**1. What is Data Security**

**1.1 Access Control and Privileges**

**1.2 Row Level Security**

**1.3 Column Level Security**

**2. Objective**

**3. Implementation**

**3.1 Schema**

**3.2 Framework**

4. Unit Testing

5. Future Work

1. What is Data Security

Data security refers to the protection of data from unauthorized access, use, disclosure, modification, or destruction. It involves implementing measures and safeguards to ensure the confidentiality, integrity, and availability of data. Data security aims to prevent unauthorized individuals or entities from accessing sensitive or private information and ensures that data remains accurate and reliable.

Data security involves various practices, technologies, and policies designed to protect data throughout its lifecycle. This includes:

* Authentication and Access Control
* Encryption
* Data Loss Prevention (DLP)
* Firewall and Network Security
* Secure Data Storage
* Employee Education and Awareness
* Security Policies and Compliance.

Data security is crucial for businesses and organizations to maintain the privacy of customer information, protect intellectual property, prevent financial losses, and maintain trust with stakeholders. It is an ongoing process that requires a combination of technology, policies, and user awareness to mitigate risks and safeguard sensitive data.

In our case, we're implementing data security measures at a granular level within a database system involving various approaches to ensure the confidentiality, integrity, and availability of data.

Here are some commonly used approaches and the ones we're implementing in current project:

| **Approaches** | **Description** | Implementation status |
| --- | --- | --- |
| **Row-Level Security (RLS)** | Controlling access to individual rows based on predefined rules and attributes. | Yes |
| **Column-Level Security (CLS)** | Restricting access to specific columns or fields within a table. | Yes |
| Data Encryption | Transforming data into unreadable format using encryption algorithms. | No |
| **Access Control and Privileges** | Managing user permissions and roles at different levels. | Yes |
| Auditing and Monitoring | Tracking user activities and detecting suspicious behavior. | No |
| Database Activity Monitoring (DAM) | Real-time monitoring and analysis of database activities. | No |
| Database Encryption | Encrypting the entire database or specific objects. | No |

By combining these approaches, organizations can implement robust and granular data security measures that align with specific security requirements and regulatory compliance obligations. It is important to assess the sensitivity of data, understand access requirements, and apply appropriate security controls to ensure data protection throughout its lifecycle.

1.1 Access Control and Privileges

Access control and privileges involve managing user permissions and roles in a database system.

It determines who can access data and resources at different levels: database, table, and column. Users are assigned specific privileges like read, write, update, delete, or execute.

Access control ensures that users have the necessary access rights while minimizing the risk of unauthorized access or data breaches. It is implemented through user accounts, roles, and granting appropriate permissions to perform specific operations within the database system. Effective access control helps maintain data integrity, protect sensitive information, and comply with security regulations.

In our case, we plan to create roles specific to each department, the access rights are associated with that department's role. Users who belong to multiple departments would be assigned multiple roles, reflecting their access rights across those departments. For example, user\_A can access information that granted to role “Sales” and “Administration”:

| Roles | Departments | Users |
| --- | --- | --- |
| Sales | Sales | user\_A, user\_B, user\_C |
| Engineering | Engineering | user\_D, user\_E |
| Finance | Finance | user\_F |
| Administration | Administration | user\_A, user\_G, user\_K |

1.2 Row Level Security

Row-level security (RLS) is a data security approach that focuses on controlling access to individual rows within a database table. It allows you to enforce fine-grained access controls based on the characteristics of the data itself. Here are a few approaches to implementing row-level security, in our case, we choose Predicate-based filtering and Stored procedures and functions.

| **Approach** | **Description** | **Implementation Status** |
| --- | --- | --- |
| Predicate-based filtering | Security policies defined using conditions or predicates to determine which rows a user can access. | Yes |
| Views and joins | Creating views that include only permitted rows based on user roles or attributes. | No |
| Stored procedures and functions | Implementing logic in stored procedures or functions to enforce row-level security policies. | Yes |
| Externalized security models | Integration with external security frameworks or IAM systems for authentication, authorization, and access controls. | No |

RLS helps organizations enforce strict data access controls and mitigate the risk of unauthorized access or data breaches. For example, users in “Sales” role can access the rows data related to Sales department.

| Roles | Tables | Rows |
| --- | --- | --- |
| Sales | Table 1 | Rows where related to Sales department |
| Engineering | Table 1 | Rows where related to Engineering department |
| Finance | Table 1 | Rows where related to Finance department |
| Administration | Table 1 | Rows where related to Admin department |

1.3 Column Level Security

Column-level security is a data security approach that focuses on controlling access to specific columns or fields within a database table. It allows to restrict or limit the visibility of sensitive or confidential data at a more granular level.

To achieve column-level security, we can employ various techniques, in our case, we choose Dynamic Data Masking.

| **Technique** | **Description** | **Implementation Status** |
| --- | --- | --- |
| Database Views | Creating views to control column-level access, providing a logical layer of abstraction and fine-grained control. | No |
| Column-level Privileges | Defining specific permissions at the column level, granting control over who can access or modify individual columns. | No |
| Data Encryption | Applying encryption to specific columns, rendering data unreadable without proper decryption keys. | No |
| Data Masking | Hiding or obfuscating sensitive information in specific columns, like credit card numbers or social security numbers. | No |
| Dynamic Data Masking | On-the-fly masking or transformation of data based on user access privileges, preventing exposure of raw data. | Yes |

By implementing column-level security measures, we can restrict access to sensitive data, ensure data confidentiality, and reduce the risk of unauthorized disclosure. It provides an additional layer of security, especially for databases that store highly sensitive information like personal identifiable information (PII), financial data, or intellectual property. For example, there are 4 columns in Table 1 users in the “Sales” role can have access to Column 1 and Column 2 in Table 1 without masking.

| Roles | Tables | Columns |
| --- | --- | --- |
| Sales | Table 1 | Column 1, Column 2 |
| Engineering | Table 1 | Column 1, Column 2, Column 3 |
| Finance | Table 1 | Column 1, Column 2, Column 3, Column 4 |
| Administration | Table 1 | Column 1, Column 2, Column 3, Column 4 |

2. Objective

Objective: Strengthen data protection and safeguard sensitive information.

Key Achievements:

* Implement robust access control measures.
* Enhance monitoring and logging capabilities.
* Establish incident response procedures.

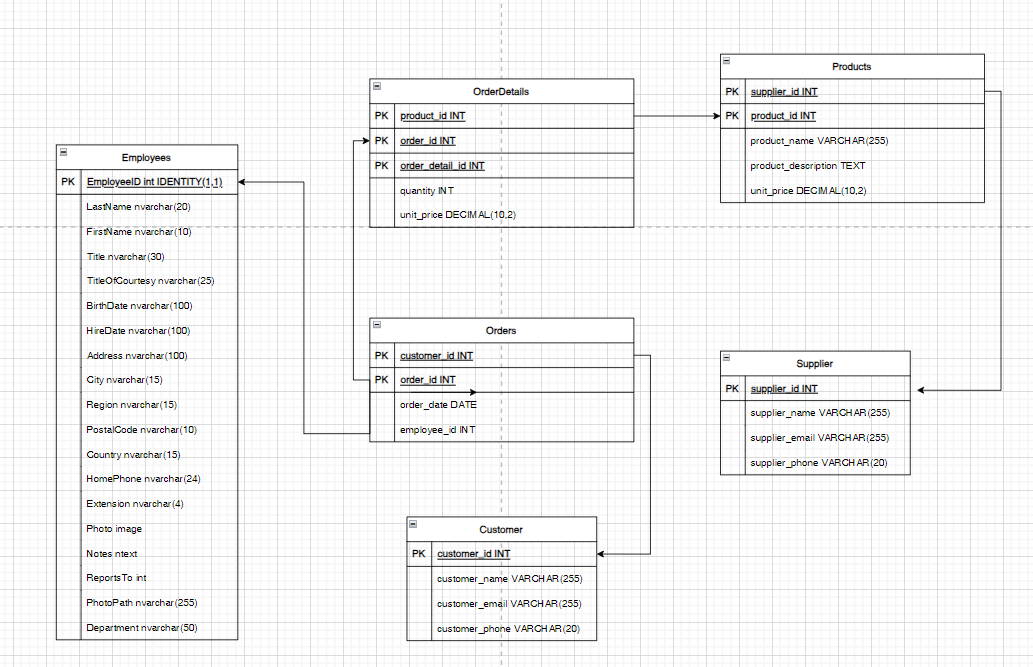
3. Implementation

* 1. **Database Schema Overview**

3.1.1 **DBO schema**

"dbo" schema is a default schema that provides a way to organize and group database objects within a SQL Server database.(in this case, we only have employees table and orders table for for demonstration purposes. )

* The column "BirthDate" can only be accessed by the "administration" team.
* Each role can only access the rows that are related to their own department.

A picture containing text, screenshot, number, font

Description automatically generated

3.1.2 **SECURITY schema**

It's highly recommended to create a separate schema for the RLS objects: predicate functions, and security policies. This helps to separate the permissions that are required on these special objects from the target tables. Additional separation for different policies and predicate functions may be needed in multi-tenant-databases, but not as a standard for every case.

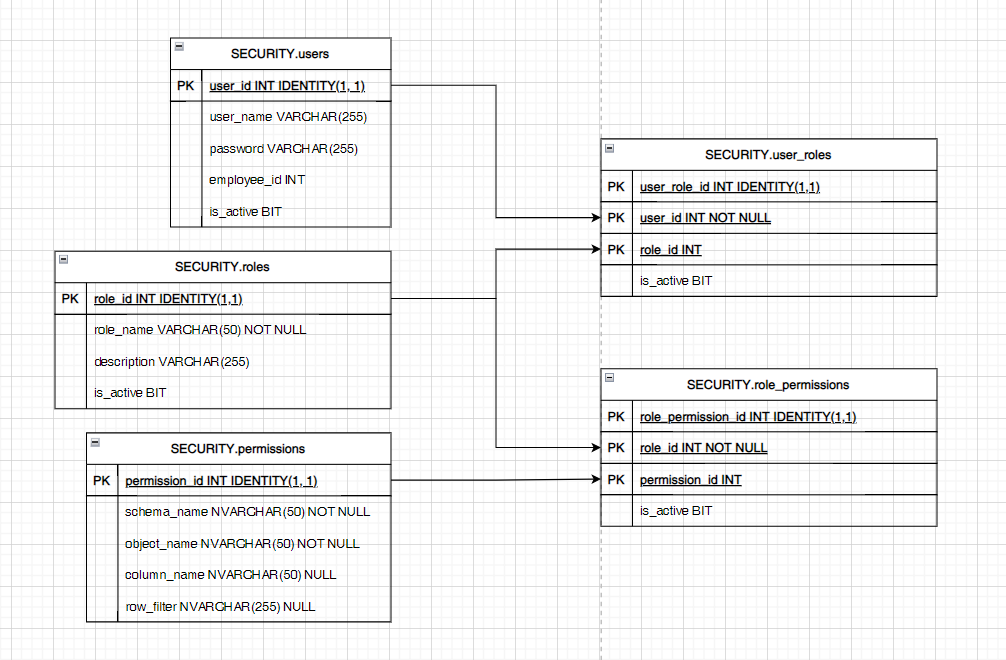
* 1. **Implementation**

Implementing the data security measures, including access control and privileges, applying column-level security (dynamic data masking), predicate-based filtering, creating row-level security functions. Adapt these steps and corresponding scripts to meet specific requirements and security models.

| Step | Description |
| --- | --- |
| 1 | create users, roles, user\_roles, permissions, role\_permissions tables |
| 2 | create column level security (dynamic data masking) procedure |
| 3 | create row level security function |

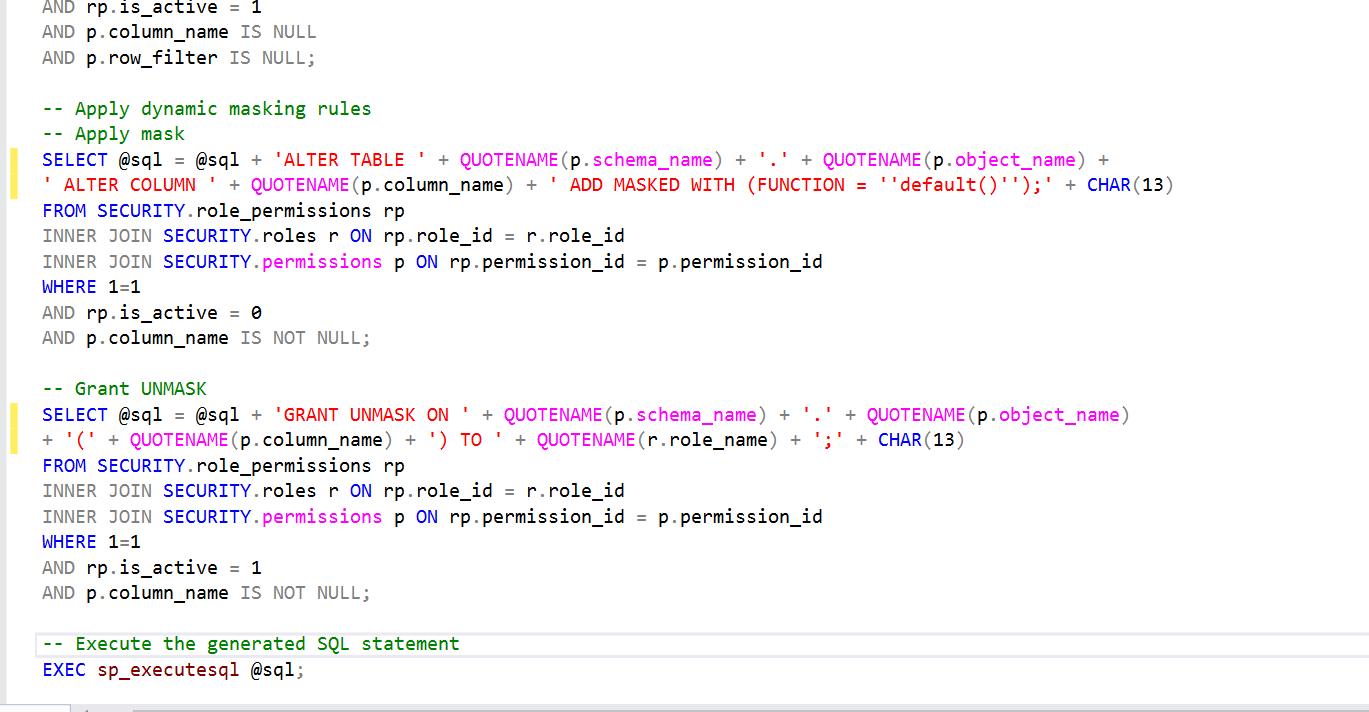
STEP 1 Create users, roles, permissions, user\_roles, role\_permissions tables

Schema diagram of the tables:



STEP 2 Create Column Level Security (Dynamic Masking) Procedure

Create a dynamic masking procedure based on the role permissions table.

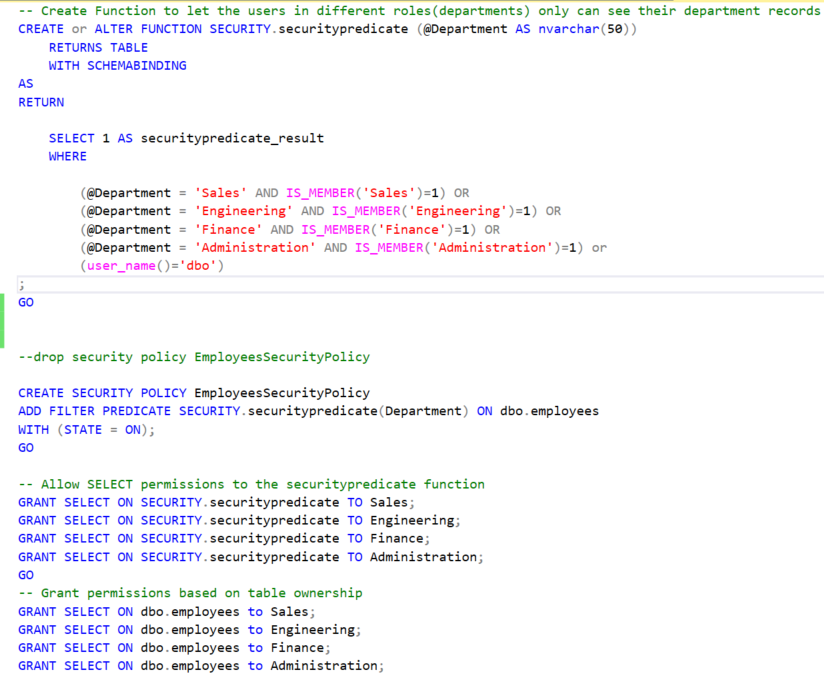


STEP 3 Create Row Level Security Function

Create a function that allows roles to access the rows belonging to their department.

Create a security policy adding the function as a filter predicate. The state must be set to ON to enable the policy.

Allow SELECT permissions to the function.



* 1. **Framework**

In general, a framework is a real or conceptual structure that supports or simplifies a process. A SQL framework is a collection of tools, libraries, and utilities that provide a structured approach to working with SQL databases. It simplifies the development process by offering a set of standardized methods, functions, and abstractions that make it easier to interact with databases and perform common database operations.

3.3.1 User

When we create or alter users in the “users” table, the users should automatically be created, or updated. Hence, we use triggers on the table to implement the rule as a framework.

3.3.2 Role

Similar to users, when we create or alter roles in the “roles” table, the roles should automatically be created or updated. Hence, we use triggers on the table to implement the rule as a framework.

3.3.3 User\_Roles

When we create, delete or alter current relationships in the “user\_roles” table, the users should automatically be added, removed or updated to the role group. Hence, we use triggers on the table to implement the rule as a framework.

3.3.4 Role\_Permissions

When we create, delete or alter current roles and permissions in the “role\_permissions” table, the role should have changed its permissions.

Column level security (masking columns) is easy to implement, we can use one stored procedure to achieve the desired result. While the row level security can be more complex, we may need multiple functions according to the requirement.

4. Unit Testing

4.1 Users

4.1.1 Add a new user to a role

*Test Case Description:*

Pranaou is new to the company and going to join the team 'Engineering', he should have the 'Engineering' role access.

*Expected Result:*

Create a user 'pranaou', and he can see the record related to the 'Engineering' department and also the masked 'BirthDate' column in dbo.employee according to the 'Engineering' role access.

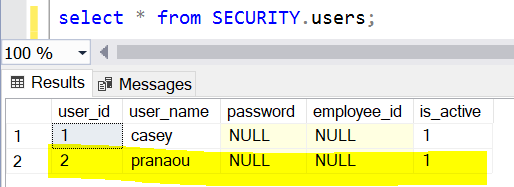
*Steps of Execution:*

1. Insert a new user in the 'users' table(it will fire the trigger to create the user in the system).

*INSERT INTO SECURITY.users (user\_name, is\_active)*

*values( 'pranaou', 1);*

2. Check if the user is inserted into the 'users' table.

**

3. Insert the relationship into the 'user\_roles' table(it will fire the trigger to add the user to the role).

*INSERT INTO SECURITY.user\_roles (user\_id, role\_id, is\_active)*

*values( (select user\_id from SECURITY.users where user\_name = 'pranaou'), (select role\_id from SECURITY.roles where role\_name =* 'Engineering' *), 1);*

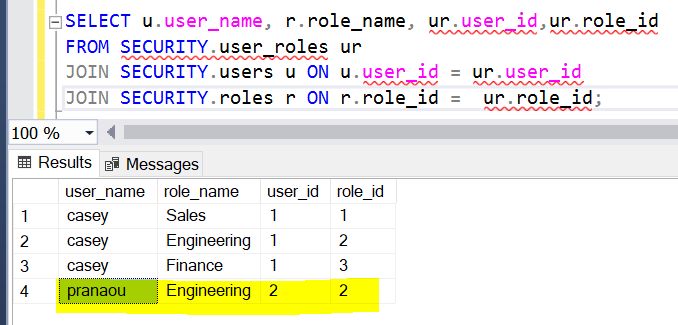
4. Check if the relation is inserted into the 'user\_roles' table.

*SELECT u.user\_name, r.role\_name, ur.user\_id,ur.role\_id*

*FROM SECURITY.user\_roles ur*

*JOIN SECURITY.users u ON u.user\_id = ur.user\_id*

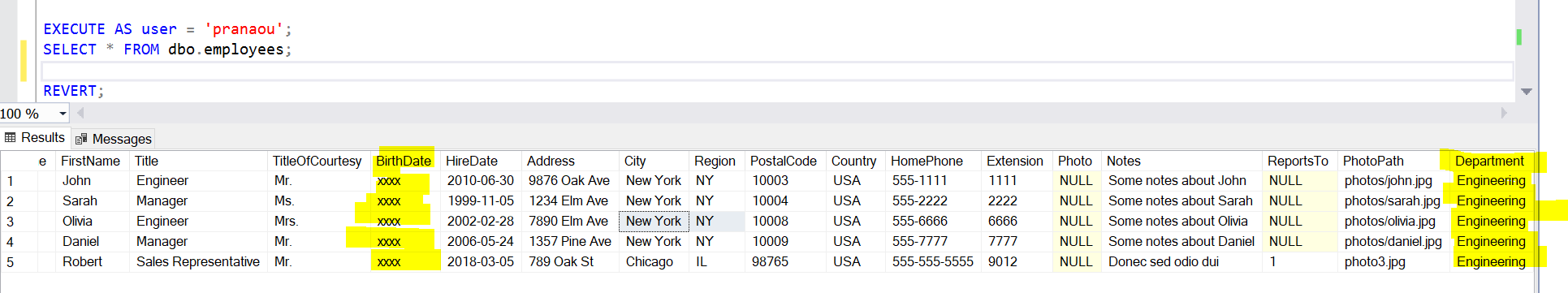
*JOIN SECURITY.roles r ON r.role\_id = ur.role\_id;*

**

3. Validation:

*EXECUTE AS user = 'pranaou';*

*SELECT \* FROM dbo.employees;*

**

*Test Case Result:*

Pass

4.1.2 Add an existing user to another role

*Test Case Description:*

Pranaou from the Engineering team is going to support the 'Administration' team for 3 months, he should also have the role access as 'Administration' and keep his current role as 'Engineering'.

*Expected Result:*

User 'pranaou' can have 'Engineering' and 'Administration' role access. He can see the unmasked 'BirthDate' column in dbo.employee as the 'Administration' role has the access.

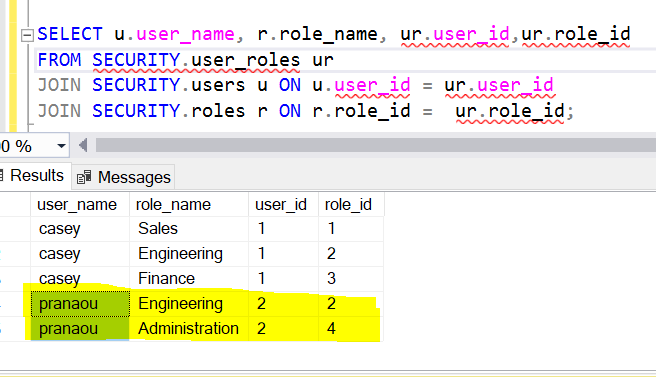
*Steps of Execution:*

1. Insert the new relationship to the users table (it will fire the trigger to add the user to the role).

*INSERT INTO SECURITY.user\_roles (user\_id, role\_id, is\_active)*

*values((select user\_id from SECURITY.users where user\_name = 'pranaou'), (select role\_id from SECURITY.roles where role\_name =* 'Administration' *), 1);*

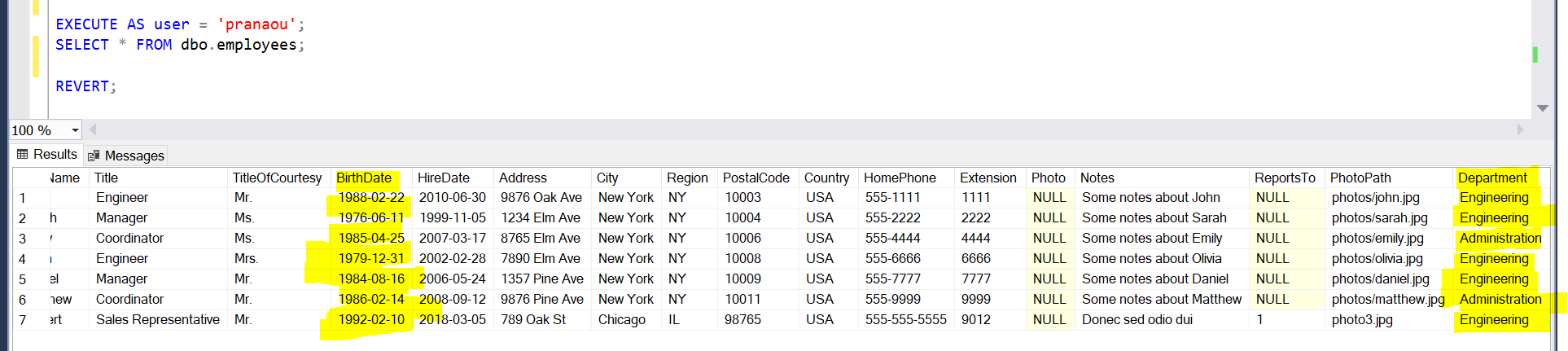
2. Check if the relation is inserted into the 'user\_roles' table.

**

3. Validation:

*EXECUTE AS user = 'pranaou';*

*SELECT \* FROM dbo.employees;*

**

*Test Case Result:*

Pass

4.1.3 Remove a user from a role

*Test Case Description:*

Upon 4.1.2, Pranaou no longer supports the Administration team, his access to “Administration” should be removed. While, he is still in the Engineering team.

*Expected Result:*

User 'pranaou' can only see the record related to the 'Engineering' department and also the masked 'BirthDate' column in dbo.employee as he only has the 'Engineering' role access and has been removed from 'Administration' role.

*Steps of Execution:*

1. Delete the user and role relationship in the '*user\_roles*' table.

*DELETE FROM SECURITY.user\_roles*

*WHERE 1=1*

*AND user\_id = (select user\_id from SECURITY.users where user\_name = 'pranaou')*

*AND role\_id = (select role\_id from SECURITY.roles where role\_name =* 'Administration' *);*

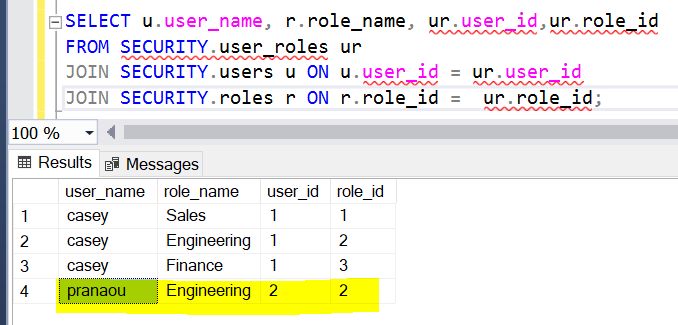
2. Check if the relation is removed from the 'user\_roles' table.

*SELECT u.user\_name, r.role\_name, ur.user\_id,ur.role\_id*

*FROM SECURITY.user\_roles ur*

*JOIN SECURITY.users u ON u.user\_id = ur.user\_id*

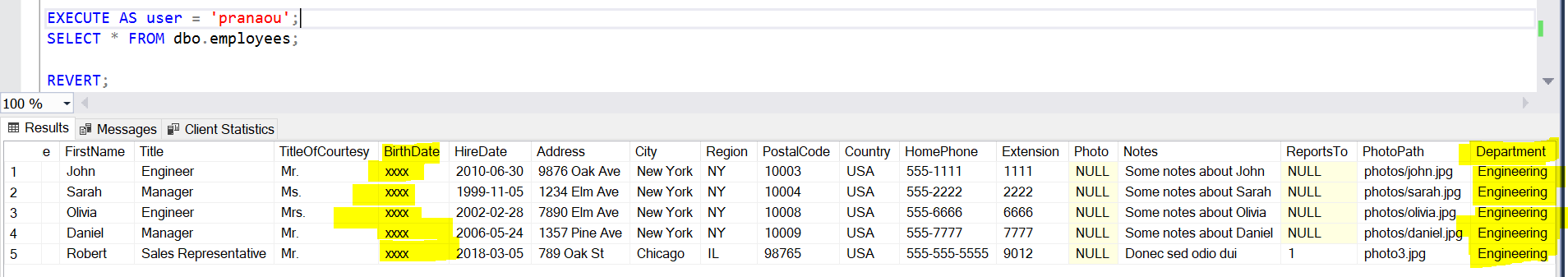
*JOIN SECURITY.roles r ON r.role\_id = ur.role\_id;*

**

3. Validation:

*EXECUTE AS user = 'pranaou';*

*SELECT \* FROM dbo.employees;*

**

*Test Case Result:*

Pass

4.1.4 Deactivate a user

*Test Case Description:*

Pranaou is leaving the Engineering team as well as the company. All his permissions should be removed.

*Expected Result:*

User 'pranaou' should not access any tables or views. And his status 'is\_active' in the 'users' table should be 0.

*Steps of Execution:*

1. Update the user table to set 'is\_active' column as 0 to deactivate the user, where user\_name is *'pranaou'* (it will fire the trigger to remove all user\_roles relationships).

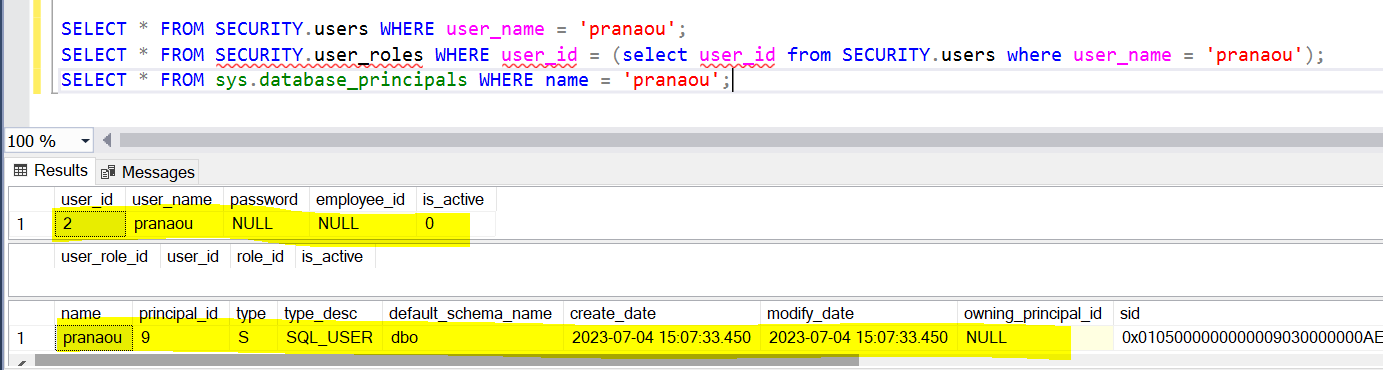
*UPDATE SECURITY.users*

*SET is\_active = 0*

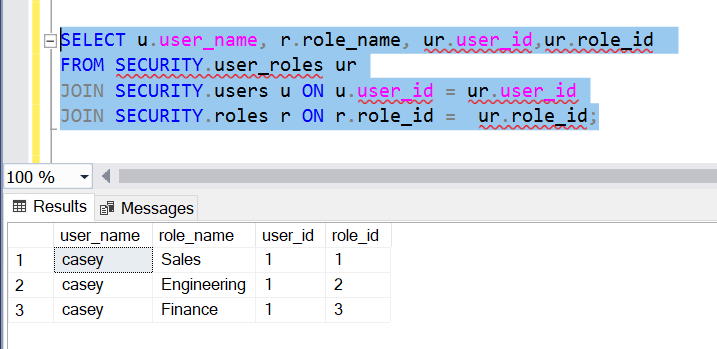
*WHERE 1=1*

*AND user\_name = 'pranaou';*

2. Check if the user has been deactivated in the table but not been deleted from the database.



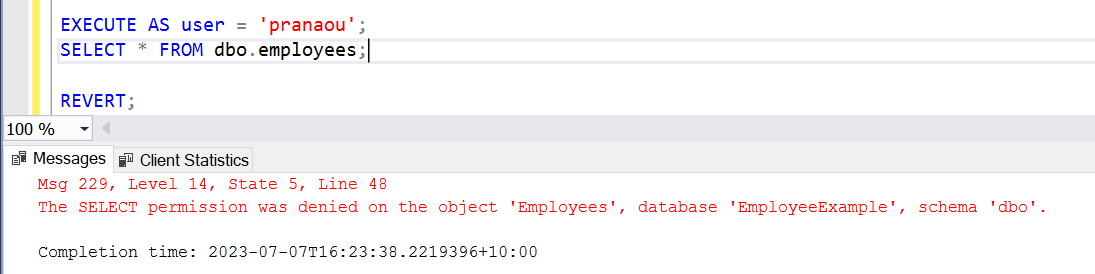
3. Check if all relations of 'pranaou' are removed from the 'user\_roles' table.



4. Validation:

*EXECUTE AS user = 'pranaou';*

*SELECT \* FROM dbo.employees;*

**

*Test Case Result:*

Pass

4.1.5 Reactive a user

*Test Case Description:*

Pranaou used to work in the company and he left the company 1 year ago, and recently he is joining the company in the Engineering team again, he should have the 'Engineering' role access.

*Expected Result:*

User 'pranaou' can see the record related to the 'Engineering' department and also the masked 'BirthDate' column in dbo.employee according to the 'Engineering' role access.

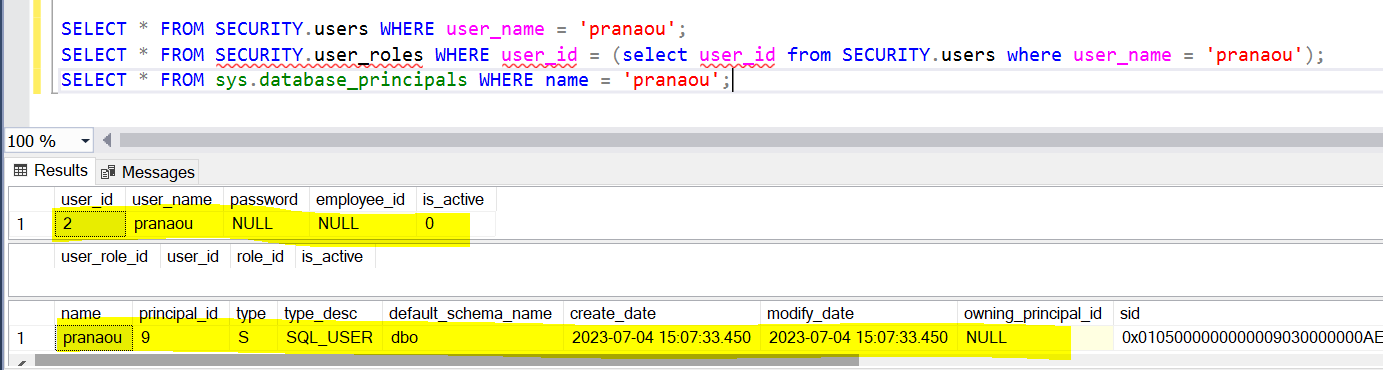
*Steps of Execution:*

1. Check if user *'pranaou' is in the database, in the users table and the relations in user\_roles table.*

*Select \* From SECURITY.users WHERE user\_name = 'pranaou';*

*Select \* From SECURITY.user\_roles WHERE user\_name = 'pranaou';*

*SELECT \* FROM sys.server\_principals WHERE name = 'pranaou';*



2. Update the users table to set '*is\_active* ' column as 1

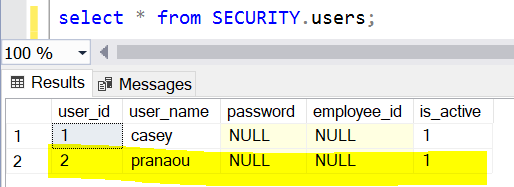
*UPDATE SECURITY.users*

*SET is\_active = 1*

*WHERE 1=1*

*AND user\_name = 'pranaou';*

3. Check if the user is active in the 'users' table.

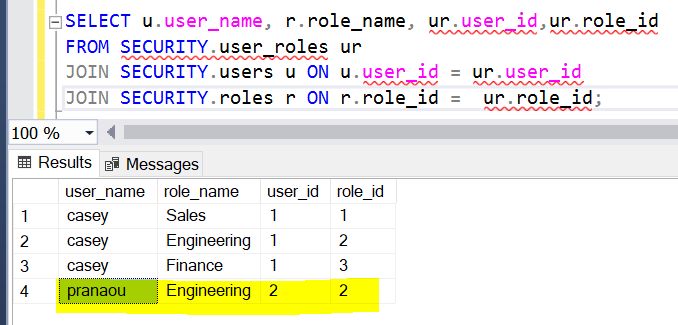
**

4. Insert the relationship into the 'user\_roles' table(it will fire the trigger to add the user to the role).

*INSERT INTO SECURITY.user\_roles (user\_id, role\_id, is\_active)*

*values( (select user\_id from SECURITY.users where user\_name = 'pranaou'), (select role\_id from SECURITY.roles where role\_name =* 'Engineering' *), 1);*

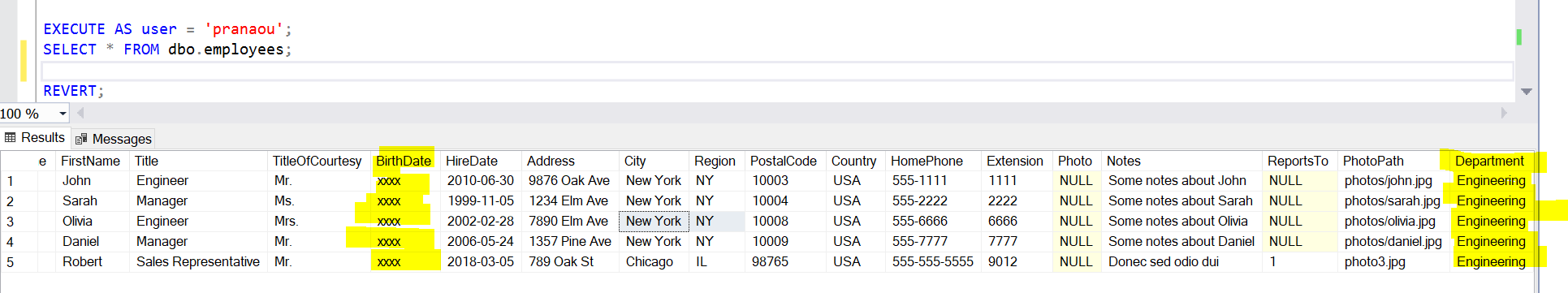
5. Check if the relation is inserted into the 'user\_roles' table.

**

6. Validation:

*EXECUTE AS user = 'pranaou';*

*SELECT \* FROM dbo.employees;*

**

*Test Case Result:*

Pass

4.2 Permissions

4.2.1 Add a new column with masking, and apply the permission

*Test Case Description:*

A new column ‘Salary’ will be created in the employees table and all current roles cannot access it.

*Expected Result:*

The current users could only see the masked ‘Salary’ column. And it wont mess up with current permissions.

*Steps of Execution:*

1. Create a ‘Salary’ column in the employees table and insert data into it.

*-- Add the 'Salary' column to the 'employees' table*

*ALTER TABLE dbo.employees*

*ADD Salary DECIMAL(10, 2);*

*-- Insert data into the 'Salary' column*

*UPDATE dbo.employees*

*SET Salary =*

*CASE*

*WHEN EmployeeID = 33 THEN 100000.00*

*WHEN EmployeeID = 16 THEN 70000.00*

*WHEN EmployeeID = 17 THEN 90000.00*

*WHEN EmployeeID = 18 THEN 60000.00*

*WHEN EmployeeID = 19 THEN 65000.00*

*WHEN EmployeeID = 20 THEN 85000.00*

*WHEN EmployeeID = 21 THEN 55000.00*

*WHEN EmployeeID = 22 THEN 70000.00*

*WHEN EmployeeID = 23 THEN 50000.00*

*WHEN EmployeeID = 24 THEN 60000.00*

*WHEN EmployeeID = 25 THEN 80000.00*

*WHEN EmployeeID = 26 THEN 90000.00*

*WHEN EmployeeID = 27 THEN 65000.00*

*WHEN EmployeeID = 28 THEN 50000.00*

*WHEN EmployeeID = 29 THEN 75000.00*

*WHEN EmployeeID = 30 THEN 90000.00*

*WHEN EmployeeID = 31 THEN 70000.00*

*WHEN EmployeeID = 32 THEN 50000.00*

*WHEN EmployeeID = 34 THEN 75000.00*

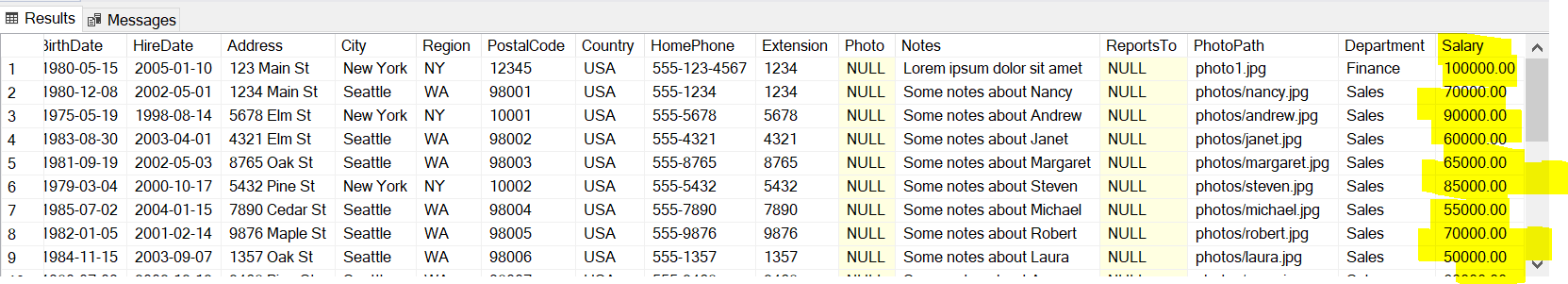
*WHEN EmployeeID = 35 THEN 60000.00*

*WHEN EmployeeID = 36 THEN 55000.00*

*ELSE NULL*

*END;*

2. Check if the data has been added.



3. Add a new permission of the column to the permissions table.

*INSERT INTO SECURITY.permissions (schema\_name, object\_name, column\_name, row\_filter)*

*VALUES*

*('dbo', 'Employees', 'Salary', NULL);*

4. Add the new permission to the role\_permissions table.

*INSERT INTO SECURITY.role\_permissions (role\_id, permission\_id, is\_active)*

*SELECT r.role\_id, p.permission\_id, 0*

*FROM SECURITY.roles r*

*CROSS JOIN SECURITY.permissions p*

*WHERE p.column\_name = 'Salary' AND r.is\_active = 1;*

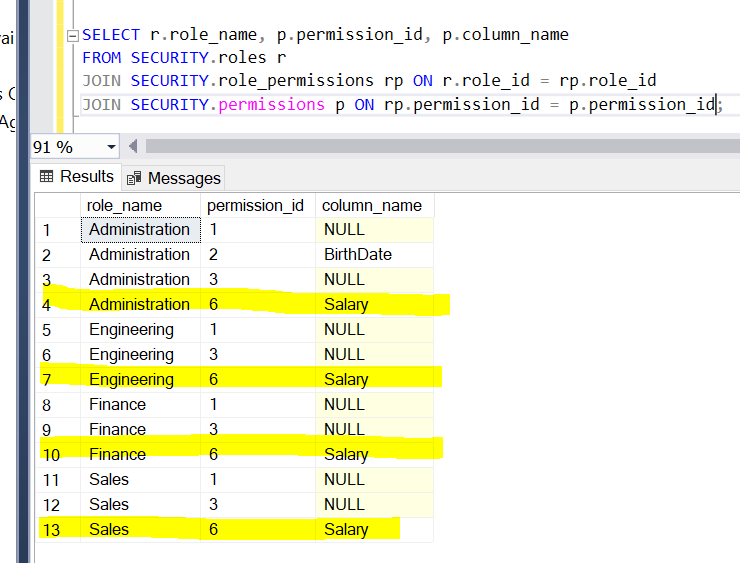
5. Check if the role\_permissions have been added to all active roles.

*SELECT r.role\_name, p.permission\_id, p.column\_name*

*FROM SECURITY.roles r*

*JOIN SECURITY.role\_permissions rp ON r.role\_id = rp.role\_id*

*JOIN SECURITY.permissions p ON rp.permission\_id = p.permission\_id;*



6. Execute the GrantAccessToRoleFunction

*EXEC SECURITY.GrantAccessToRole;*

7. Check the active users we have and their roles

*SELECT u.user\_name, r.role\_name*

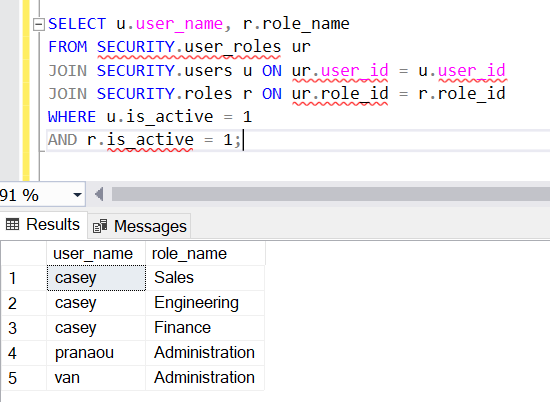
*FROM SECURITY.user\_roles ur*

*JOIN SECURITY.users u ON ur.user\_id = u.user\_id*

*JOIN SECURITY.roles r ON ur.role\_id = r.role\_id*

*WHERE u.is\_active = 1*

*AND r.is\_active = 1;*

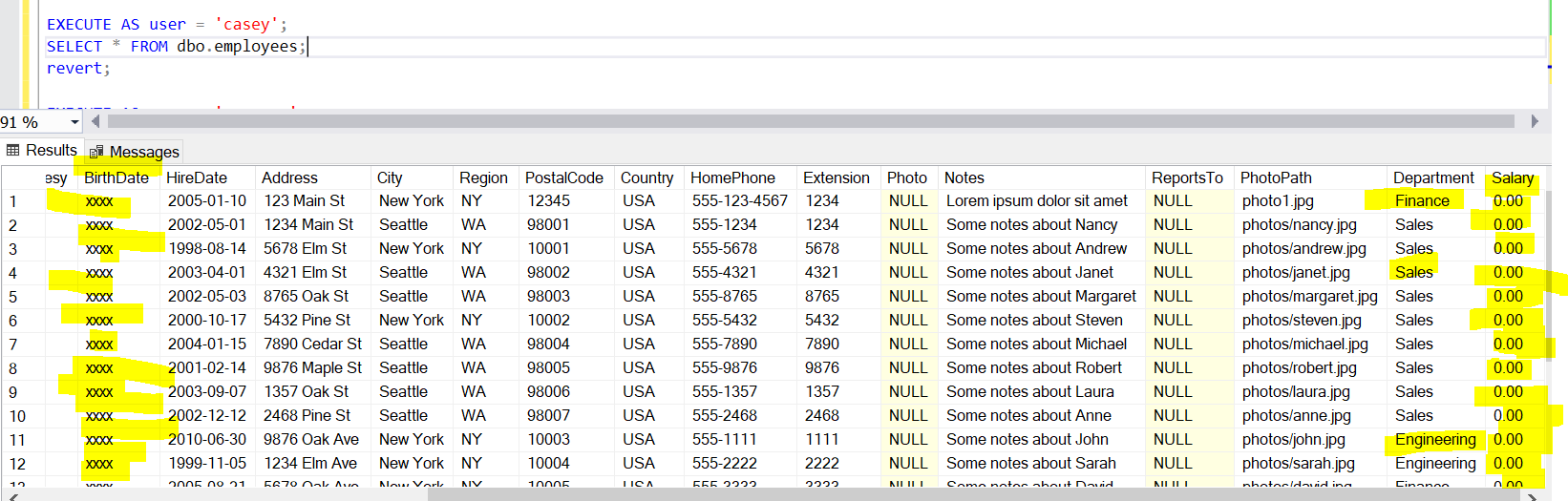


8. Validation:

Execute as current users to see the ‘Salary’ column to check if it is masked.

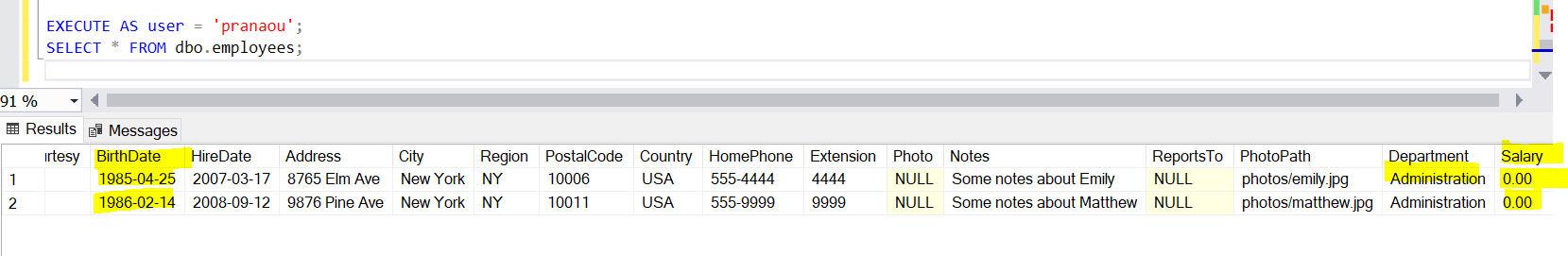
*EXECUTE AS user = 'casey';*

*SELECT \* FROM dbo.employees;*



*EXECUTE AS user = 'pranaou';*

*SELECT \* FROM dbo.employees;*

**

*Test Case Result:*

Pass

4.2.2 Add a new table

*Test Case Description:*

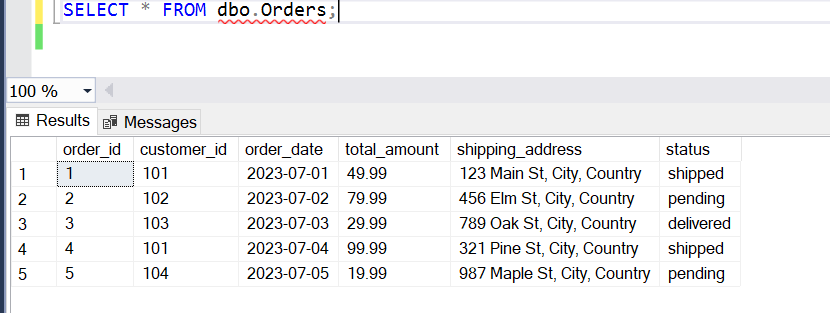
There is a new table 'Orders'. Only the users with Sales role access can see it.

*Expected Result:*

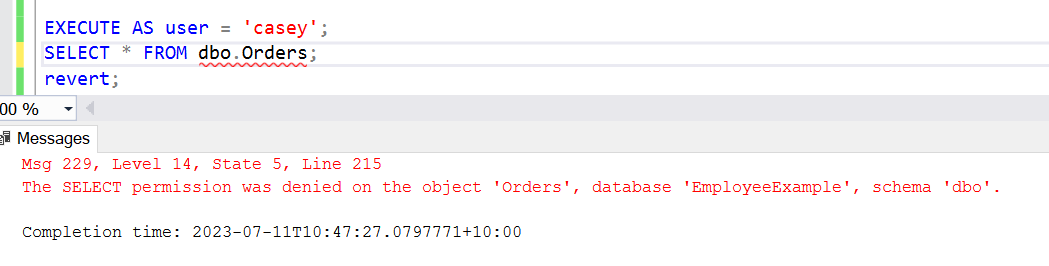
Only the users with Sales role access can have the access to query the table.

*Steps of Execution:*

1. New table ‘Order’ is created.



2. Execute as a user before the permission is created to see if the user has the access to see the table.



3. Add a new permission of the table to the permissions table.

*INSERT INTO SECURITY.permissions (schema\_name, object\_name, column\_name, row\_filter)*

*VALUES*

*('dbo', 'Orders', NULL, NULL);*

4. Add the new permission to the role\_permissions table.

*INSERT INTO SECURITY.role\_permissions (role\_id, permission\_id, is\_active)*

*SELECT r.role\_id, p.permission\_id, 1*

*FROM SECURITY.roles r*

*JOIN SECURITY.permissions p ON p.schema\_name = 'dbo'*

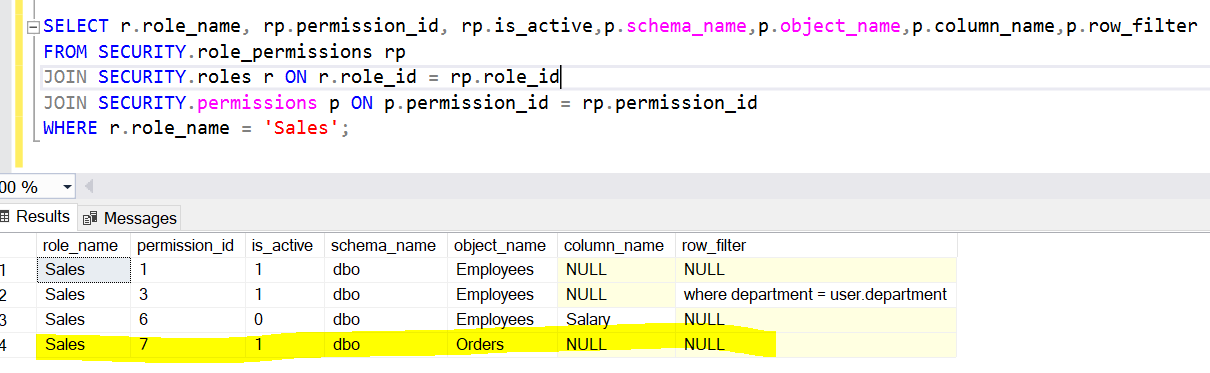
*AND p.object\_name = 'Orders'*

*AND p.column\_name IS NULL*

*AND p.row\_filter IS NULL*

*WHERE r.role\_name = 'Sales';*

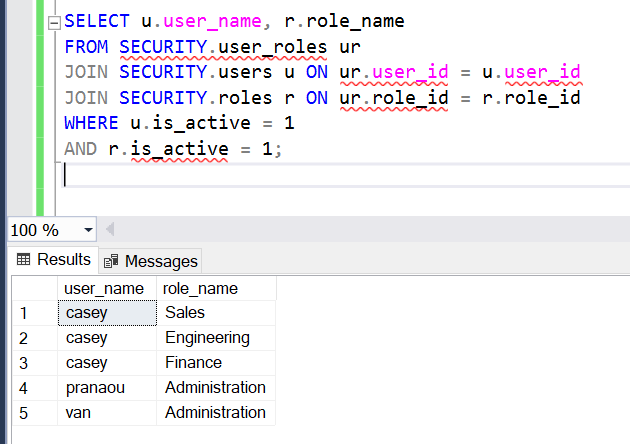
5. Check if the role\_permissions have been added to Sales.



6. Execute the GrantAccessToRoleFunction

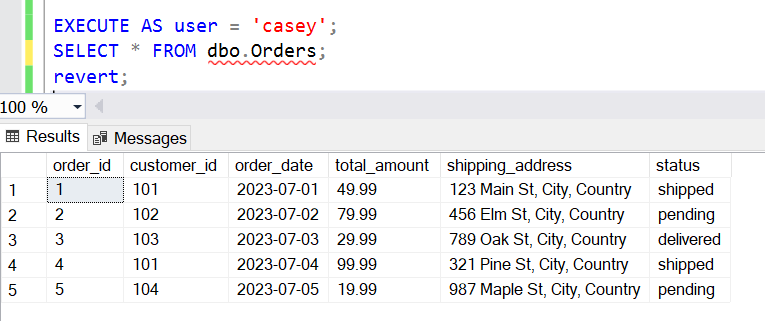
*EXEC SECURITY.GrantAccessToRole;*

7. Validation: Execute as 1 user with Sales role access and 1 user without the access to query the Orders table.



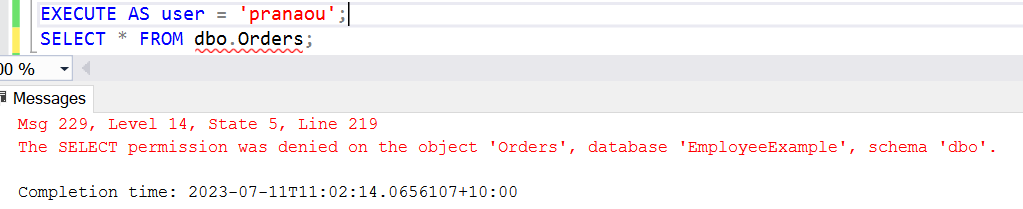
*EXECUTE AS user = 'casey';*

*SELECT \* FROM dbo.orders;*

**

*EXECUTE AS user = 'pranaou';*

*SELECT \* FROM dbo.orders;*



*Test Case Result:*

Pass

4.3 Roles

4.3.1 Add a new role

*Test Case Description:*

The company is creating an 'Audit' role. But the role should not be active until further notification.

*Expected Result:*

The role 'Audit' is created and the is\_active status equals to 0.

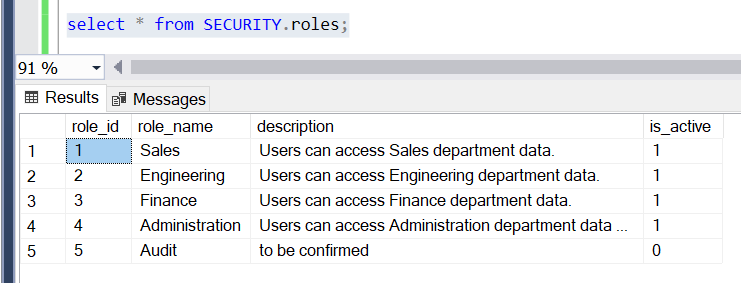
*Steps of Execution:*

1. Create a 'Audit' role in the roles table.

*INSERT INTO SECURITY.roles (role\_name, description, is\_active)*

*values( 'Audit' , 'to be confirmed',0);*

2. Check if the role has been added to the roles table, it would fire the trigger and create the role automatically.

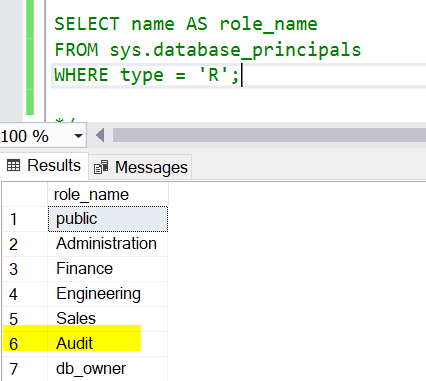


3. Check if the role has been created in the system.

*SELECT name AS role\_name*

*FROM sys.database\_principals*

*WHERE type = 'R';*



*Test Case Result:*

Pass

4.3.2 Remove a permission from a role

*Test Case Description:*

The Sales role can querythe dbo.orders table. However, the orders table should not be accessed by the Sales role until further notice. The access should be removed.

*Expected Result:*

Users in the Sales role could not query the dbo.orders table.

*Steps of Execution:*

1. Find the permission and set the permission is\_active as 0 from the Sales role in the role\_permissions table.

*UPDATE SECURITY.role\_permissions*

*SET is\_active = 0*

*WHERE role\_id IN (*

*SELECT role\_id*

*FROM SECURITY.roles*

*WHERE role\_name = 'Sales'*

*)*

*AND permission\_id IN (*

*SELECT permission\_id*

*FROM SECURITY.permissions*

*WHERE schema\_name = 'dbo'*

*AND object\_name = 'Orders'*

*AND column\_name IS NULL*

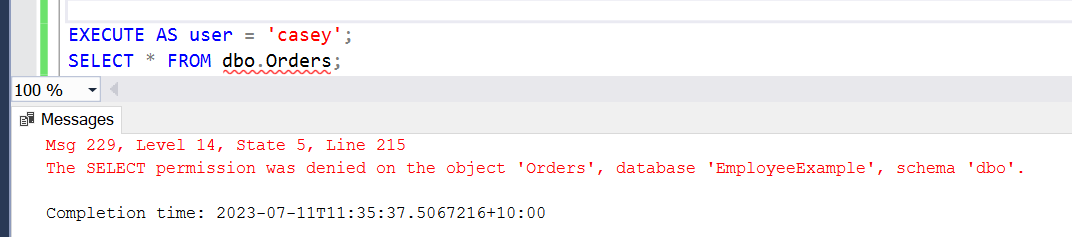
*AND row\_filter IS NULL*

*);*

2.Execute the GrantAccessToRoleFunction

*EXEC SECURITY.GrantAccessToRole;*

3. Validation: Execute as a user in the Sales role to check if the user can have the access or not.



*Test Case Result:*

Pass

5. Future Work

5.1 Risk Management

For the current case, we are using triggers to achieve the automatically creating users and user-role relationship. But we also can use refreshing to run the user-role table to cross check the roles and users and whether they match and generate reports to show the security privilege and data security level.

For the current case, if an operator truncates the table users and roles, or deletes the relationship in the role\_permissions table by mistake, it won’t fire the trigger as it is a truncate function. We may need to check procedures to run every week (or depend on requirements) as part of the framework to make sure all the requirements are met.

5.2 Timestamp

The create\_timestamp or effective\_date should also be added to the tables depending on the requirement.

5.3 Views

In the current case, we’re using masking to implement column-level security. We can also create views to achieve the same result.

5.4 Row Level Security

Currently we need to create new functions according to the business requirements. I don't see this is a problem but more as a necessary step.

Suggestions:

Security Audit: Audit tables of security had been triggered, currently we are not having a control over what had happened on our Security Schema.

User Training and Awareness: With the important and critical nature of this project, we need to educate the Database operator with details of the document and the Framework.

Incident Response Plan

Regular Updates and Patch Management

Continuous Monitoring.

Encryption and Data Protection: Evaluate the need for implementing additional encryption measures, such as using encrypt functions, transformation which allow reporters to analyze but limited the access to actual sensitive data