BACK TO OPEN BOOK SHELF

SHARE ON

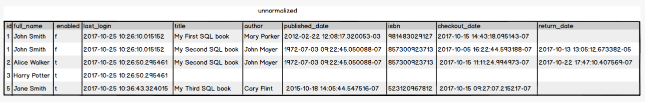
Table Relationships

Thus far in this book, all the work we've done has been with a single database table. The majority of databases you'll work with as a developer will have more than one table, and those tables will be connected together in various ways to form table relationships. In this chapter we'll explore the reasons for having multiple tables in a database, look at how to define relationships between different tables, and outline the different types of table relationships that can exist.

Normalization

At this point, our `users` table doesn't need to hold that much data for each user in our system. It stores a name for the user, whether their account is enabled or not, when they last logged in, and an id to identify each user record. In reality, the requirements of our application will mean that we need to store a lot more data than that. Our app will be used to manage a library of SQL books and allow users to check out the books and also review them.

To implement some of these requirements we could simply try adding more columns to our `users` table; the resulting table might look a little like this:



unnormalized

id	full_name	enabled	last_login	title	author	published_date	isbn	checkout_date	return_date
1	John Smith	f	2017-10-25 10:26:10.015152	My First SQL book	Mary Parker	2012-02-22 12:08:17.320053-03	981483029127	2017-10-15 14:43:18.095143-07	
1	John Smith	f	2017-10-25 10:26:10.015152	My Second SQL book	John Mayer	1972-07-03 09:22:45.050088-07	857300923713	2017-10-05 16:22:44.593188-07	2017-10-13 13:05:12.673382-05
2	Alice Walker	t	2017-10-25 10:26:50.295461	My Second SQL book	John Mayer	1972-07-03 09:22:45.050088-07	857300923713	2017-10-15 11:11:24.994973-07	2017-10-22 17:47:10.407569-07
3	Harry Potter	t	2017-10-25 10:26:50.295461						
5	Jane Smith	t	2017-10-25 10:36:43.324015	My Third SQL book	Cary Flint	2015-10-18 14:05:44.547516-07	523120967812	2017-10-15 09:27:07.215217-07	

Wow, that's a lot of information all in one table! There are other issues here as well, such as duplication of data (often referred to as 'redundancy'). For each book that a user checks out, we have to repeat all of the user data in our table. It's a similar story with the book data, if more than one person checks out a book, such as with 'My Second SQL Book', we have to repeat the book title, author, isbn, and published date.

Duplicating data in this way can lead to issues with data integrity. For example what if for one of the 'My Second SQL Book' checkouts the title is entered as 'My 2nd SQL Book' instead, or a typo had been made with the isbn on one of the rows? From looking at the data in the table, how would we know which piece of data is correct?

How do we deal with this situation? The answer is to split our data up across multiple different tables, and create relationships between them. The process of splitting up data in this way to remove duplication and improve data integrity is known as *normalization*.

Normalization is a deep topic, and there are complex sets of rules which dictate the extent to which a database is judged to be *normalized*. These rule-sets, known as 'normal forms', are beyond the scope of this book; for now there are two important things to remember:

- The *reason* for normalization is to reduce data redundancy and improve data integrity
- The *mechanism* for carrying out normalization is arranging data in multiple tables and defining relationships between them

We know that we want to split the data for our application across multiple tables, but how do we decide what those tables should be and what relationships should exist between them? When answering questions such as these it is often useful to zoom out and think at a higher level of abstraction, and this is where the process of database design comes in.

Database Design

At a high level, the process of database design involves defining**entities** to represent different sorts of data and designing **relationships** between those entities. But what do we mean by entities, and how do different entities relate to each other? Let's find out.

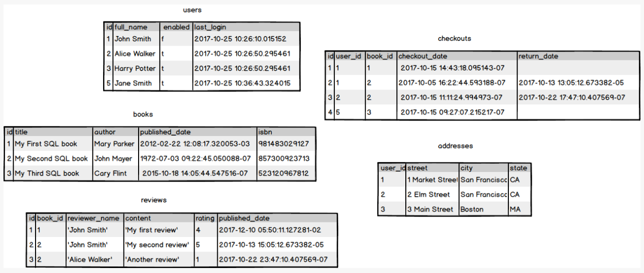
Entities

An entity represents a real world object, or a set of data that we want to model within our database; we can often identify these as the major nouns of the system we're modeling. For the purposes of this book we're going to try and keep things simple and draw a direct correlation between an entity and a single table of data; in a real database however, the data for a single entity might be stored in more than one table.

What entities might we define for our SQL Book application?

- Well, we already have a `users` table, and we can think of a**user** as a specific entity within our app; a 'user' is someone who uses our app.
- The purpose of our SQL Book app is to allow users to use books about SQL, so in this context we can think of **books** as an entity within our system.
- One of the things our users can do is to checkout books, so we could have a third entity called **checkouts** that exists between users and books.
- We also want users to be able to leave reviews of books they've read, so we might have another entity called **reviews**.
- Finally, we want to store address information for each user. Since this address data will only be used occasionally and not for every user interaction, we decide to store it in a separate table. We could potentially still think of this address data as part of the 'users' entity, but for now let's think of it as a separate entity called **addresses**.

Now we have defined the entities we need, we can plan tables to store the data for each entity. Those tables might look something like this:



users

id	full_name	enabled	last_login
1	John Smith	f	2017-10-25 10:26:10.015152
2	Alice Walker	t	2017-10-25 10:26:50.295461
3	Harry Potter	t	2017-10-25 10:26:50.295461
5	Jane Smith	t	2017-10-25 10:36:43.324015

checkouts

id	user_id	book_id	checkout_date	return_date
1	1	1	2017-10-15 14:43:18.095143-07	
2	1	2	2017-10-05 16:22:44.593188-07	2017-10-13 13:05:12.673382-05
3	2	2	2017-10-15 11:11:24.994973-07	2017-10-22 17:47:10.407569-07
4	5	3	2017-10-15 09:27:07.215217-07	

books

id	title	author	published_date	isbn
1	My First SQL book	Mary Parker	2012-02-22 12:08:17.320053-03	981483029127
2	My Second SQL book	John Mayer	1972-07-03 09:22:45.050088-07	857300923713
3	My Third SQL book	Cary Flint	2015-10-18 14:05:44.547516-07	523120967812

addresses

user_id	street	city	state
1	1 Market Street	San Francisco	CA
2	2 Elm Street	San Francisco	CA
3	3 Main Street	Boston	MA

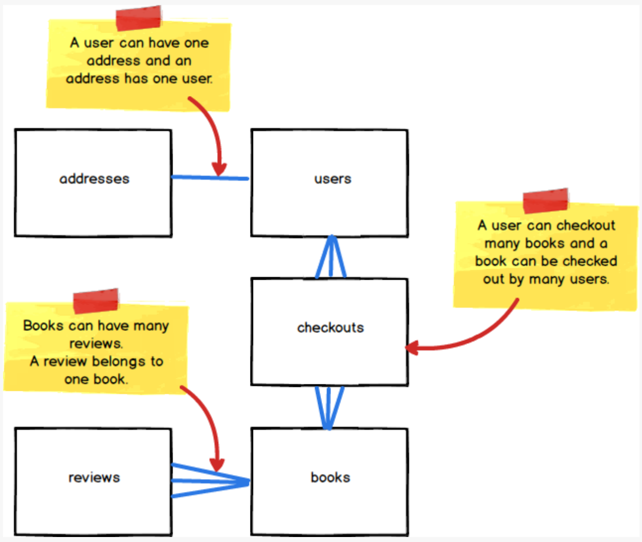
reviews

id	book_id	reviewer_name	content	rating	published_date
1	1	'John Smith'	'My first review'	4	2017-12-10 05:50:11.127281-02
2	2	'John Smith'	'My second review'	5	2017-10-13 15:05:12.673382-05
3	2	'Alice Walker'	'Another review'	1	2017-10-22 23:47:10.407569-07

Relationships

We're making good progress with our database design. We've decided on the entities we want and have formed a picture of the tables we need, the columns in those tables, and even examples of the data that those columns will hold. There's something missing though, and that's the relationships between our entities.

If we look at the diagram of our five tables, the tables are all isolated and it's not obvious how these tables should relate to each other. Let's simplify our tables a bit and explicitly define some relationships between them.


A user can have one address and an address has one user.
addresses
users
A user can checkout many books and a book can be checked out by many users.
Books can have many reviews. A review belongs to one book.
checkouts
reviews
books

This diagram shows an abstract representation of our various entities and also the relationships between them, (note: in reality we could imagine that more than one user might share the same address; this structure is intended for illustration purposes). We can think of this diagram as a simple Entity Relationship Diagram, or **ERD**. An ERD is a graphical representation of entities and their relationships to each other, and is a commonly used tool within database design.

There are different types of ERD varying from conceptual to detailed, and often using specific conventions such as crow's foot notation to model relationships. We won't go into the details of these different types, or the conventions they use, in this book. For now it's useful to simply think of an ERD as any diagram which models relationships between entities.

Keys

Okay, so we now know the tables that we need and we've also defined the relationships that should exist between those tables in our ERD, but how do we actually implement those relationships in terms of our table schema? The answer to that is to use *keys*.

In an earlier section of this book we looked at an aspect of schema called constraints, and explored how constraints act on and work with the data in our database tables. Keys are a special type of constraint used to establish relationships and uniqueness. They can be used to *identify a specific row in the current table*, or to *refer to a specific row in another table*. In this chapter we'll look at two types of keys that fulfill these particular roles: **Primary Keys**, and **Foreign Keys**.

Primary Keys

A necessary part of establishing relationships between two entities or two pieces of data is being able to identify the *data* correctly. In SQL, uniquely identifying data is critical.**A Primary Key is a unique identifier for a row of data**.

In order to act as a unique identifier, a column must contain some data, and that data should be unique to each row. If you're thinking that those requirements sound a lot like our `NOT NULL` and `UNIQUE` constraints, you'd be right; in fact, making a column a `PRIMARY KEY` is essentially equivalent to adding `NOT NULL` and `UNIQUE` constraints to that column.

The `id` column in our `users` table has both of these constraints, and we've used that column in many of our `SELECT` queries in order to uniquely identify rows; we've effectively had `id` as a primary key all along although we haven't explicitly set it as the Primary Key. Let's do that now:

```
ALTER TABLE users ADD PRIMARY KEY (id);
```

Although any column in a table can have `UNIQUE` and `NOT NULL` constraints applied to them, each table can have only one Primary Key. It is common practice for that Primary Key to be a column named `id`. If you look at the other tables we've defined for our database, most of them have an `id` column. While a column of any name can serve as the primary key, using a column named `id` is useful for mnemonic reasons and so is a popular convention.

Being able to uniquely identify a row of data in a table via that table's Primary Key column is only half the story when it comes to creating relationships between tables. The other half of this story is the Primary Key's partner, the Foreign Key.

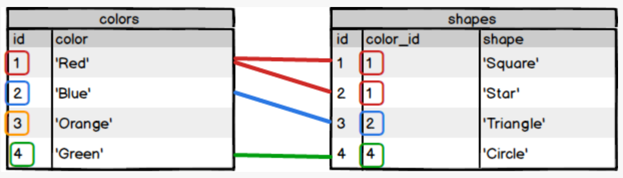
Foreign Keys

A Foreign Key allows us to associate a row in one table to a row in another table. This is done by setting a column in one table as a Foreign Key and having that column reference another table's Primary Key column. Creating this relationship is done using the `REFERENCES` keyword in this form:

```
FOREIGN KEY (fk_col_name)
REFERENCES target_table_name (pk_col_name);
```

We'll explore some specific examples of how this is used when we look at setting up various kinds of relationships later in this chapter, but in general terms you can think of this reference as creating a connection between rows in different tables.

Imagine for instance that we have two tables, one called `colors` and one called `shapes`. The `color_id` column of the `shapes` table is a Foreign Key which references the `id` column of the `colors` table.



colors

id	color
1	'Red'
2	'Blue'
3	'Orange'
4	'Green'

shapes

id	color_id	shape
1	1	'Square'
2	1	'Star'
3	2	'Triangle'
4	4	'Circle'

In the diagram above, the 'Red' row of our `colors` table is associated with the 'Square' and 'Star' rows of our `shapes` table. Similarly, 'Blue' is associated with 'Triangle' and 'Green' with 'Circle'. 'Orange' isn't currently associated with any row in the `shapes` table, but there is the potential to create such an association if we insert a row into `shapes` with a `color_id` of `3`.

By setting up this reference, we're ensuring the*referential integrity* of a relationship. Referential integrity is the assurance that a column value within a record must reference an existing value;

if it doesn't then an error is thrown. In other words, PostgreSQL won't allow you to add a value to the Foreign Key column of a table if the Primary Key column of the table it is referencing does not already contain that value. We'll discuss this concept in a bit more detail later on.

The specific way in which a Foreign Key is used as part of a table's schema depends on the type of relationship we want to define between our tables. In order to implement that schema correctly it is useful to formally describe the relationships we need to model between our entities:

1. A User can have **ONE** address. An address has only **ONE** user.
2. A review can only be about **ONE** Book. A Book can have **MANY** reviews.
3. A User can have **MANY** books that he/she may have checked out or returned. A Book can be/ have been checked out by **MANY** users.

The entity relationships described above can be classified into three relationship types:

- one-to-one
- one-to-many
- many-to-many

Let's look at them each in turn.

one-to-one

A one-to-one relationship between two entities exists when a particular entity instance exists in one table, and it can have only one associated entity instance in another table.

Example: A user can have only one address, and an address belongs to only one user.

> This example is contrived: in the real world, users can have multiple addresses and multiple people can live at the same address.

In the database world, this sort of relationship is implemented like this: the `id` that is the `PRIMARY KEY` of the `users` table is used as both the `FOREIGN KEY` *and* `PRIMARY KEY` of the `addresses` table.

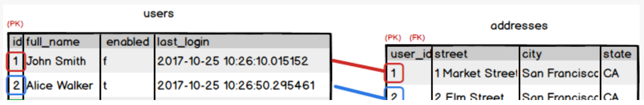
```
/*
one-to-one: User has one address
*/

CREATE TABLE addresses (
  user_id int, -- Both a primary and foreign key
  street varchar(30) NOT NULL,
  city varchar(30) NOT NULL,
  state varchar(30) NOT NULL,
  PRIMARY KEY (user_id),
  FOREIGN KEY (user_id)
      REFERENCES users (id)
      ON DELETE CASCADE
);
```

Executing the above SQL statement will create an `addresses` table, and create a relationship between it and the `users` table. Notice the `PRIMARY KEY` and `FOREIGN KEY` clauses at the end of the `CREATE` statement. These two clauses create the constraints that makes the `user_id` the Primary Key of the `addresses` table and also the Foreign Key for the `users` table.

Let's go ahead and add some data to our table.

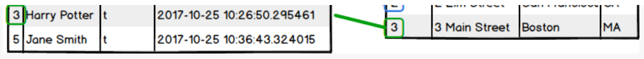
```
INSERT INTO addresses
         (user_id, street, city, state)
  VALUES (1, '1 Market Street', 'San Francisco', 'CA'),
         (2, '2 Elm Street', 'San Francisco', 'CA'),
         (3, '3 Main Street', 'Boston', 'MA');
```

The `user_id` column uses values that exist in the `id` column of the `users` table in order to connect the tables through the foreign key constraint we just created.



id	full_name	enabled	last_login
1	John Smith	f	2017-10-25 10:26:10.015152
2	Alice Walker	t	2017-10-25 10:26:50.295461

user_id	street	city	state
1	1 Market Street	San Francisco	CA
2	2 Elm Street	San Francisco	CA


3 Harry Potter t 2017-10-25 10:26:50.295461
5 Jane Smith t 2017-10-25 10:36:43.324015
3 3 Main Street Boston MA

Referential Integrity

We're going to take a slight detour here to discuss a topic that's extremely important when dealing with table relationships: referential integrity. This is a concept used when discussing relational data which states that table relationships must always be consistent. Different RDBMSes might enforce referential integrity rules differently, but the concept is the same.

The constraints we've defined for our `addresses` table enforce the one-to-one relationship we want between it and our `users` table, whereby a user can only have one address and an address must have one, and only one, user. This is an example of *referential integrity*. Let's demonstrate how this works.

What happens if we try to add another address for a user who already has one?

```
INSERT INTO addresses (user_id, street, city, state)
  VALUES (1, '2 Park Road', 'San Francisco', 'CA');
```

```
ERROR:  duplicate key value violates unique constraint "addresses_pke
DETAIL:  Key (user_id)=(1) already exists.
```

The error above occurs because we are trying to insert a value `1` into the `user_id` column when such a value already exists in that column. The `UNIQUE` constraint on the column prevents us from doing so.

How about if we try to add an address for a user who doesn't exist?

```
INSERT INTO addresses (user_id, street, city, state)
       VALUES (7, '11 Station Road', 'Portland', 'OR');
```

```
ERROR:  insert or update on table "addresses" violates foreign key co
DETAIL:  Key (user_id)=(7) is not present in table "users".
```

Here we get a different error. The `FOREIGN KEY` constraint on the `user_id` column prevents us from adding the value `7` to that column because that value is not present in the `id` column of the `users` table.

If you're wondering why we can add a user without an address but can't add an address without a user, this is due to the *modality* of the relationship between the two entities. Don't worry about exactly what this means for now, just think of it as another aspect of entity relationships.

The ON DELETE clause

You might have noticed in the table creation statement for our `addresses` table, the `FOREIGN KEY` definition included a clause which read `ON DELETE CASCADE`. Adding this clause, and setting it to `CASCADE` basically means that if the row being referenced is deleted, the row referencing it is also deleted. There are alternatives to `CASCADE` such as `SET NULL` or `SET DEFAULT` which instead of deleting the referencing row will set a new value in the appropriate column for that row.

Determining what to do in situations where you delete a row that is referenced by another row is an important design decision, and is part of the concept of maintaining referential integrity. If we don't set such clauses we leave the decision of what to do up to the RDBMS we are using. In the case of PostgreSQL, if we try to delete a row that is being referenced by a row in another table and we have no `ON DELETE` clause for that reference, then an error will be thrown.

One-to-Many

Okay, time to get back to our different table relationship types with a look at one-to-many. A one-to-many relationship exists between two entities if an entity instance in one of the tables can be associated with multiple records (entity instances) in the other table. The opposite relationship does not exist; that is, each entity instance in the second table can only be associated with one entity instance in the first table.

Example: A review belongs to only one book. A book has many reviews.

Let's set up the necessary data. First let's create our tables

```
CREATE TABLE books (
  id serial,
  title varchar(100) NOT NULL,
  author varchar(100) NOT NULL,
  published_date timestamp NOT NULL,
  isbn char(12),
  PRIMARY KEY (id),
  UNIQUE (isbn)
);

/*
 one-to-many: Book has many reviews
*/

CREATE TABLE reviews (
  id serial,
  book_id integer NOT NULL,
  reviewer_name varchar(255),
  content varchar(255),
  rating integer,
  published_date timestamp DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (id),
  FOREIGN KEY (book_id)
      REFERENCES books(id)
      ON DELETE CASCADE
);
```

These table creation statements for our `books` and `reviews` tables are fairly similar to our previous example. There's a key difference worth pointing out in the statement for our `reviews` table however:

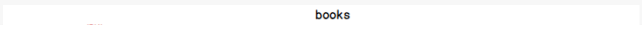
- Unlike our `addresses` table, the `PRIMARY KEY` and `FOREIGN KEY` reference different columns, `id` and `book_id` respectively. This means that the `FOREIGN KEY` column, `book_id` is not bound by the `UNIQUE` constraint of our `PRIMARY KEY` and so the same value from the `id` column of the `books` table can appear in this column more than once. In other words a book can have many reviews.

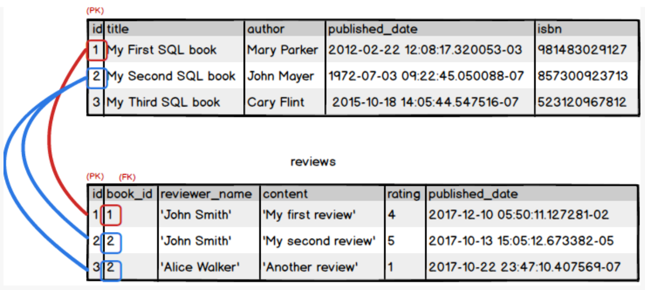
Note that the foreign key `book_id` has a `NOT NULL` constraint. In general, foreign keys on the "many" side of a one-to-many relationship should not allow `NULL`. In this case, it makes no sense to have a review that isn't tied to a book.

Now we have created our `books` and `reviews` tables, let's add some data to them.

```
INSERT INTO books
  (id, title, author, published_date, isbn)
  VALUES
      (1, 'My First SQL Book', 'Mary Parker',
          '2012-02-22 12:08:17.320053-03',
          '981483029127'),
      (2, 'My Second SQL Book', 'John Mayer',
          '1972-07-03 09:22:45.050088-07',
          '857300923713'),
      (3, 'My First SQL Book', 'Cary Flint',
          '2015-10-18 14:05:44.547516-07',
          '523120967812');


INSERT INTO reviews
  (id, book_id, reviewer_name, content, rating,
       published_date)
  VALUES
      (1, 1, 'John Smith', 'My first review', 4,
          '2017-12-10 05:50:11.127281-02'),
      (2, 2, 'John Smith', 'My second review', 5,
          '2017-10-13 15:05:12.673382-05'),
      (3, 2, 'Alice Walker', 'Another review', 1,
          '2017-10-22 23:47:10.407569-07');
```

The order in which we add the data is important here. Since a column in `reviews` references data in `books` we must first ensure that the data exists in the `books` table for us to reference.


books



id	title	author	published_date	isbn
1	My First SQL book	Mary Parker	2012-02-22 12:08:17.320053-03	981483029127
2	My Second SQL book	John Mayer	1972-07-03 09:22:45.050088-07	857300923713
3	My Third SQL book	Cary Flint	2015-10-18 14:05:44.547516-07	523120967812

id	book_id	reviewer_name	content	rating	published_date
1	1	'John Smith'	'My first review'	4	2017-12-10 05:50:11.127281-02
2	2	'John Smith'	'My second review'	5	2017-10-13 15:05:12.673382-05
3	2	'Alice Walker'	'Another review'	1	2017-10-22 23:47:10.407569-07

Just as with the `users` / `addresses` relationship, the `FOREIGN KEY` creates relationships between the `reviews` table and the `books` table. Unlike the `users` / `addresses` relationship however, both books and users can have multiple reviews. For example the `id` value of `2` for `My Second SQL Book` appears twice in the `book_id` column of the `reviews` table.

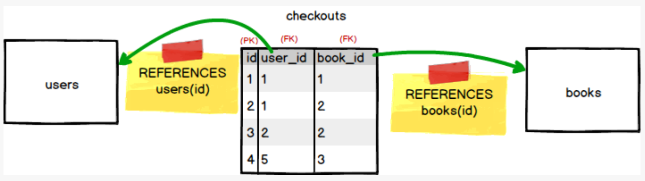
In a real database our `reviews` table would probably also have a Foreign Key reference to the `id` column in `users` table rather than have user type data directly in a `reviewer_name` column. We set up the table in this way for our example because we wanted to focus on the one-to-many relationship type. If we had added such a Foreign Key to `reviews` we'd effectively be setting up a many-to-many relationship between `books` and `users` , which is what we'll look at next.

Many-to-Many

A many-to-many relationship exists between two entities if for one entity instance there may be multiple records in the other table, and vice versa.

Example: A user can check out many books. A book can be checked out by many users (over time).

In order to implement this sort of relationship we need to introduce a third, cross-reference, table. This table holds the relationship between the two entities, by having **two** `FOREIGN KEY` s, each of which references the **PRIMARY KEY** of one of the tables for which we want to create this relationship. We already have our `books` and `users` tables, so we just need to create the cross-reference table: `checkouts` .



id	user_id	book_id
1	1	1
2	1	2
3	2	2
4	5	3

Here, the `user_id` column in `checkouts` references the `id` column in `users` , and the `book_id` column in `checkouts` references the `id` column in `books` . Each row of the `checkouts` table uses these two Foreign Keys to create an association between rows of `users` and `books` .

We can see on the first row of `checkouts` , the user with an `id` of `1` is associated with the book with an `id` of `1` . On the second row, the same user is also associated with the book with an `id` of `2` . On the third row, a different user, with an `id` of `2` is associated with the same book from the previous row. On the fourth row, the user with an `id` of `5` is associated with the book with an `id` of `3` .

Don't worry if you don't completely understand this right away. Shortly, we'll expand on what these associations look like in terms of the data in `users` and `books` . First, though, let's create our `checkouts` table and add some data to it.

```
CREATE TABLE checkouts (
  id serial,
  user_id int NOT NULL,
  book_id int NOT NULL,
  checkout_date timestamp,
  return_date timestamp,
  PRIMARY KEY (id),
  FOREIGN KEY (user_id) REFERENCES users(id)
                        ON DELETE CASCADE,
  FOREIGN KEY (book_id) REFERENCES books(id)
                        ON DELETE CASCADE
);
```

You may have noticed that our table contains a couple of other columns `checkout_date` and `return_date` . While these aren't necessary to create the relationship between the `users` and `books` table, they can provide additional context to that relationship. Attributes like a checkout date or return date don't pertain specifically to users or specifically to books, but to the *association* between a user and a book.

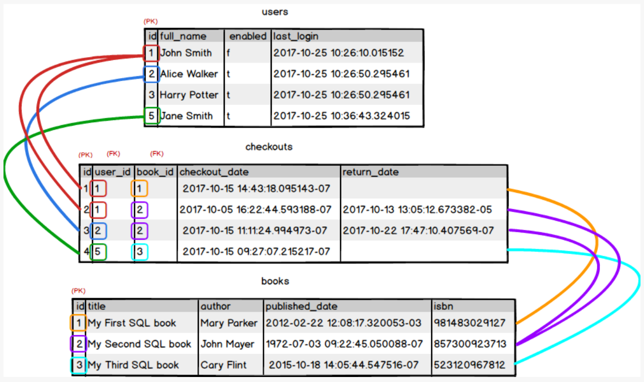
This kind of additional context can be useful within the business logic of the application using our database. For example, in order to prevent more than one user trying to check out the same book at the same time, the app could determine which books are currently checked out by querying those that have a value in the `checkout_date` column of the `checkouts` table but where the `return_date` is set to `NULL`.

As with one-to-many relationships, the foreign keys in many-to-many relationships should not allow `NULL` entries. Thus, both `user_id` and `book_id` have the `NOT NULL` constraint. In this case, it makes no sense to have a checkout row for a user if they haven't checked out a specific book, nor does it make sense to have a row for book that hasn't been checked out by a specific user.

Now that we have our `checkouts` created, we can add the data that will create the associations between the rows in `users` and `books`.

```
INSERT INTO checkouts
  (id, user_id, book_id, checkout_date, return_date)
  VALUES
    (1, 1, 1, '2017-10-15 14:43:18.095143-07',
              NULL),
    (2, 1, 2, '2017-10-05 16:22:44.593188-07',
              '2017-10-13 13:0:12.673382-05'),
    (3, 2, 2, '2017-10-15 11:11:24.994973-07',
              '2017-10-22 17:47:10.407569-07'),
    (4, 5, 3, '2017-10-15 09:27:07.215217-07',
              NULL);
```

Let's have a look at what this data looks like in terms of the relationships between the tables.



users

id	full_name	enabled	last_login
1	John Smith	f	2017-10-25 10:26:10.015152
2	Alice Walker	t	2017-10-25 10:26:50.295461
3	Harry Potter	t	2017-10-25 10:26:50.295461
5	Jane Smith	t	2017-10-25 10:36:43.324015

checkouts

id	user_id	book_id	checkout_date	return_date
1	1	1	2017-10-15 14:43:18.095143-07	
2	1	2	2017-10-05 16:22:44.593188-07	2017-10-13 13:05:12.673382-05
3	2	2	2017-10-15 11:11:24.994973-07	2017-10-22 17:47:10.407569-07
4	5	3	2017-10-15 09:27:07.215217-07	

books

id	title	author	published_date	isbn
1	My First SQL book	Mary Parker	2012-02-22 12:08:17.320053-03	981483029127
2	My Second SQL book	John Mayer	1972-07-03 09:22:45.050088-07	857300923713
3	My Third SQL book	Cary Flint	2015-10-18 14:05:44.547516-07	523120967812

Here we can see that the `id` value of `1` from the `users` table for 'John Smith' appears twice in the `user_id` column of `checkouts`, but alongside different values for `book_id` (`1` and `2`); this satisfies the 'a user can check out many books' part of the relationship. Similarly we can see that `id` value of `2` from the `books` table for 'My Second SQL Book' appears twice in the `books_id` column of `checkouts`, alongside different values for `user_id` (`1` and `2`); this satisfies the 'a book can be checked out by many users' part of the relationship.

We can perhaps think of a many-to-many relationship as combining two one-to-many relationships; in this case between `checkouts` and `users`, and between `checkouts` and `books`.

Summary

In this chapter we covered a number of different topics regarding table relationships:

- We briefly covered normalization, and how this is used to reduce redundancy and improve data integrity within a database.
- ERDs were introduced, and we discussed how these diagrams allow us to model the relationships between different entities.
- We also looked at keys, and how Primary and Foreign keys work together to create the relationships between different tables.
- Finally we looked at some of the different types of relationships that can exist between tables and how to implement these with SQL statements.

To recap, here is a list of common relationships that you'll encounter when working with SQL:

Relationship	Example
one-to-one	**A** User has **ONE** address
one-to-many	**A** Book has **MANY** reviews
many-to-many	**A** user has **MANY** books and a book has **MANY** users

Earlier in this book we looked at how to query data in a database table using `SELECT`. Now that our data is split across multiple tables, how can we structure our queries if we need data from more than one table at the same time? In order to do this, we need to *join* our tables together. In the next chapter we'll explore how to do exactly that by introducing another SQL keyword, `JOIN`.

Exercises

1. Make sure you are connected to the `encyclopedia` database. We want to hold the continent data in a separate table from the country data.

    1. Create a `continents` table with an auto-incrementing `id` column (set as the Primary Key), and a `continent_name` column which can hold the same data as the `continent` column from the `countries` table.

    2. Remove the `continent` column from the `countries` table.

    3. Add a `continent_id` column to the `countries` table of type integer.

    4. Add a Foreign Key constraint to the `continent_id` column which references the `id` field of the `continents` table.

    Hint

    › Solution

2. Write statements to add data to the `countries` and `continents` tables so that the data below is correctly represented across the two tables. Add both the countries and the continents to their respective tables in alphabetical order.

Name	Capital	Population	Continent
France	Paris	67,158,000	Europe
USA	Washington D.C.	325,365,189	North America
Germany	Berlin	82,349,400	Europe
Japan	Tokyo	126,672,000	Asia
Egypt	Cairo	96,308,900	Africa
Brazil	Brasilia	208,385,000	South America

    › Solution

3. Examine the data below:

Album Name	Released	Genre	Label	Singer Name
Born to Run	August 25, 1975	Rock and roll	Columbia	Bruce Springsteen
Purple Rain	June 25, 1984	Pop, R&B, Rock	Warner Bros	Prince
Born in the USA	June 4, 1984	Rock and roll, pop	Columbia	Bruce Springsteen
Madonna	July 27, 1983	Dance-pop, post-disco	Warner Bros	Madonna
True Blue	June 30, 1986	Dance-pop, Pop	Warner Bros	Madonna
Elvis	October 19, 1956	Rock and roll, Rhythm and Blues	RCA Victor	Elvis Presley
Sign o' the Times	March 30, 1987	Pop, R&B, Rock, Funk	Paisley Park, Warner Bros	Prince
G.I. Blues	October 1, 1960	Rock and roll, Pop	RCA Victor	Elvis Presley

    We want to create an `albums` table to hold all the above data except the singer name, and create a reference from the `albums` table to the `singers` table to link each album to the correct singer. Write the necessary SQL statements to do this and to populate the table with data. Assume Album Name, Genre, and Label can hold strings up to 100 characters. Include an auto-incrementing `id` column in the `albums` table.

    Hint

    › Solution

4. Connect to the `ls_burger` database. If you run a simple `SELECT` query to retrieve all the data from the `orders` table, you will see it is very unnormalised. We have repetition of item names and costs and of customer data.

```
SELECT * FROM orders;
```

```
 id | customer_name  |          burger          |    side     |
----+----------------+--------------------------+-------------+---
  3 | Natasha O'Shea | LS Double Deluxe Burger | Onion Rings | Ch
  2 | Natasha O'Shea | LS Cheeseburger          | Fries       |
  1 | James Bergman  | LS Chicken Burger        | Fries       | Le
  4 | Aaron Muller   | LS Burger                | Fries       |
(4 rows)
```

We want to break this table up into multiple tables. First of all create a `customers` table to hold the customer name data and an `email_addresses` table to hold the customer email data. Create a one-to-one relationship between them, ensuring that if a customer record is deleted so is the equivalent email address record. Populate the tables with the appropriate data from the current `orders` table.

Solution

5. We want to make our ordering system more flexible, so that customers can order any combination of burgers, sides and drinks. The first step towards doing this is to put all our product data into a separate table called `products`. Create a table and populate it with the following data:

Product Name	Product Cost	Product Type	Product Loyalty Points
LS Burger	3.00	Burger	10
LS Cheeseburger	3.50	Burger	15
LS Chicken Burger	4.50	Burger	20
LS Double Deluxe Burger	6.00	Burger	30
Fries	1.20	Side	3
Onion Rings	1.50	Side	5
Cola	1.50	Drink	5
Lemonade	1.50	Drink	5
Vanilla Shake	2.00	Drink	7
Chocolate Shake	2.00	Drink	7
Strawberry Shake	2.00	Drink	7

The table should also have an auto-incrementing `id` column which acts as its `PRIMARY KEY`. The `product_type` column should hold strings of up to 20 characters. Other than that, the column types should be the same as their equivalent columns from the `orders` table.

Solution

6. To associate customers with products, we need to do two more things:

   1. Alter or replace the `orders` table so that we can associate a customer with one or more orders (we also want to record an order status in this table).

   2. Create an `order_items` table so that an order can have one or more products associated with it.

Based on the order descriptions below, amend and create the tables as necessary and populate them with the appropriate data.

James has one order, consisting of a Chicken Burger, Fries, Onion Rings, and a Lemonade. It has a status of 'In Progress'.

Natasha has two orders. The first consists of a Cheeseburger, Fries, and a Cola, and has a status of 'Placed'; the second consists of a Double Deluxe Burger, a Cheeseburger, two sets of Fries, Onion Rings, a Chocolate Shake and a Vanilla Shake, and has a status of 'Complete'.

Aaron has one order, consisting of an LS Burger and Fries. It has a status of 'Placed'.

Assume that the `order_status` field of the `orders` table can hold strings of up to 20 characters.

› **Solution**

NEXT: SQL JOINS ›