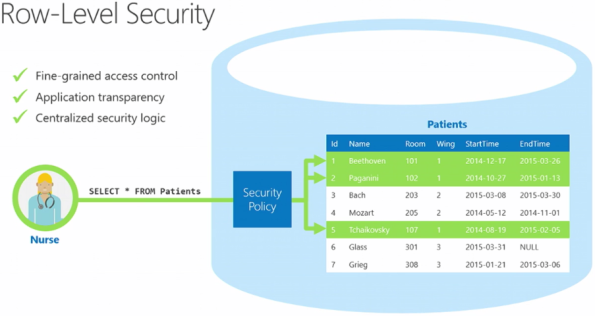
**Implementing “Row Level Security” (RLS) with “Filter Predicate” in SQL Server 2016**

## Part 1

For an Organization its Data is the most important thing, and the Org’s IT department takes lot of measures to make sure data does not fall in wrong hands. The DBA’s and Application programmers setup different layers of security on top of data so that the user is only able to see the Filtered data/rows from a table that he has access to. DB programmers normally create abstracted Views or Stored Procedures with complex logic on top of the Tables by using other master-mapping tables that contains the user-data access key. Sometimes these security logics are not upto the standards, and may have various versions on a database and thus it becomes difficult to track, update and make them fail-proof. So, with **SQL Server 2016** the new **Row Level Security** feature is going to handle this Out of the Box and as a Standard.

**RLS** or Row Level Security is a feature that enables **fine grained control** over access to rows in a table, allowing you to easily control which users can access which data with complete transparency to the application.

With this feature rows are filtered based on the **Execution Context** of the Query, rather than the current user access rights. A secure logic can be created to determine which user can see which rows and restrict any kind of data (rows) by designing a flexible and robust **Security policy** for a table.

[](https://sqlwithmanoj.files.wordpress.com/2015/07/sql-server-2016-row-level-security-04.png)

**–>** To **setup Row Level Security (RLS)** on a particular table it needs only these simple steps:

1. Create **Users** and Grant Read/SELECT access for a Particular table.

2. Create a new **Inline Table-Valued Function** that will contain the Filter Predicate for that table. This Function Predicate can be a sophisticated business logic with multiple JOINs or a simple WHERE ManagerCode = ‘userHR’.

3. Create a new **Security Policy** for this table and add the above Function (Filter) Predicate to it.

**Please note:** that these Functions & Security Policies should be unique for a table. So to create RLS for an another table, you will need to create separate Function & Security Policy.

**–> Step 1.a. Let’s create some test accounts:** I will create three users for:

1. The CEO, over-all admin of the company data.

2. HR department head

3. Finance department head

|  |  |
| --- | --- |
|  | CREATE USER userCEO WITHOUT LOGIN;  GO  CREATE USER userHR WITHOUT LOGIN;  GO  CREATE USER userFin WITHOUT LOGIN;  GO |

**–> Create a sample table [dbo].[Employee]:** with a self-referencing Manager ID column.

|  |  |
| --- | --- |
|  | CREATE TABLE dbo.Employees (      [EmpCode] NVARCHAR(50),  -- Employee ID      [EmpName] NVARCHAR(250), -- Employee/Manager Full Name      [Salary]  MONEY,         -- Fictious Salary      [MgrCode] NVARCHAR(50)   -- Manager ID  );  GO |

**-> Now insert some test records:**

|  |  |
| --- | --- |
|  | -- Top Boss CEO  INSERT INTO dbo.Employees VALUES ('userCEO' , 'CEO Top Boss'  , 800, NULL)    -- Next 2 levels under CEO  INSERT INTO dbo.Employees VALUES ('userHR'  , 'HR User'       , 700, 'userCEO');  INSERT INTO dbo.Employees VALUES ('userFin' , 'Finance User'  , 600, 'userCEO');    -- Employees under Kevin  INSERT INTO dbo.Employees VALUES ('manojp'  , 'Manoj Pandey'  , 100, 'userHR');  INSERT INTO dbo.Employees VALUES ('saurabhs', 'Saurabh Sharma', 400, 'userHR');  INSERT INTO dbo.Employees VALUES ('deepakb' , 'Deepak Biswal' , 500, 'userHR');    -- Employees under Amy  INSERT INTO dbo.Employees VALUES ('keshavk' , 'Keshav K'      , 200, 'userFin');  INSERT INTO dbo.Employees VALUES ('viveks'  , 'Vivek S'       , 300, 'userFin');  GO |

**–> Let’s check the records before applying “Row Level Security”:**

|  |  |
| --- | --- |
| 1  2 | SELECT \* FROM dbo.Employees; -- 8 rows  GO |

As a normal SEELCT and without RLS, it just ignores my Execution Context and execute the Query and return all the 8 rows.

**–>** The **Traditional** way to setup the **Row Level Security** till now was as follows (a simple example):

|  |  |
| --- | --- |
|  | -- Stored Procedure with User-Name passed as parameter:  CREATE PROCEDURE dbo.uspGetEmployeeDetails (@userAccess NVARCHAR(50))  AS  BEGIN      SELECT \*      FROM dbo.Employees      WHERE [MgrCode] = @userAccess      OR @userAccess = 'userCEO'; -- CEO, the admin should see all rows  END  GO    -- Execute the SP with different parameter values:  EXEC dbo.uspGetEmployeeDetails @userAccess = 'userHR'  -- only 3 rows  GO  EXEC dbo.uspGetEmployeeDetails @userAccess = 'userFin' -- only 2 rows  GO  EXEC dbo.uspGetEmployeeDetails @userAccess = 'userCEO' -- all 8 rows  GO |

The above method is prone to issues, like SQL Injection and any other user can apply other user’s User-Name and get the information that he is not allowed to see. With this type of method you have to apply another security layer at the application level so whenever a user executes the SP it gets executed with the same user’s User-Name.

**–>** The **new Row Level Security** feature let you:  
– apply this security at the database level  
– and there is no need to apply the WHERE clause filter for the User-Name.

This makes the security system more reliable and robust by reducing the surface area of your security system.

**–> Step 1.b. Grant Read/SELECT access on the dbo.Employee table to all 3 users:**

|  |  |
| --- | --- |
|  | GRANT SELECT ON dbo.Employees TO userCEO;  GO  GRANT SELECT ON dbo.Employees TO userHR;  GO  GRANT SELECT ON dbo.Employees TO userFin;  GO |

**–> Step 2. Let’s create an Inline Table-Valued Function to write our Filter logic:**

|  |  |
| --- | --- |
|  | CREATE FUNCTION dbo.fn\_SecurityPredicateEmployee(@mgrCode AS sysname)      RETURNS TABLE  WITH SCHEMABINDING  AS      RETURN SELECT 1 AS fn\_SecurityPredicateEmployee\_result      -- Predicate logic      WHERE @mgrCode = USER\_NAME()      OR USER\_NAME() = 'userCEO'; -- CEO, the admin should see all rows  GO |

This function returns value 1 when:

– a row in the MgrCode (i.e. the Manager Code) column is the same as the user executing the query (@@mgrCode = USER\_NAME())

– or if the user executing the query is the Top Boss user (USER\_NAME() = ‘userCEO’)

**–> Step 3. Create a security policy adding the function as a filter predicate:**

|  |  |
| --- | --- |
|  | CREATE SECURITY POLICY ManagerFilter  ADD FILTER PREDICATE dbo.fn\_SecurityPredicateEmployee(MgrCode)  -- Filter Column from dbo.Employee table  ON dbo.Employees  WITH (STATE = ON); -- The state must be set to ON to enable the policy.  GO |

The above Security Policy takes the **Filter Predicate** Logic from the associated Function and applies it to the Query as a WHERE clause.

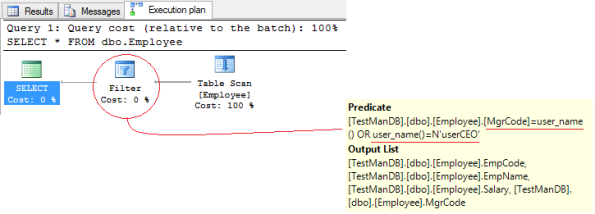
**–> Now let’s again check the records after applying “Row Level Security”:**

|  |  |
| --- | --- |
|  | SELECT \* FROM dbo.Employees; -- 0 rows,  GO |

The simple “SELECT \*” statement will fetch me zero rows after applying RLS, as my user ID is not configured to have access to any of those rows & Table.

**–>** And if you check in the **Execution Plan** of above SELECT statement **without WHERE** clause, it will show you the **Filter Predicate** that is added by the Security Policy defined in Step #3 for applying RLS on this table, which looks like this:

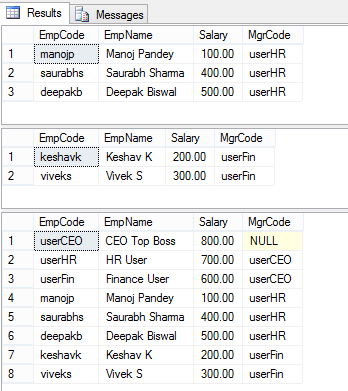
**[TestManDB].[dbo].[Employee].[MgrCode]=user\_name()  
OR user\_name()=N’userCEO’**

[](https://sqlwithmanoj.files.wordpress.com/2015/07/sql-server-2016-row-level-security-10.png)

**–> Let’s check the 3 users we created and provided them customized access to the dbo.Employee table and rows in it:**

|  |  |
| --- | --- |
|  | -- Execute as our immediate boss userHR (3 rows):  EXECUTE AS USER = 'userHR';  SELECT \* FROM dbo.Employees; -- 3 rows  REVERT;  GO    -- Execute as our immediate boss userFin:  EXECUTE AS USER = 'userFin';  SELECT \* FROM dbo.Employees; -- 2 rows  REVERT;  GO    -- Execute as our Top boss userCEO (8):  EXECUTE AS USER = 'userCEO';  SELECT \* FROM dbo.Employees; -- 8 rows  REVERT;  GO |

**–>** The results of the above 3 SELECTs looks like this:

[](https://sqlwithmanoj.files.wordpress.com/2015/07/sql-server-2016-row-level-security-01.png)

So, as you can see the three users we created resulted in different results:

– The HR & Finance users got just 3 & 2 rows, respectively.

– But the admin CEO user got all the 8 rows.

**–>** Thus, by using **RLS Filter Predicate** feature in **SQL Server 2016** you can create your own **customized Security** by creating an **Inline Table-Valued Function** that is linked to the **Security Policy** for your Table.

**–>** Check **Block Predicates** with RLS in my **[**[**next post, Part 2**](https://sqlwithmanoj.com/2016/01/09/implementing-row-level-security-rls-with-block-predicates-in-sql-server-2016-part-2/)**]**.

**–> Final Cleanup**

|  |  |
| --- | --- |
|  | DROP SECURITY POLICY [dbo].[ManagerFilter]  GO  DROP FUNCTION [dbo].[fn\_SecurityPredicateEmployee]  GO  DROP TABLE [dbo].[Employee]  GO    DROP PROCEDURE dbo.uspGetEmployeeDetails  GO |

**Part 2**

There I discussed about the **Filter Predicate** option available in the CTP 2.x release. With this option we can **restrict read** access for a user to avoid unauthorized access of rows.

**Filter Predicates** filters the read operations like **SELECT**, **UPDATE** and **DELETE**, but do not filter **INSERT** operations, and thus a user can INSERT any row (associate to another user also).

Here in this post I will talk about the new **Block Predicate** option available in the CTP 3.0 release. With this option we can **restrict write** access for specific users.

**Block Predicates** block all write operations like:

**– AFTER INSERT** and **AFTER UPDATE**

**– BEFORE UPDATE**

– and **BEFORE DELETE**

To know more about these write operations check [**MS BoL here**](https://msdn.microsoft.com/en-us/library/dn765131.aspx#Anchor_1).

I’m using the same script I used in my **[**[**previous post**](https://sqlwithmanoj.com/2015/07/13/implementing-row-level-security-rls-with-sql-server-2016/)**]** about RLS with Filter Predicates.

**–> Step #1.** Create some test accounts and test data:

|  |  |
| --- | --- |
|  | -- Let's create some test accounts:    -- Three users for The CEO, HR &amp; Finance department admin users.  CREATE USER userCEO WITHOUT LOGIN;  GO  CREATE USER userHR WITHOUT LOGIN;  GO  CREATE USER userFin WITHOUT LOGIN;  GO  CREATE USER userAdmin WITHOUT LOGIN;  GO    -- Create a sample table [dbo].[Employee]: with a self-referencing Manager ID column.  CREATE TABLE dbo.Employee (      [EmpCode] NVARCHAR(50),  -- Employee ID      [EmpName] NVARCHAR(250), -- Employee/Manager Full Name      [Salary]  MONEY,         -- Fictious Salary      [MgrCode] NVARCHAR(50)   -- Manager ID  );  GO    -- Now insert some test records:    -- Top Boss CEO  INSERT INTO dbo.Employee VALUES ('userCEO' , 'CEO Top Boss'  , 800, NULL)    -- Next 2 levels under CEO  INSERT INTO dbo.Employee VALUES ('userHR'  , 'HR User'       , 700, 'userCEO');  INSERT INTO dbo.Employee VALUES ('userFin' , 'Finance User'  , 600, 'userCEO');    -- Employees under Kevin  INSERT INTO dbo.Employee VALUES ('manojp'  , 'Manoj Pandey'  , 100, 'userHR');  INSERT INTO dbo.Employee VALUES ('saurabhs', 'Saurabh Sharma', 400, 'userHR');  INSERT INTO dbo.Employee VALUES ('deepakb' , 'Deepak Biswal' , 500, 'userHR');    -- Employees under Amy  INSERT INTO dbo.Employee VALUES ('keshavk' , 'Keshav K'      , 200, 'userFin');  INSERT INTO dbo.Employee VALUES ('viveks'  , 'Vivek S'       , 300, 'userFin');  GO    -- Let's check the records before applying "Row Level Security":  SELECT \* FROM dbo.Employee; -- 8 rows  GO |

**Output:** As a normal SEELCT and without RLS, it just ignores my Execution Context and execute the Query and return all the 8 rows.

**–> Step #2.** Grant only SELECT to the CEO-User, and Grant SELECT, INSERT,UPDATE, DELETE access on the dbo.Employee table to all 3 users:

|  |  |
| --- | --- |
|  | GRANT SELECT ON dbo.Employee TO userCEO;  GO  GRANT SELECT, INSERT, UPDATE, DELETE ON dbo.Employee TO userHR;  GO  GRANT SELECT, INSERT, UPDATE, DELETE ON dbo.Employee TO userFin;  GO  GRANT SELECT, INSERT, UPDATE, DELETE ON dbo.Employee TO userAdmin;  GO |

**The new “Row Level Security” feature lets you:**

– apply this security at the database level

– create two predicate functions, Filter & Block Predicates.

– and there is no need to apply the WHERE clause to filter out unauthorized users.

**–> Step #3a.** Create an Inline Table-Valued Function to create a **Filter Predicate** function:

|  |  |
| --- | --- |
|  | CREATE FUNCTION dbo.fn\_SecurityFilterPredicateEmployee (@mgrCode AS sysname)      RETURNS TABLE  WITH SCHEMABINDING  AS      RETURN SELECT 1 AS fn\_SecurityPredicateEmployee\_result      -- Predicate logic      WHERE @mgrCode = USER\_NAME()      OR USER\_NAME() IN ('userCEO', 'userAdmin'); -- the CEO and Admin should see all rows  GO |

**–> Step #3b.** Create an Inline Table-Valued Function to create a **Block Predicate** function:

|  |  |
| --- | --- |
|  | CREATE FUNCTION dbo.fn\_SecurityBlockPredicateEmployee (@mgrCode AS sysname)      RETURNS TABLE  WITH SCHEMABINDING  AS      RETURN SELECT 1 AS fn\_SecurityPredicateEmployee\_result      -- Predicate logic      WHERE @mgrCode = USER\_NAME()      OR USER\_NAME() = 'userAdmin'; -- the Admin should have SELECT, INSERT, UPDATE, DELETE access to all rows  GO |

Both the functions returns value 1 when a row in the MgrCode (Manager ID) column is the same as the user executing the query (@@mgrCode = USER\_NAME()) or if the user executing the query is the Top Boss user USER\_NAME() IN (‘userCEO’, ‘userAdmin’) and (USER\_NAME() = ‘userCEO’) respectively.

**–> Step #4.** Create a security policy adding both the functions for Filter & Block predicates:

|  |  |
| --- | --- |
|  | CREATE SECURITY POLICY ManagerFilter  ADD FILTER PREDICATE      dbo.fn\_SecurityFilterPredicateEmployee(MgrCode) ON dbo.Employee,  ADD BLOCK PREDICATE      dbo.fn\_SecurityBlockPredicateEmployee(MgrCode) ON dbo.Employee  WITH (STATE = ON); -- The state must be set to ON to enable the policy.  GO    -- Now let's again check the records after applying "Row Level Security":  SELECT \* FROM dbo.Employee; -- 0 rows, because my used does not have any access.  GO |

Finally with the above 4 steps we’ve configured RLS with both Filter & Block predicates, to restrict unauthorized access and modification of data.

**Note:** to check and validate **RLS with Filter Predicate** while retrieving data (or SELECT) please refer my **[**[**previous post**](https://sqlwithmanoj.com/2015/07/13/implementing-row-level-security-rls-with-sql-server-2016/)**]**.

**–>** Now, let’s check and validate **RLS with Block Predicate** while Updating data (INSERT, UPDATE, DELETE):

**Usage #1.** First we will check our top boss ‘userCEO’ account which has just read/SELECT access:

|  |  |
| --- | --- |
|  | EXECUTE AS USER = 'userCEO';      SELECT \* FROM dbo.Employee; -- 8 rows    -- 1. Insert:      INSERT INTO dbo.Employee (EmpCode, EmpName, Salary, MgrCode)      VALUES ('akashm' , 'Akash M'  , 550, 'userCEO') -- error  /\*  Msg 229, Level 14, State 5, Line 106  The INSERT permission was denied on the object 'Employee', database 'TestManDB', schema 'dbo'.  \*/    -- 2. Update:      UPDATE [dbo].[Employee] SET Salary = 900 WHERE EmpCode = 'akashm' -- error  /\*  Msg 229, Level 14, State 5, Line 111  The UPDATE permission was denied on the object 'Employee', database 'TestManDB', schema 'dbo'.  \*/    -- 3. Delete:      DELETE FROM [dbo].[Employee] WHERE EmpCode = 'akashm' -- error  /\*  Msg 229, Level 14, State 5, Line 118  The DELETE permission was denied on the object 'Employee', database 'TestManDB', schema 'dbo'.  \*/  REVERT;  GO |

As you can see above for ‘userCEO’ account all INSERT/UPDATE/DELETE operations failed on the Employee table, as it only has read/SELECT access.  
nbsp;

**Usage #2.** Now let’s check the ‘userHR’ account which as all permissions on his data:

|  |  |
| --- | --- |
|  | EXECUTE AS USER = 'userHR';      SELECT \* FROM dbo.Employee; -- 3 rows    -- 1. Insert:      INSERT INTO dbo.Employee (EmpCode, EmpName, Salary, MgrCode)      VALUES ('akashm' , 'Akash M'  , 550, 'userHR')      SELECT \* FROM dbo.Employee; -- 4 rows    -- 2: Update:      UPDATE [dbo].[Employee] SET Salary = 900 WHERE EmpCode = 'akashm'      SELECT \* FROM dbo.Employee; -- 4 rows, with Salary = 900    -- 3: Delete:      DELETE FROM [dbo].[Employee] WHERE EmpCode = 'akashm'      SELECT \* FROM dbo.Employee; -- 3 rows    -- 4. Update another user's row:      UPDATE [dbo].[Employee] SET Salary = 1000 WHERE EmpCode = 'viveks'      -- (0 row(s) affected)  REVERT;  GO |

So, the ‘userHR’ account can modify his records in all 3 INSERT/UPDATE/DELETE operations. But the 4th UPDATE operation fails where he tries to update another user (‘userFin’) record. However this does not displays any error message, but the query output message clearly shows no records were updated.

**Usage #3.** Execute as our Admin who can insert record for any User:

|  |  |
| --- | --- |
|  | EXECUTE AS USER = 'userAdmin';      SELECT \* FROM dbo.Employee; -- 8 rows    -- 1. Insert row for another user HR:      INSERT INTO dbo.Employee (EmpCode, EmpName, Salary, MgrCode)      VALUES ('akashm' , 'Akash M'  , 550, 'userHR')      SELECT \* FROM dbo.Employee; -- 9 rows    -- 2. Insert row for another user Fin:      INSERT INTO dbo.Employee (EmpCode, EmpName, Salary, MgrCode)      VALUES ('shantis' , 'Shanti S'  , 200, 'userFin')      SELECT \* FROM dbo.Employee; -- 10 rows    -- 3. Update other user's rows (HR & Fin):      UPDATE [dbo].[Employee] SET Salary = 900 WHERE EmpCode IN ('akashm', 'shantis')      SELECT \* FROM dbo.Employee; -- 10 rows total, last 2 rows with Salary = 900    -- 4. Delete other user's rows (HR & Fin)::      DELETE FROM [dbo].[Employee] WHERE EmpCode IN ('akashm', 'shantis')      SELECT \* FROM dbo.Employee; -- 8 rows  REVERT;  GO |

And here you can see the ‘userAdmin’ account is able to update all rows belonging to other users as well, as it is configured to behave like that, and this logic is built into the Security function that we build above in Step 3.b, where the Predicate logic looks like this:

|  |  |
| --- | --- |
|  | -- Predicate logic  WHERE @mgrCode = USER\_NAME()  OR USER\_NAME() = 'userAdmin'; |

**–>** Thus, by using Row Level Security feature in SQL Server 2016 you can build your own customized Security feature for Read & Write operations for Tables by creating Inline Table-Valued Functions with Filter & Block predicates which are linked to the Security Policy for respective tables.

**–> Final Cleanup**

|  |  |
| --- | --- |
|  | USE [TestManDB]  GO  DROP SECURITY POLICY [dbo].[ManagerFilter]  GO  DROP FUNCTION [dbo].[fn\_SecurityFilterPredicateEmployee]  GO  DROP FUNCTION [dbo].[fn\_SecurityBlockPredicateEmployee]  GO  DROP TABLE [dbo].[Employee]  GO |