learnsql.com

Referential Constraints and Foreign Keys in MySQL

Francisco Claria

17-21 minutes

Foreign keys and referential constraints allow you to set relationships between tables and modify some of the database engine's actions. This beginner's guide explains referential integrity and foreign key use in MySQL.

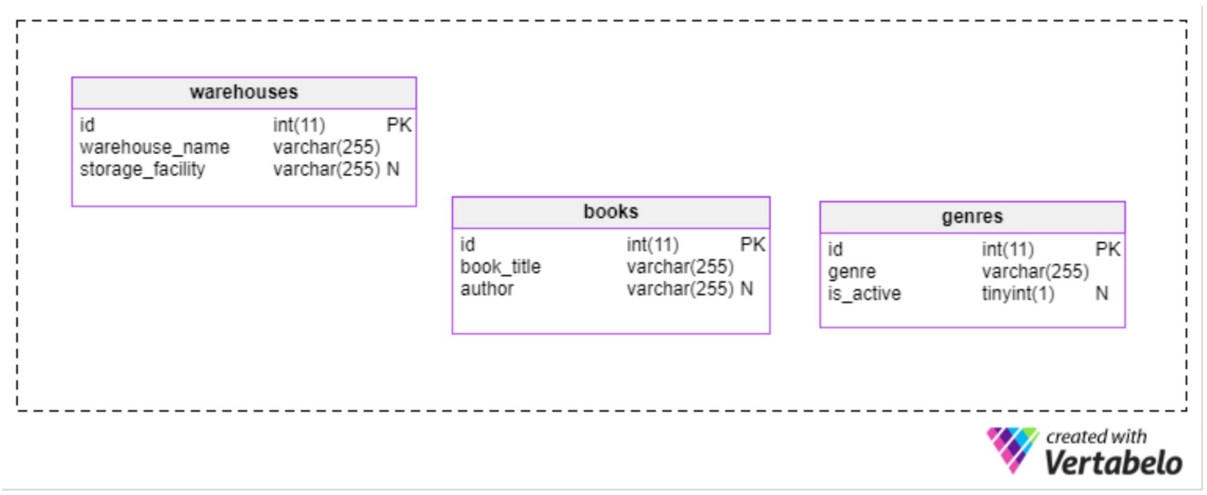
One of the most important aspects of database usage is being able to trust the information you store. Database engines provide several features that help you maintain the quality of your data, like **defining** required columns as NOT NULL and **setting** an exact data type for each column.

There are situations where information in one table has a relationship with information in another table, like having a bookseller's warehouses in one table and storing books associated with each warehouse in a separate table. Often, this relationship needs to be strengthened by including references to the other table. To do this, we define a set of "rules" known as foreign keys and referential constraints that tell the database how to reinforce these relationships.

Let's start with the simple scenario of a bookseller. This bookseller needs to store records in several interrelated database tables. The model is shown below.

The Scenario

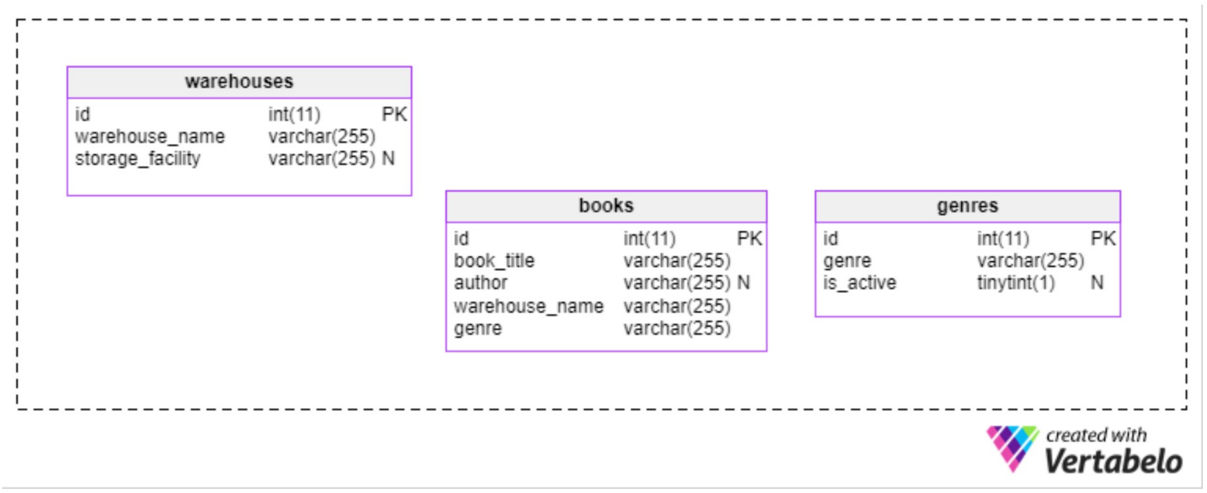
The bookseller uses a simple database that tracks **books** of various **genres**. These books are stored in several **warehouses**. This is what the data model looks like:



There is a table for each entity, but we are not yet able to associate a given book to its genre or indeed to the warehouse where it is stored. To do this, we need two new columns in the “**books**” table:

- A column to store the relevant warehouse
- A column to store the book’s genre

One possibility could be to simply add the warehouse name and genre to the “**books**” table, like this:



Let’s populate the “**warehouses**” table with some sample data:

INSERT INTO warehouses (warehouse_name, storage_facility)
VALUES ('Depo1', 'L.A');
INSERT INTO warehouses (warehouse_name, storage_facility)
VALUES ('Depo1', 'N.Y');
INSERT INTO warehouses (warehouse_name) VALUES ('Depo2');
INSERT INTO warehouses (warehouse_name) VALUES ('Depo3');

And now we’ll insert a record for a book in the “Depo3” warehouse:

INSERT INTO books (book_title, author, warehouse_name) VALUES ('Alice in Wonderland', 'Lewis Carroll', 'Depo3');

You may have already noticed how this would be prone to inconsistencies. For instance, you could save a record with a "`warehouse_name`" that doesn't exist in the "**`warehouses`**" table:

INSERT INTO fkexample2.books (book_title, author, warehouse_name) VALUES ('Alice in Wonderland', 'Lewis Carroll', 'False Depo');

Or you might update a warehouse (or genre) and forget to update the associated books:

UPDATE warehouses SET warehouse_name='Depo3-1' WHERE warehouse_name='Depo3'

In the record below, we can see there is still a book pointing to "Depo3" (which doesn't exist as far as the database knows):

id	book_title	author	warehouse_name	genre
1	Alice in Wonderland	Lewis Carroll	**Depo3**	null

There is another problem. Suppose we want to add a book to the "Depo1" warehouse:

id	warehouse_name	storage_facility
1	**Depo1**	L.A
2	**Depo1**	N.Y
3	Depo2	null
4	Depo3-1	null

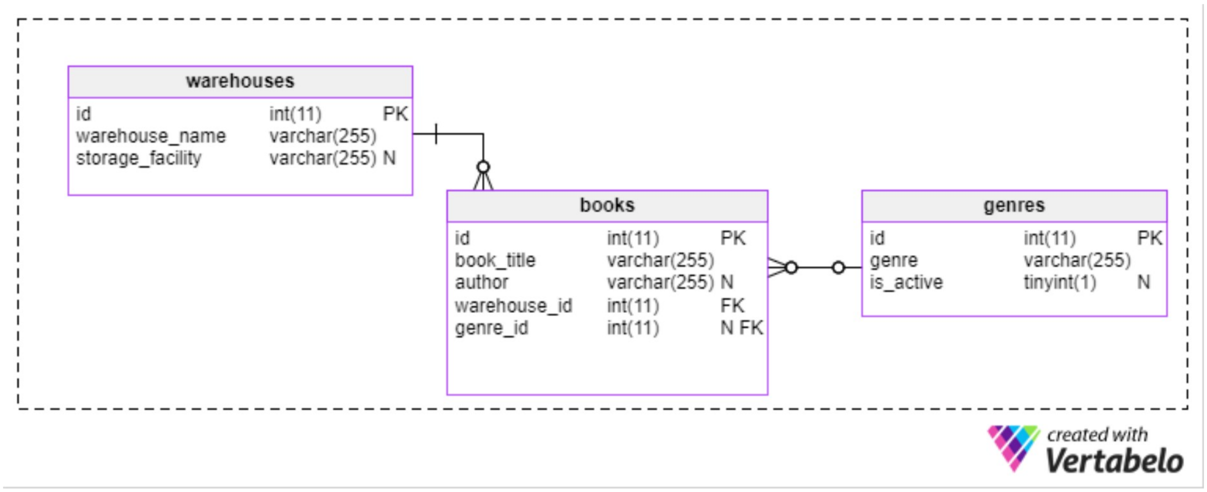
There are two warehouses called "Depo1". If we went by just this facility name, the database wouldn't know which warehouse the book belongs to.

You may think that this confusion could never happen to you. Why would you add a book to a warehouse that doesn't exist or forget to update a field? This actually happens all the time; people manually enter a wrong value or there's a fault in the software that connects to your database. Trust me, it will happen.

The fix is to model your database in a way that maintains the integrity of your data. SQL provides foreign keys (a.k.a. FKs) to help us with this task. Let's see how it works.

What Are Foreign Keys?

Let me begin by showing you how the above tables will look with foreign keys:



Foreign keys are about **reinforcing relationships at a database level**. The line between the tables in the above diagram gives you an idea of how the tables are related. Also the `FK` label next to "`warehouse_id`" and "`genre_id`" clearly identifies what columns are "linking" these tables.

We'll now back up a bit and assume we're working with the unconnected tables shown in the previous section. The first thing to do is to insert some sample data for warehouses and genres:

INSERT INTO warehouses (id, warehouse_name, storage_facility) VALUES (1, 'Depo1', 'L.A');
INSERT INTO warehouses (id, warehouse_name, storage_facility) VALUES (2, 'Depo1', 'N.Y');

INSERT INTO genres (id, genre, is_active) VALUES (1, 'fiction', '1');
INSERT INTO genres (id, genre, is_active) VALUES (2, 'action', '1');

And now we'll insert a book pointing to a warehouse with `id=1` and a genre with `id=2`:

INSERT INTO books (book_title, author, warehouse_id, genre_id)
VALUES ('Alice in Wonderland', 'Lewis Carroll', 1, 2)

This record will insert flawlessly since the data is "consistent". In other words, we are inserting a book associated with a warehouse and a genre that exist in the database. But what if the book referred to values that didn't exist? Try this one:

INSERT INTO books (book_title, author, **warehouse_id**, genre_id)
VALUES ('Doctor No', 'Ian Fleming', **10**, 2);

Error Code: 1452. Cannot add or update a child row: a foreign key constraint fails.

This `INSERT` is associating a book with a warehouse that doesn't exist in our database. Therefore, the command fails. The relationship defined in the foreign key is not met and we are prevented from storing inconsistent data.

Now that we have the general picture, let's examine foreign keys in more detail and see how they work in a variety of scenarios.

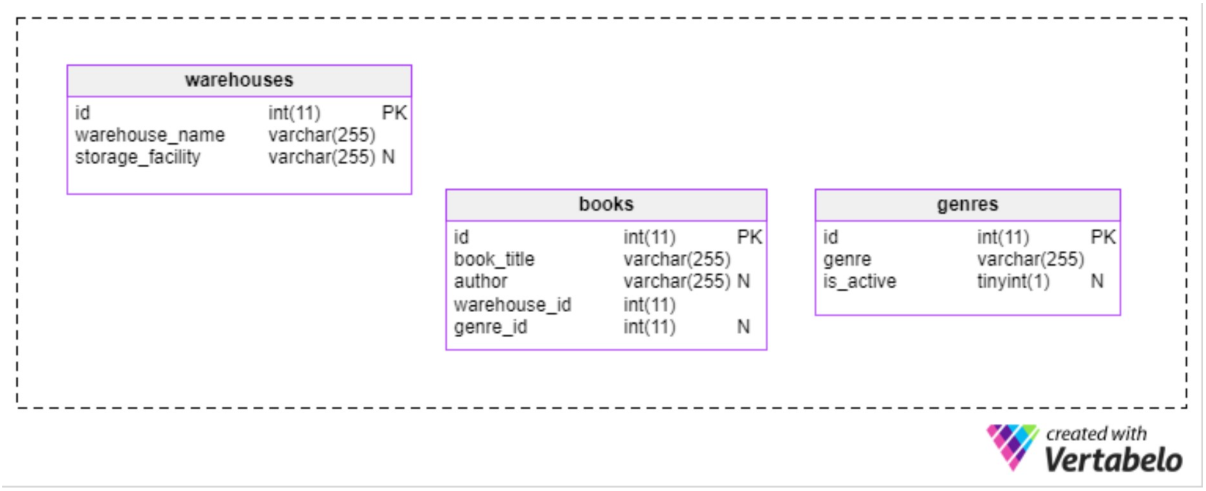
Using Foreign Keys

A foreign key is a column that uniquely references a record in another table. In our example, each table has an "`id`" column that uniquely identifies each row. We use the "`warehouse_id`" and "`genre_id`" to "connect" these tables to the "**`books`**" table. Note that primary keys are most likely to be used as foreign keys, but other `UNIQUE` columns can also be used.

Adding a Foreign Key to a Table

To add foreign keys, the first step is to create the columns in the "**`books`**" table that will store the references to the other tables. We're following a naming convention, so we'll name the columns using the table name (singular), an underscore (_), and the referenced column (in this case, “id”). As previously mentioned, this gives us “warehouse_id” and “genre_id”.

Our model now looks like this:



warehouses		
id	int(11)	PK
warehouse_name	varchar(255)	
storage_facility	varchar(255)	N

books		
id	int(11)	PK
book_title	varchar(255)	
author	varchar(255)	N
warehouse_id	int(11)	
genre_id	int(11)	N

genres		
id	int(11)	PK
genre	varchar(255)	
is_active	tinyint(1)	N

We are ready to add the foreign keys. Let’s start with the warehouse relationship:

```
ALTER TABLE books
ADD CONSTRAINT fk_books_warehouses_warehouse_id
  FOREIGN KEY (warehouse_id)
  REFERENCES warehouses (id);
```

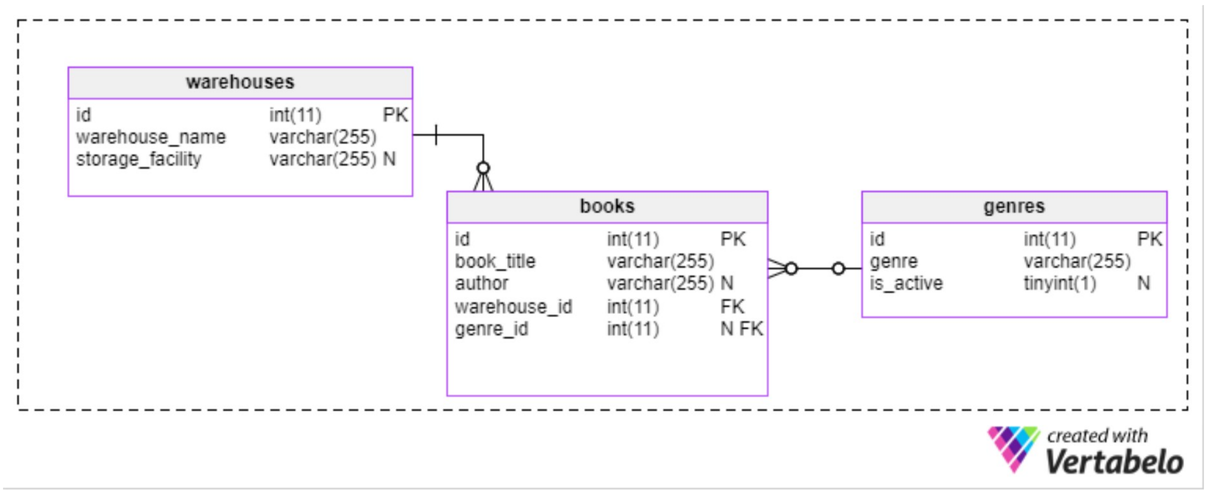
As you can see the query is self-explanatory: we modify (alter) the “**books**” table to add a constraint identified as `fk_books_warehouses_warehouse_id`. This instructs the engine to set the “`warehouse_id`” column as a reference to the “`id`” column in the “**`warehouses`**” table.

The constraint name is optional, but if you supply one it must be unique to the database. A naming convention you could use is `fk_[referencing table name]_[referenced table name]_[referencing field name]`.

Next, we do the same for the books-genres relation:

```
ALTER TABLE books
ADD CONSTRAINT fk_books_genres_genre_id
  FOREIGN KEY (genre_id)
  REFERENCES genres (id);
```

As anticipated, our model now looks like this:



Foreign Keys and Relational Integrity

How do foreign keys protect the relationships between tables and records? To answer this question, let’s use some examples. (Note: We will use the terms “parent” and “child” to describe records and tables. Put simply, a “child” record is a record that contains foreign keys. In this case, the book record is the child record because it references both the warehouse and genre tables (the parent tables). As we will see, the child record is dependent on the data in the parent tables.)

We need to populate the “`warehouses`” and “`genres`” with fresh data. First, we clean the tables of any previous records:

```
    TRUNCATE TABLE books;
    TRUNCATE TABLE genres;
    TRUNCATE TABLE warehouses;
```

And now we can insert new records:

```
INSERT INTO warehouses (warehouse_name, storage_facility)
VALUES ('Depo1', 'L.A');
INSERT INTO warehouses (warehouse_name, storage_facility)
VALUES ('Depo1', 'N.Y');
INSERT INTO warehouses (warehouse_name) VALUES ('Depo2');
INSERT INTO warehouses (warehouse_name) VALUES ('Depo3');

INSERT INTO genres (genre, is_active) VALUES ('fiction', '1');
```

INSERT INTO genres (genre, is_active) VALUES ('action', '1');
INSERT INTO genres (genre, is_active) VALUES ('horror', '0');
INSERT INTO genres (genre, is_active) VALUES ('drama', '1');

By default, the constraint we added will ensure the following:

- When inserting a new book record (i.e. a child record), the “`warehouse_id`” and “`genre_id`” fields will be compared against the “`id`” columns of “**`warehouses`**” and “**`genres`**”. If at least one of the values does not match (i.e. there is no parent record in the table) then the book record won’t be inserted; its foreign key relation is not met. Remember this record?
  INSERT INTO books (book_title, author, warehouse_id, genre_id) VALUES ('Alice in Wonderland', 'Lewis Carroll', 1,1)

  Because it points to a warehouse that doesn’t exist, it fails:

  INSERT INTO books (book_title, author, **warehouse_id**, genre_id) VALUES ('Doctor No', 'Ian Fleming', **10**, 2);

  Error Code: 1452. Cannot add or update a child row: a foreign key constraint fails

- If you delete a warehouse or genre record (a parent record) that has a book (child) record related to it, the deletion will not be performed:
  DELETE FROM warehouses WHERE id = 1; *#the warehouse where Alice in wonderland book is*

  Error Code: 1451. Cannot delete or update a parent row: a foreign key constraint fails

- The same is true for updates; any update of a parent record will not be performed if there are existing child records.
  UPDATE warehouses SET id=22 WHERE id = 1;*#the warehouse where Alice in wonderland book is*
  Error Code: 1451. Cannot delete or update a parent row: a foreign key constraint fails

However, if we try to update another warehouse that has no books, this command works:

```
UPDATE warehouses SET id=22 WHERE id = 2;
```

1 row(s) affected Rows matched: 1 Changed: 1 Warnings: 0

A Quick Note About Foreign Keys
There are some important considerations to remember when working with foreign keys: • The child table holds the reference(s) to the parent table(s), so the FOREIGN KEY clause is specified in the child table. • Foreign key columns and the columns they refer to must have matching data types; for INTEGER types, the size and type must be the same (e.g. `int(11)`). • You must have an index on foreign keys and referenced keys. This allows key checks to be done quickly, without a table scan. This index will automatically be created on the referencing table if it does not already exist. In the above example, we let the engine create the index for us. • BLOB and TEXT columns cannot be included in a foreign key. For more info, check out this link to MySQL documentation.

What are Referential Constraints?

We know we can't delete or update a warehouse or genre if there is any book record pointing to it. Let's imagine we have thousands of books in a warehouse. What if we want to delete that warehouse from our database and keep the consistency of our data? We would first delete all the books and then delete the warehouse. This will eliminate the foreign key restriction error.

This behavior can be configured by defining referential constraints. When we define the "**books**" table's foreign keys, we specify how we want the engine to "react" when an update or a delete occurs on related parent records.

The syntax is generally as follows:

… CONSTRAINT [constraint_name] FOREIGN KEY
[index_name] (index_col_name, ...)
REFERENCES tbl_name (index_col_name,...)
[ON DELETE action]
[ON UPDATE action]

It’s important to note that it is not mandatory to set both referential actions (`ON DELETE`, `ON UPDATE`). Also, we do not need to set these actions to the same outcome. Let me show you some possible situations based on our bookseller scenario:

ON DELETE CASCADE and ON UPDATE CASCADE in MySQL

When a delete occurs on the row from the parent table, setting `CASCADE` will automatically **delete the matching rows in the child table**.

In a real-world scenario, we could have a software requirement that whenever a warehouse deposit gets deleted from our system then all its associated books are also deleted. We could simply write a referential constraint, like this:

ALTER TABLE books
ADD CONSTRAINT fk_books_warehouses_warehouse_id
FOREIGN KEY (warehouse_id)
REFERENCES warehouses (id)
ON DELETE CASCADE;

When we set `ON DELETE CASCADE`, we are instructing the engine to propagate the deletion performed on the warehouse to its related books. We can now try deleting the warehouse where *Alice in Wonderland* is located and see if it works:

DELETE FROM warehouses WHERE id = 1;
1 row(s) affected

The deletion has gone through!

Likewise, say that warehouse id numbers change over time and all books pointing to the warehouse must also be updated to match. This complex operation can be solved by setting a referential constraint for the update action:

ALTER TABLE books
ADD CONSTRAINT fk_books_warehouses_warehouse_id
FOREIGN KEY (warehouse_id)
REFERENCES warehouses (id)
ON UPDATE CASCADE;

The `ON UPDATE CASCADE` tells the database that when an update occurs on the referenced column from the parent table (“**`id`**”), it must automatically update the matching rows in the child table (“**`books`**”) with the new value.

Let’s insert a new book into Warehouse 3:

```
INSERT INTO books (book_title, author, warehouse_id, genre_id) VALUES ('Jurassic Park', 'Michael Crichton', 3, 2)
```

If we update the id of Warehouse 3, the database will also update the book record:

UPDATE warehouses SET id='33' WHERE id='3';

1 row(s) affected Rows matched: 1 Changed: 1 Warnings: 0

The “`warehouse_id`” for this book is now 33, as we expected:

id	book_title	author	warehouse_id	genre_id
7	Jurassic Park	Michael Crichton	**33**	2

These constraints can be combined; we can have both actions configured for the `DELETE` and `UPDATE` actions, as shown below:

Note: You can drop the previous constraint by running:

ALTER TABLE books

```
DROP  FOREIGN KEY  fk_books_warehouses_warehouse_id
```

Now you can run:

```
ALTER TABLE books
ADD CONSTRAINT fk_books_warehouses_warehouse_id
FOREIGN KEY (warehouse_id)
REFERENCES warehouses (id)
ON DELETE CASCADE
ON UPDATE CASCADE;
```

ON DELETE SET NULL and ON UPDATE SET NULL in MySQL

Like `CASCADE`, we can use `SET NULL` on delete and update operations. The foreign key column from the child table (books) will be set to `NULL` when the parent record gets updated or deleted.

Suppose we decide that deleting a warehouse should delete all its books, but deleting a genre should not delete the books linked to that genre (since the books still exist in the warehouse). So when a genre is deleted, we simply remove the reference to the deleted genre. We do this by saving a NULL value in the “`genre_id`” column for the related books.

To achieve this, we employ `SET NULL` as a constraint when creating the foreign key:

```
ALTER TABLE books
ADD CONSTRAINT fk_books_genres_genre_id
FOREIGN KEY (genre_id)
REFERENCES genres (id)
ON DELETE SET NULL
```

Important! Before you specify the `SET NULL` action, make sure that you have declared the “`genre_id`” column in the “**`books`**” table as `NULL`able.

SET NULL can also be defined for updates, like this:

```
ALTER TABLE books
```

```
ADD CONSTRAINT fk_books_genres_genre_id
FOREIGN KEY (genre_id)
REFERENCES genres (id)
ON UPDATE SET NULL
```

This is a relatively rare situation; it’s more common to have `ON UPDATE` set to `CASCADE`. As I explained earlier, though, we don’t need to set the same action for both `UPDATE` and `DELETE`; for instance, I can `SET NULL` when a deletion occurs but use a `CASCADE` for an update, like this:

```
ALTER TABLE books
ADD CONSTRAINT fk_books_genres_genre_id
FOREIGN KEY (genre_id)
REFERENCES genres (id)
ON DELETE SET NULL
ON UPDATE CASCADE;
```

RESTRICT and NO ACTION

Setting `RESTRICT` is the same as omitting the `ON DELETE` or `ON UPDATE` clause; it rejects the delete or update operation for the parent table if the table has associated children. In other words, if you do not specify `ON DELETE` or `ON UPDATE`, by default you invoke `RESTRICT`.

In our scenario, we might only allow the deletion of unused genres (i.e. genres that have no books associated with them). If there is a book pointing to the “`genre_id`” we want to delete, the operation should be `RESTRICT`ed (rejected).

This referential constraint will do the trick:

```
ALTER TABLE books
ADD CONSTRAINT fk_books_genres_genre_id
FOREIGN KEY (genre_id)
REFERENCES genres (id)
ON DELETE RESTRICT;
```

`NO ACTION` is an SQL keyword that, in MySQL, is equivalent to `RESTRICT`. (Note: Other databases may interpret this command differently.) `NO ACTION` rejects the delete or update operation for the parent table if there is a related foreign key value in the child table.

```
ALTER TABLE books
ADD CONSTRAINT fk_books_warehouses_warehouse_id
FOREIGN KEY (warehouse_id)
REFERENCES warehouses (id)
ON DELETE NO ACTION;
```

Want to Learn More?

I've introduced some initial concepts about the complex world of data integrity and consistency. There is a lot involved in mastering this, mostly in terms of database and table design. For this reason, I encourage you to take a look at LearnSQL.com's **Creating Tables in SQL** course. It is designed to be easy to understand, yet also comprehensive enough to show you how to create optimal tables in SQL.

Never worked with SQL and don't know how to use it? That's OK! Just make sure to check the SQL Basics course first.