one "@" character (%@%@%).
- Ensures that the dot "." is not directly after "@" (%.@%) or directly before "@" (%@.%).
- Checks for consecutive dots (%..%).
- Verifies that the domain part has at least two characters after the last dot.
- Confirms that the email address doesn't start or end with "@" or ".".
- If the email address meets all these conditions, @IsValid is set to 1, indicating that the email is valid.

Cursors

1. This cursor aims to generate a report for clients with outstanding invoices. It uses a cursor to loop through each client meeting the specified criteria and executes the GenerateClientReport stored procedure for each client. The GenerateClientReport procedure fetches relevant information about the client and their outstanding invoices.

```
DECLARE @ClientID INT;
DECLARE @ClientName NVARCHAR(60);

-- Adjust the cursor to only select clients with outstanding invoices.
DECLARE ReportCursor CURSOR FOR
    SELECT DISTINCT c.ClientID, c.Name + ' ' + c.Surname
    FROM Client c
    JOIN Accounts a ON c.ClientID = a.ClientID
    WHERE EXISTS (
        SELECT 1
        FROM Invoice i
        WHERE i.AccountID = a.AccountID
          AND i.InvoiceTotal > 0 -- Assumes InvoiceTotal > 0 indicates an outstanding invoice
          AND a.AmountOutStanding > 0 -- Ensures there's an outstanding amount in the account
    );

OPEN ReportCursor;

FETCH NEXT FROM ReportCursor INTO @ClientID, @ClientName;

WHILE @@FETCH_STATUS = 0
BEGIN
    -- This PRINT statement will now only include clients with outstanding invoices.
    PRINT 'Generating report for Client: ' + @ClientName;

    -- Execute the stored procedure for the current client.
    EXEC GenerateClientReport @ClientID;

    FETCH NEXT FROM ReportCursor INTO @ClientID, @ClientName;
END

CLOSE ReportCursor;
DEALLOCATE ReportCursor;
GO

CREATE OR ALTER PROCEDURE GenerateClientReport
    @ClientID INT
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @SQL NVARCHAR(MAX);

    -- Modified the WHERE clause to include a check for positive outstanding amounts.
    SET @SQL = N'
        SELECT
            c.Name AS ClientName,
            c.Surname AS ClientSurname,
            i.InvoiceDate AS InvoiceDate,
            i.InvoiceTotal AS InvoiceTotal
        FROM
            Client c
        INNER JOIN
            Accounts a ON c.ClientID = a.ClientID AND a.AmountOutStanding > 0
        LEFT JOIN
            Invoice i ON a.AccountID = i.AccountID
        WHERE
            c.ClientID = @ClientID
            AND EXISTS ( -- Added to ensure only clients with at least one outstanding invoice are included
                SELECT 1
                FROM Invoice iv
                WHERE iv.AccountID = a.AccountID
                AND iv.InvoiceTotal > 0 -- Assumes InvoiceTotal > 0 indicates outstanding
            );
    ';

    EXEC sp_executesql @SQL, N'@ClientID INT', @ClientID;
END;
```

2. The provided SQL script is using cursors to iterate through practitioners and their respective client appointments. It prints out the practitioner's name followed by the names of clients and their appointment dates.

```
DECLARE @PractitionerID INT;
DECLARE @PractitionerName NVARCHAR(60);
DECLARE @ClientName NVARCHAR(60);
DECLARE @AppointmentDate DATETIME; -- Declare variable for AppointmentDate

-- Cursor to iterate through each practitioner that has appointments
DECLARE PractitionerCursor CURSOR FOR
    SELECT DISTINCT p.PractitionerID, p.Name + ' ' + p.Surname
    FROM Practitioner p
    JOIN ClientAppointment ca ON p.PractitionerID = ca.PractitionerID;

OPEN PractitionerCursor;
FETCH NEXT FROM PractitionerCursor INTO @PractitionerID, @PractitionerName;

WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT 'Practitioner: ' + @PractitionerName; -- Print the practitioner's name once

    -- Sub-cursor to fetch and print names of clients and their appointment dates for the current practitioner
    DECLARE ClientCursor CURSOR FOR
        SELECT c.Name + ' ' + c.Surname AS ClientName, ca.AppointmentDate
        FROM ClientAppointment ca
        INNER JOIN Client c ON ca.ClientID = c.ClientID
        WHERE ca.PractitionerID = @PractitionerID
        ORDER BY ca.AppointmentDate; -- Added ordering to sort appointments chronologically

    OPEN ClientCursor;
    FETCH NEXT FROM ClientCursor INTO @ClientName, @AppointmentDate;

    WHILE @@FETCH_STATUS = 0
    BEGIN
        -- Print each client's name along with the appointment date
        PRINT ' - Client: ' + @ClientName + ', Appointment Date: ' + CONVERT(NVARCHAR, @AppointmentDate, 120);
        FETCH NEXT FROM ClientCursor INTO @ClientName, @AppointmentDate;
    END

    CLOSE ClientCursor;
    DEALLOCATE ClientCursor;

    FETCH NEXT FROM PractitionerCursor INTO @PractitionerID, @PractitionerName;
        PRINT ' ';
END

CLOSE PractitionerCursor;
DEALLOCATE PractitionerCursor;
```

3. This SQL script retrieves information about practitioners and their qualifications from the Practitioner and PractitionerQualification tables, respectively. It uses nested cursors to iterate over practitioners and their qualifications, printing the practitioner's name, qualifications, institutions, and years obtained.

```
USE DrJKnoetze
GO

DECLARE @PractitionerID INT;
DECLARE @PractitionerName NVARCHAR(60);

-- Outer Cursor: Iterate over each Practitioner
DECLARE PractitionerCursor CURSOR FOR
    SELECT PractitionerID, Name
    FROM Practitioner;

OPEN PractitionerCursor;
FETCH NEXT FROM PractitionerCursor INTO @PractitionerID, @PractitionerName;

WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT 'Practitioner : '+ @PractitionerName;
    PRINT 'Qualifications: ';

    DECLARE @Name NVARCHAR(30), @Institution NVARCHAR(80), @YearObtained SMALLINT;

    -- Inner Cursor: Iterate over Qualifications for the current Practitioner
    DECLARE QualificationCursor CURSOR FOR
        SELECT Name, pq.Institution, pq.YearObtained
        FROM Qualification q
        INNER JOIN PractitionerQualification pq ON q.QualificationID = pq.QualificationID
        WHERE pq.PractitionerID = @PractitionerID
        ORDER BY pq.YearObtained;

    OPEN QualificationCursor;
    FETCH NEXT FROM QualificationCursor INTO @Name, @Institution, @YearObtained;

    WHILE @@FETCH_STATUS = 0
    BEGIN
        PRINT '- '+ @Name + '| ' + @Institution + ' | ' + CAST(@YearObtained AS NVARCHAR(5));
        FETCH NEXT FROM QualificationCursor INTO @Name, @Institution, @YearObtained;
    END

    CLOSE QualificationCursor;
    DEALLOCATE QualificationCursor;

    FETCH NEXT FROM PractitionerCursor INTO @PractitionerID, @PractitionerName;
        PRINT ' ';
END

CLOSE PractitionerCursor;
DEALLOCATE PractitionerCursor;
```

4. A cursor is used to iterate through clients with positive total outstanding amounts, retrieving their client ID, name, and total outstanding balance. Here are some suggestions for improving the script:

```
DECLARE @ClientID INT, @ClientName NVARCHAR(255), @TotalOutstanding MONEY;

-- Modified cursor to include a join and SUM directly in the cursor's SELECT statement,
-- filtering clients with a positive total outstanding amount.
DECLARE ClientCursor CURSOR FOR
    SELECT a.ClientID, c.Name, SUM(a.AmountOutStanding) AS TotalOutstanding
    FROM Accounts a
    JOIN Client c ON a.ClientID = c.ClientID
    GROUP BY a.ClientID, c.Name
    HAVING SUM(a.AmountOutStanding) > 0;

OPEN ClientCursor;

FETCH NEXT FROM ClientCursor INTO @ClientID, @ClientName, @TotalOutstanding;

WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT 'Client: ' + CAST(@ClientID AS NVARCHAR(10)) + ' | ' + @ClientName + ', Total Outstanding: ' + CAST(@TotalOutstanding AS NVARCHAR(20));

    FETCH NEXT FROM ClientCursor INTO @ClientID, @ClientName, @TotalOutstanding;
END

CLOSE ClientCursor;
DEALLOCATE ClientCursor;
```

Create Views

1. 2. A view in a relational database is a virtual table that contains the result set of a stored query. The purpose of this view seems to be to consolidate invoice details along with client information and total payments.

Here's a breakdown of the script:

- The view selects several columns from different tables and calculates the total payments made for each invoice.
- It selects the **InvoiceID**, **ClientName**, **ClientSurname**, **InvoiceNum**, **InvoiceDate**, **InvoiceTotal**, and calculates the **TotalPayments**.
- The tables involved in the query are **Invoice**, **Accounts**, **Client**, and **InvoicePayments**.
- It joins the **Invoice** table with the **Accounts** table on the **AccountID** column and then joins the **Accounts** table with the **Client** table on the **ClientID** column.
- It performs a left join between the **Invoice** table and the **InvoicePayments** table on the **InvoiceID** column to get the payment amounts for each invoice.
- The **GROUP BY** clause is used to group the results by **InvoiceID**, **ClientName**, **ClientSurname**, **InvoiceNum**, **InvoiceDate**, and **InvoiceTotal**.
- The **ISNULL** function is used to handle cases where there are no payments for an invoice, returning 0 instead of NULL for **TotalPayments**.

```
CREATE VIEW InvoiceDetailsView AS
SELECT
    i.InvoiceID,
    c.Name AS ClientName,
    c.Surname AS ClientSurname,
    i.InvoiceNum,
    i.InvoiceDate,
    i.InvoiceTotal,
    ISNULL(SUM(ip.Amount), 0) AS TotalPayments
FROM
    Receptionist.Invoice i
JOIN
    Receptionist.Accounts a ON i.AccountID = a.AccountID
JOIN
    Receptionist.Client c ON a.ClientID = c.ClientID
LEFT JOIN
    Receptionist.InvoicePayments ip ON i.InvoiceID = ip.InvoiceID
GROUP BY
    i.InvoiceID, c.Name, c.Surname, i.InvoiceNum, i.InvoiceDate, i.InvoiceTotal;
GO
```

2. This SQL script creates or alters a view named **Receptionist.ClientWithGuardiansView**. The purpose of this view seems to be to provide a list of child clients along with their associated guardians and their relationship.

    a. The view selects several columns from different tables to construct a view that includes child names, guardian names, and their relationship.
    b. It selects **Name** and **Surname** columns for both child and guardian, as well as **Description** for the relationship type.

c. The tables involved in the query are **Client**, **ChildGuardian**, **Guardian**, **RelationshipStatus**, and **GuardianRelationType**.
d. It joins the **Client** table with the **ChildGuardian** table on the **ClientID** column to link child clients with their guardians.
e. It then joins the **Guardian** table on **GuardianID** to retrieve guardian details.
f. Another join with the **Client** table is performed to get the names of the guardians.
g. Joins with **RelationshipStatus** and **GuardianRelationType** tables are used to get descriptive information about the relationship between child and guardian.
h. The result set provides a list of child names along with their associated guardian names and their relationship types.

```
CREATE OR ALTER VIEW Receptionist.ClientWithGuardiansView AS
SELECT
    c.Name AS ChildtName,
    c.Surname AS ChildSurname,
    cli.Name AS GuardianName,
    cli.Surname AS GuardianSurname,
    grt.Description AS GuardianRelationship
FROM
    Receptionist.Client c
JOIN
    ChildGuardian cg ON c.ClientID = cg.ChildID
JOIN
    Receptionist.Guardian g ON cg.GuardianID = g.GuardianID
JOIN
    Receptionist.Client cli ON g.ClientID = cli.ClientID
JOIN
    RelationshipStatus rs ON g.RelationshipStatusID = rs.RelationshipStatusID
JOIN
    GuardianRelationType grt ON cg.GuardianRelationType = grt.RelationshipTypeID;
GO
```

3. This SQL script creates a view named **ClientAccountSummaryView**. The purpose of this view seems to be to provide a summary of outstanding amounts for each client.

- The view selects several columns from the **Client** table and calculates the total outstanding amount for each client.
- It selects **ClientID**, **Name**, and **Surname** columns from the **Client** table.
- It calculates the **TotalOutstandingAmount** by summing the **AmountOutstanding** column from the **Accounts** table. The **ISNULL** function is used to handle cases where there are no outstanding amounts, returning 0 instead of NULL.
- The tables involved in the query are **Client** and **Accounts**.
- It performs a left join between the **Client** table and the **Accounts** table on the **ClientID** column to include all clients, even those who may not have any outstanding amounts.
- The **GROUP BY** clause is used to group the results by **ClientID**, **Name**, and **Surname**.

```
CREATE VIEW ClientAccountSummaryView AS
SELECT
    c.ClientID,
    c.Name,
    c.Surname,
    ISNULL(SUM(a.AmountOutstanding), 0) AS TotalOutstandingAmount
FROM
    Receptionist.Client c
LEFT JOIN
    Receptionist.Accounts a ON c.ClientID = a.ClientID
GROUP BY
    c.ClientID, c.Name, c.Surname;
GO
```

Stored Procedures

The purpose of this stored procedure is to update the phone number of a client identified by their **ClientID**.

- The stored procedure takes two parameters: **@ClientID** of type **INT** to identify the client, and **@NewPhoneNumber** of type **NVARCHAR(20)** to specify the new phone number.
- **SET NOCOUNT ON** is used to prevent the count of the number of rows affected by a Transact-SQL statement from being returned as part of the result set.
- Inside the procedure, a dynamic SQL statement is constructed and stored in the variable **@SQL**. Dynamic SQL is used here to allow the parameterized execution of the update statement.
- The dynamic SQL statement is an **UPDATE** statement that updates the **Phone** column in the **Client** table with the new phone number provided, for the client identified by **ClientID**.
- The **sp_executesql** system stored procedure is used to execute the dynamic SQL statement. It allows for parameterized execution of SQL commands. Parameters **@ClientID** and **@NewPhoneNumber** are passed to the dynamic SQL statement using the **@params** parameter.

```
CREATE OR ALTER PROCEDURE UpdateClientPhoneNumber
    @ClientID INT,
    @NewPhoneNumber NVARCHAR(20)
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @SQL NVARCHAR(MAX);

    SET @SQL = '
        UPDATE Receptionist.Client
        SET Phone = @NewPhoneNumber
        WHERE ClientID = @ClientID;
    ';

    EXEC sp_executesql @SQL, N'
        @ClientID INT,
        @NewPhoneNumber NVARCHAR(20)',
        @ClientID, @NewPhoneNumber;
END;
GO
```

2. The purpose of this stored procedure is to generate a report of outstanding invoices for a specific client identified by their **ClientID**.

Here's a breakdown of the script:

- The stored procedure takes one parameter: **@ClientID** of type **INT** to identify the client for whom the report is generated.
- **SET NOCOUNT ON** is used to prevent the count of the number of rows affected by a Transact-SQL statement from being returned as part of the result set.
- Inside the procedure, a dynamic SQL statement is constructed and stored in the variable **@SQL**. Dynamic SQL is used here to allow the parameterized execution of the select statement.

- The dynamic SQL statement is a **SELECT** statement that retrieves the client's name, surname, invoice date, and invoice total for outstanding invoices. It joins the **Client** table with the **Accounts** table and the **Invoice** table.
- A condition is added to the join with the **Accounts** table to include only rows with positive outstanding amounts.
- A condition is also added in the **WHERE** clause to ensure that only clients with at least one outstanding invoice are included in the report.
- The **sp_executesql** system stored procedure is used to execute the dynamic SQL statement. It allows for parameterized execution of SQL commands. Parameter **@ClientID** is passed to the dynamic SQL statement using the **@params** parameter.

```
CREATE OR ALTER PROCEDURE GenerateClientReport
    @ClientID INT
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @SQL NVARCHAR(MAX);

    -- Modified the WHERE clause to include a check for positive outstanding amounts.
    SET @SQL = N'
        SELECT
            c.Name AS ClientName,
            c.Surname AS ClientSurname,
            i.InvoiceDate AS InvoiceDate,
            i.InvoiceTotal AS InvoiceTotal
        FROM
            Receptionist.Client c
        INNER JOIN
            Receptionist.Accounts a ON c.ClientID = a.ClientID AND a.AmountOutStanding > 0
        LEFT JOIN
            Receptionist.Invoice i ON a.AccountID = i.AccountID
        WHERE
            c.ClientID = @ClientID
            AND EXISTS ( -- Added to ensure only clients with at least one outstanding invoice are included
                SELECT 1
                FROM Receptionist.Invoice iv
                WHERE iv.AccountID = a.AccountID
                AND iv.InvoiceTotal > 0 -- Assumes InvoiceTotal > 0 indicates outstanding
            );
    ';

    EXEC sp_executesql @SQL, N'@ClientID INT', @ClientID;
END;
GO
```

Data Queries

Query 1

This query retrieves information about invoices, including the client's name and surname, invoice number, date, total amount, and the total payments made for each invoice. It ensures that all invoices are returned, even if there are no payments associated with them. The GROUP BY clause ensures that the aggregation functions are applied correctly for each invoice.

```
SELECT
    i.InvoiceID,
    c.Name AS ClientName,
    c.Surname AS ClientSurname,
    i.InvoiceNum,
    i.InvoiceDate,
    i.InvoiceTotal,
    ISNULL(SUM(ip.Amount), 0) AS TotalPayments
FROM
    Receptionist.Invoice i
JOIN
    Receptionist.Accounts a ON i.AccountID = a.AccountID
JOIN
    Client c ON a.ClientID = c.ClientID
LEFT JOIN
    Receptionist.InvoicePayments ip ON i.InvoiceID = ip.InvoiceID
GROUP BY
    i.InvoiceID, c.Name, c.Surname, i.InvoiceNum, i.InvoiceDate, i.InvoiceTotal;
```

SELECT statement:

- i.InvoiceID: Selecting the InvoiceID from the Invoice table.
- c.Name AS ClientName, c.Surname AS ClientSurname: Selecting the client's first name and last name from the Client table.
- i.InvoiceNum: Selecting the invoice number from the Invoice table.
- i.InvoiceDate: Selecting the invoice date from the Invoice table.
- i.InvoiceTotal: Selecting the total amount of the invoice from the Invoice table.
- ISNULL(SUM(ip.Amount), 0) AS TotalPayments: Calculating the total payments made for each invoice. It sums up the payment amounts from the InvoicePayments table (aliased as ip). The ISNULL function is used to handle cases where there are no payments for an invoice. If there are no payments, it returns 0.

FROM clause:

- Receptionist.Invoice i: This selects data from the Invoice table, aliased as i.
- Receptionist.Accounts a: This selects data from the Accounts table, aliased as a.
- Client c: This selects data from the Client table, aliased as c.
- Receptionist.InvoicePayments ip: This selects data from the InvoicePayments table, aliased as ip.

JOIN clauses:

- JOIN Receptionist.Accounts a ON i.AccountID = a.AccountID: Joins the Invoice table (i) with the Accounts table (a) based on the AccountID foreign key relationship.
- JOIN Client c ON a.ClientID = c.ClientID: Joins the Accounts table (a) with the Client table (c) based on the ClientID foreign key relationship.
- LEFT JOIN Receptionist.InvoicePayments ip ON i.InvoiceID = ip.InvoiceID: This is a left join between the Invoice table (i) and the InvoicePayments table (ip) based on the InvoiceID foreign key relationship. This ensures that all rows from the Invoice table are included in the result set, even if there are no corresponding rows in the InvoicePayments table.

GROUP BY clause:

- GROUP BY i.InvoiceID, c.Name, c.Surname, i.InvoiceNum, i.InvoiceDate, i.InvoiceTotal: Groups the result set by InvoiceID, ClientName, ClientSurname, InvoiceNum, InvoiceDate, and InvoiceTotal. This means that the aggregation functions (such as SUM) are applied for each unique combination of these columns.

Query 2

This query fetches information about children, their associated guardians, and the relationship type between them. It links data from multiple tables using various foreign key relationships and joins, then sorts the result set by child ID and guardian ID.

```
SELECT
    ch.ChildID,
    cli.Name AS ChildName,
    cli.Surname AS ChildSurname,
    cli2.Name AS GuardianName,
    cli2.Surname AS GuardianSurname,

    grt.Description AS GuardianRelationshipType
FROM
    Receptionist.Child ch
INNER JOIN
    Client cli ON ch.ClientID = cli.ClientID
INNER JOIN
    ChildGuardian cg ON ch.ChildID = cg.ChildID
INNER JOIN
    Receptionist.Guardian gu ON cg.GuardianID = gu.GuardianID
INNER JOIN
    Client cli2 ON gu.ClientID = cli2.ClientID
INNER JOIN
    GuardianRelationType grt ON cg.GuardianRelationType = grt.RelationshipTypeID
ORDER BY
    ch.ChildID, gu.GuardianID;
```

SELECT statement:

- ch.ChildID: Selecting the ChildID from the Child table.
- cli.Name AS ChildName, cli.Surname AS ChildSurname: Selecting the child's first name and last name from the Client table, aliased as cli.
- cli2.Name AS GuardianName, cli2.Surname AS GuardianSurname: Selecting the guardian's first name and last name from the Client table, aliased as cli2.
- grt.Description AS GuardianRelationshipType: Selecting the description of the guardian relationship type from the GuardianRelationType table, aliased as grt.

FROM clause:

- Receptionist.Child ch: This selects data from the Child table, aliased as ch.
- Client cli: This selects data from the Client table, aliased as cli, representing information about children.
- ChildGuardian cg: This selects data from the ChildGuardian table, aliased as cg, representing the relationship between children and guardians.
- Receptionist.Guardian gu: This selects data from the Guardian table, aliased as gu, representing information about guardians.
- Client cli2: This selects data from the Client table again, aliased as cli2, representing information about guardians.
- GuardianRelationType grt: This selects data from the GuardianRelationType table, aliased as grt, representing the types of relationships between children and guardians.

JOIN clauses:

- INNER JOIN Client cli ON ch.ClientID = cli.ClientID: Joins the Child table (ch) with the Client table (cli) based on the ClientID foreign key relationship. This links children to their respective client information.
- INNER JOIN ChildGuardian cg ON ch.ChildID = cg.ChildID: Joins the Child table (ch) with the ChildGuardian table (cg) based on the ChildID foreign key relationship. This establishes the relationship between children and guardians.
- INNER JOIN Receptionist.Guardian gu ON cg.GuardianID = gu.GuardianID: Joins the ChildGuardian table (cg) with the Guardian table (gu) based on the GuardianID foreign key relationship. This links guardians to their respective child-guardian relationships.
- INNER JOIN Client cli2 ON gu.ClientID = cli2.ClientID: Joins the Guardian table (gu) with the Client table (cli2) based on the ClientID foreign key relationship. This links guardians to their respective client information.
- INNER JOIN GuardianRelationType grt ON cg.GuardianRelationType = grt.RelationshipTypeID: Joins the ChildGuardian table (cg) with the GuardianRelationType table (grt) based on the GuardianRelationType foreign key relationship. This associates the type of relationship between a child and a guardian.

ORDER BY clause:

- ORDER BY ch.ChildID, gu.GuardianID: Orders the result set first by ChildID in ascending order, then by GuardianID in ascending order.

Query 3

In summary, this query retrieves information about practitioners, including their ID, full name, and the number of appointments they have. It ensures that only practitioners with at least one appointment are included in the result set and orders them based on the number of appointments they have, with the highest number of appointments first.

```
SELECT
    p.PractitionerID,
    p.Name + ' ' + p.Surname AS PractitionerName,
    COUNT(ca.AppointmentID) AS NumberOfAppointments
FROM
    Doctor.Practitioner p
INNER JOIN
    Receptionist.ClientAppointment ca ON p.PractitionerID = ca.PractitionerID
GROUP BY
    p.PractitionerID,
    p.Name,
    p.Surname
HAVING
    COUNT(ca.AppointmentID) > 0
ORDER BY
    NumberOfAppointments DESC;
```

SELECT statement:

- p.PractitionerID: Selects the PractitionerID from the Practitioner table.
- p.Name + ' ' + p.Surname AS PractitionerName: Concatenates the practitioner's first name and last name with a space in between to create the full name. This is aliased as PractitionerName.
- COUNT(ca.AppointmentID) AS NumberOfAppointments: Counts the number of appointments associated with each practitioner. This is aliased as NumberOfAppointments.

FROM clause:

- Doctor.Practitioner p: This selects data from the Practitioner table, aliased as p, which likely contains information about healthcare professionals.

- Receptionist.ClientAppointment ca: This selects data from the ClientAppointment table, aliased as ca, which likely stores information about appointments.

JOIN clause:

- INNER JOIN Receptionist.ClientAppointment ca ON p.PractitionerID = ca.PractitionerID: Joins the Practitioner table (p) with the ClientAppointment table (ca) based on the PractitionerID foreign key relationship. This links practitioners to their respective appointments.

GROUP BY clause:

- GROUP BY p.PractitionerID, p.Name, p.Surname: Groups the result set by PractitionerID, Name, and Surname. This means that the aggregation function (COUNT) will be applied for each unique combination of these columns.

HAVING clause:

- HAVING COUNT(ca.AppointmentID) > 0: Filters the grouped results to include only those practitioners who have at least one appointment. This ensures that only practitioners with appointments are included in the final result set.

ORDER BY clause:

- ORDER BY NumberOfAppointments DESC: Orders the result set by the number of appointments (NumberOfAppointments) in descending order. This means that practitioners with the highest number of appointments will appear first in the result set.

Query 4

This query retrieves detailed information about client appointments, including the client's name, the type of appointment, appointment date, and the practitioner's name. It joins data from multiple tables using foreign key relationships and sorts the result set based on the appointment date.

```
SELECT
    ca.AppointmentID,
    c.Name AS ClientName,
    c.Surname AS ClientSurname,
    s.ServiceType AS AppointmentType,
    ca.AppointmentDate,
    p.Name AS PractitionerName,
    p.Surname AS PractitionerSurname
FROM
    Receptionist.ClientAppointment ca
JOIN
    Client c ON ca.ClientID = c.ClientID
JOIN
    ServiceType s ON ca.AppointmentType = s.ServiceTypeID
JOIN
    Doctor.Practitioner p ON ca.PractitionerID = p.PractitionerID
ORDER BY AppointmentDate ASC;
```

SELECT statement:

- ca.AppointmentID: Selects the AppointmentID from the ClientAppointment table.
- c.Name AS ClientName, c.Surname AS ClientSurname: Selects the client's first name and last name from the Client table, aliased as c.
- s.ServiceType AS AppointmentType: Selects the service type of the appointment from the ServiceType table, aliased as s.
- ca.AppointmentDate: Selects the appointment date from the ClientAppointment table.
- p.Name AS PractitionerName, p.Surname AS PractitionerSurname: Selects the practitioner's first name and last name from the Practitioner table, aliased as p.

FROM clause:

- Receptionist.ClientAppointment ca: This selects data from the ClientAppointment table, aliased as ca, which likely contains information about client appointments.

JOIN clauses:

- JOIN Client c ON ca.ClientID = c.ClientID: Joins the ClientAppointment table (ca) with the Client table (c) based on the ClientID foreign key relationship. This links client appointments to their respective clients.
- JOIN ServiceType s ON ca.AppointmentType = s.ServiceTypeID: Joins the ClientAppointment table (ca) with the ServiceType table (s) based on the AppointmentType foreign key relationship. This links the appointment types to their respective service types.
- JOIN Doctor.Practitioner p ON ca.PractitionerID = p.PractitionerID: Joins the ClientAppointment table (ca) with the Practitioner table (p) based on the PractitionerID foreign key relationship. This links client appointments to their respective practitioners.

ORDER BY clause:

- ORDER BY AppointmentDate ASC: Orders the result set by the appointment date (AppointmentDate) in ascending order. This means that appointments will be listed chronologically, with the earliest appointment appearing first.

Query 5

This query calculates the total outstanding amount for each client by summing up the AmountOutstanding column from the Accounts table. It ensures that all clients are included in the result set, even if they do not have any outstanding amount. Finally, it groups the result set by client ID, name, and surname.

```
SELECT
    c.ClientID,
    c.Name,
    c.Surname,
    ISNULL(SUM(a.AmountOutstanding), 0) AS TotalOutstandingAmount
FROM
    Client c
LEFT JOIN
    Receptionist.Accounts a ON c.ClientID = a.ClientID
GROUP BY
    c.ClientID, c.Name, c.Surname;
```

SELECT statement:

- c.ClientID: Selects the ClientID from the Client table.
- c.Name: Selects the client's name from the Client table.
- c.Surname: Selects the client's surname from the Client table.
- ISNULL(SUM(a.AmountOutstanding), 0) AS TotalOutstandingAmount: Calculates the total outstanding amount for each client. It sums up the AmountOutstanding column from the Accounts table (aliased as a) and replaces any NULL values with 0. This is aliased as TotalOutstandingAmount.

FROM clause:

- Client c: This selects data from the Client table, aliased as c, which likely contains information about clients.

LEFT JOIN clause:

- LEFT JOIN Receptionist.Accounts a ON c.ClientID = a.ClientID: Performs a left join between the Client table (c) and the Accounts table (a) based on the ClientID foreign key relationship. This ensures that all clients are included in the result set, even if they do not have any corresponding records in the Accounts table.

GROUP BY clause:

- GROUP BY c.ClientID, c.Name, c.Surname: Groups the result set by ClientID, Name, and Surname. This means that the aggregation function (SUM) will be applied for each unique combination of these columns.

CREATION OF USER LOGINS AS WELL AS PRIVELAGES

This script ensures that certain roles and logins exist in the database, assigns users to these roles, grants permissions to these roles, and handles errors gracefully by rolling back the transaction if any error occurs during the process. This helps manage user access and security within the database DrJKnoetze.

```
USE DrJKnoetze;
GO

BEGIN TRANSACTION;
BEGIN TRY
    -- Check and create roles if they do not exist
    IF NOT EXISTS (SELECT * FROM sys.database_principals WHERE name = N'Doctor')
        CREATE ROLE Doctor;

    IF NOT EXISTS (SELECT * FROM sys.database_principals WHERE name = N'Receptionist')
        CREATE ROLE Receptionist;

    -- Check and create logins if they do not exist
    IF NOT EXISTS (SELECT * FROM sys.server_principals WHERE name = N'DoctorLogin')
    BEGIN
        CREATE LOGIN DoctorLogin WITH PASSWORD = 'P@ssw0rd1';
        CREATE USER DoctorUser FOR LOGIN DoctorLogin;
        ALTER ROLE Doctor ADD MEMBER DoctorUser;
    END

    IF NOT EXISTS (SELECT * FROM sys.server_principals WHERE name = N'ReceptionistLogin')
    BEGIN
        CREATE LOGIN ReceptionistLogin WITH PASSWORD = 'P@ssw0rd2';
        CREATE USER ReceptionistUser FOR LOGIN ReceptionistLogin;
        ALTER ROLE Receptionist ADD MEMBER ReceptionistUser;
    END

        GRANT SELECT, INSERT, UPDATE, DELETE ON SCHEMA::Doctor TO Doctor;

        GRANT SELECT, INSERT, UPDATE, DELETE ON SCHEMA::Receptionist TO Receptionist;

    COMMIT TRANSACTION; -- If everything is successful, commit the transaction
END TRY
BEGIN CATCH
    -- If there's an error, roll back the transaction
    ROLLBACK TRANSACTION;

    -- Error handling: capture and rethrow the error
    DECLARE @ErrorMessage NVARCHAR(4000) = ERROR_MESSAGE();
    DECLARE @ErrorSeverity INT = ERROR_SEVERITY();
    DECLARE @ErrorState INT = ERROR_STATE();
    RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState);
END CATCH
```

IF NOT EXISTS...CREATE LOGIN...CREATE USER...ALTER ROLE...: Similar to the previous step, these statements check if certain logins exist (DoctorLogin and ReceptionistLogin). If they do not exist, the script creates them using the CREATE LOGIN statement. It then creates corresponding database users (DoctorUser and ReceptionistUser) using the CREATE USER statement and assigns them to their respective roles (Doctor and Receptionist) using the ALTER ROLE statement.

GRANT SELECT, INSERT, UPDATE, DELETE ON SCHEMA...: These statements grant specific permissions (SELECT, INSERT, UPDATE, DELETE) on the schemas Doctor and Receptionist to the roles Doctor and Receptionist, respectively. This allows the members of these roles to perform those operations on objects within their respective schemas.

COMMIT TRANSACTION;: If no errors occur, this statement commits the transaction, making all the changes performed by the script permanent.

END TRY: This marks the end of the try block.

BEGIN CATCH: This begins a block of code to handle any errors that might occur within the try block.

ROLLBACK TRANSACTION;: If an error occurs, this statement rolls back the transaction, undoing any changes made by the script within the transaction.

RAISERROR: This raises an error message using the RAISERROR function, providing details about the error that occurred.

END CATCH: This marks the end of the catch block.

SAMPLE DATA QUERIES

```
USE DrJKnoetze;
GO

--IF Used for initial setup
DBCC CHECKIDENT ('MedicalAid', RESEED, 0);
GO
DBCC CHECKIDENT ('Doctor.Practitioner', RESEED, 0);
GO
DBCC CHECKIDENT ('Doctor.Qualification', RESEED, 0);
GO
DBCC CHECKIDENT ('Receptionist.Client', RESEED, 0);
GO
DBCC CHECKIDENT ('VisitReason', RESEED, 0);
GO
DBCC CHECKIDENT ('Receptionist.Child', RESEED, 0);
GO
DBCC CHECKIDENT ('Receptionist.Guardian', RESEED, 0);
GO
DBCC CHECKIDENT ('GuardianRelationType', RESEED, 0);
GO
DBCC CHECKIDENT ('RelationshipStatus', RESEED, 0);
GO
DBCC CHECKIDENT ('Receptionist.Adult', RESEED, 0);
GO
DBCC CHECKIDENT ('Receptionist.Accounts', RESEED, 0);
GO
DBCC CHECKIDENT ('ServiceType', RESEED, 0);
GO
DBCC CHECKIDENT ('AccountStatus', RESEED, 0);
GO
DBCC CHECKIDENT ('Receptionist.AccountDiscount', RESEED, 0);
GO
DBCC CHECKIDENT ('Receptionist.ClientAppointment', RESEED, 0);
GO
DBCC CHECKIDENT ('Receptionist.Invoice', RESEED, 0);
GO
DBCC CHECKIDENT ('Receptionist.InvoicePayments', RESEED, 0);
GO
DBCC CHECKIDENT ('ReportTypes', RESEED, 0);
GO
DBCC CHECKIDENT ('ReportStatus', RESEED, 0);
GO
```

USE DrJKnoetze;: This statement instructs SQL Server to switch to the database named DrJKnoetze, making it the current database for subsequent operations.

DBCC CHECKIDENT ('Table_Name', RESEED, 0);: These statements are used to reset the identity seed value for each specified table. The DBCC CHECKIDENT command is used to check the current identity value for the specified table and optionally change the identity value. In this case, the RESEED option is used to reset the identity seed value to 0 for each table.

GO: The GO keyword is used to indicate the end of a batch of Transact-SQL statements. It is a batch terminator and is used to separate multiple batches of statements.

Each DBCC CHECKIDENT statement is followed by a table name and the RESEED option, indicating that the identity seed value for the respective table should be reset to 0.

Here's a breakdown of the tables for which the identity seed values are being reset:

- MedicalAid
- Doctor.Practitioner
- Doctor.Qualification
- Receptionist.Client
- VisitReason
- Receptionist.Child
- Receptionist.Guardian
- GuardianRelationType
- RelationshipStatus
- Receptionist.Adult
- Receptionist.Accounts
- ServiceType
- AccountStatus
- Receptionist.AccountDiscount
- Receptionist.ClientAppointment
- Receptionist.Invoice
- Receptionist.InvoicePayments
- ReportTypes
- ReportStatus

Doctor.Practioner Sample Data

```
INSERT INTO Doctor.Practitioner(Name, Surname) VALUES
('John', 'Doe'),
('Jane', 'Smith'),
('Michael', 'Johnson'),
('Emily', 'Brown'),
('David', 'Williams'),
('Sarah', 'Jones'),
('Daniel', 'Martinez'),
('Jessica', 'Garcia'),
('Matthew', 'Hernandez'),
('Amanda', 'Lopez'),
('Christopher', 'Young'),
('Ashley', 'King'),
('Joshua', 'Lee'),
('Megan', 'Scott'),
('Ryan', 'Green'),
('Lauren', 'Evans'),
('Justin', 'Hall'),
('Stephanie', 'Adams'),
('Brandon', 'Morris'),
('Elizabeth', 'Nelson');
-- Add more rows as needed
```

Doctor.Qualification Sample Data

```
INSERT INTO Doctor.Qualification (Name, NQFLevel) VALUES
('Bachelor of Medicine', 8),
('Doctor of Medicine', 10),
('Bachelor of Nursing', 7),
('Master of Nursing', 9),
('Bachelor of Psychology', 7),
('Master of Psychology', 9),
('Bachelor of Pharmacy', 8),
('Master of Pharmacy', 9),
('Bachelor of Dentistry', 8),
('Master of Dentistry', 9),
('Bachelor of Occupational Therapy', 7),
('Master of Occupational Therapy', 9),
('Bachelor of Physiotherapy', 7),
('Master of Physiotherapy', 9),
('Bachelor of Optometry', 7),
('Master of Optometry', 9),
('Bachelor of Social Work', 7),
('Master of Social Work', 9),
('Bachelor of Chiropractic', 7),
('Master of Chiropractic', 9);
-- Add more rows as needed
```

Doctor.PractionerQualification Sample Data

```
INSERT INTO Doctor.PractitionerQualification (PractitionerID, QualificationID, Institution, Description, YearObtained) VALUES
(1, 1, 'Harvard Medical School', 'Completed medical degree', 2010),
(2, 2, 'Johns Hopkins University School of Medicine', 'Specialized in cardiology', 2013),
(1, 3, 'Yale School of Nursing', 'Completed nursing degree', 2008),
(4, 4, 'University of Pennsylvania School of Nursing', 'Specialized in pediatrics', 2011),
(5, 5, 'Stanford University Department of Psychology', 'Completed psychology degree', 2009),
(3, 6, 'Columbia University Department of Psychology', 'Specialized in clinical psychology', 2012),
(5, 7, 'University of California, San Francisco School of Pharmacy', 'Completed pharmacy degree', 2010),
(8, 8, 'University of Michigan College of Pharmacy', 'Specialized in pharmacotherapy', 2013),
(9, 9, 'University of Maryland School of Dentistry', 'Completed dentistry degree', 2008),
(2, 10, 'University of North Carolina at Chapel Hill School of Dentistry', 'Specialized in orthodontics', 2011),
(6, 11, 'Boston University College of Health and Rehabilitation Sciences: Sargent College', 'Completed occupational therapy degree', 2009),
(4, 12, 'University of Southern California Division of Biokinesiology and Physical Therapy', 'Specialized in pediatric occupational therapy', 2012),
(5, 13, 'University of Pittsburgh School of Health and Rehabilitation Sciences', 'Completed physiotherapy degree', 2010),
(6, 14, 'Ohio State University School of Health and Rehabilitation Sciences', 'Specialized in sports physiotherapy', 2013),
(4, 15, 'University of Houston College of Optometry', 'Completed optometry degree', 2008),
(3, 16, 'Indiana University School of Optometry', 'Specialized in pediatric optometry', 2011),
(2, 17, 'University of Chicago School of Social Service Administration', 'Completed social work degree', 2009),
(3, 18, 'University of Washington School of Social Work', 'Specialized in family therapy', 2012),
(1, 19, 'Life University College of Chiropractic', 'Completed chiropractic degree', 2010),
(2, 20, 'Palmer College of Chiropractic', 'Specialized in sports chiropractic', 2013);
```

MedicalAid Sample Data

```
INSERT INTO MedicalAid (MedicalAidNum, MedicalAidName) VALUES
('MA001', 'HealthCarePlus'),
('MA002', 'WellnessFirst'),
('MA003', 'MediCareGuard'),
('MA004', 'FamilyCare'),
('MA005', 'HealthShield'),
('MA006', 'MediHelp'),
('MA007', 'GuardianCare'),
('MA008', 'LifeSaver'),
('MA009', 'CareFirst'),
('MA010', 'GuardianPlus'),
('MA011', 'FamilyGuard'),
('MA012', 'HealthWise'),
('MA013', 'CareZone'),
('MA014', 'MediPlan'),
('MA015', 'LifeCare'),
('MA016', 'FamilyFirst'),
('MA017', 'WellBeing'),
('MA018', 'MediCover'),
('MA019', 'HealthSure'),
('MA020', 'CareZone');
```

Receptionist.Client Sample Data

```
-- Sample data for Client table
INSERT INTO Receptionist.Client(IDNum, Name, Surname, DateOfBirth, Phone, WhatsAppNumber, PostalAddress, Email)
VALUES
('9304011234567', 'John', 'Doe', '2005-04-01','+27831234567', '+27831234567', '123 Main St, City', 'john.doe@example.com'),
('8506029876543', 'Jane', 'Smith', '2002-06-02','+27839876543', '+27839876543', '456 Elm St, Town', 'jane.smith@example.com'),
('8807158765432', 'Michael', 'Johnson', '2012-07-15', '+27835555555', '+27835555555', '789 Oak St, Village', 'michael.johnson@example.com'),
('9003207654321', 'Emily', 'Brown', '2003-03-20','+27838888888', '+27838888888', '101 Pine St, County', 'emily.brown@example.com'),
('9505106543210', 'David', 'Martinez', '2018-05-10', '+27834444444', '+27834444444', '202 Cedar St, Township', 'david.martinez@example.com'),
('9107155432109', 'Sarah', 'Taylor', '2009-07-15', '+27836666666', '+27836666666', '303 Maple St, Hamlet', 'sarah.taylor@example.com'),
('9205244321098', 'Christopher', 'Anderson', '2008-05-24', '+27837777777', '+27837777777', '404 Birch St, Borough', 'christopher.anderson@example.com'),
('8708133210987', 'Amanda', 'Wilson', '2010-08-13', '+27839999999', '+27839999999', '505 Walnut St, Municipality', 'amanda.wilson@example.com'),
('8902092109876', 'James', 'Garcia', '2015-02-09', '+27832222222', '+27832222222', '606 Spruce St, District', 'james.garcia@example.com'),
('8406301098765', 'Jessica', 'Lopez', '2009-06-30', '+27833333333', '+27833333333', '707 Pineapple St, Precinct', 'jessica.lopez@example.com'),
('9601020987654', 'Matthew', 'Hernandez', '2011-01-02', '+27831111111', '+27831111111', '808 Strawberry St, Sector', 'matthew.hernandez@example.com'),
('9305239876543', 'Lauren', 'Young', '2006-05-23', '+27835555555', '+27835555555', '909 Cherry St, Block', 'lauren.young@example.com'),
('8504088765432', 'Ryan', 'Miller', '2014-04-08', '+27834444444', '+27834444444', '1010 Grape St, Tract', 'ryan.miller@example.com'),
('8807197654321', 'Ashley', 'King', '2017-07-19', '+27838888888', '+27838888888', '1111 Orange St, Division', 'ashley.king@example.com'),
('9006276543210', 'Daniel', 'Lee', '2019-06-27', '+27839999999', '+27839999999', '1212 Pear St, Lot', 'daniel.lee@example.com'),
('9508065432109', 'Megan', 'Gonzalez', '2020-08-06', '+27832222222', '+27832222222', '1313 Lemon St, Unit', 'megan.gonzalez@example.com'),
('9102044321098', 'Kevin', 'White', '2014-02-04', '+27833333333', '+27833333333', '1414 Banana St, Assembly', 'kevin.white@example.com'),
('9204233210987', 'Rachel', 'Martinez', '2009-04-23', '+27834444444', '+27834444444', '1515 Kiwi St, Cluster', 'rachel.martinez@example.com'),
('8706222109876', 'Justin', 'Davis', '2018-06-22', '+27835555555', '+27835555555', '1616 Mango St, Neighborhood', 'justin.davis@example.com'),
('8908021098765', 'Brittany', 'Hernandez', '2019-08-02', '+27836666666', '+27836666666', '1717 Lime St, Suburb', 'brittany.hernandez@example.com'),
('9412021234570', 'Alice', 'Brown', '2005-12-02', '+27731234701', '+27731234701', '19 Rose St, City', 'alice.brown21@example.com'),
('8611039876550', 'Bob', 'Johnson', '2019-11-03', '+27839876702', '+27839876702', '58 Oak St, Town', 'bob.johnson22@example.com'),
('8812048765430', 'Carlos', 'Lee', '2018-12-04', '+27735555703', '+27735555703', '798 Pine St, Village', 'carlos.lee23@example.com'),
('9203057654320', 'Diana', 'Martinez', '2017-03-05', '+27836666804', '+27836666804', '1012 Maple St, Hamlet', 'diana.martinez24@example.com'),
('9504066543210', 'Evan', 'Davis', '2016-04-06', '+27837777905', '+27837777905', '1214 Birch St, Borough', 'evan.davis25@example.com'),
('9705075432101', 'Fiona', 'Garcia', '2013-05-07', '+27738888006', '+27738888006', '1316 Walnut St, Municipality', 'fiona.garcia26@example.com'),
('9806084321092', 'George', 'Hill', '2007-06-08', '+27839999107', '+27839999107', '1418 Spruce St, District', 'george.hill27@example.com'),
('9907093210983', 'Hannah', 'Adams', '2008-07-09', '+27731110208', '+27731110208', '1520 Pineapple St, Precinct', 'hannah.adams28@example.com'),
('0008102109874', 'Ian', 'Baker', '2015-08-10', '+27832221309', '+27832221309', '1622 Strawberry St, Sector', 'ian.baker29@example.com'),
('8109111098765', 'Julia', 'Clark', '2019-09-11', '+27733332410', '+27733332410', '1724 Cherry St, Block', 'julia.clark30@example.com'),
('9412021234570', 'Alice', 'Brown', '2020-12-02', '+27731234701', '+27731234701', '19 Rose St, City', 'alice.brown21@example.com'),
('8611039876550', 'Bob', 'Pietersen', '2018-11-03', '+27839876702', '+27839876702', '58 Oak St, Town', 'bob.johnson22@example.com'),
('8812048765430', 'Carlos', 'Lee', '2016-12-04', '+27735555703', '+27735555703', '798 Pine St, Village', 'carlos.lee23@example.com'),
('9203057654320', 'Diana', 'Martinez', '2013-03-05', '+27836666804', '+27836666804', '1012 Maple St, Hamlet', 'diana.martinez24@example.com'),
('9504066543210', 'Evan', 'Davis', '2018-04-06', '+27837777905', '+27837777905', '1214 Birch St, Borough', 'evan.davis25@example.com'),
('9412021234567', 'Alice', 'Wong', '1994-12-02', '+27731234621', '+27731234621', '18 Rose St, City', 'alice.wong21@example.com'),
('8611039876543', 'Bob', 'Marley', '1986-11-03', '+27839876643', '+27839876643', '57 Oak St, Town', 'bob.marley22@example.com'),
('8812048765432', 'Carlos', 'Santana', '1988-12-04', '+27735555623', '+27735555623', '789 Pine St, Village', 'carlos.santana23@example.com'),
('9304011234567', 'John', 'Doe', '1993-04-01', '+27731234567', '+27731234567', '123 Main St, City', 'john.doe1@example.com'),
('8506029876543', 'Jane', 'Smith', '1985-06-02', '+27839876543', '+27839876543', '456 Elm St, Town', 'jane.smith2@example.com'),
('9501014800086', 'Liam', 'Smith', '1995-01-01', '+27720000001', '+27720000001', '1 Elm Street, Suburb', 'liam.smith@example.com'),
('9402024800087', 'Emma', 'Johnson', '1994-02-02', '+27720000002', '+27720000002', '2 Pine Street, Suburb', 'emma.johnson@example.com'),
('9303034800088', 'Noah', 'Williams', '1993-03-03', '+27720000003', '+27720000003', '3 Oak Street, Suburb', 'noah.williams@example.com'),
('9204044800089', 'Olivia', 'Brown', '1992-04-04', '+27720000004', '+27720000004', '4 Maple Street, Suburb', 'olivia.brown@example.com'),
('9105054800090', 'Ava', 'Jones', '1991-05-05', '+27720000005', '+27720000005', '5 Cedar Street, Suburb', 'ava.jones@example.com'),
('9006064800091', 'William', 'Garcia', '1990-06-06', '+27720000006', '+27720000006', '6 Birch Street, Suburb', 'william.garcia@example.com'),
('8907074800092', 'Sophia', 'Miller', '1989-07-07', '+27720000007', '+27720000007', '7 Walnut Street, Suburb', 'sophia.miller@example.com'),
('8808084800093', 'James', 'Davis', '1988-08-08', '+27720000008', '+27720000008', '8 Chestnut Street, Suburb', 'james.davis@example.com'),
('8709094800094', 'Isabella', 'Rodriguez', '1987-09-09', '+27720000009', '+27720000009', '9 Ash Street, Suburb', 'isabella.rodriguez@example.com'),
('8610104800095', 'Benjamin', 'Martinez', '1986-10-10', '+27720000010', '+27720000010', '10 Cherry Street, Suburb', 'benjamin.martinez@example.com'),
('8511114800096', 'Mia', 'Hernandez', '1985-11-11', '+27720000011', '+27720000011', '11 Elm Street, Suburb', 'mia.hernandez@example.com'),
('8412124800097', 'Mason', 'Moore', '1984-12-12', '+27720000012', '+27720000012', '12 Pine Street, Suburb', 'mason.moore@example.com'),
('8301014800098', 'Harper', 'Taylor', '1983-01-01', '+27720000013', '+27720000013', '13 Oak Street, Suburb', 'harper.taylor@example.com'),
('8202024800099', 'Ethan', 'Anderson', '1982-02-02', '+27720000014', '+27720000014', '14 Maple Street, Suburb', 'ethan.anderson@example.com'),
('8103034800010', 'Ella', 'Thomas', '1981-03-03', '+27720000015', '+27720000015', '15 Cedar Street, Suburb', 'ella.thomas@example.com'),
('8004044800011', 'Alexander', 'Jackson', '1980-04-04', '+27720000016', '+27720000016', '16 Birch Street, Suburb', 'alexander.jackson@example.com'),
('7905054800012', 'Amelia', 'White', '1979-05-05', '+27720000017', '+27720000017', '17 Walnut Street, Suburb', 'amelia.white@example.com'),
('7806064800013', 'Michael', 'Harris', '1978-06-06', '+27720000018', '+27720000018', '18 Chestnut Street, Suburb', 'michael.harris@example.com'),
('7707074800014', 'Charlotte', 'Clark', '1977-07-07', '+27720000019', '+27720000019', '19 Ash Street, Suburb', 'charlotte.clark@example.com'),
('7608084800015', 'Elijah', 'Lewis', '1976-08-08', '+27720000020', '+27720000020', '20 Cherry Street, Suburb', 'elijah.lewis@example.com'),
('7509094800016', 'Sofia', 'Robinson', '1975-09-09', '+27720000021', '+27720000021', '21 Elm Street, Suburb', 'sofia.robinson@example.com'),
('7410104800017', 'Logan', 'Walker', '1974-10-10', '+27720000022', '+27720000022', '22 Pine Street, Suburb', 'logan.walker@example.com'),
('7311114800018', 'Avery', 'Perez', '1973-11-11', '+27720000023', '+27720000023', '23 Oak Street, Suburb', 'avery.perez@example.com'),
('7212124800019', 'Jackson', 'Young', '1972-12-12', '+27720000024', '+27720000024', '24 Maple Street, Suburb', 'jackson.young@example.com'),
('7101014800020', 'Scarlett', 'Hernandez', '1971-01-01', '+27720000025', '+27720000025', '25 Cedar Street, Suburb', 'scarlett.hernandez@example.com'),
('7002024800021', 'Grace', 'King', '1970-02-02', '+27720000026', '+27720000026', '26 Birch Street, Suburb', 'grace.king@example.com'),
('6903034800022', 'Lucas', 'Wright', '1969-03-03', '+27720000027', '+27720000027', '27 Walnut Street, Suburb', 'lucas.wright@example.com'),
('6804044800023', 'Lily', 'Scott', '1968-04-04', '+27720000028', '+27720000028', '28 Chestnut Street, Suburb', 'lily.scott@example.com'),
('6705054800024', 'Oliver', 'Torres', '1967-05-05', '+27720000029', '+27720000029', '29 Ash Street, Suburb', 'oliver.torres@example.com'),
('6606064800025', 'Madison', 'Nguyen', '1966-06-06', '+27720000030', '+27720000030', '30 Cherry Street, Suburb', 'madison.nguyen@example.com');
```

MedicalAidClient Sample Data

```
-- Sample data for MedicalAidClient table
INSERT INTO MedicalAidClient (MedicalAidID, ClientID, DepNum) VALUES
(4, 1, 2),
(1, 2, 1),
(2, 3, 3),
(3, 4, 2),
(4, 5, 1),
(5, 6, 2),
(6, 7, 1),
(7, 8, 3),
(8, 9, 2),
(9, 10, 1),
(10, 11, 2),
(11, 12, 3),
(12, 13, 1),
(13, 14, 2),
(14, 15, 1),
(15, 16, 3),
(16, 17, 2),
(17, 18, 1),
(18, 19, 2),
(19, 20, 3);
```

Visit Reason Sample Data

```
INSERT INTO VisitReason (ReasonWhy, BackgroundInfo)
VALUES
('Seeking help for anxiety', 'Patient has been experiencing severe anxiety and panic attacks, impacting daily routines.'),
('Depression', 'Exhibiting symptoms of depression including prolonged sadness, withdrawal from social activities, and changes in sleep patterns.'),
('Work-related stress', 'Patient reports high levels of stress and burnout from work, affecting mental and physical health.'),
('Marital problems', 'Seeking counseling for ongoing marital conflicts and communication issues.'),
('PTSD symptoms', 'Patient experiencing symptoms of post-traumatic stress disorder following a recent traumatic event.'),
('ADHD management', 'Looking for strategies to manage ADHD symptoms affecting work and personal life.'),
('Eating disorder concerns', 'Patient expresses concerns over eating habits and body image, suspecting an eating disorder.'),
('Insomnia', 'Difficulty falling and staying asleep, leading to chronic fatigue and decreased quality of life.'),
('Substance dependence', 'Seeking help for dependence on substances as a coping mechanism for stress.'),
('Grief counseling', 'Patient needs support in dealing with the grief of losing a family member.'),
('Chronic pain management', 'Looking for psychological support to manage chronic pain and its impact on lifestyle.'),
('Self-esteem issues', 'Patient struggles with low self-esteem and is seeking ways to build confidence.'),
('Anger management issues', 'Reports of uncontrollable anger affecting relationships and employment.'),
('Life transition challenges', 'Struggling to adjust to significant life changes, such as retirement or a new baby.'),
('Obsessive-compulsive behavior', 'Seeking help for obsessive-compulsive behaviors that are interfering with daily life.'),
('Social anxiety', 'Patient wants to address social anxiety that inhibits participation in social and professional situations.'),
('Parenting challenges', 'Seeking guidance on dealing with parenting challenges and child behavior management.'),
('Bipolar disorder management', 'Patient with diagnosed bipolar disorder seeking assistance in managing mood swings.'),
('Stress from chronic illness', 'Looking for support in coping with the stress of living with a chronic illness.'),
('Trauma recovery', 'Patient seeking help in recovering from physical and emotional trauma experienced from an accident.');
```

Receptionist.Child Sample Data

```
INSERT INTO ServiceType (ServiceType, Description, ICD10, Cost)
VALUES
('Psychotherapy', 'Psychotherapy session', 'F43.10', 150.00),
('Counseling', 'Counseling session', 'Z71.9', 120.00),
('Psychiatric Evaluation', 'Psychiatric evaluation session', 'F31.9', 200.00),
('Medication Management', 'Medication management session', 'Z51.81', 180.00),
('Behavioral Therapy', 'Behavioral therapy session', 'F90.0', 170.00),
('Family Therapy', 'Family therapy session', 'Z63.0', 160.00),
('Group Therapy', 'Group therapy session', 'F43.8', 140.00),
('Substance Abuse Counseling', 'Substance abuse counseling session', 'F10.20', 130.00),
('Anger Management', 'Anger management session', 'F43.8', 110.00),
('Art Therapy', 'Art therapy session', 'Z73.1', 100.00),
('Play Therapy', 'Play therapy session', 'Z76.5', 90.00),
('Occupational Therapy', 'Occupational therapy session', 'Z75.89', 180.00),
('Physical Therapy', 'Physical therapy session', 'Z47.1', 170.00),
('Speech Therapy', 'Speech therapy session', 'Z47.8', 160.00),
('Nutritional Counseling', 'Nutritional counseling session', 'Z71.3', 150.00),
('Social Skills Training', 'Social skills training session', 'F80.9', 140.00),
('Parent Training', 'Parent training session', 'Z76.2', 130.00),
('Life Skills Training', 'Life skills training session', 'Z73.89', 120.00),
('Stress Management', 'Stress management session', 'Z73.3', 110.00),
('Relaxation Techniques', 'Relaxation techniques session', 'Z73.89', 100.00);
```

Relationship Status Sample Data

```
INSERT INTO RelationshipStatus (Description)
VALUES
('Married'),
('Single'),
('Divorced'),
('Widowed'),
('Engaged'),
('Separated'),
('In a Relationship'),
('Complicated'),
('Open Relationship'),
('Civil Union'),
('Domestic Partnership'),
('Other');
```

Receptionist.Guardian Sample Data

```
INSERT INTO Receptionist.Guardian (ClientID, RelationshipStatusID, Occupation)
VALUES
(41, 1, 'Engineer'),
(36, 1, 'Teacher'),
(42, 1, 'Nurse'),
(39, 1, 'Graphic Designer'),
(43, 5, 'Architect'),
(44, 2, 'Software Developer'),
(45, 4, 'Accountant'),
(46, 3, 'Doctor'),
(47, 4, 'Marketing Specialist'),
(38, 2, 'Sales Manager'),
(48, 1, 'Human Resources Manager'),
(49, 1, 'Construction Worker'),
(37, 1, 'Pharmacist'),
(50, 6, 'Chef'),
(51, 7, 'Electrician'),
(40, 4, 'Mechanic'),
(52, 2, 'Artist'),
(53, 3, 'Entrepreneur'),
(54, 1, 'Consultant'),
(55, 1, 'Journalist'),
(56, 1, 'Physiotherapist'),
(57, 1, 'Dentist'),
(58, 9, 'Biologist'),
(59, 3, 'Economist'),
(60, 2, 'Veterinarian'),
(61, 2, 'Civil Engineer'),
(62, 1, 'Data Analyst'),
(63, 1, 'Social Worker'),
(64, 2, 'Copywriter'),
(65, 3, 'Interior Designer'),
(66, 1, 'Pilot'),
(41, 1, 'Engineer'),
(36, 1, 'Teacher'),
(42, 1, 'Nurse'),
(39, 1, 'Graphic Designer'),
(43, 4, 'Architect'),
(44, 2, 'Software Developer'),
(45, 4, 'Accountant'),
(46, 3, 'Doctor'),
(47, 4, 'Marketing Specialist'),
(38, 2, 'Sales Manager'),
(48, 1, 'Human Resources Manager'),
(49, 1, 'Construction Worker'),
(37, 1, 'Pharmacist'),
(50, 5, 'Chef'),
(51, 4, 'Electrician'),
(40, 3, 'Mechanic'),
(52, 5, 'Artist'),
(53, 2, 'Entrepreneur'),
(54, 1, 'Consultant'),
(55, 1, 'Journalist'),
(56, 1, 'Physiotherapist'),
(57, 1, 'Dentist'),
(58, 4, 'Biologist'),
(59, 2, 'Economist'),
(60, 4, 'Veterinarian'),
(61, 1, 'Civil Engineer'),
(62, 3, 'Data Analyst'),
(63, 4, 'Social Worker'),
(64, 1, 'Copywriter'),
(65, 1, 'Interior Designer'),
(66, 2, 'Pilot');
```

GuardianRelationType Sample Data

```
INSERT INTO GuardianRelationType (Description)
VALUES
('Parent'),
('Legal Guardian'),
('Step Parent'),
('Grandparent'),
('Sibling'),
('Aunt/Uncle'),
('Cousin'),
('Other Relative'),
('Family Friend'),
('Foster Parent'),
('Neighbor'),
('Teacher'),
('Caretaker'),
('Social Worker'),
('Legal Representative'),
('No Relation');
```

ChildGuardian Sample Data

```
INSERT INTO ChildGuardian (ChildID,GuardianID,GuardianRelationType)
VALUES
(31, 1, 1),
(16, 2, 3),
(22, 3, 4),
(30, 4, 2),
(20, 5,4),
(15, 6, 9),
(23, 7, 7),
(32, 8,3),
(19, 9,6),
(5, 10,8),
(35, 11,5),
(14, 12,1),
(24, 13,1),
(33, 14,1),
(25, 15,2),
(29, 16,1),
(9, 17,7),
(13, 18,2),
(17, 19,3),
(26, 20,4),
(34, 21,2),
(3, 22,1),
(11, 23,1),
(8, 24,1),
(6, 25,2),
(10, 26,1),
(18, 27,3),
(28, 28,2),
(7, 29,1),
(27, 30,4),
(12, 31,2),
(21, 32,3),
(1, 33,4),
(4, 34,5),
(2, 35,6);
```

Receptionist.ClientAppointment Sample Data

```
-- Sample data for Invoice table
INSERT INTO Receptionist.Invoice (AccountID, InvoiceNum, InvoiceDate, InvoiceTotal, AppointmentID)
VALUES
(1, 'INV001', '2024-04-01', 150.00, 1),
(2, 'INV002', '2024-04-02', 120.00, 2),
(3, 'INV003', '2024-04-03', 200.00, 3),
(4, 'INV004', '2024-04-04', 180.00, 4),
(5, 'INV005', '2024-04-05', 300.00, 5),
(6, 'INV006', '2024-04-06', 160.00, 6),
(7, 'INV007', '2024-04-07', 140.00, 7),
(8, 'INV008', '2024-04-08', 130.00, 8),
(9, 'INV009', '2024-04-09', 150.00, 9),
(10, 'INV010', '2024-04-10', 140.00, 10),
(1, 'INV011', '2024-04-11', 130.00, 11),
(12, 'INV012', '2024-04-12', 120.00, 12),
(13, 'INV013', '2024-04-13', 180.00, 13),
(14, 'INV014', '2024-04-14', 170.00, 14),
(15, 'INV015', '2024-04-15', 160.00, 15),
(16, 'INV016', '2024-04-16', 150.00, 16),
(17, 'INV017', '2024-04-17', 140.00, 17),
(18, 'INV018', '2024-04-18', 130.00, 18),
(19, 'INV019', '2024-04-19', 120.00, 19),
(20, 'INV020', '2024-04-20', 180.00, 8);
```

Receptionist.Invoice Sample Data

```
-- Sample data for InvoicePayments table
INSERT INTO Receptionist.InvoicePayments (InvoiceID, Amount, Date)
VALUES
(1, 150.00, '2024-04-01'),
(2, 120.00, '2024-04-02'),
(3, 200.00, '2024-04-03'),
(4, 180.00, '2024-04-04'),
(5, 170.00, '2024-04-05'),
(6, 160.00, '2024-04-06'),
(7, 140.00, '2024-04-07'),
(8, 130.00, '2024-04-08'),
(9, 150.00, '2024-04-09'),
(10, 140.00, '2024-04-10'),
(11, 130.00, '2024-04-11'),
(12, 120.00, '2024-04-12'),
(13, 180.00, '2024-04-13'),
(14, 170.00, '2024-04-14'),
(15, 160.00, '2024-04-15'),
(16, 150.00, '2024-04-16'),
(17, 140.00, '2024-04-17'),
(18, 130.00, '2024-04-18'),
(19, 120.00, '2024-04-19'),
(5, 80.00, '2024-04-20');
```

Receptionist.InvoicePayments Sample Data

```
INSERT INTO Receptionist.InvoicePayments (InvoiceID, Amount, Date)
VALUES
(1, 150.00, '2024-04-01'),
(2, 120.00, '2024-04-02'),
(3, 200.00, '2024-04-03'),
(4, 180.00, '2024-04-04'),
(5, 170.00, '2024-04-05'),
(6, 160.00, '2024-04-06'),
(7, 140.00, '2024-04-07'),
(8, 130.00, '2024-04-08'),
(9, 150.00, '2024-04-09'),
(10, 140.00, '2024-04-10'),
(11, 130.00, '2024-04-11'),
(12, 120.00, '2024-04-12'),
(13, 180.00, '2024-04-13'),
(14, 170.00, '2024-04-14'),
(15, 160.00, '2024-04-15'),
(16, 150.00, '2024-04-16'),
(17, 140.00, '2024-04-17'),
(18, 130.00, '2024-04-18'),
(19, 120.00, '2024-04-19'),
(5, 80.00, '2024-04-20');
```

ReportTypes Sample Data

```
INSERT INTO ReportTypes (Type, Description, Cost)
VALUES
('Initial Assessment', 'Initial assessment report', 100),
('Progress Report', 'Progress report on client', 80),
('Treatment Plan', 'Treatment plan report', 120),
('Final Evaluation', 'Final evaluation report', 150),
('Follow-up Report', 'Follow-up report on client', 90),
('Specialized Assessment', 'Specialized assessment report', 130),
('Behavioral Analysis', 'Behavioral analysis report', 110),
('Intervention Summary', 'Intervention summary report', 140),
('Diagnostic Report', 'Diagnostic report on client', 160),
('Therapeutic Program', 'Therapeutic program report', 170),
('Evaluation Report', 'Evaluation report on client', 180),
('Case Management Plan', 'Case management plan report', 190),
('Psychological Assessment', 'Psychological assessment report', 200),
('Educational Report', 'Educational report on client', 210),
('Cognitive Evaluation', 'Cognitive evaluation report', 220),
('Social Skills Assessment', 'Social skills assessment report', 230),
('Behavioral Intervention Plan', 'Behavioral intervention plan report', 240),
('Functional Behavior Assessment', 'Functional behavior assessment report', 250),
('Language Evaluation', 'Language evaluation report', 260),
('Occupational Therapy Report', 'Occupational therapy report on client', 270);
```

ReportStatus Sample Data

```sql
-- Sample data for ReportStatus table
INSERT INTO ReportStatus (Description)
VALUES
('Pending'),
('In Progress'),
('Completed'),
('On Hold'),
('Cancelled');
```

Doctor.Report Sample Data

```sql
-- Sample data for Report table
-- Assuming each client has at least one report entry
INSERT INTO Doctor.Report (ClientID, ReportTypeID, ReportStatusID, Notes)
VALUES
(31, 1, 1, 'Initial assessment report for John Doe'),
(16, 2, 3, 'Progress report for Jane Smith'),
(22, 3, 2, 'Treatment plan for Michael Johnson'),
(30, 4, 2, 'Final evaluation report for Emily Brown'),
(20, 5, 1, 'Follow-up report for David Martinez'),
(15, 6, 1, 'Specialized assessment report for Sarah Taylor'),
(23, 7, 2, 'Behavioral analysis report for Christopher Anderson'),
(32, 8, 3, 'Intervention summary report for Amanda Wilson'),
(19, 9, 2, 'Diagnostic report for James Garcia'),
(5, 10, 1, 'Therapeutic program report for Jessica Lopez'),
(35, 11, 3, 'Evaluation report for Matthew Hernandez'),
(14, 12, 1, 'Case management plan report for Lauren Young'),
(24, 13, 3, 'Psychological assessment report for Ryan Miller'),
(33, 14, 2, 'Educational report for Ashley King'),
(25, 15, 1, 'Cognitive evaluation report for Daniel Lee'),
(29, 16, 2, 'Social skills assessment report for Megan Gonzalez'),
(9, 17, 1, 'Behavioral intervention plan report for Kevin White'),
(13, 18, 3, 'Functional behavior assessment report for Rachel Martinez'),
(68, 19, 2, 'Language evaluation report for Justin Davis'),
(70, 14, 3, 'Occupational therapy report for Brittany Hernandez');
```

```
    COMMIT TRANSACTION; -- If everything is successful, commit the transaction
END TRY
BEGIN CATCH
    -- If there's an error, roll back the transaction
    ROLLBACK TRANSACTION;

    -- Error handling: capture and rethrow the error
    DECLARE @ErrorMessage NVARCHAR(4000) = ERROR_MESSAGE();
    DECLARE @ErrorSeverity INT = ERROR_SEVERITY();
    DECLARE @ErrorState INT = ERROR_STATE();
    RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState);
END CATCH
GO
```

COMMIT TRANSACTION;: This statement is commented out but would typically be used to commit the transaction if everything executed successfully within the TRY block.

END TRY: This marks the end of the TRY block.

BEGIN CATCH: This begins the CATCH block, which is executed if an error occurs within the TRY block.

ROLLBACK TRANSACTION;: This statement rolls back the transaction if an error occurs, reverting any changes made within the TRY block.

DECLARE @ErrorMessage NVARCHAR(4000) = ERROR_MESSAGE();: This declares a variable @ErrorMessage of type NVARCHAR(4000) and assigns it the value of the error message produced by the ERROR_MESSAGE() function. This function retrieves the text of the message generated by the last statement that caused an error.

DECLARE @ErrorSeverity INT = ERROR_SEVERITY();: This declares a variable @ErrorSeverity of type INT and assigns it the value of the severity level of the error generated by the ERROR_SEVERITY() function. Severity levels range from 0 to 25. Errors with severity levels from 11 through 19 are considered to be errors, whereas severity levels from 20 through 25 are considered to be fatal errors.

DECLARE @ErrorState INT = ERROR_STATE();: This declares a variable @ErrorState of type INT and assigns it the value of the state of the error generated by the ERROR_STATE() function. Error states provide additional information about the error, such as the line number where the error occurred.

RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState);: This raises an error using the RAISERROR function, which displays the error message, severity level, and state that were captured earlier. This effectively rethrows the error, propagating it up the call stack.

END CATCH: This marks the end of the CATCH block.

GO: This statement is used as a batch terminator, marking the end of the batch of Transact-SQL statements.

In summary, this code snippet provides error handling for the preceding code. If an error occurs within the TRY block, it is caught by the CATCH block, which rolls back the transaction and raises an error message containing details about the error. This helps to gracefully handle errors and prevent partial or inconsistent database modifications.