CREATE TABLE (Transact-SQL)

Creates a new table.

Transact-SQL Syntax Conventions

Syntax

```
CREATE TABLE
    [ database_name . [ schema_name ] . | schema_name . ] table_name
    ( { <column_definition> | <computed_column_definition>
        | <column_set_definition> }
        [ <table_constraint> ] [ ,...n ] )
    [ ON { partition_scheme_name (partition_column_name) | filegroup
        | "default" } ]
    [ { TEXTIMAGE_ON { filegroup | "default" } ]
    [ FILESTREAM_ON { partition_scheme_name | filegroup
        | "default" } ]
    [ WITH ( <table_option> [ ,...n ] ) ]
[ ; ]

<column_definition> ::=column_name <data_type>
    [ FILESTREAM ]
    [ COLLATE collation_name ]
```

```
    [ SPARSE ]
```

```
    [ NULL | NOT NULL ]
    [
        [ CONSTRAINT constraint_name ] DEFAULT constant_expression ]
      | [ IDENTITY [ ( seed ,increment ) ] [ NOT FOR REPLICATION ]
    ]
    [ ROWGUIDCOL ] [ <column_constraint> [ ...n ] ]
```

```
<data type> ::=
[ type_schema_name . ] type_name
    [ (precision [ ,scale ] | max |
        [ { CONTENT | DOCUMENT } ] xml_schema_collection) ]

<column_constraint> ::=
```

```
[ CONSTRAINT constraint_name ]
{     { PRIMARY KEY | UNIQUE }
        [ CLUSTERED | NONCLUSTERED ]
        [
            WITH FILLFACTOR = fillfactor
          | WITH ( < index_option > [ , ...n ] )
        ]
        [ ON { partition_scheme_name (partition_column_name)
            | filegroup | "default" } ]
  | [ FOREIGN KEY ]
        REFERENCES [ schema_name . ] referenced_table_name [ (ref_column ) ]
        [ ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
        [ ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
        [ NOT FOR REPLICATION ]
  | CHECK [ NOT FOR REPLICATION ] (logical_expression )
}
```

```
<computed_column_definition> ::=column_name AS computed_column_expression
[ PERSISTED [ NOT NULL ] ]
[
    [ CONSTRAINT constraint_name ]
    { PRIMARY KEY | UNIQUE }
        [ CLUSTERED | NONCLUSTERED ]
        [
            WITH FILLFACTOR = fillfactor
          | WITH ( <index_option> [ , ...n ] )
        ]
    | [ FOREIGN KEY ]
        REFERENCES referenced_table_name [ (ref_column ) ]
        [ ON DELETE { NO ACTION | CASCADE } ]
        [ ON UPDATE { NO ACTION } ]
        [ NOT FOR REPLICATION ]
    | CHECK [ NOT FOR REPLICATION ] (logical_expression )
    [ ON { partition_scheme_name (partition_column_name)
        | filegroup | "default" } ]
]

<column_set_definition> ::=column_set_name XML COLUMN_SET FOR ALL_SPARSE_COLUMNS

< table_constraint > ::=
[ CONSTRAINT constraint_name ]
{
    { PRIMARY KEY | UNIQUE }
        [ CLUSTERED | NONCLUSTERED ]
```

```

        (column [ ASC | DESC ] [ ,...n ] )
        [
            WITH FILLFACTOR = fillfactor
           |WITH ( <index_option> [ , ...n ] )
        ]
        [ ON { partition_scheme_name (partition_column_name)
            | filegroup | "default" } ]
    | FOREIGN KEY
        ( column [ ,...n ] )
        REFERENCES referenced_table_name [ (ref_column [ ,...n ] ) ]
        [ ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
        [ ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
        [ NOT FOR REPLICATION ]
    | CHECK [ NOT FOR REPLICATION ] (logical_expression )
}

<table_option> ::=
{
    DATA_COMPRESSION = { NONE | ROW | PAGE }
      [ ON PARTITIONS ( { <partition_number_expression> | <range> }
      [ , ...n ] ) ]
}

<index_option> ::=
{
    PAD_INDEX = { ON | OFF }
  | FILLFACTOR =fillfactor
  | IGNORE_DUP_KEY = { ON | OFF }
  | STATISTICS_NORECOMPUTE = { ON | OFF }
  | ALLOW_ROW_LOCKS = { ON | OFF}
  | ALLOW_PAGE_LOCKS ={ ON | OFF}
  | DATA_COMPRESSION = { NONE | ROW | PAGE }
       [ ON PARTITIONS ( { <partition_number_expression> | <range> }
       [ , ...n ] ) ]
}
<range> ::=
<partition_number_expression> TO <partition_number_expression>
```

Arguments

database_name
: Is the name of the database in which the table is created. *database_name* must specify the name of an existing database. If not specified, *database_name* defaults to the current database. The login for the current connection must

be associated with an existing user ID in the database specified by *database_name*, and that user ID must have CREATE TABLE permissions.

schema_name
Is the name of the schema to which the new table belongs.

table_name
Is the name of the new table. Table names must follow the rules for identifiers. *table_name* can be a maximum of 128 characters, except for local temporary table names (names prefixed with a single number sign (#)) that cannot exceed 116 characters.

column_name
Is the name of a column in the table. Column names must follow the rules for identifiers and must be unique in the table. *column_name* can be up to 128 characters. *column_name* can be omitted for columns that are created with a **timestamp** data type. If *column_name* is not specified, the name of a **timestamp** column defaults to **timestamp**.

computed_column_expression
Is an expression that defines the value of a computed column. A computed column is a virtual column that is not physically stored in the table, unless the column is marked PERSISTED. The column is computed from an expression that uses other columns in the same table. For example, a computed column can have the definition: **cost** AS **price** * **qty**. The expression can be a noncomputed column name, constant, function, variable, and any combination of these connected by one or more operators. The expression cannot be a subquery or contain alias data types.

Computed columns can be used in select lists, WHERE clauses, ORDER BY clauses, or any other locations in which regular expressions can be used, with the following exceptions:

- A computed column cannot be used as a DEFAULT or FOREIGN KEY constraint definition or with a NOT NULL constraint definition. However, a computed column can be used as a key column in an index or as part of any PRIMARY KEY or UNIQUE constraint, if the computed column value is defined by a deterministic expression and the data type of the result is allowed in index columns.

  For example, if the table has integer columns **a** and **b**, the computed column **a+b** may be indexed, but computed column **a+DATEPART(dd, GETDATE())** cannot be indexed because the value may change in subsequent invocations.

- A computed column cannot be the target of an INSERT or UPDATE statement.

Note

Each row in a table can have different values for columns that are involved in a computed column; therefore, the computed column may not have the same value for each row.

Based on the expressions that are used, the nullability of computed columns is determined automatically by the Database Engine. The result of most expressions is considered nullable even if only nonnullable columns are present, because possible underflows or overflows also produce NULL results. Use the COLUMNPROPERTY function with the **AllowsNull** property to investigate the nullability of any computed column in a table. An expression that is nullable can be turned into a nonnullable one by specifying ISNULL with the *check_expression* constant, where the constant is a nonnull value substituted for any NULL result. REFERENCES permission on the type is required for computed columns based on common language runtime (CLR) user-defined type expressions.

PERSISTED
Specifies that the SQL Server Database Engine will physically store the computed values in the table, and update the values when any other columns on which the computed column depends are updated. Marking a computed column as

PERSISTED lets you create an index on a computed column that is deterministic, but not precise. For more information, see Creating Indexes on Computed Columns. Any computed columns that are used as partitioning columns of a partitioned table must be explicitly marked PERSISTED. *computed_column_expression* must be deterministic when PERSISTED is specified.

ON { <partition_scheme> | *filegroup* | **"**default**"** }

Specifies the partition scheme or filegroup on which the table is stored. If <partition_scheme> is specified, the table is to be a partitioned table whose partitions are stored on a set of one or more filegroups specified in <partition_scheme>. If *filegroup* is specified, the table is stored in the named filegroup. The filegroup must exist within the database. If **"**default**"** is specified, or if ON is not specified at all, the table is stored on the default filegroup. The storage mechanism of a table as specified in CREATE TABLE cannot be subsequently altered.

ON {<partition_scheme> | *filegroup* | **"**default**"**} can also be specified in a PRIMARY KEY or UNIQUE constraint. These constraints create indexes. If *filegroup* is specified, the index is stored in the named filegroup. If **"**default**"** is specified, or if ON is not specified at all, the index is stored in the same filegroup as the table. If the PRIMARY KEY or UNIQUE constraint creates a clustered index, the data pages for the table are stored in the same filegroup as the index. If CLUSTERED is specified or the constraint otherwise creates a clustered index, and a <partition_scheme> is specified that differs from the <partition_scheme> or *filegroup* of the table definition, or vice-versa, only the constraint definition will be honored, and the other will be ignored.

Note

In this context, default is not a keyword. It is an identifier for the default filegroup and must be delimited, as in ON **"**default**"** or ON **[**default**]**. If **"**default**"** is specified, the QUOTED_IDENTIFIER option must be ON for the current session. This is the default setting. For more information, see SET QUOTED_IDENTIFIER (Transact-SQL).

Note

After you create a partitioned table, consider setting the LOCK_ESCALATION option for the table to AUTO. This can improve concurrency by enabling locks to escalate to partition (HoBT) level instead of the table. For more information, see ALTER TABLE (Transact-SQL).

TEXTIMAGE_ON { *filegroup*| **"**default**"** }

Are keywords that indicate that the **text**, **ntext**, **image**, **xml**, **varchar(max)**, **nvarchar(max)**, **varbinary(max)**, and CLR user-defined type columns are stored on the specified filegroup.

TEXTIMAGE_ON is not allowed if there are no large value columns in the table. TEXTIMAGE_ON cannot be specified if <partition_scheme> is specified. If **"**default**"** is specified, or if TEXTIMAGE_ON is not specified at all, the large value columns are stored in the default filegroup. The storage of any large value column data specified in CREATE TABLE cannot be subsequently altered.

Note

In this context, default is not a keyword. It is an identifier for the default filegroup and must be delimited, as in TEXTIMAGE_ON **"**default**"** or TEXTIMAGE_ON **[**default**]**. If **"**default**"** is specified, the QUOTED_IDENTIFIER option must be ON for the current session. This is the default setting. For more information, see SET QUOTED_IDENTIFIER (Transact-SQL).

FILESTREAM_ON { *partition_scheme_name* | filegroup | **"**default**"** }

Specifies the filegroup for FILESTREAM data.

If the table contains FILESTREAM data and the table is partitioned, the FILESTREAM_ON clause must be included and must specify a partition scheme of FILESTREAM filegroups. This partition scheme must use the same partition function and partition columns as the partition scheme for the table; otherwise, an error is raised.

If the table is not partitioned, the FILESTREAM column cannot be partitioned. FILESTREAM data for the table must be stored in a single filegroup. This filegroup is specified in the FILESTREAM_ON clause.

If the table is not partitioned and the FILESTREAM_ON clause is not specified, the FILESTREAM filegroup that has the DEFAULT property set is used. If there is no FILESTREAM filegroup, an error is raised.

- As with ON and TEXTIMAGE_ON, the value set by using CREATE TABLE for FILESTREAM_ON cannot be changed, except in the following cases:
- A CREATE INDEX statement converts a heap into a clustered index. In this case, a different FILESTREAM filegroup, partition scheme, or NULL can be specified.
- A DROP INDEX statement converts a clustered index into a heap. In this case, a different FILESTREAM filegroup, partition scheme, or **"**default**"** can be specified.

The filegroup in the FILESTREAM_ON <filegroup> clause, or each FILESTREAM filegroup that is named in the partition scheme, must have one file defined for the filegroup. This file must be defined by using a CREATE DATABASE or ALTER DATABASE statement; otherwise, an error is raised.

For related FILESTREAM topics, see Designing and Implementing FILESTREAM Storage.

[*type_schema_name***.**] *type_name*

Specifies the data type of the column, and the schema to which it belongs. The data type can be one of the following:

- A system data type.
- An alias type based on a SQL Server system data type. Alias data types are created with the CREATE TYPE statement before they can be used in a table definition. The NULL or NOT NULL assignment for an alias data type can be overridden during the CREATE TABLE statement. However, the length specification cannot be changed; the length for an alias data type cannot be specified in a CREATE TABLE statement.
- A CLR user-defined type. CLR user-defined types are created with the CREATE TYPE statement before they can be used in a table definition. To create a column on CLR user-defined type, REFERENCES permission is required on the type.

If *type_schema_name* is not specified, the SQL Server Database Engine references *type_name* in the following order:

- The SQL Server system data type.
- The default schema of the current user in the current database.
- The **dbo** schema in the current database.

precision

Is the precision for the specified data type. For more information about valid precision values, see Precision, Scale, and Length.

scale

Is the scale for the specified data type. For more information about valid scale values, see Precision, Scale, and Length.

max

Applies only to the **varchar**, **nvarchar**, and **varbinary** data types for storing 2^31 bytes of character and binary data, and 2^30 bytes of Unicode data.

CONTENT

Specifies that each instance of the **xml** data type in *column_name* can contain multiple top-level elements. CONTENT applies only to the **xml** data type and can be specified only if *xml_schema_collection* is also specified. If not specified, CONTENT is the default behavior.

DOCUMENT

Specifies that each instance of the **xml** data type in *column_name* can contain only one top-level element. DOCUMENT applies only to the **xml** data type and can be specified only if *xml_schema_collection* is also specified.

xml_schema_collection

Applies only to the **xml** data type for associating an XML schema collection with the type. Before typing an **xml** column to a schema, the schema must first be created in the database by using CREATE XML SCHEMA COLLECTION.

DEFAULT

Specifies the value provided for the column when a value is not explicitly supplied during an insert. DEFAULT definitions can be applied to any columns except those defined as **timestamp**, or those with the IDENTITY property. If a default value is specified for a user-defined type column, the type should support an implicit conversion from *constant_expression* to the user-defined type. DEFAULT definitions are removed when the table is dropped. Only a constant value, such as a character string; a scalar function (either a system, user-defined, or CLR function); or NULL can be used as a default. To maintain compatibility with earlier versions of SQL Server, a constraint name can be assigned to a DEFAULT.

constant_expression

Is a constant, NULL, or a system function that is used as the default value for the column.

IDENTITY

Indicates that the new column is an identity column. When a new row is added to the table, the Database Engine provides a unique, incremental value for the column. Identity columns are typically used with PRIMARY KEY constraints to serve as the unique row identifier for the table. The IDENTITY property can be assigned to **tinyint**, **smallint**, **int**, **bigint**, **decimal(p,0)**, or **numeric(p,0)** columns. Only one identity column can be created per table. Bound defaults and DEFAULT constraints cannot be used with an identity column. Both the seed and increment or neither must be specified. If neither is specified, the default is (1,1).

seed

Is the value used for the very first row loaded into the table.

increment

Is the incremental value added to the identity value of the previous row loaded.

NOT FOR REPLICATION

In the CREATE TABLE statement, the NOT FOR REPLICATION clause can be specified for the IDENTITY property, FOREIGN KEY constraints, and CHECK constraints. If this clause is specified for the IDENTITY property, values are not incremented in identity columns when replication agents perform inserts. If this clause is specified for a constraint, the constraint is not enforced when replication agents perform insert, update, or delete operations. For more information, see Controlling Constraints, Identities, and Triggers with NOT FOR REPLICATION.

ROWGUIDCOL

Indicates that the new column is a row GUID column. Only one **uniqueidentifier** column per table can be designated as the ROWGUIDCOL column. Applying the ROWGUIDCOL property enables the column to be referenced using

$ROWGUID. The ROWGUIDCOL property can be assigned only to a **uniqueidentifier** column. The ROWGUIDCOL keyword is not valid if the database compatibility level is 65 or lower. For more information, see sp_dbcmptlevel (Transact-SQL). User-defined data type columns cannot be designated with ROWGUIDCOL.

The ROWGUIDCOL property does not enforce uniqueness of the values stored in the column. ROWGUIDCOL also does not automatically generate values for new rows inserted into the table. To generate unique values for each column, either use the NEWID or NEWSEQUENTIALID function on INSERT statements or use these functions as the default for the column.

SPARSE

Indicates that the column is a sparse column. The storage of sparse columns is optimized for null values. Sparse columns cannot be designated as NOT NULL. For additional restrictions and more information about sparse columns, see Using Sparse Columns.

FILESTREAM

Valid only for **varbinary(max)** columns. Specifies FILESTREAM storage for the **varbinary(max)** BLOB data.

The table must also have a column of the **uniqueidentifier** data type that has the ROWGUIDCOL attribute. This column must not allow null values and must have either a UNIQUE or PRIMARY KEY single-column constraint. The GUID value for the column must be supplied either by an applicationwhen inserting data, or by a DEFAULT constraint that uses the NEWID () function.

The ROWGUIDCOL column cannot be dropped and the related constraints cannot be changed while there is a FILESTREAM column defined for the table. The ROWGUIDCOL column can be dropped only after the last FILESTREAM column is dropped.

When the FILESTREAM storage attribute is specified for a column, all values for that column are stored in a FILESTREAM data container on the file system.

COLLATE *collation_name*

Specifies the collation for the column. Collation name can be either a Windows collation name or an SQL collation name. *collation_name* is applicable only for columns of the **char**, **varchar**, **text**, **nchar**, **nvarchar**, and **ntext** data types. If not specified, the column is assigned either the collation of the user-defined data type, if the column is of a user-defined data type, or the default collation of the database.

For more information about the Windows and SQL collation names, see Windows Collation Name and SQL Collation Name.

For more information about the COLLATE clause, see COLLATE (Transact-SQL).

CONSTRAINT

Is an optional keyword that indicates the start of the definition of a PRIMARY KEY, NOT NULL, UNIQUE, FOREIGN KEY, or CHECK constraint. For more information, see Constraints.

constraint_name

Is the name of a constraint. Constraint names must be unique within the schema to which the table belongs.

NULL | NOT NULL

Determine whether null values are allowed in the column. NULL is not strictly a constraint but can be specified just like NOT NULL. NOT NULL can be specified for computed columns only if PERSISTED is also specified.

PRIMARY KEY

Is a constraint that enforces entity integrity for a specified column or columns through a unique index. Only one PRIMARY KEY constraint can be created per table.

UNIQUE

Is a constraint that provides entity integrity for a specified column or columns through a unique index. A table can have multiple UNIQUE constraints.

CLUSTERED | NONCLUSTERED

Indicate that a clustered or a nonclustered index is created for the PRIMARY KEY or UNIQUE constraint. PRIMARY KEY constraints default to CLUSTERED, and UNIQUE constraints default to NONCLUSTERED.

In a CREATE TABLE statement, CLUSTERED can be specified for only one constraint. If CLUSTERED is specified for a UNIQUE constraint and a PRIMARY KEY constraint is also specified, the PRIMARY KEY defaults to NONCLUSTERED.

FOREIGN KEY REFERENCES

Is a constraint that provides referential integrity for the data in the column or columns. FOREIGN KEY constraints require that each value in the column exists in the corresponding referenced column or columns in the referenced table. FOREIGN KEY constraints can reference only columns that are PRIMARY KEY or UNIQUE constraints in the referenced table or columns referenced in a UNIQUE INDEX on the referenced table. Foreign keys on computed columns must also be marked PERSISTED.

[*schema_name***.**] *referenced_table_name*]

Is the name of the table referenced by the FOREIGN KEY constraint, and the schema to which it belongs.

(*ref_column* [**,**... *n*] **)**

Is a column, or list of columns, from the table referenced by the FOREIGN KEY constraint.

ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT }

Specifies what action happens to rows in the table created, if those rows have a referential relationship and the referenced row is deleted from the parent table. The default is NO ACTION.

NO ACTION

The Database Engine raises an error and the delete action on the row in the parent table is rolled back.

CASCADE

Corresponding rows are deleted from the referencing table if that row is deleted from the parent table.

SET NULL

All the values that make up the foreign key are set to NULL if the corresponding row in the parent table is deleted. For this constraint to execute, the foreign key columns must be nullable.

SET DEFAULT

All the values that make up the foreign key are set to their default values if the corresponding row in the parent table is deleted. For this constraint to execute, all foreign key columns must have default definitions. If a column is nullable, and there is no explicit default value set, NULL becomes the implicit default value of the column.

Do not specify CASCADE if the table will be included in a merge publication that uses logical records. For more information about logical records, see Grouping Changes to Related Rows with Logical Records.

ON DELETE CASCADE cannot be defined if an INSTEAD OF trigger ON DELETE already exists on the table.

For example, in the AdventureWorks database, the **ProductVendor** table has a referential relationship with the **Vendor** table. The **ProductVendor.VendorID** foreign key references the **Vendor.VendorID** primary key.

If a DELETE statement is executed on a row in the **Vendor** table, and an ON DELETE CASCADE action is specified for **ProductVendor.VendorID**, the Database Engine checks for one or more dependent rows in the **ProductVendor** table. If any exist, the dependent rows in the **ProductVendor** table are deleted, and also the row referenced in the **Vendor** table.

Conversely, if NO ACTION is specified, the Database Engine raises an error and rolls back the delete action on the **Vendor** row if there is at least one row in the **ProductVendor** table that references it.

ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT }

Specifies what action happens to rows in the table altered when those rows have a referential relationship and the referenced row is updated in the parent table. The default is NO ACTION.

NO ACTION

The Database Engine raises an error, and the update action on the row in the parent table is rolled back.

CASCADE

Corresponding rows are updated in the referencing table when that row is updated in the parent table.

SET NULL

All the values that make up the foreign key are set to NULL when the corresponding row in the parent table is updated. For this constraint to execute, the foreign key columns must be nullable.

SET DEFAULT

All the values that make up the foreign key are set to their default values when the corresponding row in the parent table is updated. For this constraint to execute, all foreign key columns must have default definitions. If a column is nullable, and there is no explicit default value set, NULL becomes the implicit default value of the column.

Do not specify CASCADE if the table will be included in a merge publication that uses logical records. For more information about logical records, see Grouping Changes to Related Rows with Logical Records.

ON UPDATE CASCADE cannot be defined if an INSTEAD OF trigger ON UPDATE already exists on the table that is being altered.

For example, in the AdventureWorks database, the **ProductVendor** table has a referential relationship with the **Vendor** table: **ProductVendor.VendorID** foreign key references the **Vendor.VendorID** primary key.

If an UPDATE statement is executed on a row in the **Vendor** table, and an ON UPDATE CASCADE action is specified for **ProductVendor.VendorID**, the Database Engine checks for one or more dependent rows in the **ProductVendor** table. If any exist, the dependent rows in the **ProductVendor** table are updated, and also the row referenced in the **Vendor** table.

Conversely, if NO ACTION is specified, the Database Engine raises an error and rolls back the update action on the **Vendor** row if there is at least one row in the **ProductVendor** table that references it.

CHECK

Is a constraint that enforces domain integrity by limiting the possible values that can be entered into a column or columns. CHECK constraints on computed columns must also be marked PERSISTED.

logical_expression

Is a logical expression that returns TRUE or FALSE. Alias data types cannot be part of the expression.

column

Is a column or list of columns, in parentheses, used in table constraints to indicate the columns used in the constraint definition.

[ASC | DESC]

Specifies the order in which the column or columns participating in table constraints are sorted. The default is ASC.

partition_scheme_name

Is the name of the partition scheme that defines the filegroups onto which the partitions of a partitioned table will be mapped. The partition scheme must exist within the database.

[*partition_column_name***.**]

Specifies the column against which a partitioned table will be partitioned. The column must match that specified in the partition function that *partition_scheme_name* is using in terms of data type, length, and precision. A computed columns that participates in a partition function must be explicitly marked PERSISTED.

> ⚠ **Important**
>
> We recommend that you specify NOT NULL on the partitioning column of partitioned tables, and also nonpartitioned tables that are sources or targets of ALTER TABLE...SWITCH operations. Doing this makes sure that any CHECK constraints on partitioning columns do not have to check for null values. For more information, see Transferring Data Efficiently by Using Partition Switching.

WITH FILLFACTOR **=***fillfactor*

Specifies how full the Database Engine should make each index page that is used to store the index data. User-specified *fillfactor* values can be from 1 through 100. If a value is not specified, the default is 0. Fill factor values 0 and 100 are the same in all respects.

> ⚠ **Important**
>
> Documenting WITH FILLFACTOR = *fillfactor* as the only index option that applies to PRIMARY KEY or UNIQUE constraints is maintained for backward compatibility, but will not be documented in this manner in future releases.

column_set_name XML COLUMN_SET FOR ALL_SPARSE_COLUMNS

Is the name of the column set. A column set is an untyped XML representation that combines all of the sparse columns of a table into a structured output. For more information about column sets, see Using Column Sets.

< table_option> ::=

Specifies one or more table options.

DATA_COMPRESSION

Specifies the data compression option for the specified table, partition number, or range of partitions. The options are as follows:

NONE

Table or specified partitions are not compressed.

ROW

Table or specified partitions are compressed by using row compression.

PAGE

Table or specified partitions are compressed by using page compression.

For more information about compression, see Creating Compressed Tables and Indexes.

ON PARTITIONS **(** { <partition_number_expression> | <range> } [**,**...*n*] **)**

Specifies the partitions to which the DATA_COMPRESSION setting applies. If the table is not partitioned, the ON PARTITIONS argument will generate an error. If the ON PARTITIONS clause is not provided, the DATA_COMPRESSION option will apply to all partitions of a partitioned table.

<partition_number_expression> can be specified in the following ways:

- Provide the partition number of a partition, for example: ON PARTITIONS (2).
- Provide the partition numbers for several individual partitions separated by commas, for example: ON PARTITIONS (1, 5).
- Provide both ranges and individual partitions, for example: ON PARTITIONS (2, 4, 6 TO 8)

<range> can be specified as partition numbers separated by the word TO, for example: ON PARTITIONS (6 TO 8).

To set different types of data compression for different partitions, specify the DATA_COMPRESSION option more than once, for example:

```
WITH
(
DATA_COMPRESSION = NONE ON PARTITIONS (1),
DATA_COMPRESSION = ROW ON PARTITIONS (2, 4, 6 TO 8),
DATA_COMPRESSION = PAGE ON PARTITIONS (3, 5)
)
```

<index_option> ::=
Specifies one or more index options. For a complete description of these options, see CREATE INDEX (Transact-SQL).

PAD_INDEX = { ON | OFF }
When ON, the percentage of free space specified by FILLFACTOR is applied to the intermediate level pages of the index. When OFF or a FILLFACTOR value it not specified, the intermediate level pages are filled to near capacity leaving enough space for at least one row of the maximum size the index can have, considering the set of keys on the intermediate pages. The default is OFF.

FILLFACTOR =*fillfactor*
Specifies a percentage that indicates how full the Database Engine should make the leaf level of each index page during index creation or alteration. *fillfactor* must be an integer value from 1 to 100. The default is 0. Fill factor values 0 and 100 are the same in all respects.

IGNORE_DUP_KEY = { ON | OFF }
Specifies the error response when an insert operation attempts to insert duplicate key values into a unique index. The IGNORE_DUP_KEY option applies only to insert operations after the index is created or rebuilt. The option has no effect when executing CREATE INDEX, ALTER INDEX, or UPDATE. The default is OFF.

ON
A warning message will occur when duplicate key values are inserted into a unique index. Only the rows violating the uniqueness constraint will fail.

OFF
An error message will occur when duplicate key values are inserted into a unique index. The entire INSERT operation will be rolled back.

IGNORE_DUP_KEY cannot be set to ON for indexes created on a view, non-unique indexes, XML indexes, spatial indexes, and filtered indexes.

To view IGNORE_DUP_KEY, use sys.indexes.

In backward compatible syntax, WITH IGNORE_DUP_KEY is equivalent to WITH IGNORE_DUP_KEY = ON.

STATISTICS_NORECOMPUTE = { ON | OFF }
When ON, out-of-date index statistics are not automatically recomputed. When OFF, automatic statistics updating are enabled. The default is OFF.

ALLOW_ROW_LOCKS = { ON | OFF }
When ON, row locks are allowed when you access the index. The Database Engine determines when row locks are used. When OFF, row locks are not used. The default is ON.

ALLOW_PAGE_LOCKS = { ON | OFF }
When ON, page locks are allowed when you access the index. The Database Engine determines when page locks are used. When OFF, page locks are not used. The default is ON.

Remarks

SQL Server 2008 can have up to 2 billion tables per database. A table that has a defined column set, can have up to 30,000 columns with a maximum of 1024 non-sparse + computed columns. Tables that do not have column sets are limited to 1024 columns. The number of rows and total size of the table are limited only by the available storage. The maximum number of bytes per row is 8,060. This restriction is relaxed for tables with **varchar**, **nvarchar**, **varbinary**, or **sql_variant** columns that cause the total defined table width to exceed 8,060 bytes. The lengths of each one of these columns must still fall within the limit of 8,000 bytes, but their combined widths may exceed the 8,060 byte limit in a table. For more information, see Row-Overflow Data Exceeding 8 KB.

Each table can contain a maximum of 999 nonclustered indexes, and 1 clustered index. These include the indexes generated to support any PRIMARY KEY and UNIQUE constraints defined for the table.

Space is generally allocated to tables and indexes in increments of one extent at a time. When the table or index is created, it is allocated pages from mixed extents until it has enough pages to fill a uniform extent. After it has enough pages to fill a uniform extent, another extent is allocated every time the currently allocated extents become full. For a report about the amount of space allocated and used by a table, execute **sp_spaceused**.

The Database Engine does not enforce an order in which DEFAULT, IDENTITY, ROWGUIDCOL, or column constraints are specified in a column definition.

When a table is created, the QUOTED IDENTIFIER option is always stored as ON in the metadata for the table, even if the option is set to OFF when the table is created.

Temporary Tables

You can create local and global temporary tables. Local temporary tables are visible only in the current session, and global temporary tables are visible to all sessions. Temporary tables cannot be partitioned.

Prefix local temporary table names with single number sign (#*table_name*), and prefix global temporary table names with a double number sign (##*table_name*).

SQL statements reference the temporary table by using the value specified for *table_name* in the CREATE TABLE statement, for example:

SQL

```
CREATE TABLE #MyTempTable (cola INT PRIMARY KEY);
INSERT INTO #MyTempTable VALUES (1);
```

If more than one temporary table is created inside a single stored procedure or batch, they must have different names.

If a local temporary table is created in a stored procedure or application that can be executed at the same time by several users, the Database Engine must be able to distinguish the tables created by the different users. The Database Engine does this by internally appending a numeric suffix to each local temporary table name. The full name of a temporary table as stored in the **sysobjects** table in **tempdb** is made up of the table name specified in the CREATE TABLE statement and the system-generated numeric suffix. To allow for the suffix, *table_name* specified for a local temporary name cannot exceed 116 characters.

Temporary tables are automatically dropped when they go out of scope, unless explicitly dropped by using DROP TABLE:

- A local temporary table created in a stored procedure is dropped automatically when the stored procedure is finished. The table can be referenced by any nested stored procedures executed by the stored procedure that created the table. The table cannot be referenced by the process that called the stored procedure that created the table.
- All other local temporary tables are dropped automatically at the end of the current session.
- Global temporary tables are automatically dropped when the session that created the table ends and all other tasks have stopped referencing them. The association between a task and a table is maintained only for the life of a single Transact-SQL statement. This means that a global temporary table is dropped at the completion of the last Transact-SQL statement that was actively referencing the table when the creating session ended.

A local temporary table created within a stored procedure or trigger can have the same name as a temporary table that was created before the stored procedure or trigger is called. However, if a query references a temporary table and two temporary tables with the same name exist at that time, it is not defined which table the query is resolved against. Nested stored procedures can also create temporary tables with the same name as a temporary table that was created by the stored procedure that called it. However, for modifications to resolve to the table that was created in the nested procedure, the table must have the same structure, with the same column names, as the table created in the calling procedure. This is shown in the following example.

SQL

```
CREATE PROCEDURE dbo.Test2
AS
CREATE TABLE #t(x INT PRIMARY KEY);
INSERT INTO #t VALUES (2);
SELECT Test2Col = x FROM #t;
GO
CREATE PROCEDURE dbo.Test1
AS
CREATE TABLE #t(x INT PRIMARY KEY);
INSERT INTO #t VALUES (1);
SELECT Test1Col = x FROM #t;
EXEC Test2;
GO
CREATE TABLE #t(x INT PRIMARY KEY);
INSERT INTO #t VALUES (99);
```

```
GO
EXEC Test1;
GO
```

Here is the result set.

```
(1 row(s) affected)

Test1Col
-----------
1

(1 row(s) affected)

Test2Col
-----------
2
```

When you create local or global temporary tables, the CREATE TABLE syntax supports constraint definitions except for FOREIGN KEY constraints. If a FOREIGN KEY constraint is specified in a temporary table, the statement returns a warning message that states the constraint was skipped. The table is still created without the FOREIGN KEY constraints. Temporary tables cannot be referenced in FOREIGN KEY constraints.

We recommend using table variables instead of temporary tables. Temporary tables are useful when indexes must be created explicitly on them, or when the table values must be visible across multiple stored procedures or functions. Generally, table variables contribute to more efficient query processing. For more information, see table (Transact-SQL).

Partitioned Tables

Before creating a partitioned table by using CREATE TABLE, you must first create a partition function to specify how the table becomes partitioned. A partition function is created by using CREATE PARTITION FUNCTION. Second, you must create a partition scheme to specify the filegroups that will hold the partitions indicated by the partition function. A partition scheme is created by using CREATE PARTITION SCHEME. Placement of PRIMARY KEY or UNIQUE constraints to separate filegroups cannot be specified for partitioned tables. For more information, see Partitioned Tables and Indexes.

PRIMARY KEY Constraints

- A table can contain only one PRIMARY KEY constraint.
- The index generated by a PRIMARY KEY constraint cannot cause the number of indexes on the table to exceed 999 nonclustered indexes and 1 clustered index.
- If CLUSTERED or NONCLUSTERED is not specified for a PRIMARY KEY constraint, CLUSTERED is used if there are no clustered indexes specified for UNIQUE constraints.
- All columns defined within a PRIMARY KEY constraint must be defined as NOT NULL. If nullability is not specified, all columns participating in a PRIMARY KEY constraint have their nullability set to NOT NULL.
- If a primary key is defined on a CLR user-defined type column, the implementation of the type must support binary ordering. For more information, see CLR User-Defined Types.

UNIQUE Constraints

- If CLUSTERED or NONCLUSTERED is not specified for a UNIQUE constraint, NONCLUSTERED is used by default.
- Each UNIQUE constraint generates an index. The number of UNIQUE constraints cannot cause the number of indexes on the table to exceed 999 nonclustered indexes and 1 clustered index.
- If a unique constraint is defined on a CLR user-defined type column, the implementation of the type must support binary or operator-based ordering. For more information, see CLR User-Defined Types.

FOREIGN KEY Constraints

- When a value other than NULL is entered into the column of a FOREIGN KEY constraint, the value must exist in the referenced column; otherwise, a foreign key violation error message is returned.
- FOREIGN KEY constraints are applied to the preceding column, unless source columns are specified.
- FOREIGN KEY constraints can reference only tables within the same database on the same server. Cross-database referential integrity must be implemented through triggers. For more information, see CREATE TRIGGER (Transact-SQL).
- FOREIGN KEY constraints can reference another column in the same table. This is referred to as a self-reference.
- The REFERENCES clause of a column-level FOREIGN KEY constraint can list only one reference column. This column must have the same data type as the column on which the constraint is defined.
- The REFERENCES clause of a table-level FOREIGN KEY constraint must have the same number of reference columns as the number of columns in the constraint column list. The data type of each reference column must also be the same as the corresponding column in the column list.
- CASCADE, SET NULL or SET DEFAULT cannot be specified if a column of type **timestamp** is part of either the foreign key or the referenced key.
- CASCADE, SET NULL, SET DEFAULT and NO ACTION can be combined on tables that have referential relationships with each other. If the Database Engine encounters NO ACTION, it stops and rolls back related CASCADE, SET NULL and SET DEFAULT actions. When a DELETE statement causes a combination of CASCADE, SET NULL, SET DEFAULT and NO ACTION actions, all the CASCADE, SET NULL and SET DEFAULT actions are applied before the Database Engine checks for any NO ACTION.
- The Database Engine does not have a predefined limit on either the number of FOREIGN KEY constraints a table can contain that reference other tables, or the number of FOREIGN KEY constraints that are owned by other tables that reference a specific table.

  Nevertheless, the actual number of FOREIGN KEY constraints that can be used is limited by the hardware configuration and by the design of the database and application. We recommend that a table contain no more than 253 FOREIGN KEY constraints, and that it be referenced by no more than 253 FOREIGN KEY constraints. The effective limit for you may be more or less depending on the application and hardware. Consider the cost of enforcing FOREIGN KEY constraints when you design your database and applications.
- FOREIGN KEY constraints are not enforced on temporary tables.
- FOREIGN KEY constraints can reference only columns in PRIMARY KEY or UNIQUE constraints in the referenced table or in a UNIQUE INDEX on the referenced table.
- If a foreign key is defined on a CLR user-defined type column, the implementation of the type must support binary ordering. For more information, see CLR User-Defined Types.

- A column of type **varchar(max)** can participate in a FOREIGN KEY constraint only if the primary key it references is also defined as type **varchar(max)**.

DEFAULT Definitions

- A column can have only one DEFAULT definition.
- A DEFAULT definition can contain constant values, functions, SQL-92 niladic functions, or NULL. The following table shows the niladic functions and the values they return for the default during an INSERT statement.

SQL-92 niladic function	**Value returned**
CURRENT_TIMESTAMP	Current date and time.
CURRENT_USER	Name of user performing an insert.
SESSION_USER	Name of user performing an insert.
SYSTEM_USER	Name of user performing an insert.
USER	Name of user performing an insert.

- *constant_expression* in a DEFAULT definition cannot refer to another column in the table, or to other tables, views, or stored procedures.
- DEFAULT definitions cannot be created on columns with a **timestamp** data type or columns with an IDENTITY property.
- DEFAULT definitions cannot be created for columns with alias data types if the alias data type is bound to a default object.

CHECK Constraints

- A column can have any number of CHECK constraints, and the condition can include multiple logical expressions combined with AND and OR. Multiple CHECK constraints for a column are validated in the order they are created.
- The search condition must evaluate to a Boolean expression and cannot reference another table.
- A column-level CHECK constraint can reference only the constrained column, and a table-level CHECK constraint can reference only columns in the same table.

  CHECK CONSTRAINTS and rules serve the same function of validating the data during INSERT and UPDATE statements.
- When a rule and one or more CHECK constraints exist for a column or columns, all restrictions are evaluated.
- CHECK constraints cannot be defined on **text**, **ntext**, or **image** columns.

Additional Constraint Information

- An index created for a constraint cannot be dropped by using DROP INDEX; the constraint must be dropped by using ALTER TABLE. An index created for and used by a constraint can be rebuilt by using DBCC DBREINDEX.
- Constraint names must follow the rules for identifiers, except that the name cannot start with a number sign (#). If *constraint_name* is not supplied, a system-generated name is assigned to the constraint. The constraint name appears in any error message about constraint violations.
- When a constraint is violated in an INSERT, UPDATE, or DELETE statement, the statement is ended. However, when SET XACT_ABORT is set to OFF, the transaction, if the statement is part of an explicit transaction, continues to be processed. When SET XACT_ABORT is set to ON, the whole transaction is rolled back. You can also use the ROLLBACK TRANSACTION statement with the transaction definition by checking the @@ERROR system function.
- When ALLOW_ROW_LOCKS = ON and ALLOW_PAGE_LOCK = ON, row-, page-, and table-level locks are allowed when you access the index. The Database Engine chooses the appropriate lock and can escalate the lock from a row or page lock to a table lock. For more information, see Lock Escalation (Database Engine). When ALLOW_ROW_LOCKS = OFF and ALLOW_PAGE_LOCK = OFF, only a table-level lock is allowed when you access the index. For more information about configuring the locking granularity for an index, see Customizing Locking for an Index.
- If a table has FOREIGN KEY or CHECK CONSTRAINTS and triggers, the constraint conditions are evaluated before the trigger is executed.

For a report on a table and its columns, use **sp_help** or **sp_helpconstraint**. To rename a table, use **sp_rename**. For a report on the views and stored procedures that depend on a table, use sys.dm_sql_referenced_entities and sys.dm_sql_referencing_entities.

Nullability Rules Within a Table Definition

The nullability of a column determines whether that column can allow a null value (NULL) as the data in that column. NULL is not zero or blank: NULL means no entry was made or an explicit NULL was supplied, and it typically implies that the value is either unknown or not applicable.

When you use CREATE TABLE or ALTER TABLE to create or alter a table, database and session settings influence and possibly override the nullability of the data type that is used in a column definition. We recommend that you always explicitly define a column as NULL or NOT NULL for noncomputed columns or, if you use a user-defined data type, that you allow the column to use the default nullability of the data type. Sparse columns must always allow NULL.

When column nullability is not explicitly specified, column nullability follows the rules shown in the following table.

Column data type	**Rule**
Alias data type	The Database Engine uses the nullability that is specified when the data type was created. To determine the default nullability of the data type, use **sp_help**.
CLR user-defined type	Nullability is determined according to the column definition.
System-supplied data type	If the system-supplied data type has only one option, it takes precedence. **timestamp** data types must be NOT NULL. When any session settings are set ON by using SET:

	• ANSI_NULL_DFLT_ON = ON, NULL is assigned.
	• ANSI_NULL_DFLT_OFF = ON, NOT NULL is assigned.
	• When any database settings are configured by using ALTER DATABASE:
	• ANSI_NULL_DEFAULT_ON = ON, NULL is assigned.
	• ANSI_NULL_DEFAULT_OFF = ON, NOT NULL is assigned.
	• To view the database setting for ANSI_NULL_DEFAULT, use the **sys.databases** catalog view

When neither of the ANSI_NULL_DFLT options is set for the session and the database is set to the default (ANSI_NULL_DEFAULTis OFF), the default of NOT NULL is assigned.

If the column is a computed column, its nullability is always automatically determined by the Database Engine. To find out the nullability of this type of column, use the COLUMNPROPERTY function with the **AllowsNull** property.

Note

The SQL Server ODBC driver and Microsoft OLE DB Provider for SQL Server both default to having ANSI_NULL_DFLT_ON set to ON. ODBC and OLE DB users can configure this in ODBC data sources, or with connection attributes or properties set by the application.

Data Compression

System tables cannot be enabled for compression. When you are creating a table, data compression is set to NONE, unless specified otherwise. If you specify a list of partitions or a partition that is out of range, an error will be generated. For a more information about data compression, see Creating Compressed Tables and Indexes.

To evaluate how changing the compression state will affect a table, an index, or a partition, use the sp_estimate_data_compression_savings stored procedure.

Permissions

Requires CREATE TABLE permission in the database and ALTER permission on the schema in which the table is being created.

If any columns in the CREATE TABLE statement are defined to be of a CLR user-defined type, either ownership of the type or REFERENCES permission on it is required.

If any columns in the CREATE TABLE statement have an XML schema collection associated with them, either ownership of the XML schema collection or REFERENCES permission on it is required.

Examples

A. Using PRIMARY KEY constraints

The following example shows the column definition for a PRIMARY KEY constraint with a clustered index on the `EmployeeID` column of the `Employee` table (allowing the system to supply the constraint name) in the `AdventureWorks` sample database.

```
EmployeeID int
PRIMARY KEY CLUSTERED
```

B. Using FOREIGN KEY constraints

A FOREIGN KEY constraint is used to reference another table. Foreign keys can be single-column keys or multicolumn keys. This following example shows a single-column FOREIGN KEY constraint on the `SalesOrderHeader` table that references the `SalesPerson` table. Only the REFERENCES clause is required for a single-column FOREIGN KEY constraint.

```
SalesPersonID int NULL
REFERENCES SalesPerson(SalesPersonID)
```

You can also explicitly use the FOREIGN KEY clause and restate the column attribute. Note that the column name does not have to be the same in both tables.

```
FOREIGN KEY (SalesPersonID) REFERENCES SalesPerson(SalesPersonID)
```

Multicolumn key constraints are created as table constraints. In the AdventureWorks database, the `SpecialOfferProduct` table includes a multicolumn PRIMARY KEY. The following example shows how to reference this key from another table; an explicit constraint name is optional.

```
CONSTRAINT FK_SpecialOfferProduct_SalesOrderDetail FOREIGN KEY
 (ProductID, SpecialOfferID)
REFERENCES SpecialOfferProduct (ProductID, SpecialOfferID)
```

C. Using UNIQUE constraints

UNIQUE constraints are used to enforce uniqueness on nonprimary key columns. The following example enforces a restriction that the `Name` column of the `Product` table must be unique.

```
Name nvarchar(100) NOT NULL
UNIQUE NONCLUSTERED
```

D. Using DEFAULT definitions

Defaults supply a value (with the INSERT and UPDATE statements) when no value is supplied. For example, the AdventureWorks database could include a lookup table listing the different jobs employees can fill in the company. Under a column that describes each job, a character string default could supply a description when an actual description is not entered explicitly.

```
DEFAULT 'New Position - title not formalized yet'
```

In addition to constants, DEFAULT definitions can include functions. Use the following example to get the current date for an entry.

```
DEFAULT (getdate())
```

A niladic-function scan can also improve data integrity. To keep track of the user that inserted a row, use the niladic-function for USER. Do not enclose the niladic-functions with parentheses.

```
DEFAULT USER
```

E. Using CHECK constraints

The following example shows a restriction made to values that are entered into the `CreditRating` column of the `Vendor` table. The constraint is unnamed.

```
CHECK (CreditRating >= 1 and CreditRating <= 5)
```

This example shows a named constraint with a pattern restriction on the character data entered into a column of a table.

```
CONSTRAINT CK_emp_id CHECK (emp_id LIKE
'[A-Z][A-Z][A-Z][1-9][0-9][0-9][0-9][0-9][FM]'
OR emp_id LIKE '[A-Z]-[A-Z][1-9][0-9][0-9][0-9][0-9][FM]')
```

This example specifies that the values must be within a specific list or follow a specified pattern.

```
CHECK (emp_id IN ('1389', '0736', '0877', '1622', '1756')
OR emp_id LIKE '99[0-9][0-9]')
```

F. Showing the complete table definition

The following example shows the complete table definitions with all constraint definitions for table `PurchaseOrderDetail` created in the AdventureWorks database. Note that to run the sample, the table schema is changed to `dbo`.

SQL

```
CREATE TABLE [dbo].[PurchaseOrderDetail]
(
    [PurchaseOrderID] [int] NOT NULL
        REFERENCES Purchasing.PurchaseOrderHeader(PurchaseOrderID),
    [LineNumber] [smallint] NOT NULL,
    [ProductID] [int] NULL
        REFERENCES Production.Product(ProductID),
    [UnitPrice] [money] NULL,
    [OrderQty] [smallint] NULL,
    [ReceivedQty] [float] NULL,
    [RejectedQty] [float] NULL,
    [DueDate] [datetime] NULL,
    [rowguid] [uniqueidentifier] ROWGUIDCOL  NOT NULL
        CONSTRAINT [DF_PurchaseOrderDetail_rowguid] DEFAULT (newid()),
    [ModifiedDate] [datetime] NOT NULL
        CONSTRAINT [DF_PurchaseOrderDetail_ModifiedDate] DEFAULT (getdate()),
    [LineTotal]  AS (([UnitPrice]*[OrderQty])),
    [StockedQty]  AS (([ReceivedQty]-[RejectedQty])),
CONSTRAINT [PK_PurchaseOrderDetail_PurchaseOrderID_LineNumber]
    PRIMARY KEY CLUSTERED ([PurchaseOrderID], [LineNumber])
    WITH (IGNORE_DUP_KEY = OFF)
)
ON [PRIMARY];
```

G. Creating a table with an xml column typed to an XML schema collection

The following example creates a table with an `xml` column that is typed to XML schema collection `HRResumeSchemaCollection`. The `DOCUMENT` keyword specifies that each instance of the `xml` data type in *column_name* can contain only one top-level element.

SQL

```
USE AdventureWorks;
GO
CREATE TABLE HumanResources.EmployeeResumes
   (LName nvarchar(25), FName nvarchar(25),
    Resume xml( DOCUMENT HumanResources.HRResumeSchemaCollection) );
```

H. Creating a partitioned table

The following example creates a partition function to partition a table or index into four partitions. Then, the example creates a partition scheme that specifies the filegroups in which to hold each of the four partitions. Finally, the example creates a table that uses the partition scheme. This example assumes the filegroups already exist in the database.

```
CREATE PARTITION FUNCTION myRangePF1 (int)
AS RANGE LEFT FOR VALUES (1, 100, 1000) ;
GO
CREATE PARTITION SCHEME myRangePS1
AS PARTITION myRangePF1
TO (test1fg, test2fg, test3fg, test4fg) ;
GO
CREATE TABLE PartitionTable (col1 int, col2 char(10))
ON myRangePS1 (col1) ;
GO
```

Based on the values of column `col1` of `PartitionTable`, the partitions are assigned in the following ways.

Filegroup	**test1fg**	**test2fg**	**test3fg**	**test4fg**
Partition	1	2	3	4
Values	col 1 <= 1	col1 > 1 AND col1 <= 100	col1 > 100 AND col1 <= 1,000	col1 > 1000

I. Using the uniqueidentifier data type in a column

The following example creates a table with a `uniqueidentifier` column. The example uses a PRIMARY KEY constraint to protect the table against users inserting duplicated values, and it uses the `NEWSEQUENTIALID()` function in the `DEFAULT` constraint to provide values for new rows. The ROWGUIDCOL property is applied to the `uniqueidentifier` column so that it can be referenced using the $ROWGUID keyword.

SQL

```
CREATE TABLE dbo.Globally_Unique_Data
(guid uniqueidentifier CONSTRAINT Guid_Default DEFAULT NEWSEQUENTIALID() ROWGUIDCOL,
    Employee_Name varchar(60)
CONSTRAINT Guid_PK PRIMARY KEY (guid) );
```

J. Using an expression for a computed column

The following example shows the use of an expression (`(low + high)/2`) for calculating the `myavg` computed column.

SQL

```
CREATE TABLE dbo.mytable
( low int, high int, myavg AS (low + high)/2 ) ;
```

K. Creating a computed column based on a user-defined type column

The following example creates a table with one column defined as user-defined type `utf8string`, assuming that the type's assembly, and the type itself, have already been created in the current database. A second column is defined based on `utf8string`, and uses method `ToString()` of **type(class)**`utf8string` to compute a value for the column.

```
CREATE TABLE UDTypeTable
( u utf8string, ustr AS u.ToString() PERSISTED ) ;
```

L. Using the USER_NAME function for a computed column

The following example uses the `USER_NAME()` function in the `myuser_name` column.

SQL

```
CREATE TABLE dbo.mylogintable
( date_in datetime, user_id int, myuser_name AS USER_NAME() ) ;
```

M. Creating a table that has a FILESTREAM column

The following example creates a table that has a `FILESTREAM` column `Photo`. If a table has one or more `FILESTREAM` columns, the table must have one `ROWGUIDCOL` column.

```
CREATE TABLE dbo.EmployeePhoto
    (
    EmployeeId int NOT NULL PRIMARY KEY,
    ,Photo varbinary(max) FILESTREAM NULL
    ,MyRowGuidColumn uniqueidentifier NOT NULL ROWGUIDCOL
        UNIQUE DEFAULT NEWID()
    )
```

N. Creating a table that uses row compression

The following example creates a table that uses row compression.

```
CREATE TABLE T1
(c1 int, c2 nvarchar(200) )
WITH (DATA_COMPRESSION = ROW);
```

For additional data compression examples, see Creating Compressed Tables and Indexes.

O. Creating a table that has sparse columns and a column set

The following examples show to how to create a table that has a sparse column, and a table that has two sparse columns and a column set. The examples use the basic syntax. For more complex examples, see Using Sparse Columns and Using Column Sets.

To create a table that has a sparse column, execute the following code.

```
CREATE TABLE T1
(c1 int PRIMARY KEY,
C2 varchar(50) SPARSE NULL ) ;
```

To create a table that has two sparse columns and a column set named `CSet`, execute the following code.

```
CREATE TABLE T1
(c1 int PRIMARY KEY,
C2 varchar(50) SPARSE NULL,
C3 int SPARSE NULL,
CSet XML COLUMN_SET FOR ALL_SPARSE_COLUMNS ) ;
```

See Also

Reference

ALTER TABLE (Transact-SQL)
COLUMNPROPERTY (Transact-SQL)
CREATE INDEX (Transact-SQL)
CREATE VIEW (Transact-SQL)
Data Types (Transact-SQL)
DROP INDEX (Transact-SQL)
sys.dm_sql_referenced_entities (Transact-SQL)
sys.dm_sql_referencing_entities (Transact-SQL)
DROP TABLE (Transact-SQL)
CREATE PARTITION FUNCTION (Transact-SQL)
CREATE PARTITION SCHEME (Transact-SQL)
CREATE TYPE (Transact-SQL)
EVENTDATA (Transact-SQL)
sp_help (Transact-SQL)
sp_helpconstraint (Transact-SQL)
sp_rename (Transact-SQL)
sp_spaceused (Transact-SQL)

Other Resources

Designing and Implementing FILESTREAM Storage

Community Additions

Use of #, ## and @, @@ syntaxt is deprecated in SQL 2008 and will be disabled in the future version.

Check this artcle, almost at the end the above mentioned features are listed as deprecated and no reference as to what replacement will look like.

http://msdn.microsoft.com/en-us/library/ms143729.aspx

So what is the new syntax which we should use?

[Gail Erickson MSFT]. The deprecation notice could be worded more clearly. What is actually being deprecated is the use of # and ## when those are the only characters used to name the object. For example, CREATE TABLE # (c1 int) will not be allowed in a future release. However, using # or ## as the beginning of a temporary table name is not deprecated. That is, CREATE TABLE #MyTempTable (c1 int) is allowed. The use of @, @@, and @@SomeName will not be allowed as user-defined object names and there is no replacement. You will need to rename any objects that use just that symbol as the object name or object names that begin with @@. Hope that helps.

Gail Erickson
2/3/2010