**Veiw**

View can be described as virtual table which derived its data from one or more than one table columns.It is stored in the database. It is used to implements the security mechanism in the Sql Server. For example, suppose there is table called Employeeinfo whose structure is given below:

Views are virtual tables that are compiled at run time. The data associated with views are not physically stored in the view, but it is stored in the base tables of the view. A view can be made over one or more database tables. Generally we put those columns in view that we need to retrieve/query again and again. Once you have created the view, you can query view like as table. We can make index, trigger on view.

In Sql Server we make views for security purpose since it restricts the user to view some columns/fields of the table(s). Views show only those columns that are present in the query which is used to make view.One more advantage of Views is, data abstraction since the end user is not aware of all the data present in database table.

Create table EmployeeInfo(EmpId int, EmpName nvarchar(200),

EmpLogin nvarchar(20), Emppassword nvarchar(20) , EmploymentDate datetime )

And it contains the following data

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EmpId** | **EmpName** | **EmpLogin** | **Emppassword** | **EmploymentDate** |
| 1 | Vivek Johari | Vivek | VikJoh | 29/01/2006 |
| 2 | Virender Singh | Virender | Virender | 06/02/2007 |
| 3 | Raman Thakur | Raman | Raman | 14/05/2007 |
| 4 | Uma Dutt Sharma | Uma | Uma | 30/03/2008 |
| 5 | Ravi Kumar Thakur | Ravi | Ravi | 30/06/2007 |

Now suppose that the Administrator do not want that the users have excess to the table EmployeeInfo which contains the some critical information (Emplogin, EmpPassword etc) of the Employees. So he can create a view which gives the empid, empname, employmentdate as the output and give the permission for the view to the user. In this way the administrator do not need to bother about giving the access permission for the table to the user.

The syntax for creating a View is given below:

 Collapse | [Copy Code](http://www.codeproject.com/Articles/127726/Introduction-to-Sql-Server-Views)

Create View Viewname As

Select Column1, Column2 From Tablename

Where (Condition) Group by (Grouping Condition) having (having Condition)

For example,

 Collapse | [Copy Code](http://www.codeproject.com/Articles/127726/Introduction-to-Sql-Server-Views)

Create View View\_Employeeinfo As

Select EmpId, EmpName, employmentdate From EmployeeInfo

Now user can use the view View\_EmployeeInfo as a table to get the empid , empname and employmentdate information of the employees by using the giving query

 Collapse | [Copy Code](http://www.codeproject.com/Articles/127726/Introduction-to-Sql-Server-Views)

Select \* from View\_EmployeeInfo where empid=2

It would gives the following result

|  |  |  |
| --- | --- | --- |
| **EmpId** | **EmpName** | **EmploymentDate** |
| 2 | Virender Singh | 06/02/2007 |

We can also use Sql Joins in the Select statement in deriving the data for the view.

 Collapse | [Copy Code](http://www.codeproject.com/Articles/127726/Introduction-to-Sql-Server-Views)

Create table EmpProjInfo (EmpId int, Projectname nvarchar(200))

and it contains the following data

|  |  |
| --- | --- |
| **EmpId** | **Projectname** |
| 1 | Abcbank |
| 2 | AtoZfinancialsol |
| 3 | learningsystem |
| 4 | ebooksystem |
| 5 | AtoZfinancialsol |

Now we can create a view Vw\_EmployeeProj which gives the information about the Employees and its projects

 Collapse | [Copy Code](http://www.codeproject.com/Articles/127726/Introduction-to-Sql-Server-Views)

Create view Vw\_EmployeeProj As

Select EmployeeInfo.EmpId, EmployeeInfo.EmpName,

EmpProjInfo.Projectname from EmployeeInfo inner join

EmpProjInfo on EmployeeInfo.EmpId=EmpProjInfo.EmpId

GRANT

In order for a user to be able to do something, he or she must be given permission to do it. We do this via the GRANT command. However, before we demonstrate that, let's do some setup of a test role and a test user in a test database I've created (aptly called TestDB):

Create Test User

USE TestDB;

GO

CREATE ROLE TestRole;

GO

CREATE USER TestUser WITHOUT LOGIN;

GO

EXEC sp\_addrolemember @rolename = 'TestRole', @membername = 'TestUser';

GO

Create Tables and Permissions

Now let's create a schema, a couple of tables, and let's GRANT the ability to select against the first table.

CREATE SCHEMA Test;

GO

CREATE TABLE Test.TestTable (TableID int);

GO

GRANT SELECT ON OBJECT::Test.TestTable TO TestRole;

GO

CREATE TABLE Test.TestTable2 (TableID int);

GO

Test Harness Queries

Once that is done, let's use two "test harnesses" to test the user's ability to access the tables in question. Note that with the current permissions, the user should only be able to issue a SELECT against the first table.

-- Test Harness to verify how permissions work for Test.TestTable.

EXECUTE AS USER = 'TestUser';

GO

SELECT \* FROM Test.TestTable;

GO

REVERT;

GO

-- Test Harness to verify how permissions work for Test.TestTable2.

EXECUTE AS USER = 'TestUser';

GO

-- This should fail initially, as there is no permission for this table

SELECT \* FROM Test.TestTable2;

GO

REVERT;

GO

Seeing the Permissions

In order to see the permissions that are granted, we'll use the **sys.database\_permissions** catalog view. If you issue this query now, you'll see the first GRANT we made. Re-use this query to see the permissions after each change:

-- Query sys.database\_permissions to see applicable permissions

SELECT dp.class\_desc, s.name AS 'Schema', o.name AS 'Object', dp.permission\_name,

dp.state\_desc, prin.[name] AS 'User'

FROM sys.database\_permissions dp

JOIN sys.database\_principals prin

ON dp.grantee\_principal\_id = prin.principal\_id

JOIN sys.objects o

ON dp.major\_id = o.object\_id

JOIN sys.schemas s

ON o.schema\_id = s.schema\_id

WHERE LEFT(o.name, 9) = 'TestTable'

AND dp.class\_desc = 'OBJECT\_OR\_COLUMN'

UNION ALL

SELECT dp.class\_desc, s.name AS 'Schema', '-----' AS 'Object', dp.permission\_name,

dp.state\_desc, prin.[name] AS 'User'

FROM sys.database\_permissions dp

JOIN sys.database\_principals prin

ON dp.grantee\_principal\_id = prin.principal\_id

JOIN sys.schemas s

ON dp.major\_id = s.schema\_id

WHERE dp.class\_desc = 'SCHEMA';

REVOKE

REVOKE undoes a permission, whether it's a GRANT or a DENY (more on DENY in a minute). If you issue the following REVOKE and then check the permissions, you'll note that the GRANT that was previously present for Test.Table1. After issuing the revoke command, re-run the test harness queries above against that table and you'll see that the user cannot query the table any longer.

-- Let's undo the permission using REVOKE;

REVOKE SELECT ON OBJECT::Test.TestTable FROM TestRole;

Remember, REVOKE doesn't cancel a GRANT. It doesn't block a GRANT. It removes a permission at the level specified to the security principal (user or role) specified. That's why we say it undoes a permission.

DENY

DENY blocks access. DENY trumps all other access. If a user has both a GRANT and a DENY on a given object, by whatever means, the DENY will take effect. For instance, let's consider the case of a GRANT SELECT against the Test schema. This would give the ability to issue a SELECT against any table or view in the Test schema. Try just applying this permission, re-checking the permission, and then testing the user's access to both Test.TestTable and Test.TestTable2. You'll see the user can now issue a SELECT query against both tables.

If you're not familiar with schemas and how they affect permissions, see this [tip on nested permissions due to securables](https://www.mssqltips.com/sqlservertip/1983/sql-server-nested-securable-permissions/). Sometimes, if you look for an explicit permissions against a table or stored procedure, you won't see it. However, the user can execute the SELECT or EXECUTE respectively. If this is the case, then the permission is on a securable that contains the object. That's what we're doing here. The Test schema contains the TestTable and TestTable2 tables. So if a user has SELECT permissions against the Test schema, it also has SELECT permission against any tables and views within the Test schema.

-- Permission at the schema level

GRANT SELECT ON SCHEMA::Test TO TestRole;

GO

Now let's apply a DENY. In this case I'm applying a DENY explicitly to the test user instead of going through a role. And I'm only applying it to the Test.TestTable object. Now re-run the test harness queries. You'll see the access is denied. However, you can still query Test.TestTable2. There isn't a DENY applied against it.

-- Specific DENY will block the GRANT

DENY SELECT ON OBJECT::Test.TestTable TO TestUser;

And if you re-run the permissions script, you'll see all the permissions granted, to include the DENY.

CREATE USER foo WITHOUT LOGIN;

GO

CREATE TABLE dbo.a(id INT);

CREATE TABLE dbo.b(id INT);

GO

CREATE VIEW dbo.v

AS

SELECT a.id FROM a INNER JOIN b ON a.id = b.id;

GO

GRANT SELECT ON dbo.v TO foo;

GO

EXECUTE AS USER = N'foo';

GO

-- works:

SELECT id FROM dbo.v;

GO

-- Msg 229, SELECT denied:

SELECT id FROM dbo.a;

GO

REVERT;

Note that this assumes foo has not been granted elevated privileges through explicit permissions on the schema or database, or through role or group membership.

Since you are using tables in multiple databases (sorry I missed the end of that first sentence initially), you also may need explicit grants on the table(s) in the database where the view does not exist. In order to avoid granting select to the table(s), you could create a view in each database, and then join the views.

Create two databases and a login:

CREATE DATABASE d1;

GO

CREATE DATABASE d2;

GO

USE [master];

GO

CREATE LOGIN blat WITH PASSWORD = 'x', CHECK\_POLICY = OFF;

GO

In database d1, create a user, then create a table and a simple view against that table. Grant select to the user *only* against the view:

USE d1;

GO

CREATE USER blat FROM LOGIN blat;

GO

CREATE TABLE dbo.t1(id INT);

GO

CREATE VIEW dbo.v1

AS

SELECT id FROM dbo.t1;

GO

GRANT SELECT ON dbo.v1 TO blat;

GO

Now, in the second database, create the user, then create another table and a view that joins that table to the view in d1. Grant select only to the view.

USE d2;

GO

CREATE USER blat FROM LOGIN blat;

GO

CREATE TABLE dbo.t2(id INT);

GO

CREATE VIEW dbo.v2

AS

SELECT v1.id FROM dbo.t2

INNER JOIN d1.dbo.v1 AS v1

ON t2.id = v1.id;

GO

GRANT SELECT ON dbo.v2 TO blat;

GO

Now launch a new query window and change the credentials to be for the login blat (EXECUTE ASdoes not work here). Then run the following from the context of either database, and it should work fine:

SELECT id FROM d1.dbo.v2;

These should both yield Msg 229 errors:

SELECT id FROM d1.dbo.t1;

GO

SELECT id FROM d2.dbo.t2;

Results:

Msg 229, Level 14, State 5, Line 1   
The SELECT permission was denied on the object 't1', database 'd1', schema 'dbo'.   
Msg 229, Level 14, State 5, Line 3   
The SELECT permission was denied on the object 't2', database 'd2', schema 'dbo'.

f you want to give your user all read permissions, you could use:

EXEC sp\_addrolemember N'db\_datareader', N'your-user-name'

That adds the default db\_datareader role (read permission on all tables) to that user.

There's also a db\_datawriter role - which gives your user all WRITE permissions (INSERT, UPDATE, DELETE) on all tables:

EXEC sp\_addrolemember N'db\_datawriter', N'your-user-name'

If you need to be more granular, you can use the GRANT command:

GRANT SELECT, INSERT, UPDATE ON dbo.YourTable TO YourUserName

GRANT SELECT, INSERT ON dbo.YourTable2 TO YourUserName

GRANT SELECT, DELETE ON dbo.YourTable3 TO YourUserName

and so forth - you can granularly give SELECT, INSERT, UPDATE, DELETE permission on specific tables.

This is all very well documented in the [MSDN Books Online for SQL Server](http://msdn.microsoft.com/en-us/library/ms187965.aspx).

And yes, you can also do it graphically - in SSMS, go to your database, then Security > Users, right-click on that user you want to give permissions to, then Properties adn at the bottom you see "Database role memberships" where you can add the user to db roles.



Indexes

Indexes allow the database application to find data fast; without reading the whole table.

An index can be created in a table to find data more quickly and efficiently.

The users cannot see the indexes, they are just used to speed up searches/queries.

**Note:** Updating a table with indexes takes more time than updating a table without (because the indexes also need an update). So you should only create indexes on columns (and tables) that will be frequently searched against.

SQL CREATE INDEX Syntax

Creates an index on a table. Duplicate values are allowed:

CREATE INDEX index\_name  
ON table\_name (column\_name)

SQL CREATE UNIQUE INDEX Syntax

Creates a unique index on a table. Duplicate values are not allowed:

CREATE UNIQUE INDEX index\_name  
ON table\_name (column\_name)

**Note:** The syntax for creating indexes varies amongst different databases. Therefore: Check the syntax for creating indexes in your database.

CREATE INDEX Example

The SQL statement below creates an index named "PIndex" on the "LastName" column in the "Persons" table:

CREATE INDEX PIndex  
ON Persons (LastName)

If you want to create an index on a combination of columns, you can list the column names within the parentheses, separated by commas:

CREATE INDEX PIndex  
ON Persons (LastName, FirstName)

|  |  |  |
| --- | --- | --- |
| Clustered | A clustered index sorts and stores the data rows of the table or view in order based on the  clustered index key. The clustered index is implemented as a B-tree index structure that  supports fast retrieval of the rows, based on their clustered index key values. | [Clustered and Nonclustered Indexes Described](http://technet.microsoft.com/en-us/library/ms190457.aspx)  [Create Clustered Indexes](http://technet.microsoft.com/en-us/library/ms186342.aspx) |
| Nonclustered | A nonclustered index can be defined on a table or view with a clustered index or on a heap.  Each index row in the nonclustered index contains the nonclustered key value and a row  locator. This locator points to the data row in the clustered index or heap having the key value.  The rows in the index are stored in the order of the index key values,  but the data rows are not guaranteed to be in any particular order  unless a clustered index is created on the table. |  |