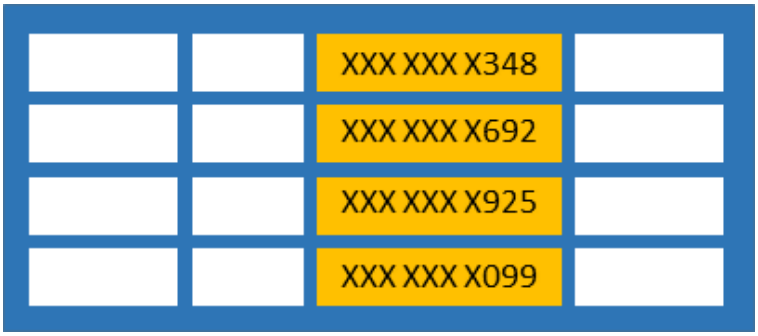
Dynamic Data Masking

09/26/2016 • 9 minutes to read • Contributors

In this article

Defining a Dynamic Data Mask
Permissions
Best Practices and Common Use Cases
Querying for Masked Columns
Limitations and Restrictions
Security Note: Bypassing masking using inference or brute-force techniques
Examples
See Also

THIS TOPIC APPLIES TO: SQL Server Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse



Dynamic data masking limits (DDM) sensitive data exposure by masking it to non-privileged users. It can be used to greatly simplify the design and coding of security in your application.

Dynamic data masking helps prevent unauthorized access to sensitive data by enabling customers to designate how much of the sensitive data to reveal with minimal impact on the application layer. DDM can be configured on the database to hide sensitive data in the result sets of queries over designated database fields, while the data in the database is not changed. Dynamic data masking is easy to use with

FEEDBACK

existing applications, since masking rules are applied in the query results. Many applications can mask sensitive data without modifying existing queries.

- A central data masking policy acts directly on sensitive fields in the database.
- Designate privileged users or roles that do have access to the sensitive data.
- DDM features full masking and partial masking functions, as well as a random mask for numeric data.
- Simple Transact-SQL commands define and manage masks.

As an example, a call center support person may identify callers by several digits of their social security number or credit card number, but those data items should not be fully exposed to the support person. A masking rule can be defined that masks all but the last four digits of any social security number or credit card number in the result set of any query. For another example, by using the appropriate data mask to protect personally identifiable information (PII) data, a developer can query production environments for troubleshooting purposes without violating compliance regulations.

The purpose of dynamic data masking is to limit exposure of sensitive data, preventing users who should not have access to the data from viewing it. Dynamic data masking does not aim to prevent database users from connecting directly to the database and running exhaustive queries that expose pieces of the sensitive data. Dynamic data masking is complementary to other SQL Server security features (auditing, encryption, row level security...) and it is highly recommended to use this feature in conjunction with them in addition in order to better protect the sensitive data in the database.

Dynamic data masking is available in SQL Server 2016 and Azure SQL Database, and is configured by using Transact-SQL commands. For additional information about configuring dynamic data masking by using the Azure portal, see [Get started with SQL Database Dynamic Data Masking (Azure portal)](#).

Defining a Dynamic Data Mask

A masking rule may be defined on a column in a table, in order to obfuscate the data in that column. Four types of masks are available.

Function	Description	Examples

Default	Full masking according to the data types of the designated fields.
	
	For string data types, use XXXX or fewer Xs if the size of the field is less than 4 characters (**char**, **nchar**, **varchar**, **nvarchar**, **text**, **ntext**).
	
	For numeric data types use a zero value (**bigint**, **bit**, **decimal**, **int**, **money**, **numeric**, **smallint**,
		Example column definition syntax:
		`Phone# varchar(12) MASKED WITH (FUNCTION = 'default()') NULL`
		
		Example alter syntax:
		`ALTER COLUMN Gender ADD MASKED WITH (FUNCTION = 'default()')`

	smallmoney, **tinyint**, **float**, **real**). For date and time data types use 01.01.1900 00:00:00.0000000 (**date**, **datetime2**, **datetime**, **datetimeoffset**, **smalldatetime**, **time**). For binary data types use a single byte of ASCII value 0 (**binary**, **varbinary**, **image**).	

Email	Masking method which exposes the first letter of an email address and the constant suffix ".com", in the form of an email address. . `aXXX@XXXX.com`.	Example definition syntax:
		`Email varchar(100) MASKED WITH (FUNCTION = 'email()') NULL`
		
		Example alter syntax:
		`ALTER COLUMN Email ADD MASKED WITH (FUNCTION = 'email()')`

Random	A random masking function for use on any numeric type to mask the original value with a random value within a specified range.	Example definition syntax:
		`Account_Number bigint MASKED WITH (FUNCTION = 'random([start range], [end range])')`
		
		Example alter syntax:
		`ALTER COLUMN [Month] ADD MASKED WITH (FUNCTION = 'random(1, 12)')`

Custom String	Masking method which exposes the first and last letters and adds a custom padding string in the middle.
	`prefix,[padding],suffix`
	
	Note: If the original value is too short to complete the entire mask, part of the prefix or suffix will not be exposed.
		Example definition syntax:
		`FirstName varchar(100) MASKED WITH (FUNCTION = 'partial(prefix, [padding],suffix)') NULL`
		
		Example alter syntax:
		`ALTER COLUMN [Phone Number] ADD MASKED WITH (FUNCTION = 'partial(1,"XXXXXXX",0)')`
		
		Additional examples:

```
ALTER COLUMN [Phone
Number] ADD MASKED WITH
(FUNCTION =
'partial(5,"XXXXXXX",0)')
```

```
ALTER COLUMN [Social
Security Number] ADD
MASKED WITH (FUNCTION =
'partial(0,"XXX-XX-
",4)')
```

Permissions

You do not need any special permission to create a table with a dynamic data mask, only the standard **CREATE TABLE** and **ALTER** on schema permissions.

Adding, replacing, or removing the mask of a column, requires the **ALTER ANY MASK** permission and **ALTER** permission on the table. It is appropriate to grant **ALTER ANY MASK** to a security officer.

Users with **SELECT** permission on a table can view the table data. Columns that are defined as masked, will display the masked data. Grant the **UNMASK** permission to a user to enable them to retrieve unmasked data from the columns for which masking is defined.

The **CONTROL** permission on the database includes both the **ALTER ANY MASK** and **UNMASK** permission.

Best Practices and Common Use Cases

- Creating a mask on a column does not prevent updates to that column. So although users receive masked data when querying the masked column, the same users can update the data if they have write permissions. A proper access control policy should still be used to limit update permissions.
- Using `SELECT INTO` or `INSERT INTO` to copy data from a masked column into another table results in masked data in the target table.
- Dynamic Data Masking is applied when running SQL Server Import and Export. A database containing masked columns will result in a backup file with masked data (assuming it is exported by a user without **UNMASK** privileges), and the imported database will contain statically masked data.

Querying for Masked Columns

Use the **sys.masked_columns** view to query for table-columns that have a masking function applied to them. This view inherits from the **sys.columns** view. It returns all columns in the **sys.columns** view, plus the **is_masked** and **masking_function** columns, indicating if the column is masked, and if so, what masking function is defined. This view only shows the columns on which there is a masking function applied.

Copy

```
SELECT c.name, tbl.name as table_name, c.is_masked, c.masking_function
FROM sys.masked_columns AS c
JOIN sys.tables AS tbl
    ON c.[object_id] = tbl.[object_id]
WHERE is_masked = 1;
```

Limitations and Restrictions

A masking rule cannot be defined for the following column types:

- Encrypted columns (Always Encrypted)
- FILESTREAM
- COLUMN_SET or a sparse column that is part of a column set.
- A mask cannot be configured on a computed column, but if the computed column depends on a column with a MASK, then the computed column will return masked data.
- A column with data masking cannot be a key for a FULLTEXT index.

  For users without the **UNMASK** permission, the deprecated **READTEXT**, **UPDATETEXT**, and **WRITETEXT** statements do not function properly on a column configured for Dynamic Data Masking.

  Adding a dynamic data mask is implemented as a schema change on the underlying table, and therefor cannot be performed on a column with dependencies. To work around this restriction, you can first remove the dependency, then add the dynamic data mask and then re-create the dependency. For example, if the dependency is due to an index dependent on that column, you can drop the index, then add the mask, and then re-create the dependent index.

Security Note: Bypassing masking using inference or brute-force techniques

Dynamic Data Masking is designed to simplify application development by limiting data exposure in a set of pre-defined queries used by the application. While Dynamic Data Masking can also be useful to

prevent accidental exposure of sensitive data when accessing a production database directly, it is important to note that unprivileged users with ad-hoc query permissions can apply techniques to gain access to the actual data. If there is a need to grant such ad-hoc access, Auditing should be used to monitor all database activity and mitigate this scenario.

As an example, consider a database principal that has sufficient privileges to run ad-hoc queries on the database, and tries to 'guess' the underlying data and ultimately infer the actual values. Assume that we have a mask defined on the `[Employee].[Salary]` column, and this user connects directly to the database and starts guessing values, eventually inferring the `[Salary]` value of a set of Employees:

Copy

```
SELECT ID, Name, Salary FROM Employees
WHERE Salary > 99999 and Salary < 100001;
```

Id	Name	Salary
62543	Jane Doe	0
91245	John Smith	0

This demonstrates that Dynamic Data Masking should not be used as an isolated measure to fully secure sensitive data from users running ad-hoc queries on the database. It is appropriate for preventing accidental sensitive data exposure, but will not protect against malicious intent to infer the underlying data.

It is important to properly manage the permissions on the database, and to always follow the minimal required permissions principle. Also, remember to have Auditing enabled to track all activities taking place on the database.

Examples

Creating a Dynamic Data Mask

The following example creates a table with three different types of dynamic data masks. The example populates the table, and selects to show the result.

Copy

```
CREATE TABLE Membership
  (MemberID int IDENTITY PRIMARY KEY,
   FirstName varchar(100) MASKED WITH (FUNCTION = 'partial(1,"XXXXXXX",0)') NULL,
   LastName varchar(100) NOT NULL,
```

```
   Phone# varchar(12) MASKED WITH (FUNCTION = 'default()') NULL,
   Email varchar(100) MASKED WITH (FUNCTION = 'email()') NULL);

INSERT Membership (FirstName, LastName, Phone#, Email) VALUES
('Roberto', 'Tamburello', '555.123.4567', 'RTamburello@contoso.com'),
('Janice', 'Galvin', '555.123.4568', 'JGalvin@contoso.com.co'),
('Zheng', 'Mu', '555.123.4569', 'ZMu@contoso.net');
SELECT * FROM Membership;
```

A new user is created and granted **SELECT** permission on the table. Queries executed as the `TestUser` view masked data.

Copy

```
CREATE USER TestUser WITHOUT LOGIN;
GRANT SELECT ON Membership TO TestUser;

EXECUTE AS USER = 'TestUser';
SELECT * FROM Membership;
REVERT;
```

The result demonstrates the masks by changing the data from

`1 Roberto Tamburello 555.123.4567 RTamburello@contoso.com`

into

`1 RXXXXXXX Tamburello xxxx RXXX@XXXX.com`

Adding or Editing a Mask on an Existing Column

Use the **ALTER TABLE** statement to add a mask to an existing column in the table, or to edit the mask on that column.
The following example adds a masking function to th `LastName` column:

Copy

```
ALTER TABLE Membership
ALTER COLUMN LastName ADD MASKED WITH (FUNCTION = 'partial(2,"XXX",0)');
```

The following example changes a masking function on the `LastName` column:

Copy

```
ALTER TABLE Membership
ALTER COLUMN LastName varchar(100) MASKED WITH (FUNCTION = 'default()');
```