LAB - Check Constraint

In this lab, you will learn how to use SQL Server `CHECK` constraint to enforce domain integrity.

The `CHECK` constraint allows you to specify the values in a column that must satisfy a Boolean expression.

For example, to require positive unit prices, you can use:

```
CREATE SCHEMA test;
GO

CREATE TABLE test.products(
    product_id INT IDENTITY PRIMARY KEY,
    product_name VARCHAR(255) NOT NULL,
    unit_price DEC(10,2) CHECK(unit_price > 0)
);
```

As you can see, the `CHECK` constraint definition comes after the data type. It consists of the keyword `CHECK` followed by a logical expression in parentheses:

```
CHECK(unit_price > 0)
```

You can also assign the constraint a specific name by using the `CONSTRAINT` keyword as follows:

```
CREATE TABLE test.products(
    product_id INT IDENTITY PRIMARY KEY,
    product_name VARCHAR(255) NOT NULL,
    unit_price DEC(10,2) CONSTRAINT positive_price CHECK(unit_price > 0)
);
```

The explicit names help classify the error messages and allow you to refer to the constraints when you want to modify them.

If you don’t specify a constraint name this way, SQL Server automatically generates a name for you.

See the following insert statement:

```
INSERT INTO test.products(product_name, unit_price)
VALUES ('Awesome Free Bike', 0);
```

SQL Server issued the following error:

```
The INSERT statement conflicted with the CHECK constraint "positive_price". The
conflict occurred in database "BikeStores", table "test.products", column
'unit_price'.
```

The error occurred because the unit price is not greater than zero as specified in the `CHECK` constraint.

The following statement works fine because the logical expression defined in the `CHECK` constraint evaluates to `TRUE`:

```
INSERT INTO test.products(product_name, unit_price)
VALUES ('Awesome Bike', 599);
```

`CHECK` constraint and `NULL`

The `CHECK` constraints reject values that cause the Boolean expression to evaluate to `FALSE`.

Because `NULL` evaluates to `UNKNOWN` and not `FALSE`, it can be used in the expression to bypass a constraint.

For example, you can insert a product whose unit price is `NULL` as shown in the following query:

```
INSERT INTO test.products(product_name, unit_price)
VALUES ('Another Awesome Bike', NULL);
```

Here is the output:

```
(1 row affected)
```

SQL Server inserted `NULL` into the `unit_price` column and did not return an error.

To fix this, you need to use a `NOT NULL` constraint for the `unit_price` column.

`CHECK` constraint referring to multiple columns

A `CHECK` constraint can refer to multiple columns. For instance, you store regular and discounted prices in the `test.products` table and you want to ensure that the discounted price is always lower than the regular price:

```
CREATE TABLE test.products(
    product_id INT IDENTITY PRIMARY KEY,
    product_name VARCHAR(255) NOT NULL,
    unit_price DEC(10,2) CHECK(unit_price > 0),
    discounted_price DEC(10,2) CHECK(discounted_price > 0),
    CHECK(discounted_price < unit_price)
);
```

The first two constraints for `unit_price` and `discounted_price` should look familiar.

The third constraint uses a new syntax which is not attached to a particular column. Instead, it appears as a separate line item in the comma-separated column list.

The first two column constraints are column constraints, whereas the third one is a table constraint.

Note that you can write column constraints as table constraints. However, you cannot write table constraints as column constraints. For example, you can rewrite the above statement as follows:

```
CREATE TABLE test.products(
    product_id INT IDENTITY PRIMARY KEY,
    product_name VARCHAR(255) NOT NULL,
    unit_price DEC(10,2),
    discounted_price DEC(10,2),
    CHECK(unit_price > 0),
    CHECK(discounted_price > 0),
    CHECK(discounted_price > unit_price)
);
```

or even:

```
CREATE TABLE test.products(
    product_id INT IDENTITY PRIMARY KEY,
    product_name VARCHAR(255) NOT NULL,
    unit_price DEC(10,2),
    discounted_price DEC(10,2),
    CHECK(unit_price > 0),
    CHECK(discounted_price > 0 AND discounted_price > unit_price)
);
```

You can also assign a name to a table constraint in the same way as a column constraint:

```
CREATE TABLE test.products(
    product_id INT IDENTITY PRIMARY KEY,
    product_name VARCHAR(255) NOT NULL,
    unit_price DEC(10,2),
    discounted_price DEC(10,2),
    CHECK(unit_price > 0),
    CHECK(discounted_price > 0),
    CONSTRAINT valid_prices CHECK(discounted_price > unit_price)
);
```

Adding `CHECK` constraints to an existing table

To add a `CHECK` constraint to an existing table, you use the `ALTER TABLE ADD CONSTRAINT` statement.

Suppose you have the following `test.products` table:

```
CREATE TABLE test.products(
    product_id INT IDENTITY PRIMARY KEY,
    product_name VARCHAR(255) NOT NULL,
    unit_price DEC(10,2) NOT NULL
);
```

To add a `CHECK` constraint to the `test.products` table, you use the following statement:

```
ALTER TABLE test.products
ADD CONSTRAINT positive_price CHECK(unit_price > 0);
```

To add a new column with a `CHECK` constraint, you use the following statement:

```
ALTER TABLE test.products
ADD discounted_price DEC(10,2)
CHECK(discounted_price > 0);
```

To add a `CHECK` constraint named `valid_price`, you use the following statement:

```
ALTER TABLE test.products
ADD CONSTRAINT valid_price
CHECK(unit_price > discounted_price);
```

Remove `CHECK` constraints

To remove a `CHECK` constraint, you use the `ALTER TABLE DROP CONSTRAINT` statement:

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

If you assign a `CHECK` constraint a specific name, you can refer the name in the statement.

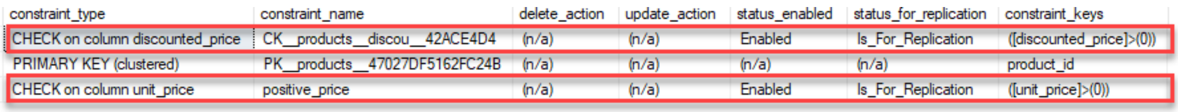
However, in case you did not assign the `CHECK` constraint a particular name, then you need to find it using the following statement:

```
EXEC sp_help 'table_name';
```

For example:

```
EXEC sp_help 'test.products';
```

This statement issues a lot of information including constraint names:



constraint_type	constraint_name	delete_action	update_action	status_enabled	status_for_replication	constraint_keys
CHECK on column discounted_price	CK__products__discou__42ACE4D4	(n/a)	(n/a)	Enabled	Is_For_Replication	([discounted_price]>(0))
PRIMARY KEY (clustered)	PK__products__47027DF5162FC24B	(n/a)	(n/a)	(n/a)	(n/a)	product_id
CHECK on column unit_price	positive_price	(n/a)	(n/a)	Enabled	Is_For_Replication	([unit_price]>(0))

The following statement drops the `positive_price` constraint:

```
ALTER TABLE test.products
DROP CONSTRAINT positive_price;
```

Disable `CHECK` constraints for insert or update

To disable a `CHECK` constraint for insert or update, you use the following statement:

```
ALTER TABLE table_name
NOCHECK CONSTRAINT constraint_name;
```

The following statement disables the `valid_price` constraint:

```
ALTER TABLE test.products
NOCHECK CONSTRAINT valid_price;
```

In this lab, you have learned how to use the SQL Server `CHECK` constraint to limit the values that can be inserted or updated to one or more columns in a table.