**Designing a High-Performance SQL Databases for Efficient Behavioral Health Data Analysis**

**Project Overview**

**Project Name**: Raise and Grow Database Management System

**Project Manager**: Emmanuel Atangana Seme

**Date**: March 2021

## **Project Outlines**

1. Introduction
   1. Purpose
   2. Problem Statement
   3. Expected Outcome
   4. Sample Key Questions that can be Addressed by the Database System
2. Project Scope
   1. Primary Focus Areas
   2. Stakeholders
3. RG’s Current Challenges
   1. Decentralized and Manual Data Management Processes
   2. Difficulty in Tracking Client Progress and Staff Performance
   3. Inefficiencies in Generating Compliance Reports
   4. Inefficiencies in Generating Compliance Reports

4. Database Design

4.1 Key Features

4.2 Relationships and Schema

4.3 Creating an Entity-Relationship Diagram (ERD)

5. Implementation Plan

5.1 Implementation Plan

5.2 Database Development

5.3 Views and Stored Procedures

5.4 Testing and Deployment

6. Sample Problems Scenarios that could be addressed Using this Database

6.1 Monitoring Client Progress

6.2 Incident Trends Analysis

6.3 Staff Certification Tracking

6.4 Evaluating Program Effectiveness

* 1. Resource Utilization

1. Expected Benefits
   1. Expected Benefits
   2. Decision-Making
   3. Care Quality
   4. Compliance
   5. Efficiency
   6. Scalability
2. General Conclusion
3. **Introduction**

Raise and Grow (RG) is dedicated to supporting adults with developmental disabilities and behavioral challenges by providing high-quality residential care and tailored programs. However, managing client data, staff schedules, program performance, and compliance reporting has become ever more complex with existing systems. This project aims to design and implement a centralized, secure, and scalable database solution. By restructuring data processes and enhancing operational efficiency, this database will empower RG;s management processes to deliver better care, track progress in real-time, and meet compliance requirements with ease.

* 1. **Purpose**

The purpose of this project is to develop and implement a robust database system for Raise and Grow (RG), a structure in charge of adults with developmental disabilities and behavioral challenges, The database aims to improve the management of behavioral health services. The database will centralize client, staff, and program data, streamline operations, and provide actionable insights to support decision-making. This will enable RG to enhance care quality, reduce inefficiencies, and ensure compliance with regulatory standards.

* 1. **Problem Statement**

Raise and Grow currently faces challenges with decentralized data management and manual processes, which lead to inefficiencies, errors, and limited insights. Indeed, tracking client progress, managing staff schedules, and monitoring program outcomes are time-consuming and prone to inaccuracies. In addition, preparing compliance reports requires significant effort, diverting valuable resources from main services. Without an integrated database, RG struggles to address these challenges effectively, struggles that impact the quality of care and managerial efficiency.

* 1. **Expected Outcome**

The success of this project will be measured by its ability to rationalize operations and improve service delivery at RG. Key performance indicators (KPIs) include a significant reduction in time spent on manual data entry and reporting, demonstrated by a 30% improvement in operational efficiency within the first six months. Success will also be evident through enhanced care quality, tracked via a reduction in behavioral incidents and improved client outcomes, as well as timely and accurate compliance reporting with “zero” missed deadlines. Additionally, user’s satisfaction among staff and stakeholders will be monitored, with an expected 85% reporting improved accessibility and usability of the system. Scalability and data security will be validated by the database’s ability to handle increased workloads and maintain data integrity without breaches or downtime.

* 1. **Sample Key Questions that can be Addressed by the Database System**

As part of the implementation of a robust database for Raise and Grow, the system is designed to solve several critical questions that currently challenge operational efficiency and service delivery. The database will provide centralized data access and real-time insights and enable the organization to answer pressing operational, compliance, and care-related questions:

**(a) Type 1 key questions: Client Progress and Behavioral Health**

* + What are the trends (development) in behavioral health for specific clients, and how effective are their current intervention plans?
  + Are there recurring patterns in incidents, and how can these be mitigated?

**(b) Type 2 key questions: Program Performance and Effectiveness**

* + Which programs are delivering the most significant positive outcomes for clients?
  + What adjustments are needed in existing programs to enhance their impact?

**(c) Type 3 key questions: Staff Management and Scheduling**

* + Are staff certifications and training up to date to meet program requirements?
  + How effectively are staff members matched with clients to maximize compatibility and care quality?

**(d) Type 4 key questions: Compliance and Reporting**

* + Are all compliance reports prepared and submitted on time, with the necessary supporting data?
  + What insights can be drawn from compliance data to inform strategic decisions?

**(e) Type 5 key questions: Resource Optimization**

* + Are resources such as staff hours and funding being used efficiently across programs?
  + What improvements can be made to balance workloads and ensure consistent coverage?

**(f) Type 6 key questions: Client Placement and Compatibility**

* + How successful have recent client placements been in terms of reduced behavioral incidents and improved client satisfaction?
  + Which available placements are best suited to meet the needs of incoming clients?

**(g) Type 7 key questions: Operational Efficiency and Growth:**

* + What bottlenecks in data management have been resolved since implementing the database?
  + How can the system scale support the organization’s growth and new program development?

1. **Project Scope**

**2.1 Primary Focus Areas**

The project will centralize data management by consolidating client, staff, program, incident, and resource data into a single, secure platform, ensuring easy access and efficient handling. It will automate key processes such as incident reporting, progress tracking, and compliance reporting, reducing manual effort and minimizing errors. Behavioral health monitoring will be enhanced through real-time tracking of client behavior patterns and outcomes, enabling timely and effective interventions. The database will be designed to scale with the organization’s growth, ensuring flexibility to accommodate new programs and increased data. Additionally, it will support compliance by providing automated tools for accurate and timely reporting to regulatory bodies. Finally, data security and privacy will be prioritized with role-based access control and robust encryption to safeguard sensitive information.

**2.2 Stakeholders**

The stakeholders benefiting from this project include clients, whose data will be securely managed to ensure personalized care and better outcomes. Direct Support Professionals (DSPs) will use the system to access client profiles, report incidents, and manage schedules more efficiently. Program managers and specialists will monitor program effectiveness and update behavior plans based on real-time data. Behavior specialists will leverage the system’s insights to refine strategies and provide timely interventions. Administrative staff will streamline resource allocation, staff certifications, and compliance reporting. Regulatory bodies like DSHS and DDA will receive accurate and timely reports, ensuring adherence to standards. Finally, the leadership team will utilize database insights for strategic planning and organizational development.

1. **RG’s Current Challenges**
   1. **Decentralized and Manual Data Management Processes**

Raise and Grow currently relies on scattered and manual methods for managing critical data, with client, staff, and program information stored in multiple locations and formats. This decentralization results in inconsistencies, data redundancies, and inefficiencies, making it difficult to retrieve accurate information in a timely manner.

**3.2 Difficulty in Tracking Client Progress and Staff Performance**

The absence of a centralized system creates significant challenges in monitoring client outcomes and evaluating staff effectiveness. Tracking behavioral trends, progress in support plans, and staff compliance with certifications or training schedules is cumbersome and often incomplete, leading to missed opportunities for timely interventions and improvements.

**3.3 Inefficiencies in Generating Compliance Reports**

Reports for regulatory bodies such as DSHS and DDA are a time-consuming and error-prone process due to the reliance on manual data compilation. These inefficiencies increase the risk of inaccuracies, delays, and potential non-compliance, diverting valuable resources away from core operations.

**3.4 Limited Scalability and Adaptability of Current Systems**

The existing systems are not designed to scale with the organization's growth or adapt to new programs and evolving needs. As Raise and Grow expands its services, the current infrastructure struggles to accommodate increased client data, staff records, and program requirements, hindering operational efficiency and organizational development.

1. **Database Design**

**4.1 Key Features**

The database system will include several essential features to simplify operations and improve efficiency. A centralized data repository will consolidate all clients, staffs, programs, and resource information into a single platform for easy access and management. The system will automate key processes such as incident logging, progress tracking, and compliance reporting, reducing manual workload and minimizing errors. Role-based access control will ensure data security, allowing only authorized users to access sensitive information. Real-time dashboards and reporting tools will provide actionable insights, enabling timely decision-making. The system will also be designed to scale seamlessly, accommodating organizational growth and new program requirements while maintaining high performance.

**4.2 Relationships and Schema**

The database design will follow a relational structure with well-defined entities and relationships to ensure data consistency and integrity. Key entities include:

* **Clients Table**: Stores personal details, medical history, behavioral profiles, and program placements, linking to incidents and behavior plans.
* **Staff Table**: Maintains staff details, certifications, training records, and program assignments, linked to clients and programs.
* **Programs Table**: Tracks program details, client placements, and associated staff, with relationships to behavior plans and incidents.
* **Table Incidents**: Logs behavioral incidents, linked to specific clients and staff for detailed analysis.
* **Behavior Plans Table**: Stores personalized intervention plans and progress notes, tied to client profiles.
* **Resources Table**: Lists external affiliations, training materials, and related program resources.
* **Compliance Reports Table**: Tracks compliance documentation and reporting timelines, linked to programs and regulatory requirements.

The schema is designed to establish relationships between these entities, ensuring data is interconnected and accessible for comprehensive analysis. For example, a query can retrieve a client’s progress, their associated behavior plan, and the staff involved in their care, providing a complete picture in one view. This relational approach supports both operational needs and strategic decision-making.

**4.3 Creating an Entity-Relationship Diagram (ERD)**

Relationships between entities:

* + **Clients** to **Programs**: One client belongs to one program (1:N).
  + **Staff** to **Programs**: A staff can work in multiple programs (M:N).
  + **Incidents** to **Clients**: Incidents are related to a specific client (1:N).
  + **Behavior Plans** to **Clients**: Each client can have multiple behavior plans (1:N).
  + **Programs** to **Resources**: Programs may use multiple resources (M:N).
  + **Programs** to **Compliance Reports**: Programs generate multiple compliance reports (M:N).

**Visualizing the Entity-Relationship Diagram (ERD) using dbdiagram.io Tool**

A screenshot of a computer program

Description automatically generated

1. **Implementation Plan**

**5.1 Requirements Gathering**

The first phase of the implementation plan involves gathering and analyzing requirements to ensure the database system meets organizational needs. This step includes collaborating with stakeholders such as program managers, staff, and administrators to identify critical data points like client profiles, program details, incidents, and compliance requirements. The process involves documenting existing workflows, understanding pain points in current data management practices, and defining the desired outcomes, such as improved data accessibility, automated reporting, and streamlined operations. This phase ensures that the database structure, functionality, and features are aligned with operational goals and regulatory obligations.

* 1. **Database Development**

The database development phase focuses on designing and building a robust relational database system based on the requirements gathered. This includes creating the schema with properly structured tables, defining relationships between entities (e.g., clients and programs), and enforcing data integrity through constraints such as primary and foreign keys. During this phase, initial data is populated, and queries, views, and stored procedures are developed to support the system's functionality. The goal is to create a scalable and secure database that meets operational and reporting needs.

* We first create the Database, and we call it “Raise\_Grow”



* Then we create each table of the database. We first ensure that these tables are not do yet exist. If the Table already exists, we simply drop it.

A screenshot of a computer code

Description automatically generated

This SQL script checks if specific user-defined tables (ComplianceReports, Resources, BehaviorPlans, Incidents, Clients, Staff, Programs) exist in the Raise\_Grow database using the OBJECT\_ID function with “U” (indicating user-defined tables). If a table already exists, it is dropped using DROP TABLE. By doing this, we ensure there are no conflicts when recreating or modifying these tables later. This process is commonly used in development or testing to maintain schema consistency.

A computer program code with text

Description automatically generated

(1)

The scripts (1) create a Programs table with a unique ProgramID as the primary key, auto-incrementing with each new entry. The table includes a Name (required, max 255 characters), a Description (long text), a SpecialistID (integer reference), and StartDate and EndDate to define the program's timeline. It is designed to store detailed information about programs, ensuring each entry is uniquely identifiable and organized.

A computer screen shot of a computer

Description automatically generated

(2)

The scripts (2) create a Clients table with a unique ClientID as the primary key, auto-incrementing for each new entry. The table stores client details, including Name (required, max 255 characters), DOB (date of birth), Gender (max 10 characters), Address, MedicalHistory, and BehavioralProfile (all in text format for detailed information). It also includes a CurrentProgramID column as a foreign key referencing the ProgramID in the Programs table, establishing a relationship between clients and the programs they are associated with.

A computer code on a white background

Description automatically generated

(3)

The script (3) creates a Staff table with a unique StaffID as the primary key, auto-incrementing for each new entry. It includes columns for Name (required, max 255 characters), Role (max 100 characters), Certifications (detailed text), TrainingCompleted (date), and Availability (text format). Additionally, it has a ProgramID column as a foreign key referencing the ProgramID in the Programs table, establishing a relationship between staff members and their associated programs. This table captures comprehensive staff details and their program affiliations.

A computer screen shot of text

Description automatically generated (4)

The script (4) creates an Incidents table with a unique IncidentID as the primary key, auto-incrementing for each new record. It includes the ClientID (a required foreign key referencing the Clients table), Date (the incident date, required), Type (description of the incident type, max 100 characters), Description (detailed text about the incident), and FollowUpActions (text describing actions taken after the incident). This table tracks incidents associated with clients, ensuring relational integrity with the Clients table.

A computer screen shot of a white background

Description automatically generated with medium confidence

(5)

The script (6) creates a BehaviorPlans table with a unique PlanID as the primary key, auto-incrementing for each new record. It includes a ClientID (a required foreign key referencing the Clients table), DateCreated (the date the behavior plan was created, required), PlanDetails (text field for detailed behavior plan information), and ProgressNotes (text field for tracking progress). This table links behavior plans to specific clients, ensuring each plan is associated with the appropriate client in the Clients table.

A computer code with black text

Description automatically generated with medium confidence

(6)

The script (6) creates a Resources table with a unique ResourceID as the primary key, auto-incrementing for each new entry. It includes columns for Name (required, max 255 characters), Type (describing the type of resource, max 100 characters), Description (text field for detailed information about the resource), and AssociatedProgramID (a foreign key referencing the ProgramID in the Programs table). This table is designed to store information about various resources and their association with specific programs.

A computer code with text

Description automatically generated with medium confidence

(7)

The script (8) creates a ComplianceReports table with a unique ReportID as the primary key, auto-incrementing for each new record. The table includes DateGenerated (required date of report creation), ProgramID (required foreign key referencing the Programs table to associate the report with a specific program), and ComplianceDetails (a text field for detailed information about the compliance report). This table is designed to store and manage compliance-related reports for various programs.

* Populate the database with the data.

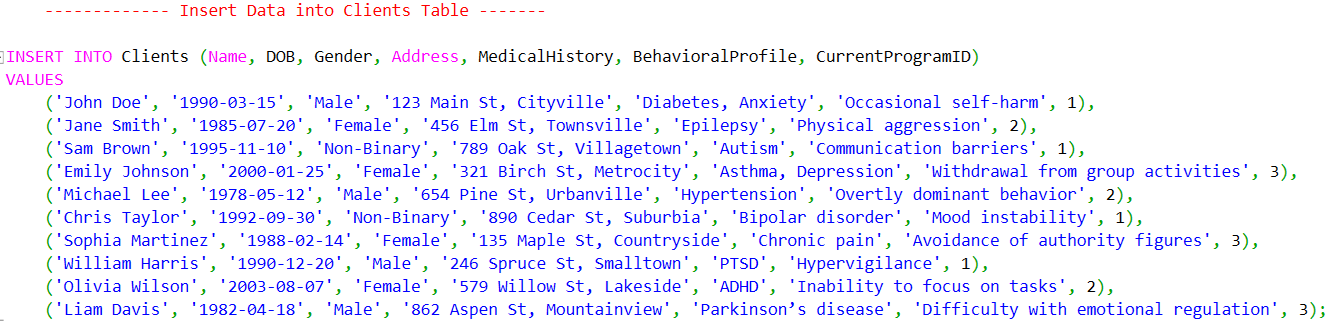
After creating these tables, the next step is to populate them with data using the INSERT INTO statement. Begin by inserting data into the parent tables such as Programs, as they are referenced by foreign keys in other tables. Once the parent tables are populated, proceed to insert data into the dependent tables like Clients, Staff, Incidents, and others, ensuring that the foreign key values correspond to valid entries in the parent tables to maintain referential integrity.

A screenshot of a computer

Description automatically generated

Each row represents a unique program with its name, description, specialist in charge (SpecialistID), start date (StartDate), and optional end date (EndDate). The “NULL” in EndDate indicates ongoing programs without a specified end date.

* After inserting records into the Programs table, the next step is to insert data into the **Clients** table, as it does not have foreign key dependencies on any other table except Programs.



* Next, we insert data into the **Staff** table.

A screenshot of a computer

Description automatically generated

* Then, we insert data into the Resources Table

A screenshot of a computer

Description automatically generated

* After inserting resources, the next step is to populate the **ComplianceReports** table.

A screenshot of a computer

Description automatically generated

* Next, we insert data into the **Incidents** table.

A close up of text

Description automatically generated

* The Next step is to populate the Behaviors Plans Table with data

A close up of text

Description automatically generated

* Finally, we insert data into the Staff Table

A computer screen shot of text

Description automatically generated

With the completion of the data insertion process across all tables, we now move to the **Verification Queries** step. This step ensures that the inserted data is accurate, complete, and adheres to the defined relationships and constraints. By executing targeted SELECT queries, we will validate the integrity of each table and confirm that all foreign key references are functioning as expected, setting the stage for confident use of the database in future operations. We will combine this step with the view procedures.

* 1. **Views and Stored Procedures**

Views and Stored Procedures are essential tools for efficient data management and retrieval. The views and stored will help us also verify and validate the integrity of each table and confirm that all foreign key references are functioning as expected.

The **Views** are virtual tables that provide a simplified, focused, and reusable way to query data from one or more tables. They are particularly useful for presenting complex “joins” queries or frequently used queries in a user-friendly format.

The **Stored Procedures**: These are precompiled SQL scripts stored in the database that perform specific tasks or queries. They enhance database functionality by reducing repetitive query writing, improving performance, and providing security through encapsulation.

This step will enhance data interaction, efficiency, and ensure consistency in accessing and processing information.

* **Views**

Views are virtual tables that simplify complex queries and provide a user-friendly way to access data. They are particularly useful for creating reports and managing information that is frequently accessed or retrieved. For instance, the following view script lists Clients and their current programs.

**A computer screen shot of a program

Description automatically generated**

The above script creates a view named “ClientProgramDetails”, which combines data from the Clients and Programs tables using an INNER JOIN on the CurrentProgramID and ProgramID columns. The view selects key columns, including the client's ID and name (ClientID, ClientName), the program's name and description (ProgramName, ProgramDescription), and the program's start and end dates. This view simplifies querying by providing a consolidated dataset of clients and their associated program details.

To retrieve all columns from the ClientProgramDetails view, we can retrieve all columns from the Clients table to view after it has been successfully created:

A close up of a word

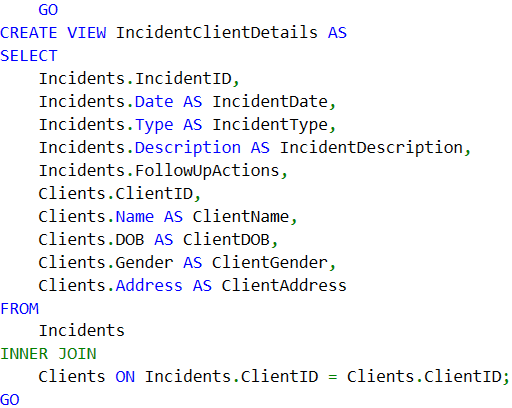
Description automatically generated

And we get the following output:

A screenshot of a computer program

Description automatically generated

* **View of the Incidents and Client Details**



* To retrieve all data from the view:

A screenshot of a computer

Description automatically generated

* **Behavior Plans and Client Details**

**A screen shot of a computer program

Description automatically generated**

* To retrieve all data from the view

**A close up of a text

Description automatically generated**

* The view Lists behavior plans for each client, including the plan creation date, details, and progressnotes.

**A computer screen shot of a program

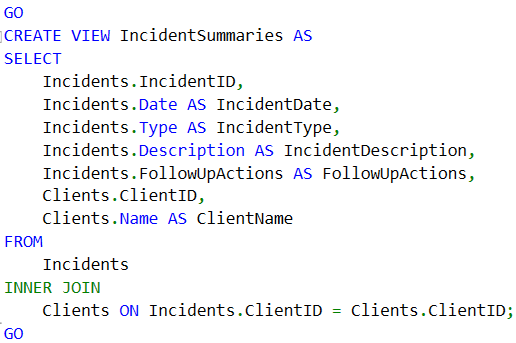
Description automatically generated**

****

A screenshot of a computer

Description automatically generated

Likewise, we create the view of all the other tables. To create the view for the View for Incident Summaries, we use the following script:

****

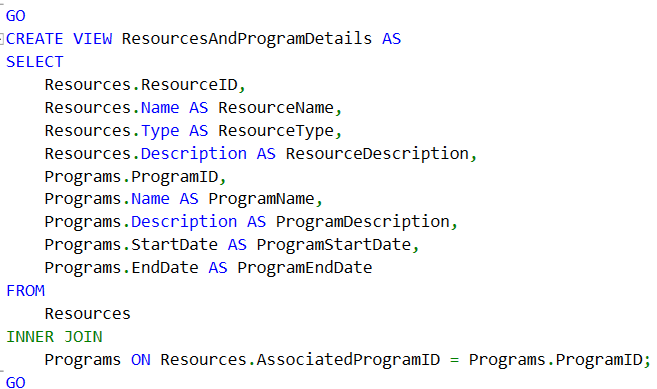
****

This view simplifies incident reporting by summarizing incident details by client.

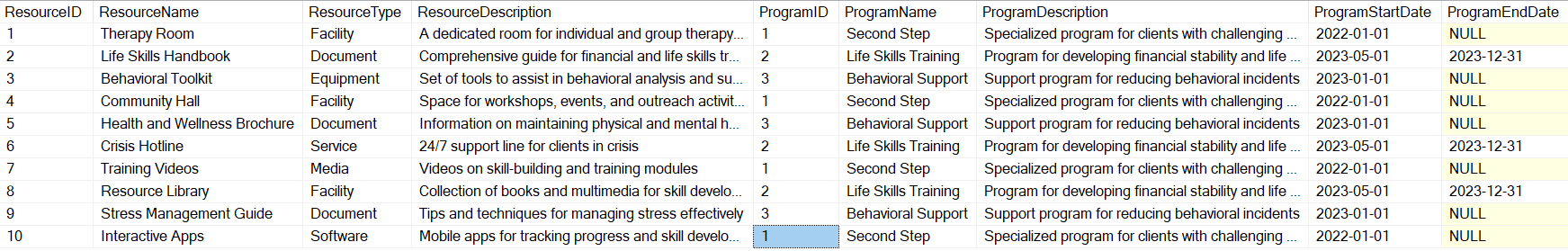
**A screenshot of a computer

Description automatically generated**

* **Resources and Program Details View**

****

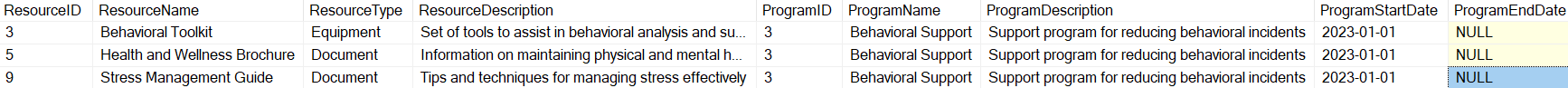




The view provides information about resources and the programs they are associated with, including resource type and descriptions

* To filter resources for a specific program:





This view makes it easy to analyze the relationship between resources and their associated programs

* **Compliance Reports and Program Details**

**A screenshot of a computer program

Description automatically generated**

This view provides a comprehensive dataset that shows compliance reports along with the details of the associated programs, making it easy to track compliance status across different programs.



A screenshot of a computer

Description automatically generated

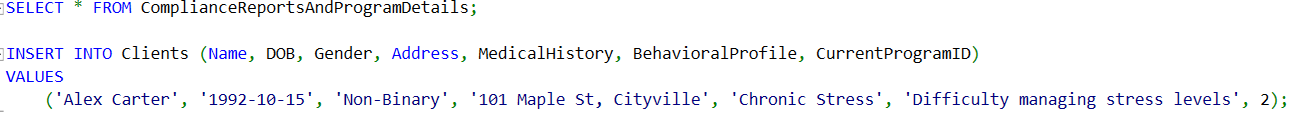
* 1. **Testing and Deployment**

Through the process of testing, we ensure that the database system meets all functional, performance, and security requirements. For that step, we perform a variety of tests should:

* Functional Testing**:** We verify that all tables, views, and stored procedures work as expected.
* We do the Test CRUD (Create, Read, Update, Delete) operations on each table.
* Validate relationships between tables, such as foreign key constraints.
* Test all views by running SELECT queries to ensure they return accurate data.

**5.4.1 A few series of Tests**

First, we insert a new client into the Clients table and assign them to an existing program using the CurrentProgramID.



To confirm the client has been added and assigned to the correct program:





To verify the program details assigned to the client:

A screenshot of a computer program

Description automatically generated

This test ensures the new client is successfully added and properly linked to an existing program via the CurrentProgramID.

A close-up of a sign

Description automatically generated

To test the Performance of the database, we run at least one complex query. For instance, we can run a query involving multiple joins and large datasets.

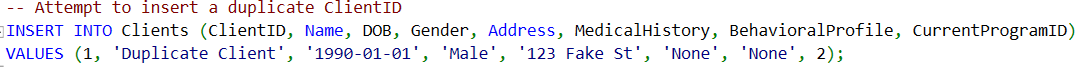
A screenshot of a computer program

Description automatically generated

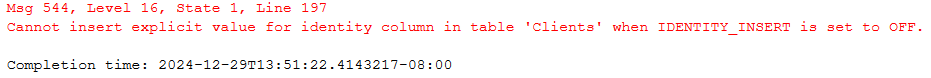
* **Data Integrity Testing**

Data integrity testing ensures that the database maintains accuracy, consistency, and validity of the data. The focus is to verify that the implemented constraints (like primary keys, foreign keys, unique constraints, and NOT NULL constraints) are working correctly and that no invalid data can be inserted.

* Primary Key Constraint Test

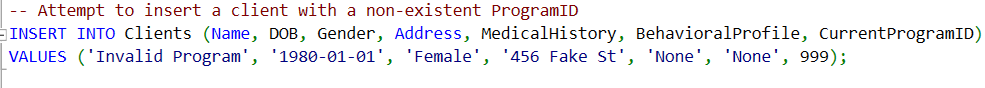


We obtain the following error message:

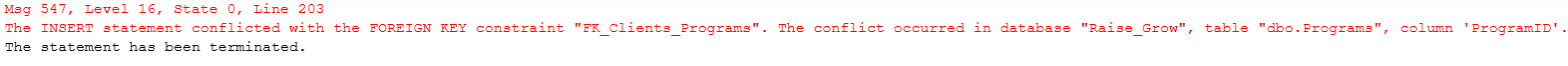


The error indicates that you are attempting to explicitly insert a value into an identity column (ClientID in this case) while the IDENTITY\_INSERT setting for the table is turned off. By default, the identity column automatically generates values, so explicit inserts are not allowed unless you temporarily enable IDENTITY\_INSERT.

* **Foreign Key Constraint Testing :** Ensure invalid CurrentProgramID values cannot be inserted into the Clients table.

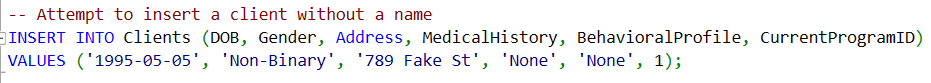
****

The output shows the following error:

****

* **NOT NULL Constraint Test**

This test verifies that NOT NULL columns cannot have null values.

****

The output shows the following error preventing duplicate entries.

****

* **Data Consistency Test**

This test checks for orphaned records (e.g., clients without a valid program).

A close-up of a white background

Description automatically generated

The query returns no results if referential integrity is maintained ( which is an expected output) 

* **Data Accuracy Test**

This test validates data with domain rules, such as DOB being in the past.

A white background with black text

Description automatically generated

The query returns no results as expected.



The purpose of these tests is to ensure that data remains accurate and consistent, constraints are properly implemented and enforced, and relationships between tables are valid. Conducting these tests after data insertion and periodically during database use helps maintain data integrity and ensures the database functions as intended.

* **Security Testing**

The first security test is to Validate “Role-Based Access Control”, that is, ensure that users have appropriate permissions based on their roles and are restricted from accessing unauthorized data or performing unauthorized actions. For this, we create or define roles with specific access rights. For example:

A screenshot of a computer program

Description automatically generated

Then, we assign or grant permission to each role on the database objects:

A screenshot of a computer screen

Description automatically generated

Then, we assign Roles to Users, that is, we add database users and assign them to roles:

A screenshot of a computer screen

Description automatically generated

We obtain the following error message

A close-up of a computer error

Description automatically generated

1. **Sample Problems Scenarios that could be addressed Using this Database**

The database is designed to address various operational challenges and support advanced analytics by providing structured, queryable data. By leveraging SQL's powerful features, the database enables users to extract valuable insights, automate processes, and enhance decision-making. This section presents real-world scenarios where SQL techniques such as joins, filtering, subqueries, and aggregate functions solve complex problems, demonstrating the database's utility in streamlining operations and driving data-driven outcomes.

* 1. **Monitoring Client Progress**

**Scenario**: Case managers need a summary of clients' behavioral progress, including details of their behavior plans and progress notes.

**Solution:** Use a query or view to consolidate information from the Clients and BehaviorPlans tables.

A computer screen shot of a program

Description automatically generated

**Usage:**

* To view all client progress summaries:

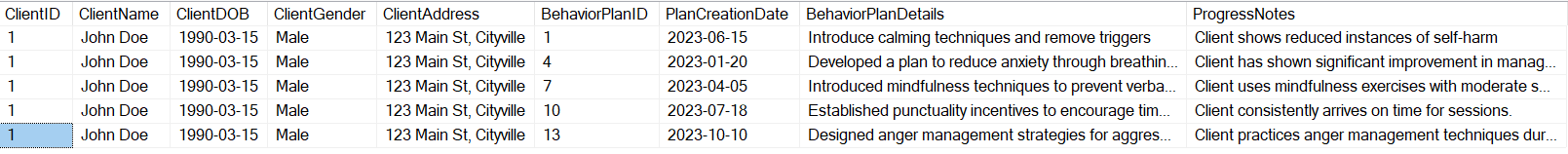


A screenshot of a computer

Description automatically generated

* To filter progress for a specific client:





* 1. **Incident Trends Analysis**

**Scenario:** The program manager needs to identify clients with frequent incidents and analyze common types of incidents across programs to improve interventions and resource allocation.

* **Query to analyze incident trends by type**

A screenshot of a computer code

Description automatically generated

A screenshot of a computer

Description automatically generated

* **Query to identify clients with frequent incidents**

A screen shot of a computer code

Description automatically generated

A screenshot of a computer

Description automatically generated

* 1. **Staff Certification Tracking**

**Scenario:** Ensure that staff members working in specialized programs (e.g., Second Step) hold the required certifications for their roles.

**Solution:**

We query the Staff and Programs tables to verify if certifications match program requirements by identifying missing or mismatched certifications

A computer code with text

Description automatically generated

**Explanation:**

* + **Joins**: The query joins the Staff table with the Programs table using the ProgramID foreign key to identify which staff members are assigned to which programs.
  + **Filter**: It filters for staff assigned to the specialized program Second Step.
  + It checks if staff certifications do not include the required certification (e.g., %Behavioral Therapy%).
  + **Columns**: Displays staff details (StaffID, Name, Role, Certifications) alongside their assigned program details (ProgramID, ProgramName, ProgramDescription).
  + **Ordering**: Results are ordered alphabetically by staff name for readability.



* 1. **Evaluating Program Effectiveness**

**Scenario:** Determine program effectiveness by evaluating client engagement. This involves analyzing the number of clients enrolled in each program and the frequency of incidents associated with those clients.

**Solution:**

Query the Clients, Programs, and Incidents tables to correlate program enrollment with the number of incidents, providing insights into client engagement and program outcomes.

A screen shot of a computer program

Description automatically generated

A white grid with black text

Description automatically generated

**Explanation:**

* **Joins**: LEFT JOIN between Programs and Clients to matches programs with the clients enrolled in them.
  + LEFT JOIN between Clients and Incidents to correlate clients with their associated incident reports.
  + **Aggregations**: COUNT(DISTINCT Clients.ClientID): Counts the total number of unique clients enrolled in each program.
  + COUNT(Incidents.IncidentID): Counts the total number of incidents reported for clients in each program.
  + **Grouping**: Groups the results by ProgramID and ProgramName to aggregate data for each program.
  + **Ordering**: Sorts programs by the highest number of clients (TotalClients DESC) and then by the lowest number of incidents (TotalIncidents ASC), emphasizing programs with high engagement and low issues.

Solution**:**

A screen shot of a computer code

Description automatically generated

**Explanation:**

* + **Joins**: Joins the ComplianceReports table with the Programs table using the ProgramID to associate each compliance report with the corresponding program.
  + **Selected Columns**: From Programs, includes ProgramID, ProgramName, and ProgramDescription to provide context for each program.
  + From ComplianceReports, includes ReportID, ReportDate, and ComplianceDetails to display detailed compliance information.
  + **Ordering**: Results are sorted by the most recent compliance reports (ReportDate DESC) and then alphabetically by program name (ProgramName).

The output is as follows:

A screenshot of a computer

Description automatically generated

* 1. **Resource Utilization**

**Scenario:**

Identify the resources most commonly associated with programs to analyze their utilization and ensure efficient allocation.

**Solution:**

We query the Resources and Programs tables to find associations between resources and programs, and count how often each resource is linked to programs.

A screenshot of a computer program

Description automatically generated

**Explanation:**

* + **Joins**: Combines the Resources table with the Programs table using AssociatedProgramID to determine which resources are associated with which programs.
  + **Aggregations**: Uses COUNT(Resources.AssociatedProgramID) to calculate how many times a resource is associated with programs.
  + **Grouping**: Groups the results by ResourceID, ResourceName, ResourceType, and program details (ProgramID, ProgramName) to aggregate data by resource and program association.
  + **Ordering**: Sorts the results by the number of program associations (ProgramAssociations DESC) and then alphabetically by resource name for clarity.
  1. **Tracking Client Incidents and Follow-Ups**

**Scenario:**

Caseworkers need a detailed history of incidents and the corresponding follow-up actions for a specific client to monitor progress and address recurring issues.

**Solution:**

We query the Incidents and Clients tables to retrieve incident records and follow-up actions for a specific client.

A screen shot of a computer program

Description automatically generated

**Explanation:**

* + **Joins**: Links the Incidents table with the Clients table using the ClientID foreign key to associate incidents with client details.
  + **Columns**: Displays client information (ClientID, Name, DOB, Gender) along with incident details (IncidentID, Date, Type, Description) and follow-up actions.
  + **Filters**: Filters the results to show data for a specific client by matching the client’s name in the WHERE clause.
  + **Ordering**: Sorts incidents in descending order of date (IncidentDate DESC) to show the most recent incidents first.
  1. **Tracking Client Incidents and Follow-Ups**

**Scenario:**

Caseworkers require a detailed history of incidents and the corresponding follow-up actions for a specific client to track progress and manage recurring issues effectively.

**Solution:**

Use the following query to retrieve incident records and follow-up actions for a specific client by filtering data from the Incidents and Clients tables.

A screen shot of a computer

Description automatically generated

**Explanation:**

* + **Joins**: The INNER JOIN links the Incidents table with the Clients table based on the ClientID, ensuring that each incident is associated with its corresponding client.
  + **Columns**: Includes client details like ClientID, Name, DOB, Gender, and Address.
  + Lists incident details such as IncidentID, IncidentDate, IncidentType, Description, and FollowUpActions.
  + **Filters**: The WHERE clause filters data to focus on a specific client by name (Clients.Name = 'John Doe'). This can be modified to filter by ClientID if needed.
  + **Ordering**: Sorts results by IncidentDate in descending order (DESC) to display the most recent incidents first.
  1. **Resource Allocation Efficiency**

**Scenario:**

Analyze how many resources are allocated to each program and identify potential gaps to improve resource allocation.

**Solution:**

Use a grouped query to count the number of resources allocated to each program, providing insights into resource distribution.

A screenshot of a computer program

Description automatically generated

**Explanation:**

* + **Joins**: A LEFT JOIN is used to include all programs, even those without allocated resources, ensuring potential gaps are identified.
  + **Aggregations**: COUNT(Resources.ResourceID): Counts the number of resources allocated to each program.
  + **Grouping**: Groups results by ProgramID and ProgramName to aggregate the resource count for each program.
  + **Ordering**: Results are sorted in ascending order of Total Resources allocated to highlight programs with fewer resources, and then alphabetically by program name.

A white grid with black text

Description automatically generated

* 1. **Behavior Plan Effectiveness**

**Scenario:**

Measure the effectiveness of behavior plans by evaluating the progress notes of all clients, highlighting significant improvements.

**Solution:**

Use the following SQL query to analyze progress notes for all clients and summarize improvements based on their behavior plans.

A computer screen shot of a program

Description automatically generated

**Explanation:**

* + **Joins**: The INNER JOIN links the Clients table with the BehaviorPlans table using ClientID to associate clients with their behavior plans.
  + **Filters**: The WHERE clause filters progress notes for keywords like "improve" or "success" to identify plans showing positive progress.
  + **Columns**: Includes client details (ClientID, Name, DOB, Gender) and behavior plan details (PlanID, PlanCreationDate, PlanDescription, ProgressSummary).
  + **Ordering**: Sorts results by the most recent plans (PlanCreationDate DESC) and then alphabetically by client name.
  1. **Incident and Resource Correlation**

**Scenario:**

Evaluate if specific resources (e.g., calming apps) correlate with a reduction in incidents by analyzing the relationship between resource allocation and client incidents.

**Solution:**

Combine data from the Resources and Incidents tables, linking them through associated programs and analyzing the number of incidents for programs using specific resources.

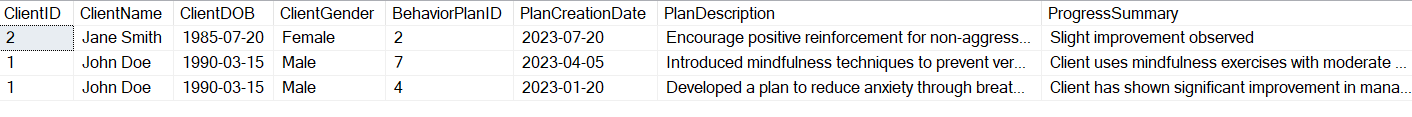
A screenshot of a computer program

Description automatically generated

**Explanation:**

* + **Joins**: Combines Resources and Programs using AssociatedProgramID to identify which resources are linked to each program.
  + Links Programs with Incidents to analyze incidents associated with the programs.
  + **Filters**: The WHERE clause filters for specific resources, such as calming apps, by using keywords (e.g., %Calming App%).
  + **Aggregations**: COUNT(Incidents.IncidentID) calculates the total number of incidents for each program using a specific resource.
  + **Grouping**: Groups results by ResourceID, ResourceName, ResourceType, and program details to provide insights into resource utilization and incident trends.
  + **Ordering**: Sorts by the total number of incidents in ascending order (TotalIncidents ASC) to highlight resources correlated with fewer incidents.

The output is displayed below:



These scenarios illustrate how the database can be leveraged to address real-world operational challenges, streamline decision-making, and improve service delivery.

* 1. **Identify Clients with No Incidents**

**Scenario:**

Case managers need to identify clients who have not been involved in any reported incidents, allowing them to focus resources on clients requiring more support.

**Solution:**

Use a query to find clients in the Clients table who do not have matching entries in the Incidents table.

A screen shot of a computer

Description automatically generated

**Explanation:**

* + **Joins**: LEFT JOIN connects Clients with Incidents to include all clients, even those without incident records.
  + LEFT JOIN links Clients with Programs to include program details for each client.
  + **Filters**: WHERE Incidents.ClientID IS NULL: Ensures only clients with no matching incidents in the Incidents table are included.
  + **Columns**: Includes client details (ClientID, Name, DOB, Gender, Address) and their associated program name for context.
  + **Ordering**: Sorts the results alphabetically by client name (Clients.Name).
  1. **Program Utilization Trends**

**Scenario:**

The administration needs to monitor the number of active clients enrolled in each program over time to track utilization trends and allocate resources effectively.

**Solution:**

Use a query to group clients by program and evaluate their enrollment trends over time.

A computer screen shot of a program code

Description automatically generated

**Explanation:**

* + **COUNT(Clients.ClientID)**: Counts the number of active clients in each program.
  + The alias ActiveClients is used for display but cannot be referenced in the ORDER BY clause directly.
  + **GROUP BY**: Groups the data by program details to calculate the number of clients per program.
  + **ORDER BY**: Orders the results by the count of active clients (COUNT(Clients.ClientID) DESC), showing the most utilized programs first.

A screenshot of a computer

Description automatically generated

* 1. **Staffing Gaps**

**Scenario:**

Identify programs with insufficient staff coverage by analyzing the client-to-staff ratio for each program. This helps in reallocating staff to ensure adequate coverage.

**Solution:**

Use a query to compare the number of clients and staff assigned to each program, highlighting programs with low staff-to-client ratios.

A computer screen shot of a program code

Description automatically generated

**Explanation:**

* + **Joins**: LEFT JOIN links Programs to Clients and Staff, ensuring all programs are included, even those without assigned clients or staff.
  + **Aggregations**: COUNT(DISTINCT Clients.ClientID): Counts the number of unique clients enrolled in each program.
  + COUNT(DISTINCT Staff.StaffID): Counts the number of unique staff members assigned to each program.
  + **Coverage Status**: Uses a CASE statement to categorize programs:
    - 'No Staff Assigned': Programs with zero staff.
    - 'Insufficient Staff': Programs with a client-to-staff ratio exceeding 10.
    - 'Adequate Staff': Programs with sufficient staff coverage.
  + **NULLIF**: Prevents division by zero by replacing zero staff count with NULL.
  + **Ordering**: Sorts by CoverageStatus (to prioritize gaps) and then by the number of clients (TotalClients).

The output is as follows:

A screenshot of a computer

Description automatically generated

* 1. **High-Need Clients**

**Scenario:**

Identify clients with frequent incidents to prioritize resource allocation and targeted interventions.

**Solution:**

Use a query to count the number of incidents reported for each client and identify those with the highest frequency of occurrences.

**A screen shot of a computer code

Description automatically generated**

**Explanation:**

* + **Joins**: LEFT JOIN links the Clients table with the Incidents table, ensuring all clients are included in the analysis, even those with no incidents.
  + **Aggregations**: COUNT(Incidents.IncidentID): Counts the number of incidents associated with each client.
  + **Grouping**: Groups data by ClientID, ClientName, DOB, and Gender to calculate incident counts for each client.
  + **HAVING Clause**: Filters the results to show only clients with more than 5 incidents (adjustable threshold).
  + **Ordering**: Sorts the results by the total number of incidents in descending order, prioritizing clients with the most incidents.

The output as follows:



* 1. **Resource Utilization by Client**

**Scenario:**

Track which resources are most frequently used by specific clients within their assigned programs to evaluate individual resource engagement.

**Solution:**

Use a query to join the Resources table with the Clients table via their assigned programs, identifying resources utilized by each client.

A screenshot of a computer program

Description automatically generated

**Explanation:**

* + **Joins**: INNER JOIN connects Clients to Programs using CurrentProgramID to link clients to their assigned programs.
  + Another INNER JOIN links Programs to Resources via AssociatedProgramID to identify resources used in each program.
  + **Selected Columns**: Includes client details (ClientID, ClientName, DOB, Gender).
  + Includes program details (ProgramID, ProgramName) to identify the assigned program for each client.
  + Includes resource details (ResourceID, ResourceName, ResourceType, ResourceDescription) to show which resources are available to the client.
  + **Ordering**: Results are sorted alphabetically by client name and then by resource name for clarity.

A portion of output is as follows:

A screenshot of a computer

Description automatically generated

* 1. **Longest Tenure in Programs**

**Scenario:**

Identify the clients who have been enrolled in their current programs for the longest duration to recognize loyalty or evaluate long-term outcomes.

**Solution:**

Use a query to calculate the duration of enrollment for each client in their current program and sort by the longest duration.

A computer code with text

Description automatically generated with medium confidence

**Explanation:**

* + **Joins**: INNER JOIN connects Clients to Programs using CurrentProgramID to associate clients with their current programs.
  + **Selected Columns**: Includes client details (ClientID, ClientName).
  + Includes program details (ProgramID, ProgramName, ProgramStartDate) to provide context about the assigned program.
  + Calculates the enrollment duration (EnrollmentDurationInDays) using DATEDIFF, which measures the number of days between the program's start date and the current date.
  + **Ordering**: Sorts results in descending order of EnrollmentDurationInDays to display the longest-tenured clients first.

A screenshot of a program

Description automatically generated

* 1. **Compliance Monitoring**

**Scenario:**

Identify programs with overdue compliance reports by finding programs where the last compliance report was generated over a year ago. This helps ensure timely updates and adherence to compliance standards.

**Solution:**

Use a query to compare the most recent compliance report date for each program with the current date and flag overdue reports.

A screenshot of a computer program

Description automatically generated

**Explanation:**

* + **Joins**: LEFT JOIN connects Programs to ComplianceReports to include all programs, even those without recent reports.
  + **Aggregations**: MAX(ComplianceReports.DateGenerated): Finds the most recent compliance report date for each program.
  + **Calculated Columns**: DATEDIFF(DAY, MAX(ComplianceReports.DateGenerated), GETDATE()): Calculates the number of days since the most recent report.
  + **Filters**: The HAVING clause filters programs where the last compliance report was generated more than 365 days ago (overdue reports).
  + **Grouping**: Groups results by ProgramID and ProgramName to aggregate data for each program.
  + **Ordering**: Sorts results by the number of days since the last report in descending order, prioritizing the most overdue programs.

A screenshot of a computer

Description automatically generated

* 1. **Clients by Gender**

**Scenario:**

Analyze the distribution of clients based on gender to identify demographic patterns and guide program planning or resource allocation.

**Solution:**

Group clients by gender and count the total number of clients in each group.

A screen shot of a computer code

Description automatically generated

**Explanation:**

* + **Grouping**: Groups the clients by the Gender column to calculate the distribution of clients across different genders.
  + **Aggregations**: COUNT(ClientID): Counts the total number of clients in each gender group.
  + **Ordering**: Sorts the results by the total number of clients in descending order (TotalClients DESC) to display the largest groups first.
  + **Columns**: ClientGender: Shows the gender category (e.g., Male, Female, Non-Binary).
  + TotalClients: Displays the number of clients in each gender group.

A screenshot of a computer

Description automatically generated

* 1. **Behavioral Trends by Program**

**Scenario:**

Analyze trends in behavior plans for each program to understand which interventions are commonly applied and their frequency.

**Solution:**

Join the BehaviorPlans table with the Programs table to group behavior plans by program and count the number of plans applied to each program.

A screenshot of a computer program

Description automatically generated

**Explanation:**

* + **Data Type Conversion**: CAST(BehaviorPlans.PlanDetails AS VARCHAR(MAX)): Converts the TEXT column PlanDetails to a compatible VARCHAR(MAX) type for use with STRING\_AGG.
  + **Functionality**: STRING\_AGG: Concatenates all behavior plan details for each program into a single string, separated by ;.
  + **Grouping**: Groups by ProgramID and ProgramName to aggregate behavior plan trends for each program.
  + **Ordering**: Sorts results by the number of behavior plans in descending order (TotalBehaviorPlans DESC).

The output:

A screenshot of a white grid

Description automatically generated

* 1. **Client Overlap Between Programs**

**Scenario:**

Identify clients who have participated in multiple programs over time to evaluate cross-program engagement or transitions.

**Solution:**

Query the Clients table to detect client IDs that appear under multiple CurrentProgramID values by grouping and counting unique program associations.

A screenshot of a computer code

Description automatically generated

**Explanation:**

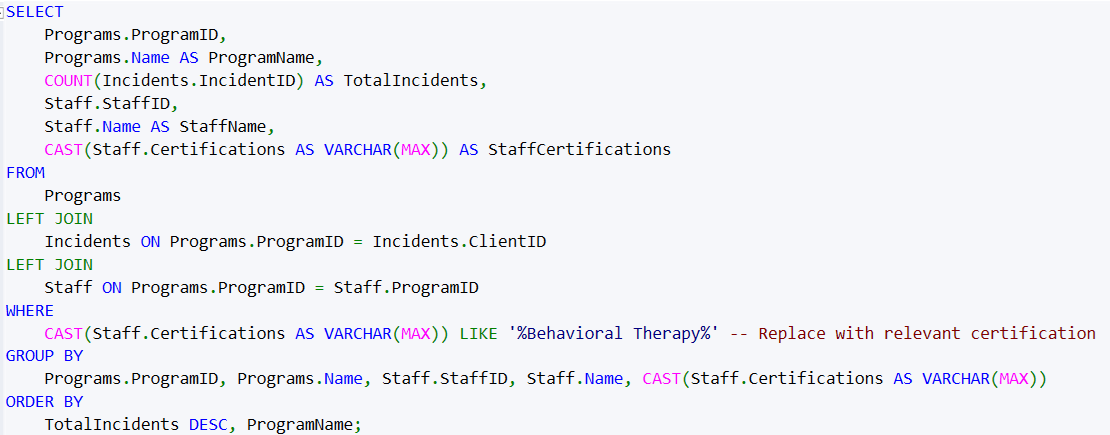
* + **Grouping**: Groups clients by ClientID and ClientName to aggregate their program associations.
  + **Aggregations**: COUNT(DISTINCT Clients.CurrentProgramID): Counts the number of unique programs each client has participated in.
  + **HAVING Clause**: Filters results to include only clients who have been associated with more than one program (COUNT(DISTINCT Clients.CurrentProgramID) > 1).
  + **Ordering**: Sorts results by the total number of programs in descending order (TotalPrograms DESC) and then alphabetically by client name.
  1. **Specialized Training Effectiveness**

**Scenario:**

Evaluate whether staff with specialized training are assigned to programs with higher needs by analyzing staff certifications and incident rates in their assigned programs.

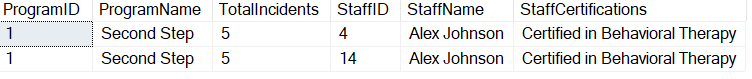
**Solution:**

Query the Staff, Programs, and Incidents tables to compare staff certifications with the incident rates of the programs they are assigned to.



**Explanation:**

* + **Data Type Conversion**: CAST(Staff.Certifications AS VARCHAR(MAX)): Converts the TEXT column to VARCHAR(MAX), allowing for operations like LIKE or comparisons.
  + **Filter**: WHERE CAST(Staff.Certifications AS VARCHAR(MAX)) LIKE '%Behavioral Therapy%': Filters for staff with certifications relevant to program needs.
  + **Grouping**: Groups by program and staff details to aggregate incident counts and evaluate alignment.
  + **Ordering**: Sorts by the number of incidents (TotalIncidents DESC) and then by program name.



* 1. **Clients and Their Assigned Staff**

**Scenario Description:**

The management wants a report that lists all clients along with the staff members assigned to their respective programs. This report evaluates whether clients are adequately supported by program staff.

**Solution Using INNER JOIN:**

We will use an INNER JOIN to combine the Clients, Staff, and Programs tables. The linking is performed through:

* CurrentProgramID in the Clients table.
* ProgramID in the Staff table.

A portion of the output is:

A computer screen shot of a program

Description automatically generated

**Breakdown of the Query:**

1. **Selected Columns**:
   * **From Clients Table**:
     + ClientID, Name (aliased as ClientName), DOB, and Gender provide key demographic information about each client.
   * **From Programs Table**:
     + ProgramID and Name (aliased as ProgramName) identify the program to which each client is assigned.
   * **From Staff Table**:
     + StaffID, Name (aliased as StaffName), and Role provide details about the staff members supporting each program.
2. **Joins**:
   * **INNER JOIN Clients and Programs**:
     + Links the Clients table with the Programs table using CurrentProgramID in Clients and ProgramID in Programs. This ensures each client is matched to their assigned program.
   * **INNER JOIN Programs and Staff**:
     + Links the Programs table with the Staff table using ProgramID in both tables. This ensures each program is matched to its assigned staff members.
3. **Ordering**:
   * **Programs.Name**: Groups and sorts results by program name.
   * **Clients.Name**: Sorts clients alphabetically within each program.
   * **Staff.Name**: Sorts staff members alphabetically within each program-client group.
4. **Aliases**:
   * Columns are aliased for better readability in the result set (e.g., Programs.Name AS ProgramName).

A screenshot of a computer

Description automatically generated

* 1. **Scenario: Clients Without Incidents**

**Scenario:**

Management requires a report that lists all clients, including those without any recorded incidents. This report can help identify clients who may need less intervention or those for whom incidents might not have been properly logged.

**Solution Using LEFT JOIN:**

A LEFT JOIN is used to combine the Clients table with the Incidents table, ensuring that all clients are included in the result, even if they do not have matching records in the Incidents table.

A computer screen shot of a computer code

Description automatically generated

**Explanation of the Query:**

1. **Joins**:
   * **LEFT JOIN Clients and Programs**:
     + Links the Clients table with the Programs table via CurrentProgramID, ensuring all clients are associated with their respective programs.
   * **LEFT JOIN Clients and Incidents**:
     + Links the Clients table with the Incidents table via ClientID, ensuring all clients are included even if they have no recorded incidents.
2. **Selected Columns**:
   * **From Clients**:
     + Includes demographic and contact details (ClientID, Name, DOB, Gender, Address).
   * **From Programs**:
     + Includes the name of the program (ProgramName) to which the client is assigned.
   * **From Incidents**:
     + Uses COUNT(Incidents.IncidentID) to calculate the total number of incidents for each client. Clients without incidents will have a count of 0.
3. **Grouping**:
   * Groups results by all Clients and Programs columns to aggregate incident counts for each client.
4. **Ordering**:
   * Sorts results by TotalIncidents in ascending order to prioritize clients with no incidents, and then alphabetically by client name.
   1. **Scenario: Programs Without Assigned Clients**

Management seeks to identify programs that currently have no enrolled clients. This allows them to pinpoint underutilized programs that may require reevaluation, additional resources, or potential closure.

**Solution Using RIGHT JOIN:**

A RIGHT JOIN is used to ensure that all programs are included in the results, even if there are no clients assigned to them. Programs without assigned clients will appear with NULL values for client details.

A computer screen shot of a program

Description automatically generated

**Explanation of the Query:**

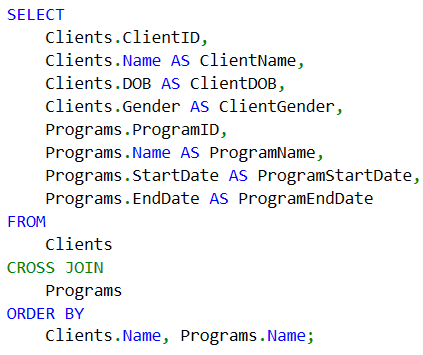
1. **Joins**:
   * **LEFT JOIN**:
     + Combines the Programs table with the Clients table using ProgramID in Programs and CurrentProgramID in Clients.
     + Ensures all programs are included, even if no clients are enrolled.
2. **Columns**:
   * **From Programs**:
     + Includes ProgramID, Name (aliased as ProgramName), StartDate, and EndDate to provide key program details.
   * **From Clients**:
     + COUNT(Clients.ClientID) calculates the total number of clients assigned to each program. Programs with no clients will have a count of 0.
3. **Grouping**:
   * Groups results by program details (ProgramID, Name, StartDate, EndDate) to aggregate client counts for each program.
4. **Filter**:
   * **HAVING COUNT(Clients.ClientID) = 0**:
     + Filters results to include only programs with no enrolled clients.
5. **Ordering**:
   * Sorts the results alphabetically by ProgramName for clarity.
   1. **Scenario: Generating All Possible Client-Program Combinations**

**Scenario Description:**

Management needs to generate all possible client-to-program assignments to simulate enrollment scenarios or analyze compatibility for future assignments. This helps in planning and optimizing resource allocation.

**Solution Using CROSS JOIN:**

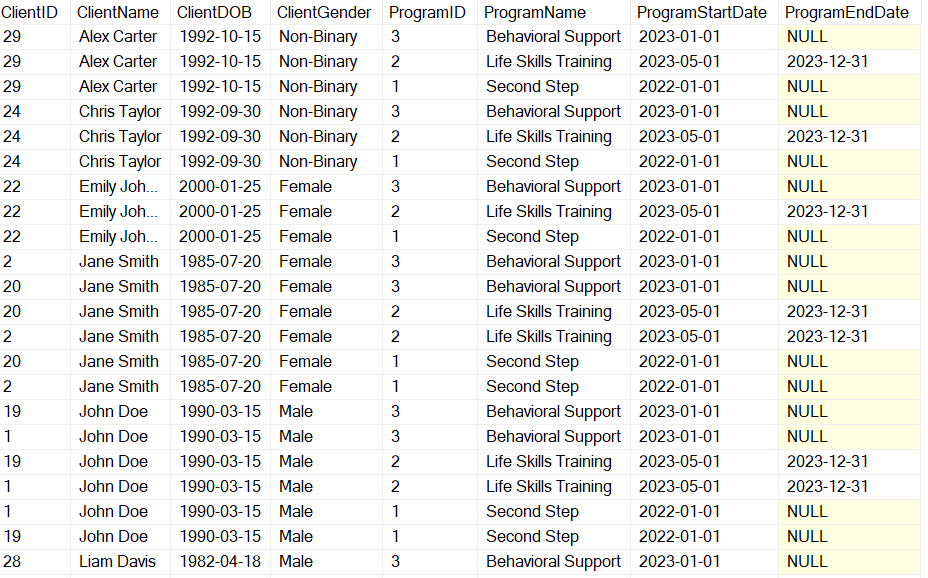
A CROSS JOIN combines each row from the Clients’ table with every row from the Programs table, producing all possible combinations of clients and programs.



**Explanation of the Query:**

1. **CROSS JOIN**:
   * Combines every row from the Clients table with every row from the Programs table.
   * Produces a Cartesian product, resulting in all possible client-program combinations.
2. **Selected Columns**:
   * **From Clients**:
     + Includes ClientID, Name (aliased as ClientName), DOB, and Gender to provide client details.
   * **From Programs**:
     + Includes ProgramID, Name (aliased as ProgramName), StartDate, and EndDate to provide program details.
3. **Ordering**:
   * Sorts results alphabetically by ClientName and then by ProgramName for better readability.

A part of the outcome is:



* 1. **Scenario: Identifying Clients Without Programs and Programs Without Clients**

**Scenario Description:**

Management wants to:

1. Identify clients who are not currently assigned to any program.
2. Identify programs that currently have no clients enrolled. This ensures no client is overlooked, and no program is underutilized.

**Solution Using FULL OUTER JOIN:**

A FULL OUTER JOIN combines all rows from the Clients and Programs tables, showing matches where they exist and NULL values for unmatched rows in either table.



**Explanation of the Query:**

1. **FULL OUTER JOIN**:
   * Combines all rows from the Clients and Programs tables.
   * Rows with matches are included, and NULL values are shown for unmatched rows.
2. **Filter**:
   * **WHERE Clients.CurrentProgramID IS NULL**:
     + Identifies programs without clients.
   * **WHERE Programs.ProgramID IS NULL**:
     + Identifies clients without programs.
3. **Selected Columns**:
   * + **From Clients**: Includes ClientID, Name (aliased as ClientName), DOB, and Gender.
     + **From Programs**: Includes ProgramID, Name (aliased as ProgramName), StartDate, and EndDate.
4. **Ordering**:
   * Sorts results alphabetically by ClientName and ProgramName for clarity.

The output is :



* 1. **Program Enrollment**

**Scenario Description:**

Management seeks to know how many clients are enrolled in each program to assess program popularity and workload distribution.

**SQL Query for Program Enrollment**:

A computer screen shot of a program code

Description automatically generated

**Explanation:**

* Counts the number of clients (COUNT(Clients.ClientID)) enrolled in each program.
* Groups by ProgramID and ProgramName to aggregate the client count per program.
* Includes programs with zero clients using LEFT JOIN.

A screenshot of a computer

Description automatically generated

**(i). Count Total Incidents for Each Client**:

A screenshot of a computer code

Description automatically generated

**Explanation:**

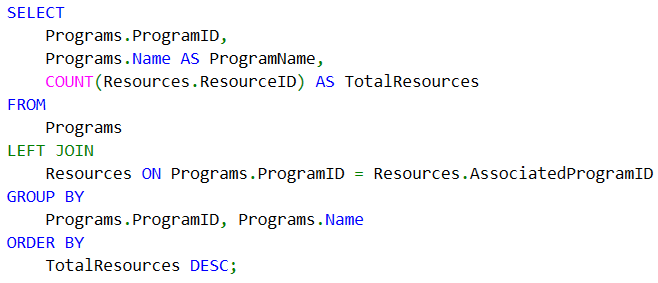
* Counts the total number of incidents (COUNT(Incidents.IncidentID)) per client.
* Groups by ClientID and ClientName to summarize incidents for each client.
* Includes clients without incidents using LEFT JOIN.

The output is

A table with numbers and letters

Description automatically generated

**(ii). Count Clients by Gender:**



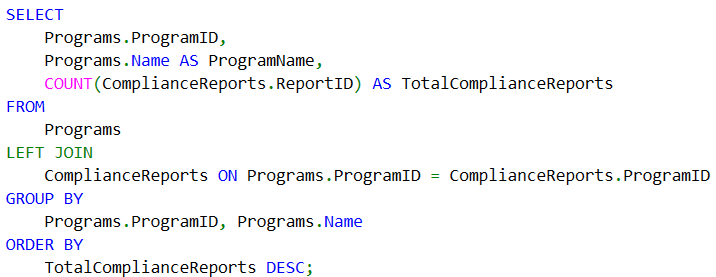
**Explanation:**

* Counts the total number of resources (COUNT(Resources.ResourceID)) assigned to each program.
* Groups by ProgramID and ProgramName to aggregate resource counts.
* Includes programs without resources using LEFT JOIN

A screenshot of a phone

Description automatically generated

**(iii). Count Compliance Reports Per Program**:



**Explanation:**

* Counts the total number of compliance reports (COUNT(ComplianceReports.ReportID)) per program.
* Groups by ProgramID and ProgramName to summarize reports for each program.
* Includes programs without compliance reports using LEFT JOIN.

A screenshot of a computer

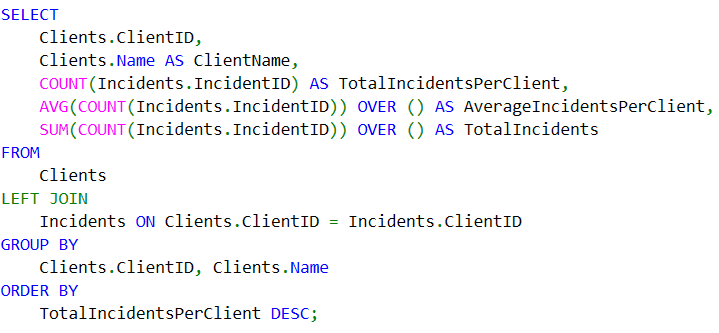
Description automatically generated

* 1. **Calculating the Average Number of Incidents Per Client and Total Incidents**

**Scenario Description:**

Management needs to analyze the average number of incidents per client and the total number of incidents across all clients. This provides insights into overall behavioral trends and client-specific patterns.

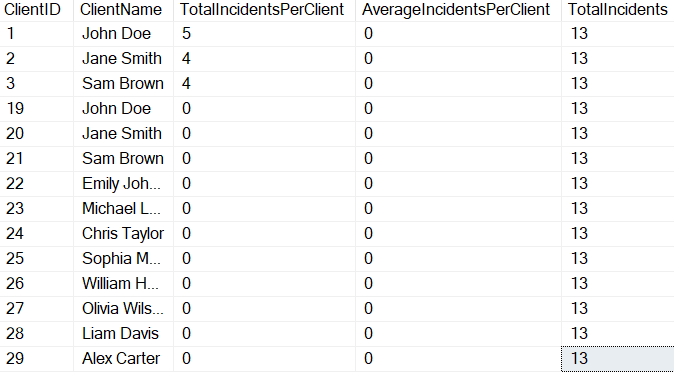
**Solution**:



**Explanation of the Query:**

1. **Columns**:
   * Clients.ClientID, Clients.Name: Identifies each client.
   * COUNT(Incidents.IncidentID): Counts the total number of incidents for each client.
   * AVG(COUNT(Incidents.IncidentID)) OVER (): Calculates the average number of incidents per client using a window function.
   * SUM(COUNT(Incidents.IncidentID)) OVER (): Computes the total number of incidents across all clients.
2. **LEFT JOIN**:
   * Ensures all clients are included, even if they have no incidents.
3. **Grouping**:
   * Groups by ClientID and ClientName to calculate the incident count for each client.
4. **Ordering**:

* Sorts by TotalIncidentsPerClient in descending order to prioritize clients with the most incidents.



**Alternative Query for Overall Averages and Totals:**

A screen shot of a computer code

Description automatically generated

**Explanation of Alternative Query:**

1. **Subquery**:
   * Calculates TotalIncidents for each client in the subquery.
2. **Outer Query**:
   * Uses AVG to compute the average number of incidents per client.
   * Uses SUM to compute the total number of incidents across all clients.

A close up of a number

Description automatically generated

* 1. **Scenario: Calculate Total and Average Resources Assigned to Programs**

**Scenario Description:**

Management needs to evaluate resource distribution by calculating the total number of resources assigned to programs and the average number of resources per program. This helps identify under-resourced or over-resourced programs.

**Solution:**

**A screenshot of a computer program

Description automatically generated**

* + **Programs.ProgramID, Programs.Name**: Identifies each program and its name.
  + **COUNT(Resources.ResourceID)**: Counts the total number of resources assigned to each program.
  + **AVG(COUNT(Resources.ResourceID)) OVER ()**: Calculates the average number of resources assigned across all programs using a window function.
  + **SUM(COUNT(Resources.ResourceID)) OVER ()**: Computes the total number of resources assigned across all programs.
* **LEFT JOIN**: Ensures all programs are included in the analysis, even if they have no resources assigned.
* **Grouping**: Groups by ProgramID and ProgramName to aggregate resource counts for each program.
* **Ordering**: Sorts results by TotalResourcesAssigned in descending order to highlight programs with the highest number of resources.

The output is as follows:

A screenshot of a computer

Description automatically generated

**Alternative Query for Overall Metrics Only:**

Suppose we need only the total and average resources without program-specific details:

A screenshot of a computer program

Description automatically generated

The output is:



* 1. **Scenario: Summing Incident Follow-Ups**

**Scenario Description:**

Management seeks to analyze the total number of follow-up actions taken for incidents across all clients to assess responsiveness and intervention efforts.

**Solution:**

**A close-up of a white background

Description automatically generated**

**Explanation:**

* + **CAST(FollowUpActions AS VARCHAR(MAX))**: Converts the TEXT column FollowUpActions to VARCHAR(MAX), which is compatible with the COUNT function.
  + **COUNT(IncidentID)**: Counts the total number of incidents in the Incidents table.
  + **COUNT(CAST(FollowUpActions AS VARCHAR(MAX)))**: Counts the number of non-NULL FollowUpActions entries after converting them to VARCHAR(MAX).

**Alternative Query:**

We can simply Count Non-NULL Follow-Up Actions without casting

A close-up of a white background

Description automatically generated

* 1. **Comprehensive Report on Client and Program Metrics Using Aggregate Functions**

**Scenario Description:**

Management requires a detailed summary report for each program, including:

1. Total number of clients enrolled.
2. Average number of incidents per client.
3. Maximum and minimum incidents counts among clients.
4. Total resources assigned to each program.

This report combines multiple aggregate functions (COUNT, AVG, MIN, MAX, SUM) to analyze program utilization and client behavior.

**Solution:**

A screenshot of a computer program

Description automatically generated

The output is the following:

A screenshot of a computer

Description automatically generated

**Explanation:**

* + **Aggregate Functions**: **COUNT(DISTINCT Clients.ClientID)**:
    - Counts the total number of unique clients enrolled in each program.
    - **AVG(ClientIncidents.TotalIncidents)**: Calculates the average number of incidents per client within each program.
    - **MAX(ClientIncidents.TotalIncidents)**: Identifies the client with the maximum number of incidents in the program.
    - **MIN(ClientIncidents.TotalIncidents)**: Identifies the client with the minimum number of incidents in the program.
    - **COUNT(Resources.ResourceID)**: Counts the total number of resources assigned to each program.
    - **Joins**: **Programs to Clients (LEFT JOIN)**: Links programs to their enrolled clients using ProgramID and CurrentProgramID.
    - **Programs to Resources (LEFT JOIN)**: Links programs to their assigned resources using ProgramID and AssociatedProgramID.
  + **Clients to Incidents (LEFT JOIN with Subquery)**:
    - Aggregates incident counts per client in a subquery and links it to clients.
  + **Subquery**: The subquery calculates the total number of incidents (COUNT(IncidentID)) for each client and groups by ClientID.
  + **Grouping**: Groups by Programs.ProgramID and Programs.Name to aggregate data for each program.
  + **Ordering**: Sorts results alphabetically by ProgramName.
  1. **Scenario: Incident and Follow-Up Analysis**

**Scenario Description:**

Management requires an analysis of all incidents to:

1. Determine the total number of incidents.
2. Calculate the average time taken for follow-up actions.
3. Identify the earliest and latest incident dates.

**Solution :**

A computer screen shot of text

Description automatically generated

**Explanation:**

* + **Conversion to VARCHAR(MAX)**: **CAST(FollowUpActions AS VARCHAR(MAX))**: Converts the TEXT column to VARCHAR(MAX) to make it compatible with TRY\_CAST.
  + **TRY\_CAST**: **TRY\_CAST(... AS DATETIME)**: Attempts to convert the VARCHAR(MAX) value to DATETIME. Invalid values are treated as NULL.
  + **CASE Statement**: Ensures that only valid follow-up dates are included in the DATEDIFF calculation.
  + **Aggregate Functions**: **AVG**: Calculates the average time in days between the incident date (Date) and valid follow-up actions.
  + **COUNT**: Counts the total number of incidents.
  + **MIN and MAX**: Identify the earliest and latest incident dates.

The output is



* 1. **Scenario: Ranking Clients by the Number of Incidents Within Each Program**

**Scenario Description:**

Management requires a report to identify the most incident-prone clients within each program. Clients should be ranked based on the number of incidents they have within their respective programs.

A screenshot of a computer code

Description automatically generated

**Explanation:**

* + **Joins**: **LEFT JOIN Programs and Clients**:
    - Links the Programs table to the Clients table using ProgramID and CurrentProgramID.
    - **LEFT JOIN Clients and Incidents**: Links the Clients table to the Incidents table using ClientID.
    - **Selected Columns**: **From Programs**: Includes ProgramID and Name to identify the program.
    - **From Clients**: Includes ClientID and Name to identify the client.
    - **From Incidents**: Counts the number of incidents for each client using COUNT(Incidents.IncidentID).
  + **RANK() Function**: **PARTITION BY Programs.ProgramID**:
    - Groups the ranking by program, so clients are ranked within their respective programs.
    - **ORDER BY COUNT(Incidents.IncidentID) DESC**: Orders the clients within each program by the number of incidents in descending order.
  + **Grouping**: Groups by Programs.ProgramID, Programs.Name, Clients.ClientID, and Clients.Name to calculate incident counts per client.
  + **Ordering**: Sorts by Programs.ProgramID and IncidentRank for better readability.

A portion of the output is as follows:

A screenshot of a table

Description automatically generated

* 1. **Scenario: Cumulative Incident Count by Date**

**Scenario Description:**

Management needs to analyze the cumulative number of incidents over time to understand trends and identify peaks in incident frequency.

A computer code with text

Description automatically generated with medium confidence

**Explanation:**

* + **COUNT(IncidentID)**: Counts the total number of incidents for each date.
  + **SUM(COUNT(IncidentID)) OVER (ORDER BY Date)**: Computes the cumulative sum of incidents up to each date using a window function.
  + **GROUP BY Date**: Groups incidents by their occurrence date.
  + **Ordering**: Sorts the results by Date to show trends over time.

A white table with black text

Description automatically generated

**Scenario: Percentile of Resources Assigned to Each Program**

**Scenario Description:**

Management seeks to understand how resource allocation compares across programs by calculating percentiles for the number of resources assigned to each program.

* 1. **Scenario: Average Number of Incidents Per Client Across Programs**

**Scenario Description:**

Management needs a report that calculates:

1. The average number of incidents per client for each program.
2. Each client’s incident count compared to the program’s average.

This helps identify clients with unusually high or low incident counts relative to their program peers.

**Solution:**

A screenshot of a computer code

Description automatically generated

**Explanation:**

* + **Joins**: **LEFT JOIN Programs and Clients**:
    - Links the Programs table to the Clients table using ProgramID and CurrentProgramID.
    - **LEFT JOIN Clients and Incidents**: Links the Clients table to the Incidents table using ClientID.
  + **Aggregate Functions**: **COUNT(Incidents.IncidentID)**:
    - Counts the number of incidents for each client.
  + **AVG(COUNT(Incidents.IncidentID)) OVER (PARTITION BY Programs.ProgramID)**:
    - Calculates the average number of incidents per client for each program using a window function.
  + **CASE Statement**: Compares each client’s incident count to the program’s average:
    - **Above Average**: If the client’s incident count is greater than the program’s average.
    - **Average**: If the client’s incident count equals the program’s average.
    - **Below Average**: If the client’s incident count is less than the program’s average.
  + **Grouping**: Groups by ProgramID, ProgramName, ClientID, and ClientName to calculate the client-level and program-level metrics.
  + **Ordering**: Sorts the results by ProgramID and ClientID for clarity.

A portion of the output is:

A screenshot of a computer

Description automatically generated

* 1. **Categorizing Clients Based on Incident Count**

**Scenario Description:**

Management requires a classification of clients into risk categories based on their total number of incidents:

* **Low-Risk**: 0 incidents.
* **Moderate-Risk**: 1–2 incidents.
* **High-Risk**: More than 2 incidents.

This helps prioritize clients for interventions and resource allocation.

A screenshot of a computer code

Description automatically generated

**Explanation:**

* + - **Joins**: **LEFT JOIN**: Ensures all clients are included, even if they have no incidents.
  + **COUNT(Incidents.IncidentID)**: Counts the total number of incidents associated with each client.
  + **CASE Statement**: Implements conditional logic to classify clients:
    - **Low-Risk**: Clients with 0 incidents.
    - **Moderate-Risk**: Clients with 1–2 incidents.
    - **High-Risk**: Clients with more than 2 incidents.
  + **Grouping**: Groups by ClientID and ClientName to calculate incident counts and classify each client.
  + **Ordering**: Sorts by RiskCategory and then by TotalIncidents in descending order for easier prioritization.

A portion of the output is as follows:

A table with numbers and text

Description automatically generated

* 1. **Flagging Programs with Insufficient Resources**

**Scenario Description:**

Management wants to identify programs with fewer than two resources and flag them as "Under-Resourced."

A screenshot of a computer program code

Description automatically generated

**Explanation:**

* + **Joins**: **LEFT JOIN** ensures that all programs are included, even if they have no resources assigned.
  + **COUNT(Resources.ResourceID)**: Counts the total number of resources assigned to each program.
  + **CASE Statement**: Flags programs with fewer than two resources as **Under-Resourced**.
  + Flags others as **Sufficiently Resourced**.
  + **Grouping**: Groups by ProgramID and ProgramName to calculate resource counts per program.
  + **Ordering**: Sorts by ResourceStatus and TotalResources in ascending order for clarity.

A screenshot of a graph

Description automatically generated

* 1. **Flagging Clients Who Have Not Updated Behavior Plans**

**Scenario Description:**

Identify clients whose behavior plans were created more than 90 days ago and flag them as "Outdated."

A computer screen shot of a program

Description automatically generated

**Explanation:**

* + **Joins**: **LEFT JOIN** ensures that all clients are included, even if they have no behavior plans.
  + **DATEDIFF(DAY, BehaviorPlans.DateCreated, GETDATE())**: Calculates the number of days since the behavior plan was created.
  + **CASE Statement**: Flags plans created more than 90 days ago as **Outdated**.
  + Flags others as **Up-to-Date**.
  + **Filter**: **WHERE BehaviorPlans.DateCreated IS NOT NULL** ensures only clients with behavior plans are included.
  + **Ordering**: Sorts by PlanStatus and PlanCreationDate for clarity.

The output is

A screenshot of a table

Description automatically generated

* 1. **Comprehensive Program and Client Analysis**

**Scenario Description:**

Management needs a detailed report covering:

1. Total clients and their average number of incidents per program.
2. Clients with the most incidents in each program.
3. Programs with the fewest resources flagged as "Under-Resourced."

This query combines grouping, ranking, subqueries, joins, and conditional logic for a comprehensive analysis.

WITH ClientIncidentCounts AS (

SELECT

Programs.ProgramID,

Programs.Name AS ProgramName,

Clients.ClientID,

Clients.Name AS ClientName,

COUNT(Incidents.IncidentID) AS TotalIncidents

FROM

Programs

LEFT JOIN

Clients ON Programs.ProgramID = Clients.CurrentProgramID

LEFT JOIN

Incidents ON Clients.ClientID = Incidents.ClientID

GROUP BY

Programs.ProgramID, Programs.Name, Clients.ClientID, Clients.Name

),

ProgramClientSummary AS (

SELECT

ProgramID,

ProgramName,

COUNT(ClientID) AS TotalClients,

AVG(TotalIncidents) AS AvgIncidentsPerClient

FROM

ClientIncidentCounts

GROUP BY

ProgramID, ProgramName

),

TopClientsByProgram AS (

SELECT

ProgramID,

ProgramName,

ClientID,

ClientName,

TotalIncidents,

RANK() OVER (PARTITION BY ProgramID ORDER BY TotalIncidents DESC) AS IncidentRank

FROM

ClientIncidentCounts

),

ProgramResourceSummary AS (

SELECT

Programs.ProgramID,

Programs.Name AS ProgramName,

COUNT(Resources.ResourceID) AS TotalResources,

CASE

WHEN COUNT(Resources.ResourceID) < 2 THEN 'Under-Resourced'

ELSE 'Sufficiently Resourced'

END AS ResourceStatus

FROM

Programs

LEFT JOIN

Resources ON Programs.ProgramID = Resources.AssociatedProgramID

GROUP BY

Programs.ProgramID, Programs.Name

)

SELECT

pcs.ProgramID,

pcs.ProgramName,

pcs.TotalClients,

pcs.AvgIncidentsPerClient,

tcbp.ClientID AS TopClientID,

tcbp.ClientName AS TopClientName,

tcbp.TotalIncidents AS TopClientIncidents,

prs.TotalResources,

prs.ResourceStatus

FROM

ProgramClientSummary pcs

LEFT JOIN

TopClientsByProgram tcbp ON pcs.ProgramID = tcbp.ProgramID AND tcbp.IncidentRank = 1

LEFT JOIN

ProgramResourceSummary prs ON pcs.ProgramID = prs.ProgramID

ORDER BY

pcs.ProgramName;

The output is

A screenshot of a computer

Description automatically generated

* 1. **Using LIKE**

**Scenario:** Find clients with names starting with "J".

A close up of words

Description automatically generated

The output is

A white background with black text

Description automatically generated

* 1. **Using INNER JOIN**

**Scenario:** List all clients and their current program details.

A close-up of words

Description automatically generated

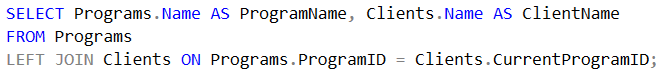
An INNER JOIN combines rows from the Clients and Programs tables based on a matching condition. The condition specified is: “Clients.CurrentProgramID = Programs.ProgramID”. This ensures that only rows where a CurrentProgramID in the Clients table matches a ProgramID in the Programs table are included in the result. The portion of the output is

A screenshot of a computer

Description automatically generated

* 1. **Using LEFT JOIN**

**Scenario:** Find all programs and the clients associated with them, including programs without any clients.



This SQL query uses a RIGHT JOIN to combine data from the Clients and Programs tables, ensuring that all rows from the Programs table are included, even if they do not have a matching record in the Clients table. The portion of the result is shown below:

A table of text on a white background

Description automatically generated

**d. Using RIGHT JOIN**

**Scenario:** List all clients and their assigned programs, even if a client is not currently assigned to a program.

A screenshot of a computer

Description automatically generated

The A RIGHT JOIN ensures that **all rows from the Programs table** appear in the result, even if there is no matching CurrentProgramID in the Clients table. If no client is associated with a particular program, columns from the Clients table will show NULL.

A screenshot of a computer

Description automatically generated

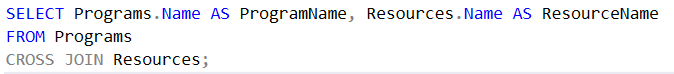
The portion of the output is :

A screenshot of a computer

Description automatically generated

* 1. **Using CROSS JOIN**

**Scenario:** Create a combination of all programs and all resources to analyze potential program-resource assignments.



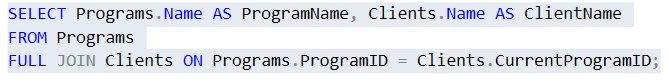
The A CROSS JOIN generates all possible combinations of rows from the two tables. If the Programs table has 3 rows and the Resources table has 4 rows, the result will have 3×4 = 12 rows. The result is:

A screenshot of a computer program

Description automatically generated

* 1. **Using FULL JOIN**

**Scenario:** Show all programs and clients, including unassigned programs or clients.



The Full Join combines rows from both tables. It includes all rows from the Programs table, even if there are no matching rows in the Clients table, and also all rows from the Clients table, even if there are no matching rows in the Programs table. If no match exists, the corresponding columns from the unmatched table will be NULL. The portion of the output is shown below:

A screenshot of a program

Description automatically generated

* 1. **Using COUNT**

**Scenario: Count the total number of clients enrolled in each program.**

A screenshot of a computer code

Description automatically generated

The output is



* 1. **Using AVERAGE, SUM**

**Scenario:** Calculate the average number of incidents per client and the total number of incidents recorded.

A screenshot of a computer program

Description automatically generated

This query calculates the average number of incidents per client as well as the total number of incidents. The output is displayed below:

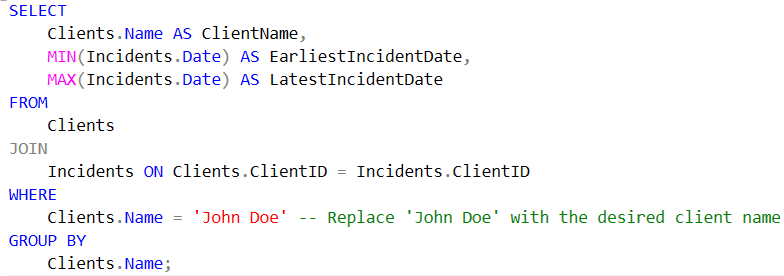
A close-up of a sign

Description automatically generated

* 1. **Using Aggregate Functions**

**Scenario: Finding the Earliest and Latest Incident Dates for a Given Client named John Doe**

This query helps in understanding the history of a client, which can be critical for creating tailored behavior plans or tracking progress over time.



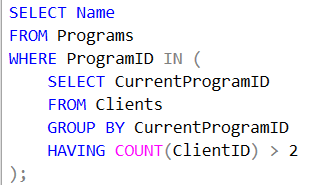
The output is as follows:

A close up of a number

Description automatically generated

**15. Using a Subquery**

**Scenario:** List of all programs with more than 2 clients enrolled.



The result is the following:

A blue rectangle with black text

Description automatically generated

* 1. **Window Functions**

**Scenario:** Rank clients by the number of incidents they have.

A screenshot of a computer code

Description automatically generated

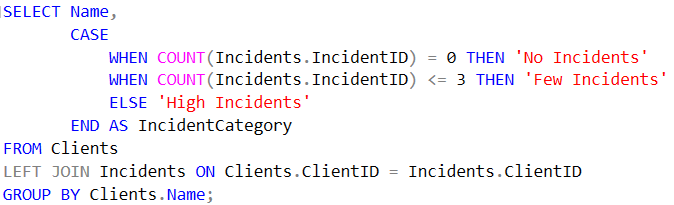
The portion of results is displayed below:

A screenshot of a data

Description automatically generated

* 1. **Using IF**

**Scenario:** Categorize clients based on the number of incidents they have.



The portion of the output is the following:

A table of information

Description automatically generated with medium confidence

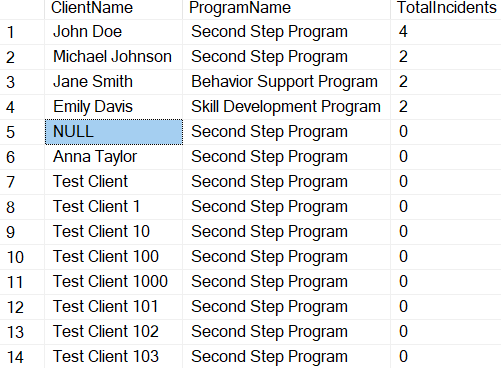
* 1. **1Using Complex Queries**

**Scenario:** List of clients with their program details and the count of incidents, ordered by the number of incidents.

A computer screen shot of text

Description automatically generated

The partial of the results is the following:



1. **Expected Benefits**

The implementation of this project brings numerous advantages that span across operational efficiency, decision-making, quality of care, regulatory compliance, and scalability. By leveraging optimized processes, data-driven insights, and advanced tools, organizations can streamline workflows, enhance patient outcomes, and meet both current and future needs. The following subsections outline the specific benefits anticipated from this initiative.

**7.1 Efficiency**

Efficiency is one of the core benefits of this project, as it aims to streamline workflows and automate repetitive tasks. By minimizing manual interventions, the likelihood of errors is reduced, allowing for faster and more accurate data processing. The optimized processes ensure that resources such as time and effort are utilized effectively, freeing up staff to focus on higher-priority tasks. Additionally, improved data retrieval mechanisms make it easier for users to access critical information promptly, thereby enhancing overall productivity and operational performance.

**7.2 Decision-Making**

Improved decision-making is another significant outcome, driven by the availability of accurate, real-time data. The integration of advanced analytics and reporting tools provides actionable insights that enable organizations to make data-driven decisions confidently. Predictive models further support proactive strategies, helping to identify trends, patterns, and areas requiring intervention. With enhanced visibility into key performance metrics, stakeholders can make timely and informed decisions that drive organizational success and improve outcomes.

**7.3 Care Quality**

The project directly contributes to improved care quality by ensuring the accuracy and reliability of patient data. Streamlined data management enables healthcare providers to access comprehensive and up-to-date records, leading to better diagnoses and treatment plans. Predictive analytics also play a vital role in identifying at-risk patients, allowing for timely interventions and preventive care. By reducing administrative burdens, healthcare professionals can dedicate more time to patient care, ultimately enhancing satisfaction and outcomes for both patients and providers.

**7.4 Compliance**

Compliance with regulatory requirements is a critical benefit of this project, ensuring adherence to industry standards and legal frameworks such as HIPAA, GDPR, and other applicable guidelines. Robust data management practices, including secure access controls and audit trails, provide transparency and accountability in operations. The system helps mitigate risks associated with non-compliance, such as penalties and reputational damage, by maintaining data integrity and enabling accurate reporting. This proactive approach to compliance builds trust and confidence among stakeholders.

**7.5** **Efficiency**

Efficiency is one of the core benefits of this project, as it aims to streamline workflows and automate repetitive tasks. By minimizing manual interventions, the likelihood of errors is reduced, allowing for faster and more accurate data processing. The optimized processes ensure that resources such as time and effort are utilized effectively, freeing up staff to focus on higher-priority tasks. Additionally, improved data retrieval mechanisms make it easier for users to access critical information promptly, thereby enhancing overall productivity and operational performance.

**7.6 Scalability**

Scalability ensures that the project can adapt to growing organizational demands and evolving technologies. The infrastructure is designed to handle increasing data volumes and accommodate more users without compromising performance. The system's flexibility allows seamless integration with other tools and platforms, ensuring future compatibility and functionality expansion. This ability to scale effectively prepares the organization for long-term growth, enabling it to meet current challenges while being ready to embrace future innovations and opportunities.

1. **General Conclusion**

This project initiative serves as a comprehensive solution to current challenges while laying a strong foundation for future success. It empowers Raise and Grow organization to remain competitive, responsive, and efficient in an ever-evolving environment, driving meaningful and sustainable outcomes.