SQL FOREIGN KEY Constraint

The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.

A FOREIGN KEY is a field (or collection of fields) in one table, that refers to the PRIMARY KEY in another table.

The table with the foreign key is called the child table, and the table with the primary key is called the referenced or parent table.

Let’s consider two tables, Persons Table<PersonID LastName FirstName Age> and Orders Table <OrderID OrderNumber PersonID>.

The "PersonID" column in the "Orders" table points to the "PersonID" column in the "Persons" table.

The "PersonID" column in the "Persons" table is the PRIMARY KEY in the "Persons" table.

The "PersonID" column in the "Orders" table is a FOREIGN KEY in the "Orders" table.

The FOREIGN KEY constraint prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the parent table.

SQL FOREIGN KEY on CREATE TABLE

The following SQL creates a FOREIGN KEY on the "PersonID" column when the "Orders" table is created:

```
CREATE TABLE Orders (
    OrderID int NOT NULL,
    OrderNumber int NOT NULL,
    PersonID int,
    PRIMARY KEY (OrderID),
    FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)
);
```

To allow naming of a FOREIGN KEY constraint, and for defining a FOREIGN KEY constraint on multiple columns, use the following SQL syntax:

```
CREATE TABLE Orders (
   OrderID int NOT NULL,
   OrderNumber int NOT NULL,
   PersonID int,
   PRIMARY KEY (OrderID),
   CONSTRAINT FK_PersonOrder FOREIGN KEY (PersonID)
   REFERENCES Persons(PersonID)
);
```

DROP a FOREIGN KEY Constraint

You can anytime drop this key

To drop a FOREIGN KEY constraint, use the following SQL:

```
ALTER TABLE Orders
DROP FOREIGN KEY FK_PersonOrder;
```

Let's focus, Foreign keys are a central concept in SQL databases; they allow us to enforce data consistency. Usually they work with primary keys to connect two database tables, like a virtual bridge. All SQL developers need to know what SQL foreign keys are, how they work, what data values are allowed in them, and how they're created.

When designing a database, it often happens that an element can't be fully represented in just one table. In that case, we use a foreign key to link both tables in the database.

Suppose we have a database with country data stored in a table called **country**. We also want to represent the world's major cities in this database. Initially, we think of creating a single table called **city** to store the city name, a numeric city_id, and the country where this city is located. For this last attribute, we add a column called country_id that references a record in the **country** table. Thus, part of the city information – the name of the country where the city is located – is actually stored in the **country** table. We can say that these tables are related.

Here, city has a column called country_id that is the link between both tables. The value in the column city.country_id should refer to a valid country_id in the country table. Using the city.country_id value, we can join the city table with country and find the record associated with any city. So far , so good.

However, the database doesn't know that both tables are related. Moreover, the database will allow a user to put an invalid value in city.country_id (e.g. something that points to a non-existent record in the country table). A foreign key constraint will inform the database about the relation between the tables.

A foreign key constraint is a database constraint that binds two tables. Before we can create a foreign key on the table **city**, we need a primary key or a unique constraint on the **country** table. In the code below, we drop and re-create both tables – this time, defining the primary and foreign keys:

```
-- We first create the country table with a primary key

DROP TABLE country;

CREATE TABLE country (

   country_id      INTEGER,

   name          VARCHAR(50),

   population      INTEGER,

   PRIMARY KEY      (country_id)

);
```

Once we created the FOREIGN KEY, the database will verify that the values stored in city.country_id are valid country.country_id values.

Foreign key validations:

Behind all foreign key validations there is a single rule: The database must ensure that every record in the child table always refers to a valid record in the parent table. There can be several scenarios where this validation is executed:

- If an INSERT INTO city (the child table) is executed, the database must validate that the value of the city.country_id column refers to an existing record in **country** (the parent table)
- If a DELETE FROM country (the parent table) is executed, the database must validate that the **city** table doesn't have any records pointing to the country_id being deleted.
- If an UPDATE for the column country_id on either table (child or parent) is executed, the database must avoid having a record in **city** (the child side) with a value in country_id that doesn't exist in the **country** table (the parent side).

In the case of a DELETE command, there is a ON DELETE CASCADE clause that can be used when the foreign key is created. If we used the ON DELETE CASCADE clause in the foreign key definition, then any DELETE on the parent table (**country**) will fire automatic DELETES on the child records instead of returning a validation error. In other words, if a record in the parent table is deleted, then the corresponding records in the child table will automatically be deleted.

There are also other clauses like RESTRICT, NO ACTIONS, or SET NULL that can be used when defining a foreign key. The next CREATE TABLE example shows how to create the table **city** with the CASCADE DELETE activated.

```
DROP TABLE city;

CREATE TABLE city  (

  city_id       INTEGER,

  name          VARCHAR(50),

  country_id    INTEGER,

  PRIMARY KEY   (city_id),

  FOREIGN KEY   (country_id) REFERENCES country(country_id) ON DELETE CASCADE

)
```

Multi-Column Foreign Keys

In the same way that primary keys can be made up of multiple columns, foreign keys in SQL can also be made up of multiple columns.

Suppose there is a brand of luxury watches that maintain records of the owners of all the watches they produce. Every kind of watch model is identified by a model name and a sub-model code. In the table, **watch**:

Here, the primary key for the table **watch** is a multi-column key formed by
the Model and Sub_model columns.

The database has another table called **owner** that stores information on each watch owner’s name, the purchase date, and the watch model and sub-model:

The following SQL code is used to create the primary key on the table watch and the foreign key on the table owner.

```
-- We first create the watch table with the primary key

CREATE TABLE watch (
   Model         VARCHAR(15),
   Sub_model     VARCHAR(15),
   Price         NUMERIC,
   PRIMARY KEY (model, sub_model)
);

-- We create the owner table with a multi-column foreign key

CREATE TABLE owner (
   Owner_name     VARCHAR(50),
   Model          VARCHAR(15),
   Sub_model      VARCHAR(15),
   Purchase_date  DATE,
   PRIMARY KEY    (Owner),
   FOREIGN KEY    (Model, Sub_model) REFERENCES watch(model, sub_model)
);
```

OBJECTPROPERTY() function

This is a useful information, you can use the **OBJECTPROPERTY() function** in SQL Server to check whether or not a table has one or more foreign key constraints. To do this, pass the table’s object ID as the first argument, and TableHasForeignKey as the second argument.

Or use: *SHOW KEYS FROM YourTableName WHERE Key_name = 'PRIMARY';*

Finally it’s important to mention that not all databases have the primary key constraint always on. You need to check for that before using the database.

https://www.w3schools.com/

https://LearnSQL.com

Report to Eng/ Mahmoud Algharabawi

Presented by: Nanette Nassif Mikhail.