CS61 Chapter 05. Creating, altering, and dropping tables[1]

Contents

Introduction to the Chapter and an Overview of SQL

This chapter introduces SQL and its statements that create and modify the metadata of tables. This metadata includes table and column names, data types of columns, default values for columns, and constraints on columns (NOT NULL, primary and foreign keys, unique, and check). Other chapters introduce statements that create other objects such as indexes and stored procedures. Other chapters also introduce statements that insert, delete, and update the raw data stored in tables.

SQL statements are grouped into three categories:

(1) **Data Definition Language (DDL)** statements create, alter, and drop metadata for tables and many other objects in the database. Some statements such as DROP TABLE would also remove any raw data stored in that table.

[1] This chapter was updated from SQL Server 2005 to SQL Server 2008 by Demetri Gropen

(2) Data Manipulation Language (DML) statements insert, delete, update, and access the raw data stored in tables.
(3) Data Control Language (DCL) statements grant and revoke privileges to database users to execute some or all of the DML and DDL statements.

This chapter describes some DDL statements for tables and the counterparts of these statements using GUIs:

Creating a table:
Naming the table and column(s). Specifying a data type for each column. Specifying default values for columns. Defining constraints (primary key, foreign key(s), NOT NULL, unique, and CHECK constraints that check to see that the value entered for a column is in a certain range or a set of values). Generating IDENTITY values for a column, which are autogenerated values for Student_IDs, Employee_IDs, and other surrogate keys often used as primary keys.

Not all of these must be done when you create a table.

Altering a table:
Adding, changing, dropping the metadata of an existing table (adding and dropping columns; changing data types; and adding, changing, dropping constraints).

Dropping a table:
Removing all metadata and raw data that has been stored for a table.

Renaming a table or column:
Changing the name of a table or column.

We'll learn several ways in Code Editor and Object Explorer to complete these tasks. In Chapter 16 when using Object Explorer, we'll learn how to complete them in a Diagram. This chapter focuses on tables, but the several ways we create them have analogous ways to create other objects in a database. While completing these tasks, we'll learn about both the graphical interfaces and the SQL statements.

We'll also learn how to instruct SQL Server to generate the script to create an existing table. A **script** is a series of SQL statements saved in a text file. A **text file** is a series of characters separated into lines by some end-of-line marker such as the <carriage return> character. No matter how you created the table (through one or more uses of some GUI or SQL statement), SQL Server can reverse engineer the table (one type of object) and generate the SQL statement(s) that would generate that object as it presently exists. Reasons to generate a script for an object that already exists are given later.

Creating Tables

Tables are the basic building blocks of SQL Server database where all raw data is stored. Tables are uniquely named within a database and schema and contain one or more columns. Each column has an associated data type that defines the kind of data that can be stored within it.
You have two options for creating tables. You can either write T-SQL code in Code Editor of SQL Server 2008 Management Studio (SSMS), or you can use Object Explorer / Object Explorer Details – the graphical user interface provided by the SQL Server Management Studio.

Creating Tables in Code Editor

A CREATE TABLE statement must have a table name and at least one column name with its data type. These names must satisfy naming requirements. Of course a database must exist for the table, the table creator must have permission to create tables in that database, and a table name for the same <database>.<user>.<tablename> cannot already exist.

You can qualify the new table name using the database, schema, and table name, or just the schema and table name. In its simplest form, to add a new table to the current database, you specify the table name and then list the table's new columns in parentheses, followed by their data type. Each column requires a defined data type. The data type defines and restricts the type of data the column can hold. You can specify several options for each column definition. Briefly, this might include options such as auto - sequencing identity, default values, constraints, and whether the column value may be set to Null.

You also can explicitly specify the database and owner with *DatabaseName.Owner.TableName.* Since each user in CS61 is restricted to one assigned database, rather than the default owner (your Login name), the following will create table Test with owner dbo:

```
CREATE TABLE dbo.Test
(
	Student_ID  int  NOT NULL
)
GO
```

To avoid confusion in CS61, when you create tables and other objects in Code Editor, explicitly specify owner dbo.

One way to learn syntax is to look at code. The following creates three tables with primary and foreign keys, an auto-generated value for the primary key and default values for several columns. The comments on the right refer to comments beginning on the next page about this code.

```
--Following sets the active database to be cs61db_01. --<<< COMMENT 1 >>>--
USE cs61db_01 --<<< COMMENT 2 >>>--
GO

DROP TABLE dbo.Enroll  --<<< COMMENT 3 >>>--
DROP TABLE dbo.Student
DROP TABLE dbo.Section
```

```
CREATE TABLE dbo.Student
(
      Student_ID        int  IDENTITY,     --<<< COMMENT 4 >>>--
      Last_Name         varchar(25)        NOT NULL,
      First_Name        varchar(25)        NOT NULL,
      Middle_Initial    char(1)            NULL,
      Birth_Date        smalldatetime      NOT NULL,
      Street_Address    varchar(25)        NULL,
      City              varchar(25)        NULL
            DEFAULT 'Santa Monica',        --<<< COMMENT 5 >>>--
      State             char(2)            NULL  DEFAULT 'CA',
      Zipcode           varchar(9)         NULL  DEFAULT '90405',
      Telephone         varchar(10)        NULL,
      Email             varchar(100)       NULL,

      CONSTRAINT PK_Student
            PRIMARY KEY (Student_ID) --<<< COMMENT 6 >>>--
)
GO

CREATE TABLE dbo.Section
(
      Section_Number          int  IDENTITY      NOT NULL,
      Course_Number           varchar(6)         NOT NULL,
      Room_Number             varchar(10)        NULL,
      Class_Day               varchar(9)         NULL,
      Class_Begin_Time        varchar(5)         NULL,
      Class_End_Time          varchar(5)         NULL,
      Instructor_ID           int                NULL,

      CONSTRAINT PK_Section
            PRIMARY KEY (Section_Number)
)
GO

/*Following bridge table enrolls students in sections of courses*/
CREATE TABLE dbo.Enroll
(
      Student_ID        int          NOT NULL,
      Section_Number    int          NOT NULL,

  CONSTRAINT PK_Enroll
    PRIMARY KEY (Student_ID, Section_Number), --<<< COMMENT 7 >>>--

  CONSTRAINT FK_Student_ID_Enroll
    FOREIGN KEY (Student_ID)
      REFERENCES Student (Student_ID),

  CONSTRAINT FK_Section_Number_Enroll
    FOREIGN KEY (Section_Number)
      REFERENCES Section (Section_Number)
)
GO
```

Comment 1. Comments do not execute but are ignored by the compiler. Comments begin with two hyphens (--) and can appear on separate lines or after the code on a line. Comments also can begin with `/*` and end with `*/`

Comment 2. If this is the only command in the Query window, then click the Execute button on the taskbar or press function key <F5>. If you have other statements in the Query window and you only want this one to execute, select only this statement and execute it. Also, the key words (reserved words) are usually capitalized in these notes but the statements will execute OK with any case.

Comment 3. We've batched the three statements with a GO after each statement, so there's three batches. If you tried to execute these three DDL statements and either the first or second one failed, the third would fail because referential integrity would be violated.

A better way to write SQL statements that drop tables is to first test whether the table exists and you have the permission to drop it. If both are satisfied, then execute a DROP TABLE statement.

Comment 4. Note in the line above that the left parenthesis (here is matched by a right parenthesis) at the end of the statement. Each column specification begins with a column name and is followed by a data type. To insure that a setting does not change whether you want to allow nulls in a column, the style in SQL Server is to explicitly code NULL or NOT NULL in each column.

IDENTITY will autogenerate whole numbers for this column, beginning with the seed 1 (initial value 1) and incrementing by 1. The numbers generated then are 1, 2, 3, 4, …. You can explicitly specify the seed value and the incremental value (which can be negative if you want). A seed of 1000 and an increment of -1 will begin with 1000 and continue 999, 998, … If the IDENTITY is added later after the table already has raw data, you probably will adjust the seed so no values in that column are repeated. The more general form of IDENTITY is

```
IDENTITY (<seed>, <increment>)  NOT FOR REPLICATION
```

Note that a comma separates the specification of one column from the specification for the next column.

Comment 5. The default value for the City column here is the string `Santa Monica`. The single quote marks denote that this is a string of characters.

If you were specifying the default value for some column (not shown here) with numerical data, you would not surround the number with single quote marks. Depending on the details of which numerical data type you're using, you might have

```
DEFAULT 12
DEFAULT 12.3
DEFAULT 0.1234E+02
```

For a column with data type datetime or smalldatetime, if you'd like to store the current date and time as the default, you could use either the built-in function that returns the current system date and time:

```
DEFAULT GETDATE()
```

or you could enter a literal value for the current date and time as a string such as

```
DEFAULT 'SEPTEMBER 23, 2010'  (time will be set to midnight since time is omitted)
DEFAULT 'SEPTEMBER 23, 2010 06:45PM'
```

```
DEFAULT 'SEPTEMBER 23, 2010 06:45:30.007PM'
```

or any of the other recognized formats for date and time described in the *Date and time data types* section of Chapter 04. While SQL Server can recognize many formats for dates, some that may seem natural to you as 'Sept. 23, 2010' wouldn't be recognized because the abbreviation Sept. isn't recognized.

Comment 6. Primary keys can be specified at the time you create a table or later by using the ALTER TABLE statement. The pattern of this constraint is

CONSTRAINT *ConstraintName* PRIMARY KEY (*ColumnName1*, *ColumnName2*, …)

One or more columns that make up the primary key.

Primary keys are constraints (the value(s) must be unique and not null), and have a name that must be unique in the database. The name for a primary key normally follows the pattern PK_*Tablename.*

Comment 7. This is a composite primary key.

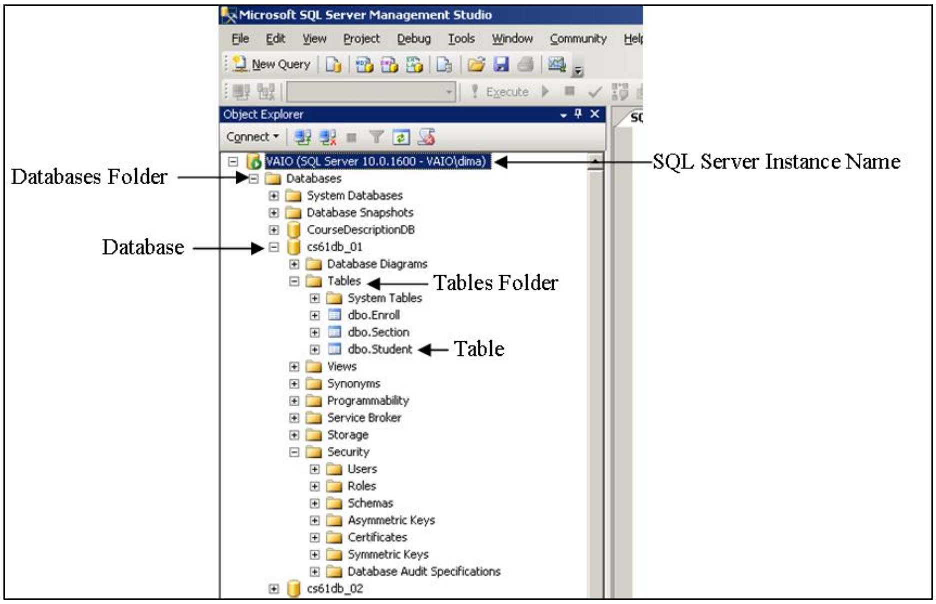
Manipulating Tables in Object Explorer

Object Explorer (OE)

Object Explorer is a component of SQL Server Management Studio (SSMS) that connects to Database Engine instances. It is a graphical user interface with which you can view and manage all the objects in the server.

Object Explorer is more than just a way to explore the database objects on a server; it is also a tool that can be used to create basic template scripts for selecting, inserting, updating, and deleting data. Object Explorer's functionality is exposed through the context menu (drop-down menu). Right-clicking on any object or folder within Object Explorer exposes the list of available options. If you cannot see Object Explorer, on the **View** menu, click **Object Explorer** or press **F7**.

Below is a sample screenshot of Object Explorer.



Object Explorer Details (OED)

The Object Explorer Details pane is a great deal like the list or detail view in Windows Explorer.

Nullability

Nullability defines whether a column can contain a NULL value. A NULL value means that the value is unknown. It does not mean that the value is zero, blank, or empty. If you don't explicitly specify that a column is NULL or NOT NULL (whether nulls are allowed) in the CREATE TABLE statement, then that value will be controlled by a default setting that may have been changed. Therefore it is a good practice to state explicitly whether a column is NULL or NOT NULL.

Let's create a sample table and insert some rows where one of the columns allows null values.

```
CREATE TABLE dbo.Person
(
      Name  nvarchar(30)       NOT NULL,
      Email nvarchar(80)       NULL
)

INSERT INTO dbo.Person (Name, Email)
VALUES
       ('Natasha', NULL),
       ('Abigail', 'abc@yahoo.com'),
       ('Sergei', 'ser@msn.com')
```

Null and COUNT Function

When using the COUNT function on a column containing NULL values, the NULL values will be omitted from the count. However, if the COUNT function is used with an asterisk, it will count all rows regardless of whether NULL values are present. This difference is demonstrated in the following code:

```
SELECT       COUNT(*)           FROM dbo.Person
SELECT       COUNT(Email)       FROM dbo.Person
```

```
-----------
3

(1 row(s) affected)


-----------
2
Warning: Null value is eliminated by an aggregate or other SET operation.

(1 row(s) affected)
```

IDENTITY Property

The IDENTITY column property allows you to define an automatically incrementing numeric value for a single column in a table. An IDENTITY column is most often used for surrogate primary key columns, as they are more compact than non-numeric data type natural keys. When a new row is inserted into a table with an IDENTITY column property, the column is inserted with a unique incremented value. The data type for an IDENTITY column can be int, tinyint, smallint, bigint, decimal, or numeric. Tables may only have one identity column defined, and the defined IDENTITY column can't have a DEFAULT or rule settings associated with it. The initial seed and step can be any integer value both positive and negative, although a column storing tinyints cannot store negatives.

An identity column is not guaranteed to be unique nor consecutive. You should always place a unique index on an identity column if your system requires uniqueness.

Below is the syntax for specifying an identity column.

```
[ IDENTITY [ ( seed ,increment ) ] ]
```

The IDENTITY property takes two values: seed and increment. seed defines the starting number for the IDENTITY column, and increment defines the value added to the previous IDENTITY column value to get the value for the next row added to the table. The default for both seed and increment is 1.

Using the IDENTITY Property During Table Creation

Below is T-SQL code to create a table with an identity column.

```
CREATE TABLE dbo.Student
(
      Student_ID        int               NOT NULL     IDENTITY(1,1),
      Student_Name      nvarchar(30)      NOT NULL
)
GO
```

Inserting Rows into Identity Columns

Let's insert some rows into the Student table.

```
INSERT INTO dbo.Student (Student_Name)
  VALUES ('Alexander'), ('Julia'), ('Mike'), ('Julia')
```

```
SELECT * FROM dbo.Student
```

```
Student_ID  Student_Name
----------- ------------------------------
1           Alexander
2           Julia
3           Mike
4           Julia

(4 row(s) affected)
```

Note that we can insert duplicate rows. In our example there are duplicate names with different identities. The identity shows the order in which the rows were inserted.

We can find the current identity seed by using DBCC checkident.

```
DBCC checkident ('dbo.Student')
```

```
Checking identity information: current identity value '4', current column value '4'.
DBCC execution completed. If DBCC printed error messages, contact your system administrator.
```

The identity value will be incremented during a transaction and will remain incremented even if that transaction is rolled back. The example below will demonstrate it.

```
BEGIN TRANSACTION

INSERT INTO dbo.Student (Student_Name)
   VALUES ('Olga')

ROLLBACK TRANSACTION

SELECT * FROM dbo.Student
DBCC checkident ('dbo.Student')
```

```
 (1 row(s) affected)
Student_ID  Student_Name
----------- ------------------------------
1           Alexander
2           Julia
3           Mike
4           Julia

(4 row(s) affected)

Checking identity information: current identity value '5', current column value '5'.
DBCC execution completed. If DBCC printed error messages, contact your system administrator.
```

The table has not changed but we can see from the checkident that the current seed has been changed. Now if we add another student, there will be a gap in a row of our identities.

```
INSERT INTO dbo.Student (Student_Name)
  VALUES ('Olga')

SELECT * FROM dbo.Student

DBCC checkident ('dbo.Student')
```

```
(1 row(s) affected)
Student_ID  Student_Name
----------- ------------------------------
1           Alexander
2           Julia
3           Mike
4           Julia
6           Olga

(5 row(s) affected)

Checking identity information: current identity value '6', current column value '6'.
DBCC execution completed. If DBCC printed error messages, contact your system administrator.
```

Inserting Specific Identity Values

We can insert a specific identity value to override the generated value. The general steps to do it are:

```
SET IDENTITY_INSERT table_name ON
--Insert rows(s) here in which value(s)for the identity columns are specified
SET IDENTITY_INSERT table_name OFF
```

For example, if you want to add a student John with student_ID = 55 to our Student table, you do it as follows:

```
SET IDENTITY_INSERT dbo.Student ON

INSERT INTO dbo.Student (Student_ID, Student_Name)
  VALUES (55, 'John')

SET IDENTITY_INSERT dbo.Student OFF

SELECT * FROM dbo.Student
```

```
Student_ID  Student_Name
----------- ------------------------------
1           Alexander
2           Julia
3           Mike
4           Julia
6           Olga
55          John

(6 row(s) affected)
```

If we insert another row into this table we will see that the seed is changed to the last inserted value.

```
INSERT INTO dbo.Student (Student_Name)
  VALUES ('Mika')

SELECT * FROM dbo.Student
```

```
(1 row(s) affected)
```

```
Student_ID  Student_Name
----------- ------------------------------
1           Alexander
2           Julia
3           Mike
4           Julia
6           Olga
55          John
56          Mika

(7 row(s) affected)
```

The identity doesn't guarantee uniqueness. We can insert a duplicate identity value as shown in the following example.

```
SET IDENTITY_INSERT dbo.Student ON

INSERT INTO dbo.Student (Student_ID, Student_Name)
VALUES (2, 'Naoko')

SET IDENTITY_INSERT dbo.Student OFF

SELECT * FROM dbo.Student
```

```
(1 row(s) affected)
Student_ID  Student_Name
----------- ------------------------------
1           Alexander
2           Julia
3           Mike
4           Julia
6           Olga
55          John
56          Mika
2           Naoko

(8 row(s) affected)
```

As you can see, we have two students with the same value in the identity column. The seed value is not changed after this insert because it changes only when manually inserted seed value is greater then the current one.

```
DBCC checkident ('dbo.Student')
```

```
Checking identity information: current identity value '56', current column value '56'.
DBCC execution completed. If DBCC printed error messages, contact your system administrator.
```

Adding an Identity Column to a Table

An identity column can be added to a table via an alter table statement if that table does not have an identity column yet. Only one identity column is allowed per table. If we try to add another identity column to a table that already has one, we will get the following error message:

```
ALTER TABLE dbo.Student
  ADD LibraryCard_ID    int    NOT NULL    IDENTITY(1,1)
GO
```

```
Msg 2744, Level 16, State 2, Line 1
```

```
Multiple identity columns specified for table 'Student'. Only one identity column per table is
allowed.
```

In the following example we will create a table and then add an identity column to it.

```
CREATE TABLE dbo.Teacher
(
      Teacher_LastName  nvarchar(30)      NOT NULL
)
GO
```

```
ALTER TABLE dbo.Teacher
  ADD Teacher_ID  int   NOT NULL    IDENTITY(1,1)
```

Global Unique Identifier

The ROWGUIDCOL property ensures a very high level of uniqueness for every database networked in the world. This unique ID is stored in a **uniqueidentifier** data type and is generated by the NEWID system function. The ROWGUIDCOL is a marker designated in a column definition, allowing you to query a table not only by the column's name, but also by the ROWGUIDCOL designator.

Although using a uniqueidentifier data type with a NEWID value for a primary key may be more unique, it takes up more space than an integer-based IDENTITY column. If you only care about unique values within the table, you may be better off using an integer surrogate key, particularly for very large tables. However, if uniqueness is an absolute requirement, with the expectation that you may be merging data sources in the future, uniqueidentifier with NEWID may be your best choice.

The only operations that can be performed against a **uniqueidentifier** value are comparisons (=, <>, <, >, <=, >=) and checking for NULL (IS NULL and IS NOT NULL). No other arithmetic operators can be used.

Constraints

Constraints are used by SQL Server to enforce column data integrity. Constraints define rules regarding the values allowed in columns. Both primary and foreign keys are forms of constraints. Other forms of constraints used for a column include UNIQUE, CHECK, and DEFAULT constraints.

Constraints can be column constraints or table constraints. A column constraint is specified as part of a column definition and applies only to that column. A table constraint is declared independently from a column definition and can apply to more than one column in a table. Table constraints must be used when more than one column must be included in a constraint.

The query optimizer also uses constraint definitions to build query execution plans. Constraints that include data type conversion may cause certain operations to fail. You should avoid using data type conversion in constraint definitions.

Primary Key

A primary key is a special type of constraint that identifies a single column or set of columns, which in turn uniquely identifies all rows in the table. A primary key enforces entity integrity, meaning that rows are guaranteed to be unambiguous and unique. Best practices recommend that every table should have a primary key. A primary key provides a way to access the record and ensures that the key is unique. A primary key column can't contain NULL values.

Only one primary key is allowed for a table, and when a primary key is designated, an underlying table index is automatically created, defaulting to a clustered index. You can also explicitly designate a non-clustered index be created when the primary key is created instead, if you have a better use for the single clustered index allowed for a table.

A composite primary key is the unique combination of more than one column in the table. In order to define a composite primary key, you must use a table constraint instead of a column constraint. Setting a single column as the primary key within the column definition is called a column constraint. Defining the primary key (single or composite) outside of the column definition is referred to as a table constraint.

The column or columns that make up the primary key must uniquely identify a single row in the table (no two rows can have the same values for all the specified columns). The ASC (ascending) and DESC (descending) options define the sorting order of the columns within the clustered or non-clustered index.

Defining a Column Constraint

Use the following syntax to define a primary key as a column constraint:

```
( column_name <data_type> [ NULL | NOT NULL ] PRIMARY KEY )
```

Defining a Table Constraint

The following syntax defines a primary key as a table constraint:

```
CONSTRAINT constraint_name PRIMARY KEY
( column [ ASC | DESC ] [ ,...n ] )
```

Creating a Table with a Primary Key

Below is an example of creating a table with a primary key that treated as a table constraint:

```
CREATE TABLE schema_name.table_name
(
	Column1 int 		NOT NULL,
	Column2 char(10) 	NULL,
	Column3 datetime 	NULL,
	CONSTRAINT PK_table_name PRIMARY KEY (Column1)
)
GO
```

Adding a Primary Key Constraint to an Existing Table

Below is an example of creating a primary key constraint for an existing table. As with the example above, it is a table constraint.

```
ALTER TABLE schema_name.table_name
   ADD CONSTRAINT PK_table_name PRIMARY KEY (Column1)
GO
```

Foreign Key

Foreign key constraints establish and enforce relationships between tables and help maintain referential integrity, which means that every value in the foreign key column must exist in the corresponding column for the referenced table, or be null. Foreign key constraints also help define domain integrity, in that they define the range of potential and allowed values for a specific column or columns. Domain integrity defines the validity of values in a column.

A table can have multiple foreign keys—and each foreign key can be based on a single or multiple (composite) key that references more than one column (referencing composite primary keys or unique indexes). Also, although the column names needn't be the same between a foreign key reference and a primary key, the primary key/unique columns must have the same data type. You also can't define foreign key constraints that reference tables across databases or servers.

Foreign keys restrict the values that can be placed within the foreign key column or columns. If the associated primary key or unique value does not exist in the reference table, the INSERT or UPDATE to the table row fails. This restriction is bidirectional in that if an attempt is made to delete a primary key, but a row referencing that specific key exists in the foreign key table, an error will be returned. All referencing foreign key rows must be deleted prior to deleting the targeted primary key or unique value; otherwise, an error will be raised.

Defining a Foreign Key Constraint

Below is the general syntax for defining a foreign key constraint.

```
CONSTRAINT constraint_name
  FOREIGN KEY (column_name)
  REFERENCES [ schema_name.] referenced_table_name [ ( ref_column ) ]
```

Creating a Table with a Foreign Key Reference

Adding a Foreign Key to an Existing Table

```
ALTER TABLE table_name
  ADD CONSTRAINT constraint_name
  FOREIGN KEY (column_name)
  REFERENCES [ schema_name.] referenced_table_name [ ( ref_column ) ]
```

Creating Recursive Foreign Key References

A foreign key in a table can be defined to reference its own primary/unique key. This technique is often used to represent recursive relationships.

Creating a New Foreign Key

In SQL Server, a foreign key must reference a primary key or column(s) with a unique constraint (One column or more than one column in a table can have a unique constraint. A table can have more than one unique constraint but only one primary key. If one column has a unique constraint, then at most that column has one null value and all other values for that column are not null. If that column also had a not null constraint or if that column were part of a primary key, then that column could not have even one null value.). If you attempt to create a foreign key without either (a) a primary key or (b) one or more columns with a unique constraint, the foreign key cannot be created. The referenced table also must exist.

Student (has other columns)

Student_ID	Last_Name
01	Smith
02	Chen
03	Sanchez
04	Yi
Primary key	

Enroll (bridge table)

Student_ID	Section_Number
01	4174
01	4175
01	1583
02	4173
04	1584
04	1583
Foreign key	**Foreign key**

Section

Section_Number	Course_Number	Class_Day
4174	CS61	Wednesday
4173	CS61	Friday
4175	CS65	Thursday
1584	CS65	Saturday
4171	CS60	Tuesday
1583	CS60	Friday
Primary key		

Allowing Cascading Changes in Foreign Keys

The effect of selecting *Cascade Update Related Fields* is, when you update a value of a primary key, the changes will be carried into (cascade to) the foreign key that references this primary key. In the present case, if you updated the Student_ID in the Student table, the change would cascade to the corresponding rows in the Enroll table so the same link persists.

The effect of *Cascade Delete Related Records* is, when you delete a record in the table with the primary key, to delete all related records in the table with the foreign key. In the present case, if you deleted a record in the Student table, all rows in the Enroll table with that Student_ID would also be deleted. This powerful, dangerous selection could have a widespread effect on what data is stored.

If the table has raw data in it, selecting *Check existing data on creation* will insure that the values in the foreign key satisfy referential integrity before the key is created. If the values would violate referential integrity, then the new foreign key would not be created.

SQL Server provides an automatic mechanism for handling changes in the primary key/unique key column, called cascading changes. In previous recipes, cascading options weren't used. You can allow cascading changes for deletions or updates using ON DELETE and ON UPDATE. The basic syntax for cascading options is as follows:

```
[ ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
[ ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
[ NOT FOR REPLICATION ]
```

NO ACTION: The default setting for a new foreign key is NO ACTION, meaning if an attempt to delete a row on the primary key/unique column occurs when there is a referencing value in a foreign key table, the attempt will raise an error and prevent the statement from executing.

CASCADE: For ON DELETE, if CASCADE is chosen, foreign key rows referencing the deleted primary key are also deleted. For ON UPDATE, foreign key rows referencing the updated primary key are also updated.

SET NULL: If the primary key row is deleted, the foreign key referencing row(s) can also be set to NULL (assuming NULL values are allowed for that foreign key column).

SET DEFAULT: If the primary key row is deleted, the foreign key referencing row(s) can also be set to a DEFAULT value. The new cascade SET DEFAULT option assumes the column has a default value set for a column. If not, and the column is nullable, a NULL value is set.

UNIQUE Constraints

UNIQUE constraints enforce uniqueness within a table on non-primary key columns. You can only have one primary key defined on a table. If you wish to enforce uniqueness on other non-primary key columns, you can use a UNIQUE constraint. A unique constraint, by definition, creates an alternate key. Unlike a PRIMARY KEY constraint, you can create multiple UNIQUE constraints for a single table. You also can designate a UNIQUE constraint for columns that allow NULL values (although only one NULL value is allowed for a single-column key per table). Like primary keys, UNIQUE constraints enforce entity integrity by ensuring that rows can be uniquely identified. The columns specified in the UNIQUE constraint definition can't have duplicate values occurring in the table; otherwise, the operation will fail with an error that a duplicate key value was found.

The UNIQUE constraint creates an underlying table index when it is created. This index can be CLUSTERED or NONCLUSTERED, although you can't create the index as CLUSTERED if a clustered index already exists for the table.

As a column constraint, the syntax to define a UNIQUE constraint during a table's creation is

```
( column_name <data_type> [ NULL | NOT NULL ] UNIQUE )
```

As a table constraint, the syntax to define a UNIQUE constraint during a table's creation is

```
CONSTRAINT constraint_name UNIQUE
  (column [ ASC | DESC ] [ ,...n ] )
```

The ASC (ascending) and DESC (descending) options define the sorting order of the columns within the clustered or non-clustered index.

Let's create a Student table where the student ID column is unique.

```
CREATE TABLE dbo.Student
(
      Student_ID          int               NOT NULL     IDENTITY(1,1),
      Student_Name        nvarchar(30)      NOT NULL
```

```
      CONSTRAINT  UQ_Student_Student_ID   UNIQUE
      (Student_ID DESC)
)
GO
```

Since IDENTITY property is not guaranteed to be unique, we use a unique constraint to enforce uniqueness.

Adding a UNIQUE Constraint to an Existing Table

```
ALTER TABLE table_name
  ADD CONSTRAINT constraint_name
  UNIQUE (column [ ASC | DESC ] [ ,...n ] )
```

CHECK Constraints

The CHECK constraint is used to define what format and values are allowed for a column. Its syntax is

```
CHECK ( logical_expression )
```

If the logical expression of CHECK evaluates to TRUE, the row will be inserted. If the CHECK constraint expression evaluates to FALSE, the row insert will fail. A CHECK constraint can also be defined at the table constraint level—where you are allowed to reference multiple columns in the expression.

Adding a CHECK Constraint to an Existing Table

```
ALTER TABLE table_name
  WITH CHECK | WITH NOCHECK
  ADD CONSTRAINT constraint_name
  CHECK ( logical_expression )
```

Adding WITH NOCHECK means that existing values are ignored going forward, and only new values are validated against the CHECK constraint. Using WITH NOCHECK may cause problems later on, as you cannot depend on the data in the table conforming to the constraint.

Disabling and Enabling a Constraint

Constraints are used to maintain data integrity, although sometimes you may need to relax the rules while performing a one-off data import or non-standard business operation. NOCHECK can also be used to disable a CHECK or FOREIGN KEY constraint, allowing you to insert rows that disobey the constraints rules.

To disable or enable all CHECK and FOREIGN KEY constraints for the table, you should use the ALL Keyword.

```
-- disable checking on all constraints
ALTER TABLE Sales.PersonCreditCard
```

```
  NOCHECK CONSTRAINT ALL
```

```
-- enable checking on all constraints
ALTER TABLE Sales.PersonCreditCard
  CHECK CONSTRAINT ALL
```

DEFAULT Constraint

If you don't know the value of a column in a row when it is first inserted into a table, you can use a DEFAULT constraint to populate that column with an anticipated or non-NULL value

Adding a DEFAULT Constraint to an Existing Table

```
ALTER TABLE table_name
  ADD CONSTRAINT constraint_name
  DEFAULT default_value
  FOR column_name
```

Dropping a Constraint from a Table

```
ALTER TABLE table_name
  DROP CONSTRAINT constraint_name
```

Altering a Table

Altering a table is changing the metadata of a table: adding and dropping keys; adding and dropping columns; changing the data type of a column; adding, changing, or dropping default value for a column; etc. Some of these may also affect raw data, e.g., if you drop a column, you will remove both the metadata and raw data for that column if the table has any rows.

The ALTER TABLE statement has these forms, all which begin with ALTER TABLE `table-name`

(1) A form to **drop constraints**[2]:

```
ALTER TABLE table-name
  DROP CONSTRAINT constraint-name
```

(2) A form to **drop columns**:

```
ALTER TABLE table-name
  DROP COLUMN column-name
```

(3) A form to **add new columns** to an existing table:

---

[2] Optional note: Oracle accepts `ALTER TABLE TableName DROP PRIMARY KEY`

```
ALTER TABLE table-name
  ADD column-name  datatype
    DEFAULT default-value
    IDENTITY (seed, increment)      Optional
    column-constraints
```

(4) A form to **add new table constraints (**primary key, foreign key, unique, check constraint) to an existing table:

```
ALTER TABLE table-name
  ADD table-constraint
```

Example:
CONSTRAINT PK_Course **PRIMARY KEY** (Course_ID)

(5) A form to **alter existing column specifications**

- change data types (including more or fewer characters in char and varchar columns, different precision and scale in decimal data types). If the table has raw data in the column, the conversion to the new data type must be handled implicitly (automatically by SQL Server). An example of a conversion that could not be implicit is a column with data type varchar(5) that stores the name 'Jim' in which the new datatype is integer. SQL Server will not implicitly convert data that is not a number into a number.

```
ALTER TABLE table-name
    ALTER COLUMN  column_name  new-data-type
```

- change a column that allows nulls to a column that does not allow nulls (i.e., has a new NOT NULL constraint)

```
ALTER TABLE table-name
    ALTER COLUMN  column_name  NOT NULL
```

- change a column with a not null constraint to a column that allows nulls

```
ALTER TABLE table-name
    ALTER COLUMN  column_name  NULL
```

Without the raw data, we'll use these tables to demonstrate the ALTER TABLE statement:

Student (has other columns)

Student_ID	Last_Name
01	Smith
02	Chen
03	Sanchez
04	Yi
Primary key	

Enroll (bridge table)

Student_ID	Section_ Number
01	4174
01	4175
01	1583
02	4173
04	1584
04	1583
Foreign key	**Foreign key**

Section

Section_ Number	Course_ Number	Class_Day
4174	CS61	Wednesday
4173	CS61	Friday
4175	CS65	Thursday
1584	CS65	Saturday
4171	CS60	Tuesday
1583	CS60	Friday
Primary key		

Using ALTER TABLE to drop a foreign key

To drop the foreign key in the Enroll table that references the primary key in the Student table, in the Query pane enter

```
ALTER TABLE Enroll DROP CONSTRAINT FK_Enroll_Student
```

TableName (Enroll)

ConstraintName (FK_Enroll_Student)
(default name created in Enterprise Manager)

or on two lines if you like

```
ALTER TABLE Enroll
  DROP CONSTRAINT FK_Enroll_Student
```

Click the *Execute* button or press <F5> to execute the command. The Results pane displays

```
The command(s) completed successfully.
```

If you try to execute the statement again, it will fail since this constraint no longer exists. The Results pane would display

```
Server: Msg 3728, Level 16, State 1, Line 1
'FK_Enroll_Student' is not a constraint.
Server: Msg 3727, Level 16, State 1, Line 1
Could not drop constraint. See previous errors.
```

Using ALTER TABLE to drop a primary key

Now that no foreign key references the primary key in the Student table, referential integrity will not be violated if you drop that key. In the Query pane type and execute only the statement[3]

```
ALTER TABLE Student DROP CONSTRAINT PK_Student
```

[3] Optional note: Oracle accepts `ALTER TABLE` *`TableName`* `DROP PRIMARY KEY`

Using ALTER TABLE to add a primary key

TableName

```
ALTER TABLE Student ADD
  CONSTRAINT PK_Student PRIMARY KEY (Student_ID)
```

ConstraintName
follows pattern PK_TableName

If more than one column,
separate the ColumnNames with commas

Using ALTER TABLE to add a foreign key

TableName

ConstraintName for this foreign key follows the pattern
FK_ColumnName_TableName
Enterprise Manager uses pattern *FK_Table1_Table2*

```
ALTER TABLE Enroll ADD
  CONSTRAINT  FK_Student_ID_Enroll
  FOREIGN KEY (Student_ID) REFERENCES Student (Student_ID)
```

ColumnName(s) for
the foreign key

TableName that
the foreign key
references

ColumnName(s)
of the primary key
that the foreign
key references

A foreign key always has the same number of columns as the primary key that it references. If the key is composite, then separate the column names with commas and list the columns in the order that the keys will connect.

Whether this statement is on one or more lines doesn't affect execution.

The 2nd and 3rd lines above

```
  CONSTRAINT  FK_Student_ID_Enroll
  FOREIGN KEY (Student_ID) REFERENCES Student (Student_ID)
```

has the same syntax to add a foreign key whether it is part of the ALTER TABLE ADD statement or the CREATE TABLE statement.

Adding a Column to an Existing Table

The syntax for adding a column to an existing table is as follows:

```
ALTER TABLE table_name
  ADD { column_name data_type } NULL
```

This method adds the column to the last column position in the table definition. When adding columns to a table that already has data in it, you will be required to add the column with NULL values allowed. You can't specify that the column be NOT NULL, because you cannot add the column to the table and simultaneously assign values to the new column. By default, the value of the new column will be NULL for every row in the table.

Now we'll add a column named Photograph to the Student table that will store in a column with data type **image** (a variable-length binary data type up to about 2GB long) a picture of the student:

```
ALTER TABLE Student ADD
  Photograph  image  NULL
```

Column Specification

A column specification can include many other features, some which you've seen before in the CREATE TABLE statement:

<*ColumnName*> <*Data type*> DEFAULT <*DefaultValue*> <NULL | NOT NULL>
IDENTITY (<*seed*>, <*increment*>) <*ColumnConstraint*>

Using ALTER TABLE to drop a column

On one or two lines, enter in the Query pane and execute

```
ALTER TABLE Student DROP COLUMN
  Photograph
```

ColumnName to be dropped

Using ALTER TABLE to alter the width or data type of a column

```
ALTER TABLE Student ALTER COLUMN
  Last_Name  varchar(30)  NULL
```

ColumnName to be altered (but you can't change the name itself this way)

New data type (it was varchar(20))

This use to be NOT NULL

Only include in the column specification the *ColumnName* and the features you're altering. Other features for that column that you've previously specified will remain unaltered.

SQL Server allows you to reduce the width of a column that stores raw data if the new data type does not cause the existing strings to be truncated. Reducing the number of trailing blanks in a char(n) column is not considered to be truncating data. But reducing the width of a column that stores '`Jim`' to char(2) will fail because the '`m`' would be truncated.

So the table remains with the properties initially specified, I'll alter the table back to the original values

```
ALTER TABLE Student ALTER COLUMN
  Last_Name  varchar(20)  NOT NULL
```

Constraining a column's value during table creation or table alteration with the CHECK constraint

The check constraint could appear in any of these statements

CREATE TABLE,
ALTER TABLE … ADD *ColumnName* (if the check constraint refers only to the new column),
ALTER TABLE … ADD CONSTRAINT (which will work if the check constraint refers one or several existing columns in the table being altered)

The syntax for the check constraint is

```
[CONSTRAINT ConstraintName] CHECK (Condition)
```

Boolean condition value is True or False.
If True, the column value being inserted or updated is accepted.
If False, an error message is returned.

Optional name for the check constraint. The name could follow the pattern CK_TableName_ColumnName. If you don't name the constraint, SQL Server will give it a cryptic name.

Suppose you had an Age column and you wished to constrain a value inserted or updated in that column to be 21 years old or older. The check constraint could be

```
CONSTRAINT  CK_Student_Age  CHECK(Age >= 21)
```

`Age >= 21` is a Boolean expression. It compares the value on the left (the value being inserted or updated in the Age column) with the value on the right (21) and uses the relational operator >=. If the age is greater than or equal to 21, the expression is True and that value for Age can be inserted or updated. If False, SQL Server rejects that insertion or update and returns an error message.

SQL Server recognizes the **relational operators** below

=	equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
<>	not equal to
!<	not less than
!>	not greater than

and recognizes the **logical operators** or **connectives** below used to form compound expressions

AND
OR
NOT

SQL Server also recognizes other operators such as

```
Age IN (21, 22, 24, 70)
```

```
City IN ('Santa Monica', 'West Los Angeles', 'Marina Del Rey')
```

```
Age BETWEEN 21 AND 75
```

which includes the lower limit (21) and upper limit (75) in the range. (The ages 21, 22, 23, …, 75 could be inserted. Ages 20, 76, and others outside the range 21–75 would be rejected if this expression were part of a check constraint.

Social_Security_Number **LIKE** '[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]'

which checks to insure that the value is composed of a string of digits and hyphens such as **'123-45-6789'**

Other uses of the LIKE operator are described along with the SELECT statement.

As another example of a check constraint clause, suppose in the Student table a column has been added named Class_Status with a varchar(9) data type. You want to constrain (restrict, limit) the values to be stored in that column to FRESHMAN, SOPHOMORE, JUNIOR, SENIOR, or GRADUATE.

The check clause would be

```
CONSTRAINT CK_Student_Class_Status
  CHECK (Class_Status IN
   ('FRESHMAN', 'SOPHOMORE', 'JUNIOR', 'SENIOR', 'GRADUATE'))
```

These are each strings of characters so they each begin and end with single-quote marks.

Changing an Existing Column Definition

The syntax for altering an existing column of a table is as follows:

```
ALTER TABLE table_name
  ALTER COLUMN column_name
  [type_name] [NULL | NOT NULL] [COLLATE collation_name]
```

There are limitations to the kind of column changes that can be made. For example, you can't alter a column that is used in an index unless the column data type is varchar, nvarchar, or varbinary – and even then, the new size of that data type must be larger than the original size. You also can't use ALTER COLUMN on columns referenced in a primary key or foreign key constraint. There are other column modification limitations documented in SQL Server Books Online.

Computed Column

The syntax for defining a computed column is as simple as follows:

```
column_name AS computed_column_expression
[ PERSISTED ]
```

A computed column is based on an expression defined when you create or alter the table, and is not physically stored in the table unless you use the PERSISTED keyword. Computed columns are sometimes useful when a calculation must be recomputed on the same data repeatedly in referencing queries.

The computed_column_expression is the calculation you wish to be performed in order to derive the column's value. Adding the PERSISTED keyword actually causes the results of the calculation to be physically stored. Any changes made to columns that are used in the computation will cause the stored value to be updated again. The stored data still can't be modified directly – the data is still computed.

Computed columns can't be used within a DEFAULT or FOREIGN KEY constraint. A calculated column can't be explicitly updated or inserted into (since its value is always derived).

Computed columns can be used within indexes, but must meet certain requirements, such as being deterministic (always returning the same result for a given set of inputs) and precise (not containing float values).

In the following table, a **computed column** is specified as the product of the values in two other columns times 1.08 (to include 8% sales tax):

```
CREATE TABLE  dbo.Line
(
      Invoice_Number    char(2),
      Line_Number       char(2),
      Item_Number       char(2),
      Quantity          integer,
      Line_Price        smallmoney,
      Total_Price       AS (Quantity*Line_Price*1.08) -- Computed column
)
```

This computed column does not store a column of values in the table, nor can you directly insert values into that column. Instead, the formula for that column is stored and is evaluated when the table is queried. Example:

```
SELECT * FROM Line
```

```
(1 row(s) affected)
Invoice_Number Line_Number Item_Number Quantity    Line_Price            Total_Price
-------------- ----------- ----------- ----------- --------------------- -------------
01             01          02          100         123.50                13338.000000

(1 row(s) affected)
```

```
SELECT Total_Price FROM Line

Total_Price
----------------
13338.000000
```

Dropping a Table Column

The syntax for dropping a column of a table is:

```
ALTER TABLE table_name
  DROP COLUMN column_name
```

Usually the word **drop** refers to removing a table or other object (both metadata and any raw data), and **delete** refers to removing a row or rows of raw data from a table, but these words are intermixed in the GUIs.

Referential integrity may block you from dropping a table. You can drop a column only if it isn't being used in a PRIMARY KEY, FOREIGN KEY, UNIQUE, or CHECK CONSTRAINT. You also can't drop a column being used in an index or that has a DEFAULT value bound to it.

Dropping a Table

Below is the statement for dropping a table:

```
DROP TABLE schema.tablename
```

The DROP command removes the table definition and its data permanently from the database. There is now undo command in SQL Server. If there are foreign key references, you must drop them first before dropping the primary key table.

Keys

Constraints place limitations on the data that can be entered into a column or columns.

Altering Keys

Altering a table is changing the metadata of a table: adding and dropping keys; adding and dropping columns; changing the data type of a column; adding, changing, or dropping default value for a column; etc. Some of these may also affect raw data, e.g., if you drop a column, you will remove both the metadata and raw data for that column if the table has any rows.

Reviewing present keys

The Customer table has a primary key referenced by a foreign key and itself has a foreign key that references another table, Representative.

Invoice table

INVOICE_ NUMBER	CUSTOMER_ NUMBER	INVOICE_ DATE	ITEM_ NUMBER	QUANTITY
01	20	May 12, 1999	70	11
02	30	February 29, 2000	60	15
03	30	September 13, 2004	20	14
04	20	July 10, 2012	10	NULL
05	60	February 17, 2015	60	20
Primary key	Foreign key, NOT NULL		Foreign key	
char(2)	char(2)	datetime	char(2)	Integer

Customer table

CUSTOMER_ NUMBER	CUSTOMER_NAME	CITY	REPRESENTATIVE_ID
10	Ballard Computer	Seattle	55
20	Computer City	Miami	33
30	Under_Score, Inc.	Atlanta	22
40	Varner User System	Naperville	NULL
50	100% Jargon	Spokane	55
60	Computing Solutions	Tucson	11
Primary key	NOT NULL		Foreign key
char(2)	varchar(20),	varchar(20)	char(2)

Inventory table

ITEM_NUMBER	DESCRIPTION	QUANTITY_ON_HAND

Representative table

REPRESENTATIVE_ID	LAST_NAME	FIRST_NAME	REGION	HIRE_DATE	PHONE

How can you review these keys in the Customer table?

Temporary Tables and Table Variables

Table Variables

A table variable is a data type that can be used within a Transact-SQL batch, stored procedure, or function—and is created and defined similarly to a table, only with a strictly defined lifetime scope. The lifetime of the table variable only lasts for the duration of the batch, function, or stored procedure. Unlike regular tables or temporary tables, table variables can't have indexes or FOREIGN KEY constraints added to them. Table variables do allow some constraints to be used in the table definition (PRIMARY KEY, UNIQUE, and CHECK).

SQL Server 2008 introduces table-valued parameters and user-defined types, which you can use to pass temporary result sets between modules.

The syntax to creating a table variable is similar to creating a table, only the DECLARE keyword is used and the table name is prefixed with an @ symbol:

```
DECLARE @TableName TABLE
(column_name <data_type> [ NULL | NOT NULL ] [ ,...n ] )
```

Table variables will be described again when we learn about creating table-valued functions. For now, you can think of it as a RAM-based table. Like all variables, a table variable is stored in RAM, and this particular form (unlike simple variables and record variables) takes the form of a table.

Key Terms

owner (creator) of a table or another object
SQL command categories
- Data Definition Language (DDL) statements affect metadata
  - CREATE TABLE
  - ALTER TABLE
  - DROP TABLE
  - RENAME
- Data Manipulation Language (DML) statements affect raw data
- Data Control Language (DCL) statements grant and revoke permissions

temporary table
- private
- global

useful system stored procedures introduced in this chapter
- sp_help
- sp_rename

```
ALTER TABLE Tablename ADD CONSTRAINT Constraintname PRIMARY KEY (Columnname(s))
ALTER TABLE Tablename ADD CONSTRAINT Constraintname FOREIGN KEY (Columnname(s))
      REFERENCES Tablename (Referenced Columnname(s))
ALTER TABLE Tablename ADD CONSTRAINT Constraintname CHECK(Condition)
ALTER TABLE Tablename DROP CONSTRAINT Constraintname
ALTER TABLE Tablename ADD COLUMN Columnname Datatype DEFAULT DefaultValue
      {NULL  | NOT NULL} IDENTITY (seed, increment) ColumnConstraint
ALTER TABLE Tablename DROP COLUMN Columnname
ALTER TABLE Tablename ALTER COLUMN Columnname [Datatype ] {NULL | NOT NULL}
```

Index for Chapter 05