**26. Back-End and Feature Testing**

**The Testing Pyramid**

**Most full-stack web applications include the following layers:**

* **A view that appears in a user’s web browser**
* **A server that handles HTTP requests**
* **A database that stores information about user interactions**

**Throughout this article, we will use the following example to illustrate the testing needs in each layer of a full stack web application.**

**Imagine you are developing a movie blog and want to build a feature that allows users to comment on your posts. Any comment can be read by any other user on the website.**

**This is how a user may interact with your web application:**

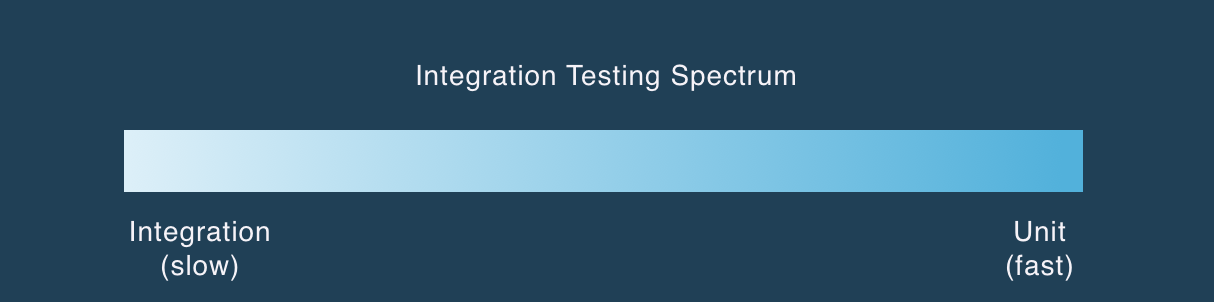
* **The browser renders comments below the blog post.**
* **When a user wants to post a comment, they enter text into a box and click the Submit button.**
* **Your web application transmits the comment to your server.**
* **Your server checks if the POST has any errors. If it does not, it creates a new comment and saves it to the web application’s database.**
* **The next user to arrive will see the last comment at the top of the list of comments.**

**In the next section, we will begin to learn about the types of tests we can run against the full-stack web application above.**

#### **Integration Tests and Unit Tests**

**We call some web applications “full stack,” because we think of the view, server, and model/database as separate layers — each one sitting on top of and relying on the one below it. This structure impacts the types of tests that we can run against our web application.**

**Developers think of tests as fitting into a spectrum. The spectrum represents the amount of the web application that a test exercises.**

**tests are isolated and fast tests that check one small behavior within your web application.**

**For example, we want to test whether our database can save a comment. Saving does not involve the view or server. We can create a test that writes directly to the database, and the test itself doesn’t need to know about the other layers.**

**A test like this is fast, but only gives you confidence that a small slice of your system is working as intended.**

**System tests System tests are a group of fully integrated tests that exercise your entire web application.**

**For example, we want to test whether our blog renders with the correct post and comments. We can write a test that checks whether a browser renders a stored blog post. This test exercises every layer of the web application:**

* **The database stores the blog post.**
* **The server sends the HTML to the browser.**
* **The browser renders the view.**

**A test like this is slow and less descriptive but provides you with confidence that a large slice of your system is working as intended. Integration tests Integration tests include everything between unit tests and system tests. They exercise multiple parts of the web application, often in different layers.**

**For example, an integration test may check whether your web application can save a server-generated comment to the database. This test integrates two layers of your web application:**

* **The server receives the comment and sends it to the database.**
* **The database stores your comment.**

**Developers often call tests like the one above end-to-end tests, because they start in the browser (one end) and traverse the stack to the database (other end).**

**Integration tests are faster than system tests, but slower than unit tests. They provide less confidence (that your feature works as expected) than system tests and more confidence than unit tests.**

#### **Shape of a testing suite**

**The goal of a full-stack web application’s testing suite is to provide you with confidence that your application works as expected while executing in a timely fashion.**

**How would you use integration and unit tests to accomplish this?**

* **You could write few integration tests that provide you with confidence, and more unit tests that execute quickly and provide you with specific feedback.**

**The number and types of unit and integration tests that you write can be mapped onto the testing pyramid.**

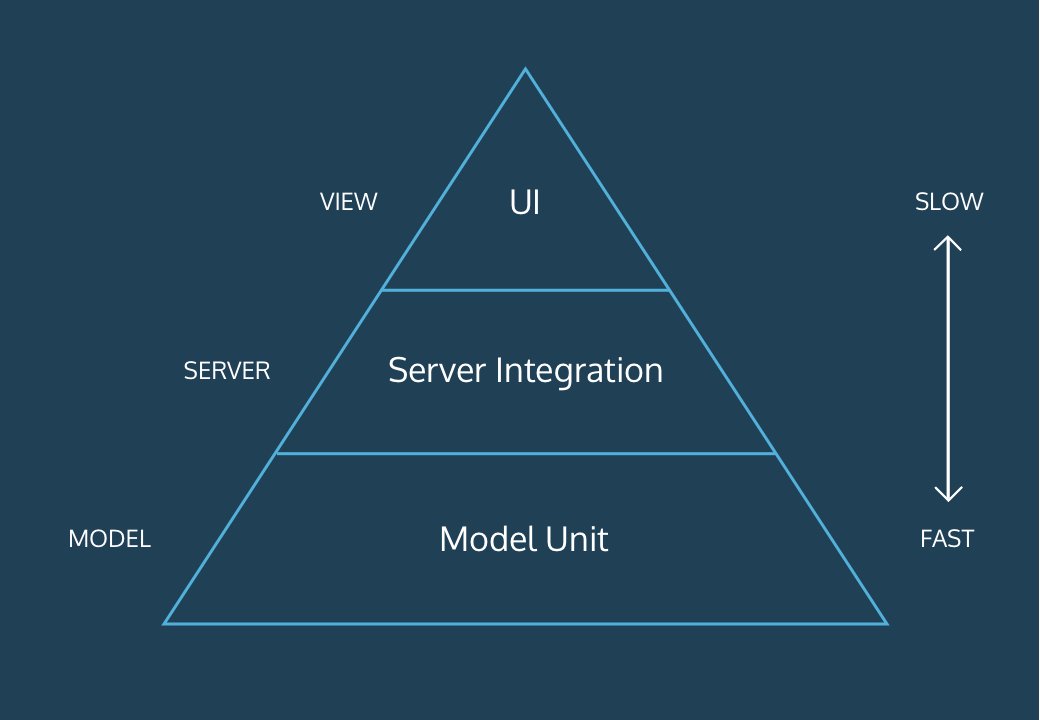
#### **The Testing Pyramid**

**The testing pyramid is an approach to structuring your test suite.**

**Browser-level integration tests sit on the top of the pyramid. This layer is the narrowest because it should have the fewest number of tests — the slow nature of browser-level tests make them more expensive than server-level tests.**

**While server tests don’t need to interact with the browser, they usually exercise parts of the model or database. They sit close to the middle of the spectrum between unit and system tests. They provide a moderate level of confidence as they may exercise multiple layers of the stack. Server tests are more expensive and provide less specific error messages than model tests.**

**Compared to browser and server tests, model and database tests exercise a smaller portion of the stack. They provide the most specific feedback, but not much confidence that the whole system is working as expected. Because they are the cheapest, you can write a lot of them without significantly slowing the amount of time it takes to run your test suite.**

****

#### **Example Testing Suite**

**Most sites that support user commenting set a limit on the number of comments that load when you arrive on the page. If an article has 300 comments, rendering the entire list would take too long and make the web page difficult to navigate.**

**Imagine that your web application already has a feature-level integration test that checks if the browser renders comments to the site.**

**Let’s consider the most efficient set of tests we could write to check that the browser renders only the last ten comments.**

**Because you already have a test at the feature layer, you decide to start by writing a server test. The test will check if:**

* **Calling a Comments.latest method returns a list of comments from your database that is ten items or less.**

**The server and feature level tests provide you with confidence that your web application returns comments that don’t exceed the maximum length. However, they don’t provide any information about which comments the model layer returns.**

**Once, you have the confidence that your model layer returns ten comments or less from your database, you can check the specifics of the Comments.latest method. You can check if:**

* **A database with more than ten comments returns only ten.**
* **A database with less than ten comments returns all of the comments.**
* **A database with zero comments returns zero comments.**
* **The list of comments is in reverse chronological order.**

**To support a ten-comment limit, we added one server test and four model tests. Notice, our tests provide greater detail as we get closer to the database. The tradeoff is that each test at the model layer provides us with less confidence that the system is working as expected.**

#### **Conclusion**

**While the testing pyramid is an approach to optimize your test suite, it’s important to think critically about the tests you write at each layer.**

**When making decisions about how to test a feature, you should ask yourself a few of these questions:**

* **Is a feature-level integration test necessary?**
* **Can I test the same behavior with server and model layer tests?**
* **How much confidence do I have with the server and model layer tests?**
* **How long does the feature test take? Will that impact my team’s workflow?**

**Outside-in test-driven development doesn’t answer these questions, but provides you with an approach to finding an answer. It helps you avoid the decision paralysis that often slows development teams, leads to extended deadlines, and incomplete implementation.**

**In addition to the efficiencies discussed above, the outside-in approach is a satisfying way to develop a web application with a full test suite.**

**At the end of this article, you will know how to approach development of features in a full-stack web application using outside-in test-driven development.**

#### **Red, Green, Refactor**

**Test-driven development (TDD) is the process of writing tests before implementation code. You use the feedback from your tests to inform the implementation of a feature or outcome.**

**A common approach to TDD is the red, green, refactor cycle. When you write a test before the implementation exists you start “in the red” phase, because your test fails and outputs a red error message. Next, you write the minimum implementation code to get your test to pass. This puts you “in the green” phase, because your test passes and outputs a green message.**

**Once you are in the green, you should consider whether your implementation is the best or most efficient approach. If you think your code could be written more efficiently or cleaner, then you enter the refactor phase. You can refactor your code with confidence, because you have tests that cover the expected behavior.**

#### **Outside-in TDD**

**Outside-in TDD is an approach that developers use to build full-stack web applications. It leverages the same red, green, refactor steps that we covered above, but with one caveat — a failing test does not always inform you to write new implementation code. Instead, it may require that you implement new functionality at a different level.**

**You start at the top of the stack, the view, and write tests as you work your way towards the database layer.**

**If a test pushes you to a lower level, you restart your red, green, refactor cycle by writing a new test. This test informs the implementation at your new layer. You continue the TDD cycle at this lower level until:**

* **You need to drop another layer to implement the desired behavior**
* **You have addressed the reason for dropping to the current layer**

**Once you address the reason for dropping a layer, you can start working your way back up the testing pyramid. If you’re in the model/database layer, you step up to the server, and run your server tests to see if you get a different response. The response should be one of the following:**

* **The test passes — you can start another red, green, refactor cycle at the server level or step up to the view layer.**
* **The test fails — the server test that pushed you to the model layer fails, but for a different reason. This is common, and indicates that you’re making progress. This failure may indicate that you need to write additional implementation at the server level, or drop back to the model.**

#### **Outside-in Example**

**We’re going to use the following as an example of how to develop a new feature with outside-in TDD: You have a movie blog and want to develop a feature that renders user comments under your blog posts. The application should render no more than ten comments when a user lands on the web page. The application should present the comments in reverse chronological order (i.e. the most recent comment should be first).**

**Let’s assume the web application generates HTML at the server — any updates to the view require implementation at the server level.**

##### **Feature Testing**

**The first step is to write a feature test that checks if your web application is rendering comments to the browser. Let’s use the following outside-in TDD approach:**

1. **Write a test that checks for the presence of a comment under a blog post.**
2. **The test fails, because your web application does not render comments.**
3. **Because your web application generates HTML at the server layer, you drop to the server to address the error.**

**Although we could continue to write feature tests to check for the number of rendered comments, we know server tests are cheaper, so we can test those details when we drop a layer.**

##### **Server Testing**

**At the server layer, we start by writing a test that informs the implementation of our server-generated HTML. Because our web application renders unique comments from the database, we want to check that the server-generated HTML is dynamic.**

1. **Write a test that checks for the presence of a dynamically generated comment element in the server HTML.**
2. **The test fails, so we add implementation for a server-generated comment.**
3. **Once we’re in the green and consider refactoring, we want to write a test that calls a method at the model layer, let’s call it Comment.latest(). At the server layer, we’ll check if the method returns comments from the database.**
4. **Because this method doesn’t exist, we must drop to the model/database layer.**

##### **Model and Database Testing**

**At the model layer, we start by writing a test that informs the implementation of our Comment.latest method. This method requires that you interface with the web application’s database.**

1. **Write a test that checks if the Comment.latest method returns ten comments when the database has eleven comments.**
2. **Implement the Comment.latest method to return ten comments, so the test is green.**
3. **Once you’ve considered refactoring, write a test that checks whether the method returns the last ten comments in reverse chronological order.**
4. **Implement and refactor**
5. **Write a test that checks if Comment.latest() returns an empty array when your database is empty.**
6. **Implement and refactor**
7. **Write a test that checks if Comment.latest returns the correct number and order of comments when your database has between zero and ten comments in it.**
8. **Implement and refactor**

#### **Taking Stock**

**At this point, your entire test suite should be green. You have written seven new tests, and the implementation code to make them pass — your web application should render the last ten comments from your database in reverse chronological order.**

**Let’s take stock of our seven new tests:**

1. **Feature: Comments are rendered to a user’s browser.**
2. **Server: The server generates an HTML field for comments.**
3. **Server: The server has access to ten comments from the database.**
4. **Model: The Comment.latest method returns ten comments from your database.**
5. **Model: The Comment.latest method returns the last ten comments in your database in reverse chronological order.**
6. **Model: The Comment.latest method returns an empty array when your database has zero comments.**
7. **Model: The Comment.latest method returns all of the comments when your database has between zero and ten comments.**

**Once your feature is working as expected, you should consider how your new tests fit into the broader test suite. The rest of the test suite could have few tests, or over one hundred. It’s time to refactor.**

#### **Refactoring Your Test Suite**

**The way you approach refactoring will vary based on the size and types of tests in your suite. One guiding light in refactoring is to optimize the suite for confidence and speed. Because we used TDD to implement our comment feature, we should feel confident that our comments are working as expected, and the feature is fully covered.**

**Consider the questions below when deciding how to refactor your suite:**

* **How much longer does it take to run my test suite with these new tests?**
* **Is the additional amount of time that your test suite takes to run acceptable?**
* **Is there overlap between any of my new tests?**
* **Is there overlap between my new tests and existing tests?**

**Let’s take a moment to consider a few of these questions in the context of our test suite.**

**How much longer does it take to run my test suite with these new tests?**

* **You can calculate this value by running your test suite before and after writing the new tests, and calculate the difference. Seven new tests, like the ones above may only add a few seconds to your suite. Let’s use our next question to think about how you can evaluate what an acceptable amount of time may be.**

**Is the additional amount of time that your test suite takes to run acceptable?**

* **Although a few seconds may seem acceptable, this time can add up as your suite grows. Even if you’re comfortable with the additional time, you should always consider whether you can make speed improvements that don’t impact confidence.**

**Is there overlap between any of my new tests?**

* **You should consider if any new tests, especially in the feature or server level, can be deleted without impacting your confidence that the comments feature works as expected. For example, our first server test checks if the server generates an HTML field for comments. Your feature-level test checks the same functionality — it also takes longer, but provides a higher level of confidence. We decide to delete the server test for reasons we will investigate when we consider our next question.**

**Is there overlap between my new tests and existing tests?**

* **Next, you should look outside your seven (now six) new tests to consider the coverage offered by the other tests in your suite. Often, your test suite will have a feature test that checks whether the web page renders as expected — this is usually good enough coverage for most new features. Given the cost of feature-level tests, and the coverage of your lower-level tests, it often makes sense to delete the new feature-level test.**

# Introduction to headless browser testing

**Which of the following is NOT something you can do with headless testing?**

**Well done! Headless testing does not involve the browser or GUIs of any sort. Investigate content of webpage in your browser.**

**What is one of the main advantages of headless testing?**

**That’s right! Because headless testing does not have the extra overhead of starting up a browser, you can run tests much more quickly. It has good performance and its typically faster than real browsers.**

**TDD FEATURE-LEVEL TESTS: Feature Test Toolbelt**

**Chai: Chai is an assertion library for Node.js and browsers that can be paired with any JavaScript testing framework.**

**const {assert} = require('chai');**

**PhantomJS:** [**PhantomJS**](http://phantomjs.org/)**is a headless browser scriptable with a JavaScript API, which allows us to write tests that mimic user interaction and then evaluate the results. It does not require us to render the application in a browser.**

**A browser runs “headless” when it doesn’t render anything to the screen, but runs in the background.**

### **WebdriverI/O**

[**WebdriverIO**](https://webdriver.io/docs/what-is-webdriverio)**provides methods that allow us to programmatically interact with the user-facing elements of our app in the headless browser that PhantomJS runs.**

### **Toolbelt High-Level Summary**

**Phantom allows us to run an instance of a headless browser so you can run tests that mimic user interaction with a web application. WebdriverIO provides the methods to interact with browser values programmatically. We can make assertions against these tests using the Chai assertion library.**

**Feature Test I: The first feature test we want to write is to check our application’s empty state. The functionality we want to test is:**

* **When a user visits the homepage, the poems section is empty**

**The testing suite for our poetry app would begin with nested describe blocks like this:**

* **describe('Poetry web app', () => {  
    describe('user visits root', () => {  
     
    });  
  });**

**Next, we add an it block to describe the behavior we want to test in our app:**

**describe('Poetry web app', () => {  
  describe('user visits root', () => {  
    it('page starts blank', () => {  
   
    });    
  });  
});**

**The Plumbing: Next, we reach for our feature testing toolbelt. We start by to using the global browser variable that is provided by WebdriverI/O.**

**The browser variable is powerful because it gives us access to the browser that Phantom is running in the background. We can simulate a user interacting with our website by calling different methods on the global browser variable in our test suite.**

**For example, we can use browser.url() to simulate a user visiting the home page of our application, which is the first behavior we want to test.**

**The .url method navigates to the URL that is passed to it as an argument. The following line of code would navigate to the Codecademy website in the Phantom browser.**

**browser.url('https://www.codecademy.com')**

**In the case of our poetry web app, we will pass in '/' as the argument, which will point the browser to the root file of our project, which in this case is our index.html.**

**The code would look like this:**

**describe('poetry web app', () => {  
  describe('user visits root', () => {  
    it('page starts blank', () => {  
      browser.url('/');  
    });    
  });  
});**

**1.**

**Imagine you are a developer working on a project that includes creating a web application with a message feature.**

**The first feature you want to test in your web app is that no messages appear on the page when a user visits the project root.**

* **Inside the describe blocks in user-visits-root-test.js file, write an it block with the string: starts blank, and an empty callback function.**

**2.Inside the it block, call the .url() method on the global browser variable and pass '/' as the argument.**

**const {assert} = require('chai');**

**describe('User visits root', () => {**

**describe('without existing messages', () => {**

**it('starts blank', () => {**

**browser.url('/')**

**});**

**});**

**});**

**Feature Test I: Assert: We want to make sure our app is in an empty state.**

**We can write a test for this behavior by deciding that poems will be listed in an HTML element with an id attribute set to poems. Then, write an assert statement to verify that the element with the ID poems is empty.**

**We can do this using the Chai assert.equal method, which evaluates if the two arguments are equal.**

**In the case of our poetry app, the assert statement would look like this:**

**assert.equal(browser.getText('#poems'), '')**

**Because we will render the poetry onto the page as text, we can evaluate the contents of the HTML element as a string.**

**The .getText method, from WebdriverI/O, gets the text content from the selected DOM element.**

**Here we are using browser.getText() to evaluate if the text in the element with the ID poems is equal to an empty string.**

**Our final code for this feature test would look like this:**

**describe('User visits root', () => {  
  describe('without existing poems', () => {  
    it('page starts blank', () => {  
      browser.url('/');  
   
      assert.equal(browser.getText('#poems'), '');  
    });  
  });  
});**

### **Instructions**

**1.**

**Use assert.equal() to evaluate if an element with ID messages:**

**<section id="messages"></section>**

**has no text in it.**

**Then run your test suite to celebrate being in the red!**

**Solution:user-visits-root-test.js**

**const {assert} = require('chai');**

**describe('User visits root', () => {**

**describe('without existing messages', () => {**

**it('starts blank', () => {**

**browser.url('/');**

**assert.equal(browser.getText('#messages'), '');**

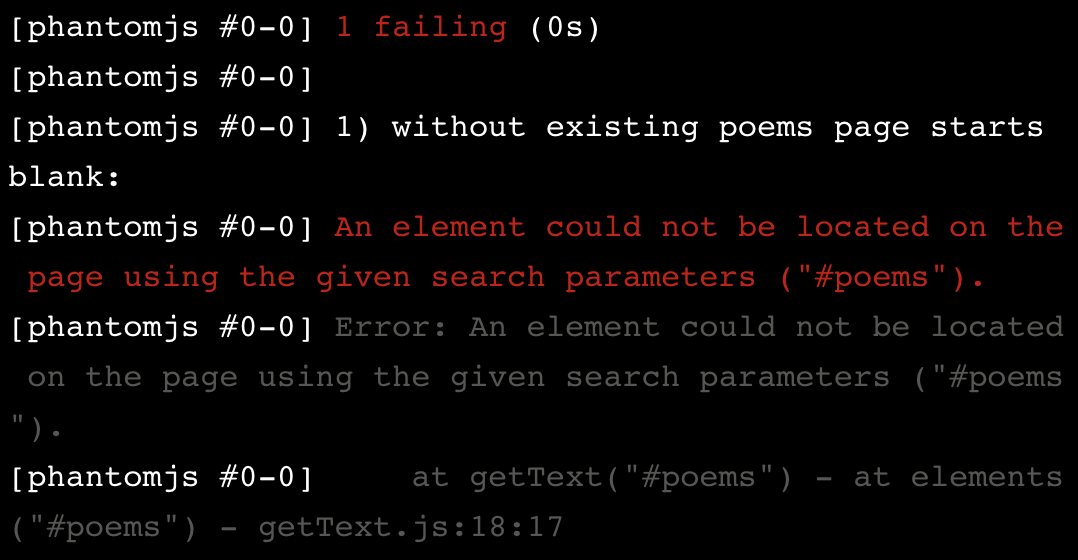
**});**

**});**

**});**

**Feature Test I: Passing**

**Considering our poetry web app example, if we ran our test suite we would get an error message that included:**

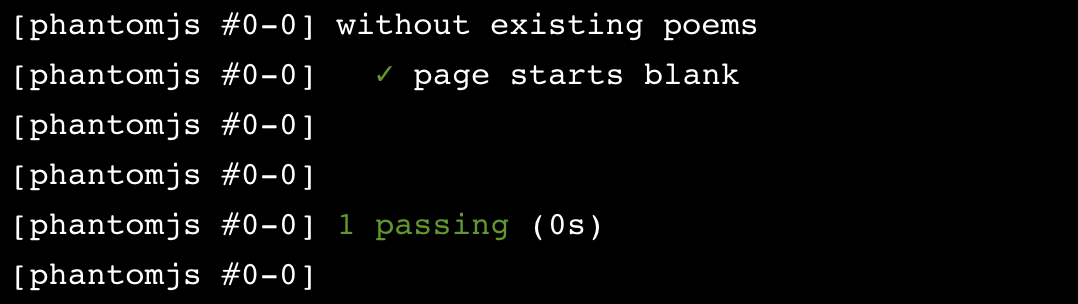
****

**The error message describes the issue in terms of HTML elements and tells us that the element we are expecting does not exist on our page. This is because we have not yet created the HTML in our index.html file.**

**Using a strict TDD approach, we would write just enough HTML code to make our test pass. Let’s do that now.**

**<section id="poems"></section>**

**When we run our tests we get a message confirming that it is passing.**

****

**We have written our first feature test and moved from the red to the green using a TDD approach.**

**We expected this test to pass because we haven’t created any poems as part of our test’s setup. Each test is discrete and isolated from the other parts of the project, so we know that the list of poems will be empty.**

### **Instructions**

**1.**

**Run your test suite in the terminal.**

**Go to index.html and create a <section> </section> element that responds to the error message you received when running your test suite.**

**Run your test suite again in the terminal to make sure you are in the green.**

**Solution:**

**const {assert} = require('chai');**

**describe('User visits root', () => {**

**describe('without existing messages', () => {**

**it('starts blank', () => {**

**browser.url('/');**

**assert.equal(browser.getText('#messages'),'');**

**});**

**});**

**});**

**Index.html**

**<section id="messages"></section>**

**Feature Test II: Setup: Now that we are in the green we should take a moment to consider refactoring our code.**

#### **Feature Test II**

**Returning to our poetry app demo, we want to write a test to check if the application saves the title and text of a user’s poem when they press the submit button.**

**The functionality we want to test is:**

1. **The user enters text into a text input element (the poem)**
2. **The user enters text into a second text input element (the title of the poem)**
3. **The user presses a submit button**

**Adding the describe and it blocks for this second test would look like this:**

**describe('demo poetry web app', () => {   
    it('saves the user poem and title', () => {  
   
    });  
  });**

**Next, we want to write the setup, exercise, and verification phases of our test.**

**In the setup phase for this test, we create variables to represent a user’s input to the title and poem fields on the home page.**

**const title = 'Words Birth Worlds';  
     const poem = 'Our words are marvelous weapons with which we could behead the sun';**

**The second test would now look like this:**

**describe('demo poetry web app', () => {   
    it('saves the user poem and title', () => {  
      const title = 'Words Birth Worlds';  
      const poem = 'Our words are marvelous weapons with which we could behead the sun';  
    });  
  });**

### **Instructions**

**1.**

**Go to your test suite for the message app in the user-visits-root-test.js file. Notice that we have added the describe and it blocks for your second test. Inside those blocks:**

* **Create a const variable named message to hold the expected value of the message input, and set it equal to a string with the value:**

**feature tests often hit every level of the TDD Testing Pyramid**

* **Below message, create a const variable named author to hold the expected value of the author input, and set it equal to username.**

**Solution: user-visits-root-test.js**

**const {assert} = require('chai');**

**describe('User visits root', () => {**

**describe('without existing messages', () => {**

**it('starts blank', () => {**

**browser.url('/');**

**assert.equal(browser.getText('#messages'),'');**

**});**

**});**

**describe('posting a message', () => {**

**it('saves the message with the author information', () => {**

**const message = "feature tests often hit every level of the TDD Testing Pyramid";**

**const author = "username";**

**});**

**});**

**});**

**Index.html**

**<section id = "messages">**

**</section>**

**Feature Test II: Exercise: First, we will set the URL of the browser to go to the root of our project using the .url method:**

**browser.url('/');**

**Next, we will use the .setValue method, which sends a sequence of keystrokes to an element, based on a string argument.**

**We will use .setValue() to mimic a user entering the title and poem into the corresponding HTML input elements at the root of our web app.**

**The first argument passed to .setValue() is the CSS selector that references an HTML element, and the second argument is the value you want to assign that element.**

**browser.setValue('input[id=title]', title);  
browser.setValue('textarea[id=poem]', poem);**

**In the example above, a text input with the ID of title will be set to a value of title. Also, the textarea with ID poem will be set to the value poem. The variables referenced here are the ones we created in the setup phase.**

**To complete the exercise phase of our test we would use the .click method to mimic a user clicking on a submit button.**

**browser.click('input[type=submit]');**

**Our second test, with the setup and exercise phases, now looks like this:**

**describe('demo poetry web app', () => {   
    it('saves the user poem and title', () => {  
      // Setup  
      const title = 'Words Birth Worlds';  
      const poem = 'Our words are marvelous weapons with which we could behead the sun';  
     // Exercise  
      browser.url('/');  
      browser.setValue('input[id=title]', title);  
      browser.setValue('textarea[id=poem]', poem);  
      browser.click('input[type=submit]');  
    });  
  });**

### **Instructions**

**1.**

**Write the exercise phase of your second feature test for the message web app.**

**Use the browser.url() method to set the URL of the headless browser to your project root.**

**Checkpoint 2 Passed**

**Stuck? Get a hint**

**2.**

**Use browser.setValue() to set the value of an HTML <input> element with the ID author to the value of your variable named author.**

**Under the line you just wrote, use browser.setValue() to set the value of an HTML <textarea> element with the ID message to the value of your variable named message.**

**Checkpoint 3 Passed**

**3.**

**Use browser.click() to simulate a user clicking on an HTML <input> with the type submit.**

**Solution: user-visits-root-test.js**

**const {assert} = require('chai');**

**describe('User visits root', () => {**

**describe('without existing messages', () => {**

**it('starts blank', () => {**

**browser.url('/');**

**assert.equal(browser.getText('#messages'),'');**

**});**

**});**

**describe('posting a message', () => {**

**it('saves the message with the author information', () => {**

**const message ='feature tests often hit every level of the TDD Testing Pyramid';**

**const author = 'username';**

**browser.url("/")**

**browser.setValue('input[id=author]', author);**

**browser.setValue('textarea[id=message]', message);**

**browser.click('input[type=submit]');**

**});**

**});**

**});**

**Indx.html**

**<section id = "messages">**

**</section>**

**Feature Test II: Verify: We have created that element already to make our first feature test pass. It is the following line of code in our index.html file:**

**<section id="poems">  
</section>**

**To add an assert statement to evaluate the behavior of our feature, we will use the browser variable, and .getText() to return the text contents of the element, with the id poem.**

**The Chai Assertion Library allows us to use the .include method to check if the string that is returned from .getText() includes the substrings of the title and poem that the user has submitted:**

**assert.include(browser.getText('#poems'), title);  
assert.include(browser.getText('#poems'), poem);**

**In both assert statements the first argument we pass to .include() is the function we created above it.**

**The .include() method works like this:**

**assert.include(haystack, needle)**

**The full second test would now look like this:**

**const {assert} = require('chai');  
   
describe('User visits root', () => {  
   
  describe('demo poetry web app', () => {   
    it('saves the user poem and title', () => {  
      // Setup  
      const title = 'Words Birth Worlds';  
      const poem = 'Our words are marvelous weapons with which we could behead the sun;  
     // Exercise  
      browser.url('/');  
      browser.setValue('input[id=title]', title);  
      browser.setValue('textarea[id=poem]', poem);  
      browser.click('input[type=submit]');  
      // Verify  
      assert.include(browser.getText('#poems'), title);  
      assert.include(browser.getText('#poems'), poem);  
    });  
  });  
});**

### **Instructions**

**1.**

**Verify your results.**

* **Write an assert statement that verifies that the value of your variable author is included in the results of your .getText function.**
* **Write an assert statement that verifies that the value of your variable message is included in the results of your .getText function.**

**Solution: user-visits-root-test.js**

**const {assert} = require('chai');**

**describe('User visits root', () => {**

**describe('without existing messages', () => {**

**it('starts blank', () => {**

**browser.url('/');**

**assert.equal(browser.getText('#messages'),'');**

**});**

**});**

**describe('posting a message', () => {**

**it('saves the message with the author information', () => {**

**//setup**

**const message ='feature tests often hit every level of the TDD Testing Pyramid';**

**const author = 'username';**

**//exercise**

**browser.url('/');**

**browser.setValue('input[id=author]', author);**

**browser.setValue('textarea[id=message]', message);**

**browser.click('input[type=submit]');**

**//verify**

**assert.include( browser.getText('#messages'), message);**

**assert.include( browser.getText('#messages'), author);**

**});**

**});**

**});**

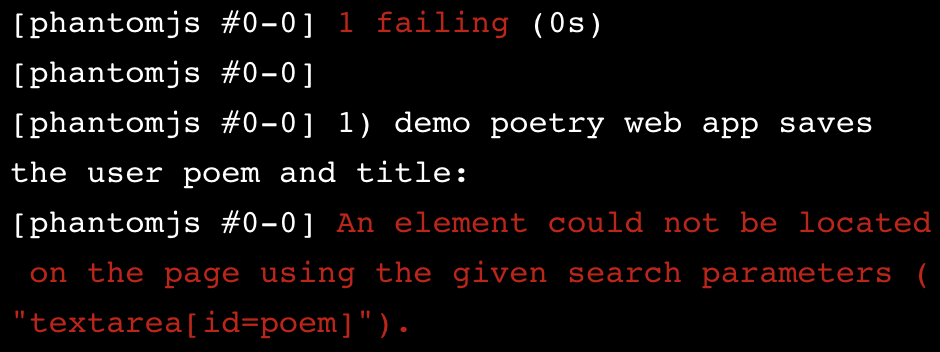
**Index.html**

**<section id = "messages">**

**</section>**

**Stuck In The Red**

**Now when we run our test, we will get a step further and receive an error message that tells us the next line of HTML code we need to write:**

****

**This error message tells us we are missing a <textarea> element with the ID, poem. We can address this by adding the following to our index.html:**

**<label for="poem">Your poem:</label>  
<textarea id="poem"></textarea>**

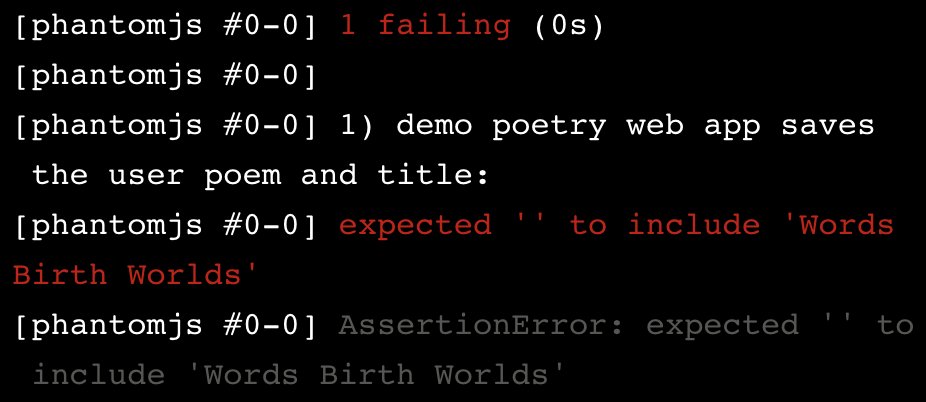
**Running the test again would give us a similar error message concerning the input element with the type equal to submit. This is the submit button referenced in our test code, and we can address this error by adding the following code to our index.html file:**

**<input type="submit">**

**The complete index.html file would now look like this:**

**<section id="poems">  
</section>  
   
<label for="title">Title</label>  
<input id="title">  
   
<label for="poem">Your poem:</label>  
<textarea id="poem"></textarea>  
   
<input type="submit">**

**Running the test suite now would give us an error message like this:**

****

**While this error message looks similar to the ones we have been seeing, it is a different type of error message, and it signals the need for a shift in our TDD process.**

**What’s different here is that the failure comes from the verification phase instead of the exercise phase. While this isn’t always the case, that means that we’ve changed the implementation code enough to get to the part of the test where we’re specifying behavior, not just the existence of elements.**

**The kind of test we need to write in response to this error will force us to drop levels in the TDD Testing Pyramid.**

**Question: Run your test and follow the error messages to address one issue at a time, until you receive an error concerning the verification phase of your test. That error that will force you to drop down to a server level test.**

* **Create an <input> element with the ID, author.**
* **Check your work**
* **Run your test suite**

**Stuck? Get a hint**

**2.**

**Create a <textarea> element with the ID, message,**

* **Check your work**
* **Run your test suite**

**3.**

**Create an <input> element with the type, submit.**

* **Check your work**
* **Run your test suite**

**Solution: index.html**

**<section id="messages"></section>**

**<label for="author">Your name:</label>**

**<input id="author">**

**<label for="message">Your message:</label>**

**<textarea id="message"></textarea>**

**<input type="submit">**

**User-visits-root-test.js**

**const {assert} = require('chai');**

**describe('User visits root', () => {**

**describe('without existing messages', () => {**

**it('starts blank', () => {**

**browser.url('/');**

**assert.equal(browser.getText('#messages'),'');**

**});**

**});**

**describe('posting a message', () => {**

**it('saves the message with the author information', () => {**

**const author = 'user name';**

**const message ='feature testing with TDD makes me feel empowered to create a better workflow';**

**browser.url('/');**

**browser.setValue('input[id=author]', author);**

**browser.setValue('textarea[id=message]', message);**

**browser.click('input[type=submit]');**

**assert.include(browser.getText('#messages'), message);**

**assert.include(browser.getText('#messages'), author);**

**});**

**});**

**});**

**Reviews**

* **When developing a new feature and practicing outside-in development, feature tests are where we’ll typically start.**
* **Feature tests often incorporate every layer of the application and — using WebDriverI/O and Mocha — exercise features in the same way that a human user would. They’re a good tool for reproducing end-user behavior.**
* **WebDriverI/O is a Node package that interacts with a “headless” instance of PhantomJS.**
* **Feedback from feature tests is usually in terms of HTML (i.e. that text or button that you said would be on the page isn’t on the page).**
* **Because feature tests typically hit every layer of a developer’s stack, they are slower than tests at lower layers, and errors thrown in feature tests can be difficult to interpret and provide little guidance on what the developer can do to resolve them.**
* **Their value, however, is in developer confidence that the software functions as expected.**

**SERVER TESTING STACK**

**Testing Framework: Chai: to test if an array foo includes an element bar using Mocha with the built-in Node assertion library, we use the JavaScript includes helper:**

**assert.ok(foo.includes(bar));**

**To improve the readability and flow of our tests, we extend the built-in Node assertion library with Chai.**

**const {assert} = require('chai');**

**The main function in Chai we are using is .include(). This allows us to rewrite the previous example as:**

**assert.include(foo, bar);**

**Include also works to check that text contains certain values:**

**assert.include('foobar', 'bar'); // Evaluates to true**

**The large set of assertion methods in the chai library enable us to write more expressive tests that are easy for developers to understand.**

### **Instructions**

**1.**

**In chai-test.js to the right, we’ve included Chai at the top of the file and set up a describe block with Mocha. Use Chai on line 9 to assert that the foo array contains the number 4. Use npm test to verify the test is passing.**

**When you are ready to move on, check your work.**

**Checkpoint 2 Passed**

**Stuck? Get a hint**

**2.**

**In JavaScript, the .pop() method removes the final element from an array and returns it. Write an assertion to verify that the variable fooPop returned from the .pop() method returns the correct element from the array. Use npm test to verify the test is passing.**

**When you are ready to move on, check your work.**

**Checkpoint 3 Passed**

**Stuck? Get a hint**

**3.**

**Since we popped the only element from the array, foo should be empty. To check this, assert that the length of foo is now zero. Use npm test to verify the test is passing.**

**When you are ready to move on, check your work.**

**Solution: const {assert} = require('chai');**

**describe('Array', () => {**

**describe('.pop()', () => {**

**it('should return a value and remove the element from the array', () => {**

**// setup**

**const foo = [4];**

**const includedNumber = 4;**

**// check setup**

**assert.include(foo, includedNumber);**

**// exercise**

**const fooPop = foo.pop();**

**assert.equal(fooPop, includedNumber);**

**// asserts**

**assert.equal(foo.length, 0);**

**});**

**});**

**});**

**Testing HTML Responses**

**We can use the jsdom library to improve this type of assertion. It allows us to select elements of the DOM and check relationships and content. To increase the readability of our tests, we abstracted the jsdom functionality into a custom function, parseTextFromHTML:**

**const parseTextFromHTML = (htmlAsString, selector) => {  
  const selectedElement = jsdom(htmlAsString).querySelector(selector);  
  if (selectedElement !== null) {  
    return selectedElement.textContent;  
  } else {  
    throw new Error(`No element with selector ${selector} found in HTML string`);  
  }  
};**

**This function takes the HTML response as a string and the desired selector as inputs and returns the textContent of the corresponding element. If no element is found, it will return a TypeError.**

### **Instructions**

**1.**

**In the panel to the right, jsdom-test.js is prepopulated with code to test that the string “Hello” is contained within the HTML response. Run the test by typing npm test in the terminal and observe it pass.**

**When you are ready to move on, check your work.**

**Checkpoint 2 Passed**

**Stuck? Get a hint**

**2.**

**Change the existing assertion to use parseTextFromHTML and assert that the string “Hello” is contained in the #bar element. The first argument should be the HTML string, foo and the second argument should be the selector, '#bar'. Run the test using npm test.**

**When you are ready to move on, check your work.**

**Checkpoint 3 Passed**

**Stuck? Get a hint**

**3.**

**Modify the HTML string foo to include the string “Hello” in the #bar element (in addition to the ‘#buzz’ element) to pass the test. Run the test using npm test to verify it passes.**

**When you are ready to move on, check your work.**

**Solution : jsdom.js**

**Solution: const {assert} = require('chai');**

**const {jsdom} = require('jsdom');**

**const parseTextFromHTML = (htmlAsString, selector) => {**

**const selectedElement = jsdom(htmlAsString).querySelector(selector);**

**if (selectedElement !== null) {**

**return selectedElement.textContent;**

**} else {**

**throw new Error(`No element with selector ${selector} found in HTML string`);**

**}**

**};**

**describe('HTML tests', () => {**

**describe('#bar', () => {**

**it('should include the string "Hello"', () => {**

**// setup**

**const foo = '<html><div id="bar">Hello</div><div id="buzz">Hello</div><html>';**

**//asserts**

**assert.include(parseTextFromHTML(foo, '#bar'), 'Hello');**

**});**

**});**

**});**

**Async / Await**

**The async/await pattern introduced in Node 8 helps us write readable descriptions of the behavior of our application which is an important part of writing good tests.**

**To use this pattern, define the function with the async keyword. Then, within the function, use the await keyword in front of the asynchronous function you are calling. For example:**

**const foo = async () => {  
  console.log(await someAsyncThing());  
  return true;  
}  
   
foo();**

**Here, we are waiting for someAsyncThing() to return before logging the result to the console.**

### **Instructions**

**1.**

**In index-test.js to the right, there is the start of a server test on the root document of our site. There are no assertions yet, but we are attempting to log the server response to the console. Run the test as is and note that we see an “undefined” response logged to the console. (The request method is covered in the next exercise)**

**When you are ready to move on, check your work.**

**Checkpoint 2 Passed**

**Stuck? Get a hint**

**2.**

**Update the function to use async in the function definition and await for the call to request. Run the tests again using npm test and note the logged response in the console.**

**When you are ready to move on, check your work.**

**Solution: index-test.js**

**const request = require('supertest');**

**const app = require('../../app');**

**describe('the homepage', () => {**

**it('returns the correct content', async () => {**

**const response = await request(app)**

**.get('/')**

**.send();**

**console.log(response.text);**

**});**

**});**

**SuperTest: you may have noticed in the previous exercise, we are using the function request to make server calls to support our tests. This is actually a reference to the SuperTest library:**

**const request = require('supertest');**

**This library was specifically designed for testing Node server responses and integrates well with Mocha and Chai. To use SuperTest, we pass the app object from our app into the request function. To make a GET request, we use .get() with the desired route as the argument:**

**await request(app)  
          .get('/')  
          .send();**

**It is also possible to perform a POST using SuperTest. We chain any desired properties or inputs to the HTTP call, and use .send() to make the request:**

**await request(app)  
          .post('/messages')  
          .type('form')  
          .send({author, message});**

### **Instructions**

**1.**

**In the pane to the right, there is the start to a server test on the root document of our site. Chain the .get() method at the end of the request. Pass the appropriate argument to get the root object of our app ('/').**

**Run npm test to verify the server response is being printed to the console.**

**When you are ready to move on, check your work.**

**Solution: index-test.js**

**const request = require('supertest');**

**const app = require('../../app');**

**describe('the homepage', () => {**

**it('returns the correct content', async () => {**

**const response = await request(app).**

**get('/');**

**console.log(response.text);**

**});**

**});**

**Summary**

**In this lesson we covered a set of technologies used for testing a Node server. These included:**

* **Chai - a library for extending the built in Node assertion library**
* **jsdom - a library for interacting and testing the DOM returned by the server (this functionality is encapsulated in our parseTextFromHTML helper function).**
* **async / await - a pattern for making asynchronous code more readable**
* **SuperTest - a library for making Node server requests and testing their responses**

### **Instructions**

**1.**

**In the pane to the right there is a test that makes use of all the technologies. Run the test using npm test and verify it passes. In the next lesson, we’ll use these concepts to further explore testing a Node server.**

**When you are ready to move on, check your work.**

**Solution: index.html**

**const express = require('express');**

**const router = express.Router();**

**router.get('/', async (req, res) => {**

**res.send('<h1 id="page-title">My Page</h1>');**

**});**

**module.exports = router;**

**index-test.js**

**const {assert} = require('chai');**

**const request = require('supertest');**

**const {jsdom} = require('jsdom');**

**const app = require('../../app');**

**const parseTextFromHTML = (htmlAsString, selector) => {**

**const selectedElement = jsdom(htmlAsString).querySelector(selector);**

**if (selectedElement !== null) {**

**return selectedElement.textContent;**

**} else {**

**throw new Error(`No element with selector ${selector} found in HTML string`);**

**}**

**};**

**describe('the homepage', () => {**

**it('the #page-title element contains the page title', async () => {**

**const pageTitle = 'My Page';**

**const response = await request(app).**

**get('/').**

**send();**

**assert.include(parseTextFromHTML(response.text, '#page-title'), pageTitle);**

**});**

**});**

**CHAPTER: 27 SQL for Back-End Development**

# What is a Relational Database Management System?

#### **What is a Database?**

**A database is a set of data stored in a computer. This data is usually structured in a way that makes the data easily accessible.**

#### **What is a Relational Database?**

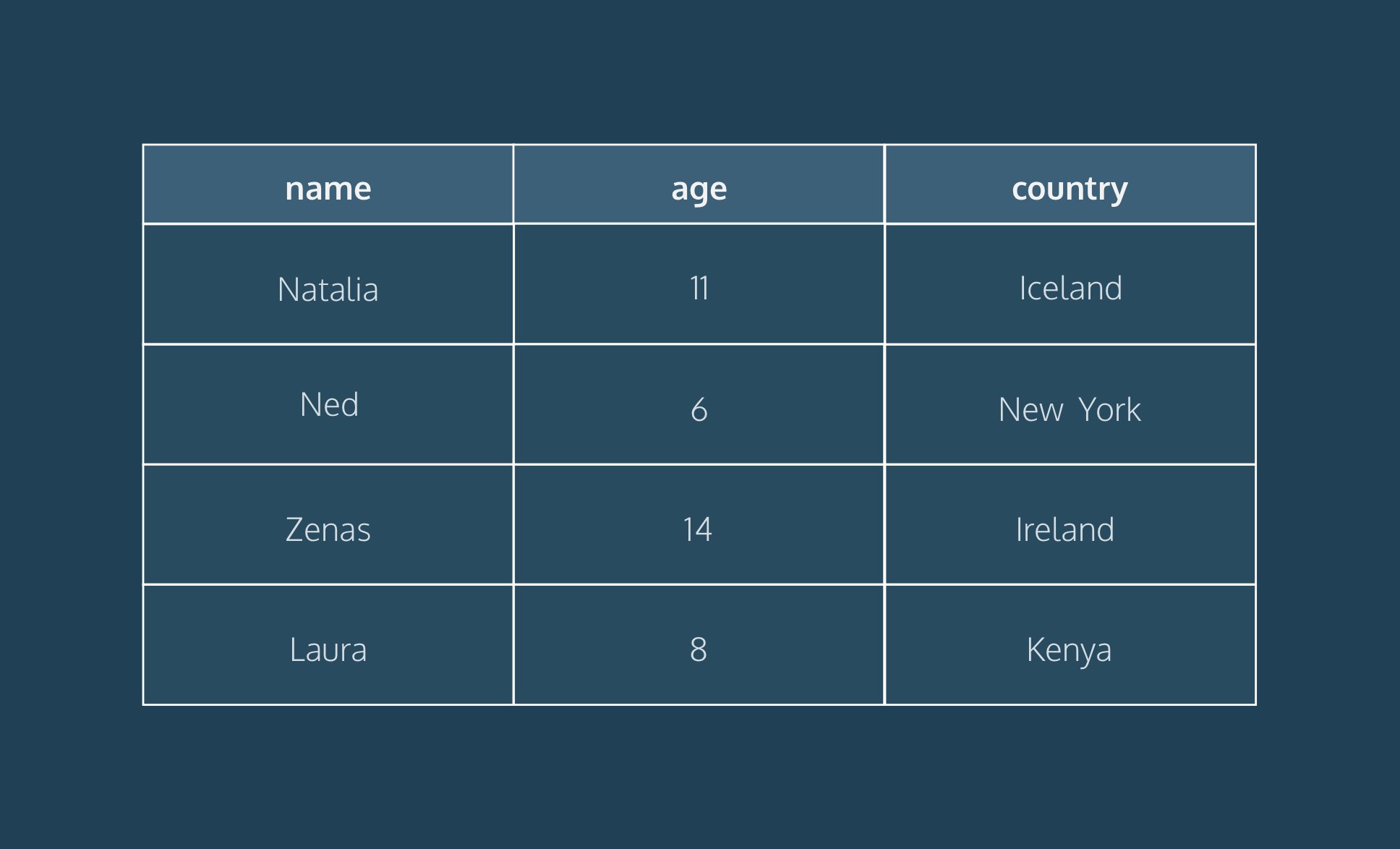
**A relational database is a type of database. It uses a structure that allows us to identify and access data in relation to another piece of data in the database. Often, data in a relational database is organized into tables.**

#### **Tables: Rows and Columns**

**Tables can have hundreds, thousands, sometimes even millions of rows of data. These rows are often called records.**

**Tables can also have many columns of data. Columns are labeled with a descriptive name (say, age for example) and have a specific data type.**

**For example, a column called age may have a type of INTEGER (denoting the type of data it is meant to hold).**

****

**In the table above, there are three columns (name, age, and country).**

**The name and country columns store string data types, whereas age stores integer data types. The set of columns and data types make up the schema of this table.**

**The table also has four rows, or records, in it (one each for Natalia, Ned, Zenas, and Laura).**

#### **What is a Relational Database Management System (RDBMS)?**

**A relational database management system (RDBMS) is a program that allows you to create, update, and administer a relational database. Most relational database management systems use the SQL language to access the database.**

#### **What is SQL?**

**SQL (Structured Query Language) is a programming language used to communicate with data stored in a relational database management system. SQL syntax is similar to the English language, which makes it relatively easy to write, read, and interpret.**

**Many RDBMSs use SQL (and variations of SQL) to access the data in tables. For example, SQLite is a relational database management system. SQLite contains a minimal set of SQL commands (which are the same across all RDBMSs). Other RDBMSs may use other variants.**

**(SQL is often pronounced in one of two ways. You can pronounce it by speaking each letter individually like “S-Q-L”, or pronounce it using the word “sequel”.)**

#### **Popular Relational Database Management Systems**

**SQL syntax may differ slightly depending on which RDBMS you are using. Here is a brief description of popular RDBMSs:**

[**MySQL**](https://www.mysql.com/)

**MySQL is the most popular open source SQL database. It is typically used for web application development, and often accessed using PHP.**

**The main advantages of MySQL are that it is easy to use, inexpensive, reliable (has been around since 1995), and has a large community of developers who can help answer questions.**

**Some of the disadvantages are that it has been known to suffer from poor performance when scaling, open source development has lagged since Oracle has taken control of MySQL, and it does not include some advanced features that developers may be used to.**

[**PostgreSQL**](https://www.postgresql.org/)

**PostgreSQL is an open source SQL database that is not controlled by any corporation. It is typically used for web application development.**

**PostgreSQL shares many of the same advantages of MySQL. It is easy to use, inexpensive, reliable and has a large community of developers. It also provides some additional features such as foreign key support without requiring complex configuration.**

**The main disadvantage of PostgreSQL is that it can be slower in performance than other databases such as MySQL. It is also slightly less popular than MySQL.**

**For more information about PostgreSQL including installation instructions, read**[**this**](https://www.codecademy.com/paths/design-databases-with-postgresql/tracks/what-is-a-database/modules/using-postgresql-on-your-own-computer/articles/installing-and-using-postgresql-locally)**article.**

[**Oracle DB**](https://www.oracle.com/database/)

**Oracle Corporation owns Oracle Database, and the code is not open sourced.**

**Oracle DB is for large applications, particularly in the banking industry. Most of the world’s top banks run Oracle applications because Oracle offers a powerful combination of technology and comprehensive, pre-integrated business applications, including essential functionality built specifically for banks.**

**The main disadvantage of using Oracle is that it is not free to use like its open source competitors and can be quite expensive.**

[**SQL Server**](https://www.microsoft.com/en-us/sql-server/sql-server-2017)

**Microsoft owns SQL Server. Like Oracle DB, the code is close sourced.**

**Large enterprise applications mostly use SQL Server.**

**Microsoft offers a free entry-level version called Express but can become very expensive as you scale your application.**

[**SQLite**](https://www.sqlite.org/)

**SQLite is a popular open source SQL database. It can store an entire database in a single file. One of the most significant advantages this provides is that all of the data can be stored locally without having to connect your database to a server.**

**SQLite is a popular choice for databases in cellphones, PDAs, MP3 players, set-top boxes, and other electronic gadgets. The SQL courses on Codecademy use SQLite.**

**For more info on SQLite, including installation instructions, read**[**this**](https://www.codecademy.com/courses/learn-sql/articles/what-is-sqlite)**article.**

#### **Using An RDBMS On Codecademy**

**On Codecademy, we use both SQLite and PostgreSQL. While this may sound confusing, don’t worry! We want to stress that the basic syntax you will learn can be used in both systems. For example, the syntax to create tables, insert data into those tables, and retrieve data from those tables are all identical. That’s one of the nice parts of learning SQL — by learning the fundamentals with one RDBMS, you can easily begin work in another.**

**That being said, let’s take a look at some of the more subtle details:**

* **File extensions — when working with databases on Codecademy, take a look at the name of the file you’re writing in. If your file ends in .sqlite, you’re using a SQLite database. If your file ends in .sql, you’re working with PostgreSQL.**
* **Data types — You’ll learn about data types very early into learning a RDBMS. One thing to note is that SQLite and PostgreSQL have slightly different data types. For example, if you want to store text in a SQLite database, you’ll use the TEXT data type. If you’re working with PostgreSQL, you have many more options. You could use varchar(n), char(n), or text. Each type has its own subtle differences. This is a good example of PostgreSQL being slightly more robust than SQLite, but the core concepts remaining the same.**
* **Built-in tables — As you work your way through more complicated lessons on databases, you’ll start to learn how to access built-in tables. For example, if you take our lesson on indexes, you’ll learn how to look at the table that the system automatically creates to keep track of what indexes exist. Depending on which RDBMS system you are using (in that lesson we’re using PostgreSQL), the syntax for doing that will be different. Any time you’re writing SQL about the database itself, rather than the data, that syntax will likely be unique to the RDBMS you’re using.**

#### **Conclusion**

**Relational databases store data in tables. Tables can grow large and have a multitude of columns and records. Relational database management systems (RDBMSs) use SQL (and variants of SQL) to manage the data in these large tables. The RDBMS you use is your choice and depends on the complexity of your application.**

**MANIPULATION: Relational Databases**

**SELECT \* FROM celebs;**

**A relational database is a database that organizes information into one or more tables. Here, the relational database contains one table.**

**A table is a collection of data organized into rows and columns. Tables are sometimes referred to as relations. Here the table is celebs.**

**A column is a set of data values of a particular type. Here, id, name, and age are the columns.**

**A row is a single record in a table. The first row in the celebs table has:**

* **An id of 1**
* **A name of Justin Bieber**
* **An age of 22**

**All data stored in a relational database is of a certain data type. Some of the most common data types are:**

* **INTEGER, a positive or negative whole number**
* **TEXT, a text string**
* **DATE, the date formatted as YYYY-MM-DD**
* **REAL, a decimal value**

**Statements**

**The code below is a SQL statement. A statement is text that the database recognizes as a valid command. Statements always end in a semicolon ;.**

**CREATE TABLE table\_name (  
   column\_1 data\_type,   
   column\_2 data\_type,   
   column\_3 data\_type  
);**

**Let’s break down the components of a statement:**

1. **CREATE TABLE is a clause. Clauses perform specific tasks in SQL. By convention, clauses are written in capital letters. Clauses can also be referred to as commands.**
2. **table\_name refers to the name of the table that the command is applied to.**
3. **(column\_1 data\_type, column\_2 data\_type, column\_3 data\_type) is a parameter. A parameter is a list of columns, data types, or values that are passed to a clause as an argument. Here, the parameter is a list of column names and the associated data type.**

**Question: 1.**

**Let’s take a closer look at the statement we wrote before. In the code editor, type:**

**SELECT \* FROM celebs;**

**Run the code to observe the results, and ask yourself:**

* **Which parts of the statement are the clauses?**
* **What table are we applying the command to?**

**So in the terminal we received a table:**

### **Query Results**

|  |  |  |
| --- | --- | --- |
| **id** | **name** | **age** |
| **1** | **Justin Bieber** | **22** |
| **2** | **Beyonce Knowles** | **33** |
| **3** | **Jeremy Lin** | **26** |
| **4** | **Taylor Swift** | **26** |

### **Database Schema**

### **celebs**

|  |  |
| --- | --- |
| **name** | **type** |
| **id** | **INTEGER** |
| **name** | **TEXT** |
| **age** | **INTEGER** |

|  |
| --- |
| **Rows: 4** |

**Create**

**CREATE statements allow us to create a new table in the database. You can use the CREATE statement anytime you want to create a new table from scratch. The statement below creates a new table named celebs.**

**CREATE TABLE celebs (  
   id INTEGER,   
   name TEXT,   
   age INTEGER  
);**

**1. CREATE TABLE is a clause that tells SQL you want to create a new table.  
2. celebs is the name of the table.  
3. (id INTEGER, name TEXT, age INTEGER) is a list of parameters defining each column, or attribute in the table and its data type:**

* **id is the first column in the table. It stores values of data type INTEGER**
* **name is the second column in the table. It stores values of data type TEXT**
* **age is the third column in the table. It stores values of data type INTEGER**

### **Instructions**

**1.**

**Now that you have a good understanding of SQL syntax, we can create a new table. We’ve cleared the database from the previous exercises. Confirm no celebs table exists by entering the following in the code editor:**

**SELECT \* FROM celebs;**

**Look at the results. The database should be empty!**

**Checkpoint 2 Passed**

**2.**

**Now that we know the database is empty, let’s create a new celebs table.**

**In the code editor, type:**

**CREATE TABLE celebs (  
   id INTEGER,   
   name TEXT,   
   age INTEGER  
);**

**Insert**

**The INSERT statement inserts a new row into a table.**

**We can use the INSERT statement when you want to add new records. The statement below enters a record for Justin Bieber into the celebs table.**

**INSERT INTO celebs (id, name, age)   
VALUES (1, 'Justin Bieber', 22);**

* **INSERT INTO is a clause that adds the specified row or rows.**
* **celebs is the table the row is added to.**
* **(id, name, age) is a parameter identifying the columns that data will be inserted into.**
* **VALUES is a clause that indicates the data being inserted.**
* **(1, 'Justin Bieber', 22) is a parameter identifying the values being inserted.**
  + **1: an integer that will be added to id column**
  + **'Justin Bieber': text that will be added to name column**
  + **22: an integer that will be added to age column**

### **Instructions**

**1.**

**Add a row to the table. In the code editor, type:**

**INSERT INTO celebs (id, name, age)   
VALUES (1, 'Justin Bieber', 22);**

**Look at the Database Schema. How many rows are in the celebs table?**

**Checkpoint 2 Passed**

**Stuck? Get a hint**

**2.**

**Add three more celebrities to the table. Beneath your previous INSERT statement type:**

**INSERT INTO celebs (id, name, age)   
VALUES (2, 'Beyonce Knowles', 33);   
   
INSERT INTO celebs (id, name, age)   
VALUES (3, 'Jeremy Lin', 26);   
   
INSERT INTO celebs (id, name, age)   
VALUES (4, 'Taylor Swift', 26);**

**Solution:**

**INSERT INTO celebs (id, name, age)**

**VALUES (1, 'Justin Bieber', 22);**

**INSERT INTO celebs (id, name, age)**

**VALUES (2, 'Beyonce Knowles', 33);**

**INSERT INTO celebs (id, name, age)**

**VALUES (3, 'Jeremy Lin', 26);**

**INSERT INTO celebs (id, name, age)**

**VALUES (4, 'Taylor Swift', 26);**

**Now table is come on terminal:**

### **Query Results**

|  |
| --- |
|  |

### **Database Schema**

### **celebs**

|  |  |
| --- | --- |
| **name** | **type** |
| **id** | **INTEGER** |
| **name** | **TEXT** |
| **age** | **INTEGER** |

|  |
| --- |
| **Rows: 4** |

**Select**

**SELECT statements are used to fetch data from a database. In the statement below, SELECT returns all data in the name column of the celebs table.**

**SELECT name FROM celebs;**

**1. SELECT is a clause that indicates that the statement is a query. You will use SELECT every time you query data from a database.  
2. name specifies the column to query data from.  
3. FROM celebs specifies the name of the table to query data from. In this statement, data is queried from the celebs table.**

**You can also query data from all columns in a table with SELECT.**

**SELECT \* FROM celebs;**

**\* is a special wildcard character that we have been using. It allows you to select every column in a table without having to name each one individually. Here, the result set contains every column in the celebs table.**

**SELECT statements always return a new table called the result set.**

### **Instructions**

**1.**

**Let’s take a closer look at SELECT and retrieve all the names in the celebs table. In the code editor, type:**

**SELECT name FROM celebs;**

**Solution: SELECT name FROM celebs;**

**Stuck? Get a hint**

**2.**

**Delete your previous SELECT statement from the code editor.**

**To SELECT all the data in the celeb table, enter the following statement in the code editor using the \* wildcard character:**

**SELECT \* FROM celebs;**

**Same as in question:**

**Alter**

**The ALTER TABLE statement adds a new column to a table. You can use this command when you want to add columns to a table. The statement below adds a new column twitter\_handle to the celebs table.**

**ALTER TABLE celebs   
ADD COLUMN twitter\_handle TEXT;**

**1. ALTER TABLE is a clause that lets you make the specified changes.  
2. celebs is the name of the table that is being changed.  
3. ADD COLUMN is a clause that lets you add a new column to a table:**

* **twitter\_handle is the name of the new column being added**
* **TEXT is the data type for the new column**

**4. NULL is a special value in SQL that represents missing or unknown data. Here, the rows that existed before the column was added have NULL (∅) values for twitter\_handle.**

### **Instructions**

**1.**

**Add a new column to the table. In the code editor, type:**

**ALTER TABLE celebs   
ADD COLUMN twitter\_handle TEXT;   
   
SELECT \* FROM celebs;**

**Solution:**

**ALTER TABLE celebs**

**ADD COLUMN twitter\_handle TEXT;**

**SELECT \* FROM celebs;**

**Table present in terminal**

### **Query Results**

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **name** | **age** | **twitter\_handle** |
| **1** | **Justin Bieber** | **22** | **Null** |
| **2** | **Beyonce Knowles** | **33** | **Null** |
| **3** | **Jeremy Lin** | **26** | **Null** |
| **4** | **Taylor Swift** | **26** | **Null** |

### **Database Schema**

### **celebs**

|  |  |
| --- | --- |
| **name** | **Type** |
| **id** | **INTEGER** |
| **name** | **TEXT** |
| **age** | **INTEGER** |
| **twitter\_handle** | **TEXT** |

|  |
| --- |
| **Rows: 4** |

**Update**

**The UPDATE statement edits a row in a table. You can use the UPDATE statement when you want to change existing records. The statement below updates the record with an id value of 4 to have the twitter\_handle @taylorswift13.**

**UPDATE celebs   
SET twitter\_handle = '@taylorswift13'   
WHERE id = 4;**

**1. UPDATE is a clause that edits a row in the table.  
2. celebs is the name of the table.  
3. SET is a clause that indicates the column to edit.**

* **twitter\_handle is the name of the column that is going to be updated**
* **@taylorswift13 is the new value that is going to be inserted into the twitter\_handle column.**

**4. WHERE is a clause that indicates which row(s) to update with the new column value. Here the row with a 4 in the id column is the row that will have the twitter\_handle updated to @taylorswift13.**

### **Instructions**

**1.**

**Update the table to include Taylor Swift’s**[**twitter handle**](https://twitter.com/taylorswift13)**. In the code editor, type:**

**UPDATE celebs   
SET twitter\_handle = '@taylorswift13'   
WHERE id = 4;   
   
SELECT \* FROM celebs;**

**solution:**

**UPDATE celebs**

**SET twitter\_handle = '@taylorswift13'**

**WHERE id = 4;**

**SELECT \* FROM celebs;**

### **Query Results**

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **name** | **age** | **twitter\_handle** |
| **1** | **Justin Bieber** | **22** | **Null** |
| **2** | **Beyonce Knowles** | **33** | **Null** |
| **3** | **Jeremy Lin** | **26** | **Null** |
| **4** | **Taylor Swift** | **26** | **@taylorswift13** |

### **Database Schema**

### **celebs**

|  |  |
| --- | --- |
| **name** | **Type** |
| **id** | **INTEGER** |
| **name** | **TEXT** |
| **age** | **INTEGER** |
| **twitter\_handle** | **TEXT** |

|  |
| --- |
| **Rows: 4** |

**Delete**

**The DELETE FROM statement deletes one or more rows from a table. You can use the statement when you want to delete existing records. The statement below deletes all records in the celeb table with no twitter\_handle:**

**DELETE FROM celebs   
WHERE twitter\_handle IS NULL;**

1. **DELETE FROM is a clause that lets you delete rows from a table.**
2. **celebs is the name of the table we want to delete rows from.**
3. **WHERE is a clause that lets you select which rows you want to delete. Here we want to delete all of the rows where the twitter\_handle column IS NULL.**
4. **IS NULL is a condition in SQL that returns true when the value is NULL and false otherwise.**

### **Instructions**

**1.**

**Delete all of the rows that have a NULL value in the twitter handle column. In the code editor, type the following:**

**DELETE FROM celebs   
WHERE twitter\_handle IS NULL;  
   
SELECT \* FROM celebs;**

**How many rows exist in the celebs table now?**

**Solution:**

**DELETE FROM celebs**

**WHERE twitter\_handle IS NULL;**

**SELECT \* FROM celebs;**

**Table present in terminal:**

### **Query Results**

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **name** | **age** | **twitter\_handle** |
| **4** | **Taylor Swift** | **26** | **@taylorswift13** |

### **Database Schema**

### **celebs**

|  |  |
| --- | --- |
| **name** | **Type** |
| **id** | **INTEGER** |
| **name** | **TEXT** |
| **age** | **INTEGER** |
| **twitter\_handle** | **TEXT** |

|  |
| --- |
| **Rows: 1** |

**Constraints**

**Constraints that add information about how a column can be used are invoked after specifying the data type for a column. They can be used to tell the database to reject inserted data that does not adhere to a certain restriction. The statement below sets constraints on the celebs table.**

**CREATE TABLE celebs (  
   id INTEGER PRIMARY KEY,   
   name TEXT UNIQUE,  
   date\_of\_birth TEXT NOT NULL,  
   date\_of\_death TEXT DEFAULT 'Not Applicable'  
);**

**1. PRIMARY KEY columns can be used to uniquely identify the row. Attempts to insert a row with an identical value to a row already in the table will result in a constraint violation which will not allow you to insert the new row.**

**2. UNIQUE columns have a different value for every row. This is similar to PRIMARY KEY except a table can have many different UNIQUE columns.**

**3. NOT NULL columns must have a value. Attempts to insert a row without a value for a NOT NULL column will result in a constraint violation and the new row will not be inserted.**

**4. DEFAULT columns take an additional argument that will be the assumed value for an inserted row if the new row does not specify a value for that column.**

### **Instructions**

**1.**

**Create a new table with constraints on the values. In the code editor type:**

**CREATE TABLE awards (  
   id INTEGER PRIMARY KEY,  
   recipient TEXT NOT NULL,  
   award\_name TEXT DEFAULT 'Grammy'  
);**

**How many tables do you see in the database schema on the right?**

**Solution:**

**CREATE TABLE awards (**

**id INTEGER PRIMARY KEY,**

**recipient TEXT NOT NULL,**

**award\_name TEXT DEFAULT 'Grammy'**

**);**

**Table :**

### **Query Results**

|  |
| --- |
|  |

### **Database Schema**

### **celebs**

|  |  |
| --- | --- |
| **name** | **type** |
| **id** | **INTEGER** |
| **name** | **TEXT** |
| **age** | **INTEGER** |
| **twitter\_handle** | **TEXT** |

|  |
| --- |
| **Rows: 1** |

### **awards**

|  |  |
| --- | --- |
| **name** | **type** |
| **id** | **INTEGER** |
| **recipient** | **TEXT** |
| **award\_name** | **TEXT** |

|  |
| --- |
| **Rows: 0** |

**SQL is a programming language designed to manipulate and manage data stored in relational databases.**

* **A relational database is a database that organizes information into one or more tables.**
* **A table is a collection of data organized into rows and columns.**

**A statement is a string of characters that the database recognizes as a valid command.**

* **CREATE TABLE creates a new table.**
* **INSERT INTO adds a new row to a table.**
* **SELECT queries data from a table.**
* **ALTER TABLE changes an existing table.**
* **UPDATE edits a row in a table.**
* **DELETE FROM deletes rows from a table.**

**Constraints add information about how a column can be used.**

### **Instructions**

**In this lesson, we have learned SQL statements that create, edit, and delete data. In the upcoming lessons, we will learn how to use SQL to retrieve information from a database!**

**1.**

Write two statements that select the first 10 rows of talks and speakers. What is the title of the talk with id = 3?

SELECT \* FROM talks

LIMIT 10;

SELECT \* FROM speakers

LIMIT 10;

**PostgreSQL Data Types**

One of the most basic methods is built into the CREATE TABLE syntax that you’ve probably already seen before.

In a CREATE TABLE statement we specify the data type for each column of a table (e.g., int, text, timestamp, etc.). In doing so, we’re telling PostgreSQL which types of values can be inserted into each column in the table. You can refer to the complete list of available data types in the [PostgreSQL documentation](https://www.postgresql.org/docs/10/datatype.html).

| **Name** | **Description** |
| --- | --- |
| boolean | true/false |
| varchar or varchar(n) | text with variable length, up to n characters (if specified) |
| date | calendar date |
| integer | whole number value between -2147483648 and +2147483647 |
| numeric(a, b) | decimal with total digits (a) and digits after the decimal point (b) |
| time | time of day (no time zone) |

To create a table that stores information about volunteers for the conference we could write the following:

CREATE TABLE volunteers (  
    id integer,  
    name varchar,  
    hours\_available integer,  
    phone\_number varchar(12),  
    email varchar  
);

In the statement above, we’ve ensured that our volunteers table will have:

* Integer values for data in columns id and hours\_available
* Text values data in columns name, phone\_number, and email

However, data types don’t prevent all unexpected data from being inserted into a table. For example, we’ve defined phone\_number as varchar(12) and might expect a 10-digit phone number formatted as XXX-XXX-XXXX. Consider the following issues that may arise:

* An incomplete value formatted like XXX-XXXX will be accepted because it’s under 12 characters.
* A value like +X XXX-XXX-XXXX will cause PostgreSQL to raise an error because it’s longer than 12 characters, even though it’s a valid entry.

Another potential issue caused by relying only on PostgreSQL data types stems from the fact that PostgreSQL will try to interpret incoming data as the data type the column has been defined as. This process, called type casting, can have mixed results.

* If one tries to insert 1.5 into our table’s hours\_available column, PostgreSQL will cast this value to integer, round the data, and insert it into the table as 2.
* If one tries to insert 1.5 into the email column, PostgreSQL will insert this into the database by casting 1.5 to '1.5' even though '1.5' is not a valid email address.

**1.**

Let’s add a table to our DB that will keep track of attendees. Write the CREATE TABLE statement to create a table named attendees with the following column names and data types.

* id — integer
* name — varchar
* total\_tickets\_reserved — integer
* standard\_tickets\_reserved — integer
* vip\_tickets\_reserved — integer

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Write a statement that inserts a row with the following values into attendees. This insert should not succeed. Inspect the error message the database returns.

* id — 1
* name — 'John Smith'
* total\_tickets\_reserved — '2.5'
* standard\_tickets\_reserved — 1
* vip\_tickets\_reserved — 1

**3.**Remember, we can’t always rely on PostgreSQL to interpret our input exactly as we’d like. In this case, PostgreSQL isn’t clever enough to convert a decimal in quotes ('2.5') to an integer (2). Alter your query to insert 2 (as an integer) into total\_tickets\_reserved. Does this succeed? : solution:

/\*create a table  and attendees is the name of table\*/

CREATE TABLE attendees (

id integer,

name varchar,

total\_tickets\_reserved integer,

standard\_tickets\_reserved integer,

vip\_tickets\_reserved integer

);

/\*insert a row in attendeed\*/

INSERT INTO attendees (id, name, total\_tickets\_reserved, standard\_tickets\_reserved, vip\_tickets\_reserved)

VALUES (1, 'John Smith', '2.5', 1, 1);

INSERT into attendees values (1, 'John Smith', 2, 1, 1);

**Nullability Constraints:** Suppose we insert a row that doesn’t contain all desired fields into our current talks table. We can do this with the statement below.

INSERT INTO talks (id, estimated\_length)  
VALUES (1, 30);

We can query this table to see how this row looks when inserted into PostgreSQL when there are no constraints in place.

SELECT \* FROM talks  
WHERE id = 1;

| **id** | **title** | **speaker\_id** | **estimated\_length** | **session\_timeslot** |
| --- | --- | --- | --- | --- |
| 1 | NULL | NULL | 30 | NULL |

As expected, we see that there are NULL values in the title, session\_timeslot and speaker\_id columns. With PostgreSQL, we can choose to reject inserts and updates that don’t include data for specific columns by adding a NOT NULL constraint on those columns.

Let’s consider how we might implement this constraint on the talks table. If we know which columns cannot be NULL before creating our table, we can add a NOT NULL constraint following the datatype in the table’s CREATE TABLE statement.

CREATE TABLE talks (  
    id integer,  
    title varchar NOT NULL,  
    speaker\_id integer NOT NULL,  
    estimated\_length integer,  
    session\_timeslot timestamp NOT NULL  
);

Let’s try the previous insert again.

INSERT INTO talks (id, estimated\_length)  
VALUES (1, 30);

Great, this statement now causes PostgreSQL to return an error.

ERROR: null value in column "title" violates not-null constraint

Detail: Failing row contains (1, null, null, 30, null).

The error message lets us know our constraint is working! We even get a helpful message that shows information about the contents of the failing row.

**1.**

Suppose we’re creating our speakers table from scratch. Write the CREATE TABLE statement that creates a table with a NOT NULL constraint on id, email, and name. Use the following types.

* id - integer
* email - varchar
* name - varchar
* organization - varchar
* title - varchar
* years\_in\_role - integer

Checkpoint

**2.**

Try inserting the following row to confirm our new constraint is working. Examine the error message. Can you identify which of the NOT NULL constraints is violated with this row?

INSERT INTO speakers (id, email, organization, title, years\_in\_role)  
VALUES (1, 'awilson@ABCcorp.com', 'ABC Corp.', 'CTO', 6);

Checkpoint 3 Passed

**3.**

For this to be a valid insert we must insert a value into each of our NOT NULL columns. Change the INSERT statement from the previous exercise to insert the value 'A. Wilson' into name

Solution:

create table speakers (

  id integer NOT NULL,

  email varchar NOT NULL,

  name varchar NOT NULL,

  organization varchar,

  title varchar,

  years\_in\_role integer

);

INSERT INTO speakers (id, email, name, organization, title, years\_in\_role)

VALUES (1, 'awilson@ABCcorp.com', 'A. Wilson', 'ABC Corp.', 'CTO', 6);

**Improving Tables with Constraints:** In PostgreSQL, we can use ALTER TABLE statements to add or remove constraints from existing tables. In fact, all of the constraints we’ll cover throughout this lesson can be added to an existing table by writing an ALTER TABLE statement!

Let’s imagine we’ve already populated our talks table with some data, but we haven’t included any constraints. Suppose that:

* The column session\_timeslot contains no NULL values
* The column title contains about 50% NULL values
* Instead, we can add a NOT NULL constraint to a column using an ALTER TABLE statement. Let’s add a NOT NULL constraint on session\_timeslot with the following statement.
* ALTER TABLE talks  
  ALTER COLUMN session\_timeslot SET NOT NULL;
* If we later decide we no longer need this constraint, we can drop a NOT NULL constraint from an existing table with the following statement:
* ALTER TABLE talks  
  ALTER COLUMN session\_timeslot DROP NOT NULL
* If we’d like to add a NOT NULL constraint to the title column, we can attempt to do so using the same syntax we used to add a constraint on session\_timeslot.
* ALTER TABLE talks  
  ALTER COLUMN title SET NOT NULL;
* However, PostgreSQL will reject the addition of the constraint and raise the following error because NULL values are already present in the column. See the error the database returns below.
* SQL Error [23502]: ERROR: column "title" contains null values
* If the table we’re attempting to add a constraint on doesn’t meet the constraint, we can *backfill* the table so that it does adhere to the constraint. *Backfilling* is a term occasionally used in DB engineering to refer to the process of adding or updating past values. In this case, we can fill our target column’s NULL values with a placeholder value using the query below.
* UPDATE talks  
  SET title = 'TBD'  
  WHERE title IS NULL;
* With the table updated so that there are no longer any nulls in title, and we can now apply the NOT NULL constraint.
* ALTER TABLE talks  
  ALTER COLUMN title SET NOT NULL;

**1.** Write an ALTER TABLE statement that adds a NOT NULL constraint on the name column of our speakers table. Assume there are no null values in name.

**2.**Suppose we’d like to implement a NOT NULL constraint on organization, but the column already contains NULL values. Write an UPDATE statement that fills the null values in organization with a placeholder value.

**3.**Write an ALTER TABLE statement that adds a NOT NULL constraint on organization.

Solution:

ALTER TABLE speakers

ALTER COLUMN name SET NOT NULL;

UPDATE speakers

SET organization = 'Unaffiliated'

WHERE organization IS NULL;

ALTER TABLE speakers

ALTER COLUMN organization SET NOT NULL;

s

**Introduction to Check Constraints**

In some situations, we might want to establish specific rules to determine what makes a row valid. For example, In our talks table, we might want to ensure that the estimated\_length column is:

* An integer
* NOT NULL
* Positive
* We can use CHECK statements to implement more precise constraints on our table. A CHECK constraint can be written into a CREATE TABLE statement, or added to an existing table with ALTER TABLE.
* To use a check constraint, we list CHECK (...) following the data type in a CREATE TABLE statement and write the condition we’d like to test for inside the parentheses.
* The condition tested for inside of parentheses of a CHECK statement must be a SQL statement that can be **evaluated as either true or false**. These statements are similar to the statements you may be familiar with in WHERE clauses when filtering rows from a table. Let’s add a CHECK statement to ensure our talks table has a positive value for estimated\_length for each row.
* ALTER TABLE talks   
  ADD CHECK (estimated\_length > 0);
* In some situations, you may want to apply multiple constraints on a single column. In this case, we’d like to also add a NOT NULL constraint on estimated\_length. We can add additional constraints on a column with multiple ALTER TABLE statements.
* Alternatively, If we know the constraints we’d like to include as we’re creating a table, we can list following the column name and datatype in a CREATE TABLE statement. To implement the constraints we’ve covered in this lesson you could write estimated\_length integer NOT NULL CHECK (estimated\_length > 0) into the CREATE TABLE statement for talks.

### Instructions

* **1.**
* Returning to our speakers table, write the ALTER TABLE statement that adds a CHECK condition on years\_in\_role. This constraint should make sure that years in role is less than 100.

Solution:

ALTER TABLE speakers

ADD CHECK (years\_in\_role < 100);

Check Constraints Continued

Inside a CHECK statement we can use a wide array of SQL syntax to create our conditions. For example, within our check constraint we can:

* Make comparisons between columns within the table
* Use logical operators like AND and OR
* Use other SQL operators you may be familiar with (IN, LIKE)

As a general rule, any logic that you might use in a WHERE statement to filter individual rows from an existing table can be applied within a CHECK, including logic that involves multiple columns or conditions. Suppose we’d like to add a check that all entries in talks have an estimated\_length between 0 and 120 minutes. We could apply such a check with the following:

ALTER TABLE talks   
ADD CHECK (estimated\_length > 0 AND estimated\_length < 120);

We can also apply constraints that apply to multiple columns. For example, suppose we want to enforce the following:

1. estimated\_length less than 120 minutes
2. year of the talk should be 2020 (we can extract this value from session\_timeslot)

We could do this by adding separate checks on each of the columns. Alternatively, we could add a single CHECK constraint that checks both conditions as shown below.

ALTER TABLE talks  
ADD CHECK (estimated\_length < 120 AND date\_part('year', session\_timeslot) = 2020);

Don’t worry if you’re not familiar with this syntax, the date\_part function in PostgreSQL just returns a portion of the date as an integer (e.g. date\_part('year' ,'2020-08-01 00:00:00'::date) = 2020).

### Instructions

**1.**

Using one CHECK statement, write an ALTER TABLE statement that adds a constraint on speakers to ensure that values for years\_in\_role between 0 and 100.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Recall our attendees table from earlier in this lesson. So far, the table had the following CREATE TABLE statement.

CREATE TABLE attendees (  
  id integer,  
  name varchar,  
  total\_tickets\_reserved integer,   
  standard\_tickets\_reserved integer,   
  vip\_tickets\_reserved integer  
);

Using one CHECK statement, write an ALTER TABLE statement that adds a constraint on attendees to ensure that standard\_tickets\_reserved + vip\_tickets\_reserved = total\_tickets\_reserved.

SOLUTION: ALTER TABLE speakers

ADD CHECK (years\_in\_role < 0 AND years\_in\_role > 100);

CREATE TABLE attendees (

  id integer,

  name varchar,

  total\_tickets\_reserved integer,

  standard\_tickets\_reserved integer,

  vip\_tickets\_reserved integer

);

**Using Unique Constraints**

When designing a PostgreSQL data model, it’s a good practice to structure tables such that rows are uniquely identifiable by some combination of attributes. Structuring your tables in this way leads to a few benefits:

* The structure of your data model and the contents of individual tables are more easily interpreted.
* Queries to access information from the table can be simpler. For example, if we’d like to query our attendees table to find out how many tickets an attendee has reserved, having a unique identifier for each attendee allows us to get a result without any intermediate aggregation.
* Identifying and implementing a PRIMARY KEY is easier on tables with UNIQUE constraints already in place.

In our attendees table, suppose that we’d like to make sure that no two people submit the same email address when they register. To do so we could apply a unique constraint on email.

To implement this constraint we could include it in our CREATE TABLE statement. To identify values in a single column as unique, we specify UNIQUE following the column name and datatype definitions, in this case we’d write email varchar UNIQUE in our CREATE TABLE statement.

Alternatively, we can add the constraint to an existing table using the following:

ALTER TABLE attendees   
ADD UNIQUE (email);

Returning to our talks table, suppose we’d like to use the combination of speaker\_id and session\_timeslot to ensure that a speaker is never booked for multiple talks at the same time. We can do this in the CREATE TABLE statement by specifying the columns that need to be jointly unique in parentheses on its own line following the column names and datatype definitions. In this case, we’d add a UNIQUE (speaker\_id, session\_timeslot) on it’s own line in the CREATE TABLE statement.

Just as with single column unique constraints, we can also and add the constraint with an ALTER TABLE statement.

ALTER TABLE talks  
ADD UNIQUE (speaker\_id, session\_timeslot)

### Instructions

**1.**

Returning to our speakers table, write an ALTER TABLE statement that includes a new constraint on email so it can be used to uniquely identify rows of the table.

Stuck? Get a hint

**2.**

Edit the CREATE TABLE statement below that defines a new table called registrations. Include a multi-column UNIQUE constraint that ensures attendees (identified by attendee\_id) are registered for only one talk at a time (identified by session\_timeslot).

CREATE TABLE registrations (  
    id integer NOT NULL,  
    attendee\_id integer NOT NULL,  
    session\_timeslot timestamp NOT NULL,  
    talk\_id integer NOT NULL,  
    \_\_\_\_ (\_\_\_\_, \_\_\_\_)  
);

SOLUTION: ALTER TABLE speakers

ADD UNIQUE (email);

CREATE TABLE registrations (

    id integer NOT NULL,

    attendee\_id integer NOT NULL,

    session\_timeslot timestamp NOT NULL,

    talk\_id  integer NOT NULL,

    UNIQUE (session\_timeslot, attendee\_id)

);

TABLE OF THIS CODE:

### Query Results

### Database Schema

### speakers

|  |  |
| --- | --- |
| **name** | **type** |
| id | integer |
| email | character varying |
| name | character varying |
| organization | character varying |
| title | character varying |
| years\_in\_role | integer |

|  |
| --- |
| **Rows: 10** |

### registrations

|  |  |
| --- | --- |
| **name** | **type** |
| id | integer |
| attendee\_id | integer |
| session\_timeslot | timestamp without time zone |
| talk\_id | integer |

|  |
| --- |
| **Rows: 0** |

**Introduction to Primary Keys:** Primary keys are essential to defining these relationships.

A primary key is a column (or set of columns) that **uniquely identifies a row within a database table**. A table can only have one primary key, and in order to be selected as a primary key a column (or set of columns) should:

* Uniquely identify that row in the table (like a UNIQUE constraint)
* Contain no null values (like a NOT NULL constraint)

Implementing a PRIMARY KEY constraint is similar to simultaneously enforcing a UNIQUE and NOT NULL constraints on a column (or set of columns). Although UNIQUE NOT NULL and PRIMARY KEY constraints function very similarly, tables are limited to one PRIMARY KEY, but not limited in how many columns can have both UNIQUE and NOT NULL constraints.

In addition to defining relationships between tables, primary keys also improve your data model in several other ways:

* Many joins will use the primary key from one table to join data with another table
* Primary keys can improve query performance
* Primary keys help to enforce data integrity within a table by ensuring that rows can be uniquely identified

Recall that the attendees table has a column named id that uniquely identifies attendees. We have also already restricted the name and email columns to be NOT NULL“. Now let’s add id as our table’s PRIMARY KEY, we can do this with an ALTER TABLE statement.

ALTER TABLE attendees  
ADD PRIMARY KEY (id);

Even with a primary key, there’s still good reason to use a combination of UNIQUE and NOT NULL constraints to enforce data integrity on other columns. In this case, the combination of a UNIQUE and NOT NULL constraint on email can be used to validate that all attendees have a value for email and each unique email can only be matched to one attendee.

Now that you’re familiar with how primary keys work, for the remainder of the lesson, you may assume that all tables in our data model have a primary key on their id column.

### Instructions

**1.**

Let’s implement a primary key on our speakers table. Write the ALTER TABLE statement that adds id as the primary key.

Stuck? Get a hint

**2.**

Great, now that we’ve implemented a primary key on id we can confirm that it upholds both UNIQUE and NOT NULL constraints. Write the INSERT statement that writes the following information into a single row of the speakers table.

* email — '[J.Saunders@ABCTech.com](mailto:J.Saunders@ABCTech.com)'
* name — 'Joan Saunders'
* organization — 'ABC Tech.'
* title — 'Sr. Data Scientist'
* years\_in\_role — 6

Note that this INSERT statement will cause PostgreSQL to return an error. What error would you expect this row to produce?

**3.**The previous exercise showed us that our primary key upholds the NOT NULL constraint. Let’s confirm that the id column also upholds a UNIQUE constraint. Insert the following row with id=1 and inspect the error message. Does it look as expected?”

* id — 1
* email — '[J.Saunders@ABCTech.com](mailto:J.Saunders@ABCTech.com)'
* name — 'Joan Saunders'
* organization — 'ABC Tech.'
* title — 'Sr. Data Scientist'
* years\_in\_role — 6

**SOLUTION:** ALTER TABLE speakers

ADD PRIMARY KEY (id);

INSERT INTO speakers(email, name, organization, title, years\_in\_role)

VALUES ('j.Saunders@ABCTech.com', 'Joan Saunders', 'ABC Tech', 'Sr. Data Scientist', 6);

INSERT INTO speakers (id, email, name, organization, title, years\_in\_role)

VALUES (1, 'j.Saunders@ABCTech.com', 'Joan Saunders', 'ABC Tech', 'Sr. Data Scientist', 6);

TABLE OF THIS CODE:

### Query Results

### Database Schema

### talks

|  |  |
| --- | --- |
| **name** | **type** |
| id | integer |
| title | character varying |
| speaker\_id | integer |
| estimated\_length | integer |
| session\_timeslot | timestamp without time zone |

|  |
| --- |
| **Rows: 20** |

### speakers

|  |  |
| --- | --- |
| **name** | **type** |
| id | integer |
| email | character varying |
| name | character varying |
| organization | character varying |
| title | character varying |
| years\_in\_role | integer |

|  |
| --- |
| **Rows: 10** |

Incorrect:

**Introduction To Foreign Keys:** Let’s use some of what we’ve learned so far about primary keys to improve registrations. Recall the registrations table that we previously created had an id primary key column, and had fields attendee\_id, session\_timeslot jointly unique, so that no attendee may register for two talks with the same timeslot.

In this model, the talks in registrations are meant to reference the talks described in talks. Because of this, we’d probably want to ensure that any talk\_id entered into registrations references an id that already appears in talks.

Referential integrity can be enforced by adding a FOREIGN KEY on the child table that references the primary key of a parent table.

Let’s work through the example described above to improve registrations. In our example above registrations is a child table of talks because entries in registrations must reference the primary key from talks. Suppose talks also has a column named id as a primary key. Now, we can update our registrations table with a foreign key using the following statement.

ALTER TABLE registrations  
ADD FOREIGN KEY (talk\_id)  
REFERENCES talks (id);

Alternatively, if we’re creating a table from scratch, we can include the following line in the CREATE TABLE statement , FOREIGN KEY (talk\_id) REFERENCES talks (id)

Suppose we now want to enter a registration for talk\_id = 100, which does not yet exist in the talks table. Trying to insert a registration for this talk yields an error because there is not a corresponding entry in talks to reference yet. The error below lets us know the constraint is working and provides a helpful error message that indicates we need to add an entry to talks before this insert will succeed.

INSERT INTO registrations VALUES (100, 1, '2020-08-15 9:00:00', 1);

SQL Error [23503]: ERROR: insert or update on table "registrations" violates foreign key constraint "registrations\_id\_fkey"

Detail: Key (talk\_id)=(100) is not present in table "talks".

**1.**

We can use foreign keys in a few other places throughout our data model. Let’s ensure that talks can’t be inserted into the talks table without the talk’s speaker already existing in the speakers table.

Write the ALTER TABLE statement that ensures talks column speaker\_id references speakers column id. You may assume there is already a primary key on speakers column id.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Let’s test out this foreign key constraint, try to insert the following row into talks and inspect the error message. Does it look as expected?”

* id — 21
* title — 'Data Warehousing Best Practices'
* speaker\_id — 101
* estimated\_length — 30
* session\_timeslot — '2020-08-15 18:00': SOLUTION:

ALTER TABLE talks

ADD FOREIGN KEY (speaker\_id)

REFERENCES speakers (id);

INSERT INTO talks (id, title, speaker\_id, estimated\_length, session\_timeslot)

VALUES (21, 'Data Warehousing Best Practices', 101, 30, '2020-08-15 18:00');

**TABLE OF THIS CODE:**

### Query Results

### Database Schema

### speakers

|  |  |
| --- | --- |
| **name** | **type** |
| id | integer |
| email | character varying |
| name | character varying |
| organization | character varying |
| title | character varying |
| years\_in\_role | integer |

|  |
| --- |
| **Rows: 10** |

### talks

|  |  |
| --- | --- |
| **name** | **type** |
| id | integer |
| title | character varying |
| speaker\_id | integer |
| estimated\_length | integer |
| session\_timeslot | timestamp without time zone |

|  |
| --- |
| **Rows: 20** |

Incorrect:

**Foreign Keys - Cascading Changes:** This behavior is sometimes explicitly specified in a CREATE TABLE statement using REFERENCES talks (id) ON DELETE RESTRICT or REFERENCES talks (id) ON UPDATE RESTRICT.

However, another strategy you may consider is adding a CASCADE clause. Rather than preventing changes, CASCADE clauses (ON UPDATE CASCADE, ON DELETE CASCADE) **cause the updates or deletes to automatically be applied to any child tables**.

For example, suppose we’d like to set up our database to automatically unregister attendees from a talk that’s been cancelled. To do this we could apply ON DELETE CASCADE to our foreign key constraint.

ALTER TABLE registrations  
ADD FOREIGN KEY (talk\_id)  
REFERENCES talks (id) ON DELETE CASCADE

When we try to delete a value from talks, we also notice that all registrations for talk\_id = 1 are removed as well. This preserves referential integrity by removing any row associated with this talk. To demonstrate, let’s first return the rows in registrations for talk\_id = 1.

SELECT \*   
FROM registrations  
WHERE talk\_id = 1;

| **id** | **attendee\_id** | **session\_timelot** | **talk\_id** |
| --- | --- | --- | --- |
| 8 | 2 | 2020-08-15 9:00:00 | 1 |
| 9 | 5 | 2020-08-15 9:00:00 | 1 |
| 10 | 8 | 2020-08-15 9:00:00 | 1 |
| 11 | 9 | 2020-08-15 9:00:00 | 1 |
| 12 | 10 | 2020-08-15 9:00:00 | 1 |
| 13 | 11 | 2020-08-15 9:00:00 | 1 |
| 14 | 3 | 2020-08-15 9:00:00 | 1 |

Great, we observe 7 registrations for talk\_id = 1 in registrations. Now, let’s use the following code to delete the talk with id = 1:

DELETE FROM talks   
WHERE id = 1;

Because we’ve specified ON DELETE CASCADE on our foreign key, the DELETE statement ran successfully even though there were still rows in registrations that referenced talk\_id = 1. Let’s check to see what affect this statement had on registrations.

SELECT \*   
FROM registrations  
WHERE talk\_id = 1;

| **id** | **attendee\_id** | **session\_timelot** | **talk\_id** |
| --- | --- | --- | --- |

As expected, records that correspond to talk\_id = 1 have been removed. In effect, ON DELETE CASCADE runs the required deletes on the child table behind the scenes and allows the user to simply run a single command to update a data model.

### Instructions

**1.**

Let’s implement a rule that says if a speaker is removed their talks are removed as well. Let’s fill the ALTER TABLE below with an ON DELETE CASCADE clause that implements this rule.

ALTER TABLE talks  
ADD FOREIGN KEY (\_\_\_\_)  
\_\_\_\_\_\_\_\_ speakers (\_\_\_\_) ON DELETE CASCADE;

Stuck? Get a hint

**2.**

Let’s check this constraint is working. Write a DELETE statement on speakers that will remove the talks in the speaker with id = 2 was scheduled to present. Inspect the schema for the talks table before and after this delete (there should be 20 rows before). How many talks was this speaker scheduled to present?

SOLUTION:

ALTER TABLE talks

ADD FOREIGN KEY (speaker\_id)

REFERENCES speakers (id) ON DELETE CASCADE;

DELETE FROM speakers

WHERE id = 2;

TABLE OF THIS CODE:

### Query Results

### Database Schema

### speakers

|  |  |
| --- | --- |
| **name** | **type** |
| id | integer |
| email | character varying |
| name | character varying |
| organization | character varying |
| title | character varying |
| years\_in\_role | integer |

|  |
| --- |
| **Rows: 9** |

### talks

|  |  |
| --- | --- |
| **name** | **type** |
| id | integer |
| title | character varying |
| speaker\_id | integer |
| estimated\_length | integer |
| session\_timeslot | timestamp without time zone |

|  |
| --- |
| **Rows: 18** |

**PostgreSQL Constraints Review:**

Constraints are rules a DB engineer defines as part of the data model to gain more control over what values are allowed in specific columns and tables.

Specifically, Constraints:

* Reject rows containing values that shouldn’t be inserted into a database table, which can help with preserving data integrity and quality.
* Raise an error when they’re violated, which can also help with debugging applications that write to the database.

There are quite a few types of constraints:

* Data types — Are your first line of defense, these rules aren’t constraints but can help reject incorrect data from your database.
* NOT NULL constraints — Reject incoming rows from your table when critical information is missing from a row.
* CHECK constraints — Give you more control over what rules you’d like to apply to your tables. These constraints will allow you to reject a row if it fails the criteria you’ve defined.
* UNIQUE constraints — Help with defining unique values in a table, they also create an index which can improve query and join performance.
* PRIMARY KEY constraints — A column or combination of columns that uniquely identify a row and are both NOT NULL and UNIQUE. PRIMARY KEYs are unique to a table, and will often be used in joins between tables.
* FOREIGN KEY constraints — Allow you to maintain referential integrity between two tables by validating the entry in one also appears in the other. Referential integrity depends on FOREIGN KEY constraints.

**TABLE**

### Query Results

### Database Schema

### talks

|  |  |
| --- | --- |
| **name** | **type** |
| id | integer |
| title | character varying |
| speaker\_id | integer |
| estimated\_length | integer |
| session\_timeslot | timestamp without time zone |

|  |
| --- |
| **Rows: 20** |

### speakers

|  |  |
| --- | --- |
| **name** | **type** |
| id | integer |
| email | character varying |
| name | character varying |
| organization | character varying |
| title | character varying |
| years\_in\_role | integer |

|  |
| --- |
| **Rows: 10** |

### registrations

|  |  |
| --- | --- |
| **name** | **type** |
| id | integer |
| attendee\_id | integer |
| session\_timeslot | timestamp without time zone |
| talk\_id | integer |

|  |
| --- |
| **Rows: 80** |

### attendees

|  |  |
| --- | --- |
| **name** | **type** |
| id | integer |
| name | integer |
| total\_tickets\_reserved | integer |
| standard\_tickets\_reserved | integer |
| vip\_tickets\_reserved | integer |

|  |
| --- |
| **Rows: 0** |

**SQL QUERIES:** In this lesson, we will be learning different SQL commands to **query** a single table in a database.

One of the core purposes of the SQL language is to retrieve information stored in a database.

Queries allow us to communicate with the database by asking questions and returning a result set with data relevant to the question.

We will be querying a database with one table named movies.

SELECT \* FROM movies;

SELECT is used every time you want to query data from a database and \* means all columns.

SELECT name, genre, year

FROM movies;

**TABLEL OF THIS CODE**

### Query Results

|  |  |  |
| --- | --- | --- |
| **name** | **genre** | **year** |
| Avatar | action | 2009 |
| Jurassic World | action | 2015 |

**As**

Knowing how SELECT works, suppose we have the code below:

SELECT name AS 'Titles'  
FROM movies;

AS is a keyword in SQL that allows you to rename a column or table using an alias. Here we renamed the name column as Titles.

Some important things to note:

* Although it’s not always necessary, it’s best practice to surround your aliases with single quotes.
* When using AS, the columns are not being renamed in the table. The aliases only appear in the result

Distinct: DISTINCT is used to return unique values in the output. It filters out all duplicate values in the specified column(s).

For instance,

SELECT tools   
FROM inventory;

might produce:

| **tools** |
| --- |
| Hammer |
| Nails |
| Nails |
| Nails |

By adding DISTINCT before the column name,

SELECT DISTINCT tools   
FROM inventory;

the result would now be:

| **tools** |
| --- |
| Hammer |
| Nails |

Filtering the results of a query is an important skill in SQL. It is easier to see the different possible genres in the movie table after the data has been filtered than to scan every row in the table.

**Where:** Following this format, the statement below filters the result set to only include top rated movies (IMDb ratings greater than 8):

SELECT \*  
FROM movies  
WHERE imdb\_rating > 8;

How does it work?

1. WHERE clause filters the result set to only include rows where the following condition is true.
2. imdb\_rating > 8 is the condition. Here, only rows with a value greater than 8 in the imdb\_rating column will be returned.

The > is an operator. Operators create a condition that can be evaluated as either true or false.

Comparison operators used with the WHERE clause are:

* = equal to
* != not equal to
* > greater than
* < less than
* >= greater than or equal to
* <= less than or equal to

There are also some special operators that we will learn more about in the upcoming exercises.

SELECT \*

FROM movies

WHERE year > 2014;

**All table year is present which is >**

Like I

LIKE can be a useful operator when you want to compare similar values.

The movies table contains two films with similar titles, ‘Se7en’ and ‘Seven’.

How could we select all movies that start with ‘Se’ and end with ‘en’ and have exactly one character in the middle?

SELECT \*   
FROM movies  
WHERE name LIKE 'Se\_en';

* LIKE is a special operator used with the WHERE clause to search for a specific pattern in a column.
* name LIKE 'Se\_en' is a condition evaluating the name column for a specific pattern.
* Se\_en represents a pattern with a wildcard character.

The \_ means you can substitute any individual character here without breaking the pattern. The names Seven and Se7en both match this pattern.

**So only that table comes which ending by en.**

**Like II**

The percentage sign % is another wildcard character that can be used with LIKE.

This statement below filters the result set to only include movies with names that begin with the letter ‘A’:

SELECT \*   
FROM movies  
WHERE name LIKE 'A%';

% is a wildcard character that matches zero or more missing letters in the pattern. For example:

* A% matches all movies with names that begin with letter ‘A’
* %a matches all movies that end with ‘a’

We can also use % both before and after a pattern:

SELECT \*   
FROM movies   
WHERE name LIKE '%man%';

Here, any movie that *contains* the word ‘man’ in its name will be returned in the result.

LIKE is not case sensitive. ‘Batman’ and ‘Man of Steel’ will both appear in the result of the query above.

**EXAMPLE:**

**1.**

In the text editor, type:

SELECT \*   
FROM movies  
WHERE name LIKE '%man%';

How many movie titles contain the word ‘man’?

Checkpoint

ANSWER:

### Query Results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **id** | **name** | **genre** | **year** | **imdb\_rating** |
| 9 | Pirates of the Caribbean: Dead Mans Chest | action | 2006 | 7.3 |
| 10 | Iron Man 3 | action | 2013 | 7.3 |

EXAMPLE: Edit the query so that it selects all the information about the movie titles that *begin* with the word ‘The’.

You might need a space in there!

Checkpoint

ANSWER:

### Query Results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **id** | **name** | **genre** | **year** | **imdb\_rating** |
| 3 | The Avengers | action | 2012 | 8.1 |
| 4 | The Dark Knight | action | 2008 | 9.0 |
| 8 | The Dark Knight Rises | action | 2012 | 8.5 |
|  |  |  |  |  |

**Is Null:** Unknown values are indicated by NULL.

It is not possible to test for NULL values with comparison operators, such as = and !=.

Instead, we will have to use these operators:

* IS NULL
* IS NOT NULL

To filter for all movies with an IMDb rating:

SELECT name  
FROM movies   
WHERE imdb\_rating IS NOT NULL;

**EXAMPLE:** Now let’s do the opposite.

Write a query to find all the movies *without* an IMDb rating.

Select only the name column!

Checkpoint

**ANSWER:**

SELECT name

FROM movies

WHERE imdb\_rating IS NULL;

**TABLE:**

### Query Results

|  |
| --- |
| **name** |
| The Good, the Bad and the Ugly |
| Dawn of the Dead |
| Shawn of the Dead |

**Between**

The BETWEEN operator is used in a WHERE clause to filter the result set within a certain *range*. It accepts two values that are either numbers, text or dates.

For example, this statement filters the result set to only include movies with years from 1990 up to, *and including* 1999.

SELECT \*  
FROM movies  
WHERE year BETWEEN 1990 AND 1999;

When the values are text, BETWEEN filters the result set for within the alphabetical range.

In this statement, BETWEEN filters the result set to only include movies with names that begin with the letter ‘A’ up to, *but not including* ones that begin with ‘J’.

SELECT \*  
FROM movies  
WHERE name BETWEEN 'A' AND 'J';

However, if a movie has a name of simply ‘J’, it would actually match. This is because BETWEEN goes *up to* the second value — up to ‘J’. So the movie named ‘J’ would be included in the result set but not ‘Jaws’

**EXAMPLE:** Using the BETWEEN operator, write a query that selects all information about movies whose name begins with the letters ‘D’, ‘E’, and ‘F’.

**ANSWER:**

SELECT \*

FROM movies

WHERE name BETWEEN 'D'AND 'G';

**TABLE:**

### Query Results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **id** | **name** | **genre** | **year** | **imdb\_rating** |
| 203 | Dances with Wolves | drama | 1990 | 8.0 |
| 128 | Dark Shadows | horror | 2012 | 6.2 |
| 222 | Dawn of the Dead | horror | 1978 |  |

**Similarly we can find the year….**

**And**

Sometimes we want to *combine multiple conditions* in a WHERE clause to make the result set more specific and useful.

One way of doing this is to use the AND operator. Here, we use the AND operator to only return 90’s romance movies.

SELECT \*   
FROM movies  
WHERE year BETWEEN 1990 AND 1999  
   AND genre = 'romance';

* year BETWEEN 1990 AND 1999 is the 1st condition.
* genre = 'romance' is the 2nd condition.
* AND combines the two conditions.

With AND, *both* conditions must be true for the row to be included in the result.

**1.**

In the previous exercise, we retrieved every movie released in the 1970’s.

Now, let’s retrieve every movie released in the 70’s, that’s also well received.

In the code editor, type:

SELECT \*  
FROM movies  
WHERE year BETWEEN 1970 AND 1979  
  AND imdb\_rating > 8;

Checkpoint 2 Passed

**2.**

Remove the previous query.

Suppose we have a picky friend who only wants to watch old horror films.

Using AND, write a new query that selects all movies made prior to 1985 that are also in the horror genre.

Checkpoint

/\*SELECT \*

FROM movies

WHERE year BETWEEN 1970 AND 1979

  AND imdb\_rating > 8;\*/

SELECT \*

FROM movies

WHERE year < 1985

  AND genre = 'horror';

**Or**

Similar to AND, the OR operator can also be used to combine multiple conditions in WHERE, but there is a fundamental difference:

* AND operator displays a row if *all* the conditions are true.
* OR operator displays a row if *any* condition is true.

Suppose we want to check out a new movie or something action-packed:

SELECT \*  
FROM movies  
WHERE year > 2014  
   OR genre = 'action';

* year > 2014 is the 1st condition.
* genre = 'action' is the 2nd condition.
* OR combines the two conditions.

With OR, if *any* of the conditions are true, then the row is added to the result.

Example: Let’s test this out:

SELECT \*  
FROM movies  
WHERE year > 2014  
   OR genre = 'action';

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Suppose we are in the mood for a good laugh or a good cry.

Using OR, write a query that returns all movies that are either a romance or a comedy.

Solution:

Checkpoint

/\*SELECT \*

FROM movies

WHERE year > 2014

   OR genre = 'action';\*/

SELECT \*

FROM movies

WHERE genre = 'romance' OR genre = 'comedy';

**Order By**

That’s it with WHERE and its operators. Moving on!

It is often useful to list the data in our result set in a particular order.

We can sort the results using ORDER BY, either alphabetically or numerically. Sorting the results often makes the data more useful and easier to analyze.

For example, if we want to sort everything by the movie’s title from A through Z:

SELECT \*  
FROM movies  
ORDER BY name;

* ORDER BY is a clause that indicates you want to sort the result set by a particular column.
* name is the specified column.

Sometimes we want to sort things in a decreasing order. For example, if we want to select all of the well-received movies, sorted from highest to lowest by their year:

SELECT \*  
FROM movies  
WHERE imdb\_rating > 8  
ORDER BY year DESC;

* DESC is a keyword used in ORDER BY to sort the results in descending order (high to low or Z-A).
* ASC is a keyword used in ORDER BY to sort the results in ascending order (low to high or A-Z).

The column that we ORDER BY doesn’t even have to be one of the columns that we’re displaying.

Note: ORDER BY always goes after WHERE (if WHERE is present).

### Instructions

**1.**

Suppose we want to retrieve the name and year columns of all the movies, ordered by their name alphabetically.

Type the following code:

SELECT name, year  
FROM movies  
ORDER BY name;

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Your turn! Remove the previous query.

Write a new query that retrieves the name, year, and imdb\_rating columns of all the movies, ordered highest to lowest by their ratings.

Answer:

/\*SELECT name, year

FROM movies

ORDER BY name;\*/

SELECT name, year, imdb\_rating

FROM movies

ORDER BY imdb\_rating DESC;

Limit

We’ve been working with a fairly small table (fewer than 250 rows), but most SQL tables contain hundreds of thousands of records. In those situations, it becomes important to cap the number of rows in the result.

For instance, imagine that we just want to see a few examples of records.

SELECT \*  
FROM movies  
LIMIT 10;

LIMIT is a clause that lets you specify the maximum number of rows the result set will have. This saves space on our screen and makes our queries run faster.

Here, we specify that the result set can’t have more than 10 rows.

LIMIT always goes at the very end of the query. Also, it is not supported in all SQL databases.

### Instructions

**1.**

Combining your knowledge of LIMIT and ORDER BY, write a query that returns the top 3 highest rated movies.

Select all the columns.

Answer:

Checkpoint 2 Passed

SELECT \*

FROM movies

ORDER BY imdb\_rating DESC

LIMIT 3;

**Case**

A CASE statement allows us to create different outputs (usually in the SELECT statement). It is SQL’s way of handling [if-then](https://en.wikipedia.org/wiki/Conditional_(computer_programming)) logic.

Suppose we want to condense the ratings in movies to three levels:

* If the rating is above 8, then it is Fantastic.
* If the rating is above 6, then it is Poorly Received.
* Else, Avoid at All Costs.

SELECT name,  
 CASE  
  WHEN imdb\_rating > 8 THEN 'Fantastic'  
  WHEN imdb\_rating > 6 THEN 'Poorly Received'  
  ELSE 'Avoid at All Costs'  
 END  
FROM movies;

* Each WHEN tests a condition and the following THEN gives us the string if the condition is true.
* The ELSE gives us the string if all the above conditions are false.
* The CASE statement must end with END.

In the result, you have to scroll right because the column name is very long. To shorten it, we can rename the column to ‘Review’ using AS:

SELECT name,  
 CASE  
  WHEN imdb\_rating > 8 THEN 'Fantastic'  
  WHEN imdb\_rating > 6 THEN 'Poorly Received'  
  ELSE 'Avoid at All Costs'  
 END AS 'Review'  
FROM movies;

### Instructions

**1.**

Let’s try one on your own.

Select the name column and use a CASE statement to create the second column that is:

* ‘Chill’ if genre = 'romance'
* ‘Chill’ if genre = 'comedy'
* ‘Intense’ in all other cases

Optional: Rename the whole CASE statement to ‘Mood’ using AS.

Answer:

SELECT name,

 CASE

  WHEN genre = 'romance' THEN 'Chill'

  WHEN genre = 'comedy'  THEN 'Chill'

  ELSE 'Intense'

 END AS 'Mood'

FROM movies;

Let’s summarize:

* SELECT is the clause we use every time we want to query information from a database.
* AS renames a column or table.
* DISTINCT return unique values.
* WHERE is a popular command that lets you filter the results of the query based on conditions that you specify.
* LIKE and BETWEEN are special operators.
* AND and OR combines multiple conditions.
* ORDER BY sorts the result.
* LIMIT specifies the maximum number of rows that the query will return.
* CASE creates different outputs.

**AGGREGATE FUNCTIONS:** Calculations performed on multiple rows of a table are called **aggregates**.

Count

The fastest way to calculate how many rows are in a table is to use the COUNT() function.

COUNT() is a function that takes the name of a column as an argument and counts the number of non-empty values in that column.

SELECT COUNT(\*)  
FROM table\_name;

Here, we want to count every row, so we pass \* as an argument inside the parenthesis.

### Instructions

**1.**

Let’s count how many apps are in the table.

In the code editor, run:

SELECT COUNT(\*)   
FROM fake\_apps;

Checkpoint 2 Passed

Answer:

SELECT COUNT(\*)   
FROM fake\_apps;

Checkpoint

**2.**

Add a WHERE clause in the previous query to count how many free apps are in the table.

Checkpoint 3 Passed

WHERE price = 0;  
**Sum**

SQL makes it easy to add all values in a particular column using SUM().

SUM() is a function that takes the name of a column as an argument and returns the sum of all the values in that column.

What is the total number of downloads for all of the apps combined?

SELECT SUM(downloads)  
FROM fake\_apps;

This adds all values in the downloads column.

### Instructions

**1.**

Let’s find out the answer!

In the code editor, type:

SELECT SUM(downloads)  
FROM fake\_apps;

Checkpoint

ANSWER:

SELECT SUM(downloads)

FROM fake\_apps;

Max / Min

The MAX() and MIN() functions return the highest and lowest values in a column, respectively.

How many downloads does the most popular app have?

SELECT MAX(downloads)  
FROM fake\_apps;

The most popular app has 31,090 downloads!

MAX() takes the name of a column as an argument and returns the largest value in that column. Here, we returned the largest value in the downloads column.

MIN() works the same way but it does the exact opposite; it returns the smallest value.

### Instructions

**1.**

What is the least number of times an app has been downloaded?

In the code editor, type:

SELECT MIN(downloads)  
FROM fake\_apps;

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Delete the previous query.

Write a new query that returns the price of the most expensive app.

Checkpoint 3 Passed

ANSWER:

/\*SELECT MIN(downloads)

FROM fake\_apps;

SELECT Max(downloads)

FROM fake\_apps;

\*/

SELECT Max(price)

FROM fake\_apps;

**Average**

SQL uses the AVG() function to quickly calculate the average value of a particular column.

The statement below returns the average number of downloads for an app in our database:

SELECT AVG(downloads)  
FROM fake\_apps;

The AVG() function works by taking a column name as an argument and returns the average value for that column.

### Instructions

**1.**

Calculate the average number of downloads for all the apps in the table.

In the code editor, type:

SELECT AVG(downloads)  
FROM fake\_apps;

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Remove the previous query.

Write a new query that calculates the average price for all the apps in the table.

Checkpoint 3 Passed

ANSWER:

/\*SELECT AVG(downloads)

FROM fake\_apps;

\*/

SELECT AVG(price)

FROM fake\_apps;

**Round**

By default, SQL tries to be as precise as possible without rounding. We can make the result table easier to read using the ROUND() function.

ROUND() function takes two arguments inside the parenthesis:

1. a column name
2. an integer

It rounds the values in the column to the number of decimal places specified by the integer.

SELECT ROUND(price, 0)  
FROM fake\_apps;

Here, we pass the column price and integer 0 as arguments. SQL rounds the values in the column to 0 decimal places in the output.

### Instructions

**1.**

Let’s return the name column and a rounded price column.

In the code editor, type:

SELECT name, ROUND(price, 0)  
FROM fake\_apps;

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Remove the previous query.

In the last exercise, we were able to get the average price of an app ($2.02365) using this query:

SELECT AVG(price)  
FROM fake\_apps;

Now, let’s edit this query so that it rounds this result to 2 decimal places.

This is a tricky one!

Checkpoint 3 Passed

ANSWER:

/\*SELECT name, ROUND(price, 0)

FROM fake\_apps;

\*/

SELECT ROUND(AVG(price), 2)

FROM fake\_apps;

**Group By I:** For instance, we might want to know the mean IMDb ratings for all movies each year. We could calculate each number by a series of queries with different WHERE statements, like so:

SELECT AVG(imdb\_rating)  
FROM movies  
WHERE year = 1999;  
   
SELECT AVG(imdb\_rating)  
FROM movies  
WHERE year = 2000;  
   
SELECT AVG(imdb\_rating)  
FROM movies  
WHERE year = 2001;

and so on.

Luckily, there’s a better way!

We can use GROUP BY to do this in a single step:

SELECT year,  
   AVG(imdb\_rating)  
FROM movies  
GROUP BY year  
ORDER BY year;

GROUP BY is a clause in SQL that is used with aggregate functions. It is used in collaboration with the SELECT statement to arrange identical data into groups.

The GROUP BY statement comes after any WHERE statements, but before ORDER BY or LIMIT.

### Instructions

**1.**

In the code editor, type:

SELECT price, COUNT(\*)   
FROM fake\_apps  
GROUP BY price;

Here, our aggregate function is COUNT() and we arranged price into groups.

What do you expect the result to be?

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

In the previous query, add a WHERE clause to count the total number of apps that have been downloaded more than 20,000 times, at each price.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Remove the previous query.

Write a new query that calculates the total number of downloads for each category.

Select category and SUM(downloads).

Checkpoint 4 Passed

SOLUTION:

/\*SELECT price, COUNT(\*)

FROM fake\_apps

WHERE downloads > 20000

GROUP BY price;

\*/

SELECT category, SUM(downloads)

FROM fake\_apps

GROUP BY category;

**Group By II**

Sometimes, we want to GROUP BY a calculation done on a column.

For instance, we might want to know how many movies have IMDb ratings that round to 1, 2, 3, 4, 5. We could do this using the following syntax:

SELECT ROUND(imdb\_rating),  
   COUNT(name)  
FROM movies  
GROUP BY ROUND(imdb\_rating)  
ORDER BY ROUND(imdb\_rating);

However, this query may be time-consuming to write and more prone to error.

SQL lets us use column reference(s) in our GROUP BY that will make our lives easier.

* 1 is the first column selected
* 2 is the second column selected
* 3 is the third column selected

and so on.

The following query is equivalent to the one above:

SELECT ROUND(imdb\_rating),  
   COUNT(name)  
FROM movies  
GROUP BY 1  
ORDER BY 1;

Here, the 1 refers to the first column in our SELECT statement, ROUND(imdb\_rating).

### Instructions

**1.**

Suppose we have the query below:

SELECT category,   
   price,  
   AVG(downloads)  
FROM fake\_apps  
GROUP BY category, price;

Write the exact query, but use column reference numbers instead of column names after GROUP BY.

Checkpoint 2 Passed

ANSWER:

SELECT category,

   price,

   AVG(downloads)

FROM fake\_apps

GROUP BY category, price;

Having: HAVING is very similar to WHERE. In fact, all types of WHERE clauses you learned about thus far can be used with HAVING.

We can use the following for the problem:

SELECT year,  
   genre,  
   COUNT(name)  
FROM movies  
GROUP BY 1, 2  
HAVING COUNT(name) > 10;

* When we want to limit the results of a query based on values of the individual rows, use WHERE.
* When we want to limit the results of a query based on an aggregate property, use HAVING.

HAVING statement always comes after GROUP BY, but before ORDER BY and LIMIT.

### Instructions

**1.**

Suppose we have the query below:

SELECT price,   
   ROUND(AVG(downloads)),  
   COUNT(\*)  
FROM fake\_apps  
GROUP BY price;

It returns the average downloads (rounded) and the number of apps – at each price point.

However, certain price points don’t have very many apps, so their average downloads are less meaningful.

Add a HAVING clause to restrict the query to price points that have more than 10 apps.

Checkpoint

ANSWER:

SELECT price,

   ROUND(AVG(downloads)),

   COUNT(\*)

FROM fake\_apps

GROUP BY price

HAVING COUNT(\*) > 10;

* COUNT(): count the number of rows
* SUM(): the sum of the values in a column
* MAX()/MIN(): the largest/smallest value
* AVG(): the average of the values in a column
* ROUND(): round the values in the column

Aggregate functions combine multiple rows together to form a single value of more meaningful information.

* GROUP BY is a clause used with aggregate functions to combine data from one or more columns.
* HAVING limit the results of a query based on an aggregate property.

**Combining Tables with SQL**

Combining tables manually is time-consuming. Luckily, SQL gives us an easy sequence for this: it’s called a JOIN.

If we want to combine orders and customers, we would type:

SELECT \*  
FROM orders  
JOIN customers  
  ON orders.customer\_id = customers.customer\_id;

Let’s break down this command:

1. The first line selects all columns from our combined table. If we only want to select certain columns, we can specify which ones we want.
2. The second line specifies the first table that we want to look in, orders
3. The third line uses JOIN to say that we want to combine information from orders with customers.
4. The fourth line tells us how to combine the two tables. We want to match orders table’s customer\_id column with customers table’s customer\_id column.

Because column names are often repeated across multiple tables, we use the syntax table\_name.column\_name to be sure that our requests for columns are unambiguous. In our example, we use this syntax in the ON statement, but we will also use it in the SELECT or any other statement where we refer to column names.

For example: Instead of selecting *all* the columns using \*, if we only wanted to select orders table’s order\_id column and customers table’s customer\_name column, we could use the following query:

SELECT orders.order\_id,  
   customers.customer\_name  
FROM orders  
JOIN customers  
  ON orders.customer\_id = customers.customer\_id;

**QUESTION:** Join orders table and subscriptions table and select all columns.

Make sure to join on the subscription\_id column.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Don’t remove the previous query.

Add a second query after your first one that only selects rows from the join where description is equal to ‘Fashion Magazine’.

Checkpoint

ANSWER:

-- First query

SELECT \*

FROM orders

JOIN subscriptions

  ON orders.subscription\_id =

      subscriptions.subscription\_id;

-- Second query

SELECT \*

FROM orders

JOIN subscriptions

  ON orders.subscription\_id = subscriptions.subscription\_id

WHERE subscriptions.description = 'Fashion Magazine';

**Inner Joins:**

Suppose we are working for The Codecademy Times, a newspaper with two types of subscriptions:

* print newspaper
* online articles

Some users subscribe to just the newspaper, some subscribe to just the online edition, and some subscribe to both.

There is a newspaper table that contains information about the newspaper subscribers.

Count the number of subscribers who get a print newspaper using COUNT().

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Don’t remove your previous query.

There is also an online table that contains information about the online subscribers.

Count the number of subscribers who get an online newspaper using COUNT().

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Don’t remove your previous queries.

Join newspaper table and online table on their id columns (the unique ID of the subscriber).

How many rows are in this table?

Checkpoint

**ANSWER:**

SELECT COUNT(\*)

FROM newspaper;

SELECT COUNT(\*)

FROM online;

SELECT COUNT(\*)

FROM newspaper

JOIN online

  ON newspaper.id = online.id;

**Left Joins:** What if we want to combine two tables and keep some of the un-matched rows?

SQL lets us do this through a command called LEFT JOIN. A *left join* will keep all rows from the first table, regardless of whether there is a matching row in the second table.

SELECT \*  
FROM table1  
LEFT JOIN table2  
  ON table1.c2 = table2.c2;

1. The first line selects all columns from both tables.
2. The second line selects table1 (the “left” table).
3. The third line performs a LEFT JOIN on table2 (the “right” table).
4. The fourth line tells SQL how to perform the join (by looking for matching values in column c2).
5. **1.**
6. Let’s return to our newspaper and online subscribers.
7. Suppose we want to know how many users subscribe to the print newspaper, but not to the online.
8. Start by performing a left join of newspaper table and online table on their id columns and selecting all columns.
9. Checkpoint 2 Passed
10. Stuck? Get a hint
11. **2.**
12. Don’t remove your previous query.
13. In order to find which users do *not* subscribe to the online edition, we need to add a WHERE clause.
14. Add a second query after your first one that adds the following WHERE clause and condition:
15. WHERE online.id IS NULL
16. This will select rows where there was no corresponding row from the online table.
17. Checkpoint

**ANSWER:**

SELECT \*

FROM newspaper

LEFT JOIN online

  ON newspaper.id = online.id;

SELECT \*

FROM newspaper

LEFT JOIN online

 ON newspaper.id = online.id

WHERE online.id IS NULL;

**Primary Key vs Foreign Key**

Let’s return to our example of the magazine subscriptions. Recall that we had three tables: orders, subscriptions, and customers.

Each of these tables has a column that uniquely identifies each row of that table:

* order\_id for orders
* subscription\_id for subscriptions
* customer\_id for customers

These special columns are called **primary keys**.

Primary keys have a few requirements:

* None of the values can be NULL.
* Each value must be unique (i.e., you can’t have two customers with the same customer\_id in the customers table).
* A table can not have more than one primary key column.

Let’s reexamine the orders table:

| **order\_id** | **customer\_id** | **subscription\_id** | **purchase\_date** |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 2017-01-01 |
| 2 | 2 | 2 | 2017-01-01 |
| 3 | 3 | 1 | 2017-01-01 |

Note that customer\_id (the primary key for customers) and subscription\_id (the primary key for subscriptions) both appear in this.

When the primary key for one table appears in a different table, it is called a **foreign key**.

So customer\_id is a primary key when it appears in customers, but a foreign key when it appears in orders.

In this example, our primary keys all had somewhat descriptive names. Generally, the primary key will just be called id. Foreign keys will have more descriptive names.

Why is this important? The most common types of joins will be joining a foreign key from one table with the primary key from another table. For instance, when we join orders and customers, we join on customer\_id, which is a foreign key in orders and the primary key in customers.

### Instructions

**1.**

Suppose Columbia University has two tables in their database:

* The classes table contains information on the classes that the school offers. Its primary key is id.
* The students table contains information on all students in the school. Its primary key is id. It contains the foreign key class\_id, which corresponds to the primary key of classes.

Perform an inner join of classes and students using the primary and foreign keys described above, and select all the columns.

Checkpoint 2 Passed

**ANSWER:**

SELECT \*

FROM classes

JOIN students

  ON classes.id = students.class\_id;

Cross Join

So far, we’ve focused on matching rows that have some information in common.

Sometimes, we just want to combine all rows of one table with all rows of another table.

For instance, if we had a table of shirts and a table of pants, we might want to know all the possible combinations to create different outfits.

Our code might look like this:

SELECT shirts.shirt\_color,  
   pants.pants\_color  
FROM shirts  
CROSS JOIN pants;

* The first two lines select the columns shirt\_color and pants\_color.
* The third line pulls data from the table shirts.
* The fourth line performs a CROSS JOIN with pants.

Notice that cross joins don’t require an ON statement. You’re not really joining on any columns!

If we have 3 different shirts (white, grey, and olive) and 2 different pants (light denim and black), the results might look like this:

| **shirt\_color** | **pants\_color** |
| --- | --- |
| white | light denim |
| white | black |
| grey | light denim |
| grey | black |
| olive | light denim |
| olive | black |

3 shirts × 2 pants = 6 combinations!

This clothing example is fun, but it’s not very practically useful.

A more common usage of CROSS JOIN is when we need to compare each row of a table to a list of values.

Let’s return to our newspaper subscriptions. This table contains two columns that we haven’t discussed yet:

* start\_month: the first month where the customer subscribed to the print newspaper (i.e., 2 for February)
* end\_month: the final month where the customer subscribed to the print newspaper

Suppose we wanted to know how many users were subscribed during each month of the year. For each month (1, 2, 3) we would need to know if a user was subscribed. Follow the steps below to see how we can use a CROSS JOIN to solve this problem.

### Instructions

**1.**

Eventually, we’ll use a cross join to help us, but first, let’s try a simpler problem.

Let’s start by counting the number of customers who were subscribed to the newspaper during March.

Use COUNT(\*) to count the number of rows and a WHERE clause to restrict to two conditions:

* start\_month <= 3
* end\_month >= 3

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Don’t remove the previous query.

The previous query lets us investigate one month at a time. In order to check across all months, we’re going to need to use a cross join.

Our database contains another table called months which contains the numbers between 1 and 12.

Select all columns from the cross join of newspaper and months.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Don’t remove your previous queries.

Create a third query where you add a WHERE statement to your cross join to restrict to two conditions:

* start\_month <= month
* end\_month >= month

This will select all months where a user was subscribed.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Don’t remove your previous queries.

Create a final query where you aggregate over each month to count the number of subscribers.

Fill in the blanks in the following query:

SELECT month,  
  COUNT(\*)  
FROM \_\_\_\_\_\_\_\_  
CROSS JOIN \_\_\_\_\_\_\_\_  
WHERE \_\_\_\_\_\_\_\_ AND \_\_\_\_\_\_\_\_  
GROUP BY \_\_\_\_\_\_\_\_;

Checkpoint

**ANSWER:**

SELECT COUNT(\*)

FROM newspaper

WHERE start\_month <= 3

  AND end\_month >= 3;

SELECT \*

FROM newspaper

CROSS JOIN months;

SELECT \*

FROM newspaper

CROSS JOIN months

WHERE start\_month <= month

  AND end\_month >= month;

SELECT month,

  COUNT(\*)

FROM newspaper

CROSS JOIN months

WHERE start\_month <= month

  AND end\_month >= month

GROUP BY month;

Union

Sometimes we just want to stack one dataset on top of the other. Well, the UNION operator allows us to do that.

Suppose we have two tables and they have the same columns.

table1:

| **pokemon** | **type** |
| --- | --- |
| Bulbasaur | Grass |
| Charmander | Fire |
| Squirtle | Water |

table2:

| **pokemon** | **type** |
| --- | --- |
| Snorlax | Normal |

If we combine these two with UNION:

SELECT \*  
FROM table1  
UNION  
SELECT \*  
FROM table2;

The result would be:

| **pokemon** | **type** |
| --- | --- |
| Bulbasaur | Grass |
| Charmander | Fire |
| Squirtle | Water |
| Snorlax | Normal |

SQL has strict rules for appending data:

* Tables must have the same number of columns.
* The columns must have the same data types in the same order as the first table.

### Instructions

**1.**

Let’s return to our newspaper and online subscriptions. We’d like to create one big table with both sets of data.

Use UNION to stack the newspaper table on top of the online table.

Checkpoint 2 Passed

**ANSWER:**

SELECT COUNT(\*)

FROM newspaper

WHERE start\_month <= 3

  AND end\_month >= 3;

SELECT \*

FROM newspaper

CROSS JOIN months;

SELECT \*

FROM newspaper

CROSS JOIN months

WHERE start\_month <= month

  AND end\_month >= month;

SELECT month,

  COUNT(\*)

FROM newspaper

CROSS JOIN months

WHERE start\_month <= month

  AND end\_month >= month

GROUP BY month;

With

Often times, we want to combine two tables, but one of the tables is the result of another calculation.

Let’s return to our magazine order example. Our marketing department might want to know a bit more about our customers. For instance, they might want to know how many magazines each customer subscribes to. We can easily calculate this using our orders table:

SELECT customer\_id,  
   COUNT(subscription\_id) AS 'subscriptions'  
FROM orders  
GROUP BY customer\_id;

This query is good, but a customer\_id isn’t terribly useful for our marketing department, they probably want to know the customer’s name.

We want to be able to join the results of this query with our customers table, which will tell us the name of each customer. We can do this by using a WITH clause.

WITH previous\_results AS (  
   SELECT ...  
   ...  
   ...  
   ...  
)  
SELECT \*  
FROM previous\_results  
JOIN customers  
  ON \_\_\_\_\_ = \_\_\_\_\_;

* The WITH statement allows us to perform a separate query (such as aggregating customer’s subscriptions)
* previous\_results is the alias that we will use to reference any columns from the query inside of the WITH clause
* We can then go on to do whatever we want with this temporary table (such as join the temporary table with another table)

Essentially, we are putting a whole first query inside the parentheses () and giving it a name. After that, we can use this name as if it’s a table and write a new query using the first query.

### Instructions

**1.**

Place the whole query below into a WITH statement, inside parentheses (), and give it name previous\_query:

SELECT customer\_id,  
   COUNT(subscription\_id) AS 'subscriptions'  
FROM orders  
GROUP BY customer\_id

Join the temporary table previous\_query with customers table and select the following columns:

* customers.customer\_name
* previous\_query.subscriptions

Check the answer in the hint below.

Checkpoint 2 Passed

**ANSWER:**

WITH previous\_query AS (

SELECT customer\_id,

       COUNT(subscription\_id) as subscriptions

FROM orders

GROUP BY customer\_id

)

SELECT customers.customer\_name,

previous\_query.subscriptions

FROM previous\_query /\*table name\*/

JOIN customers /\*joining  table\*/

  ON customers.customer\_id = previous\_query.customer\_id;

Let’s summarize what we’ve learned so far:

* JOIN will combine rows from different tables if the join condition is true.
* LEFT JOIN will return every row in the left table, and if the join condition is not met, NULL values are used to fill in the columns from the right table.
* Primary key is a column that serves a unique identifier for the rows in the table.
* Foreign key is a column that contains the primary key to another table.
* CROSS JOIN lets us combine all rows of one table with all rows of another table.
* UNION stacks one dataset on top of another.
* WITH allows us to define one or more temporary tables that can be used in the final query.

### **CHAPTER: 28 PostgreSQL Database:**

# What is CRUD?

Create, Read, Update, Delete. A model should have the ability to perform at most these four functions in order to be complete. If an action cannot be described by one of these four operations, then it should potentially be a model of its own.

The CRUD paradigm is common in constructing web applications, because it provides a memorable framework for reminding developers of how to construct full, usable models. For example, let’s imagine a system to keep track of library books. In this hypothetical library database, we can imagine that there would be a **books** resource, which would store **book** objects. Let’s say that the **book** object looks like this:

“book”: {  
  "id": <Integer>,  
  “title”: <String>,  
  “author”: <String>,  
  “isbn”: <Integer>  
}

Create — This would consist of a function which we would call when a new library book is being added to the catalog. The program calling the function would supply the values for **“title”**, **“author”**, and **“isbn”**. After this function is called, there should be a new entry in the **books** resource corresponding to this new book. Additionally, the new entry is assigned a unique **id**, which can be used to access this resource later.

Read — This would consist of a function which would be called to see all of the books currently in the catalog. This function call would not alter the books in the catalog - it would simply retrieve the resource and display the results. We would also have a function to retrieve a single book, for which we could supply the title, author, or ISBN. Again, this book would not be modified, only retrieved.

Update — There should be a function to call when information about a book must be changed. The program calling the function would supply the new values for **“title”**, **“author”**, and **“isbn”**. After the function call, the corresponding entry in the **books** resource would contain the new fields supplied.

Delete — There should be a function to call to remove a library book from the catalog. The program calling the function would supply one or more values (**“title”**, **“author”**, and/or **“isbn”**) to identify the book, and then this book would be removed from the **books** resource. After this function is called, the **books** resource should contain all of the books it had before, except for the one just deleted.

### CRUD and REST

In a REST environment, CRUD often corresponds to the HTTP methods POST, GET, PUT, and DELETE, respectively. These are the fundamental elements of a persistent storage system.

#### Create

To create resources in a REST environment, we most commonly use the HTTP POST method. POST creates a new resource of the specified resource type.

For example, let’s imagine that we are adding a new food item to the stored list of dishes for this restaurant, and the **dish** objects are stored in a **dishes** resource. If we wanted to create the new item, we would use a POST request:

Request:

POST http://www.myrestaurant.com/dishes/

Body -

{  
  "dish": {  
    "name": “Avocado Toast”,  
    "price": 8  
  }  
}

This creates a new item with a **name** value of **“Avocado Toast”** and a **price** value of 8. Upon successful creation, the server should return a header with a link to the newly-created resource, along with a HTTP response code of 201 (CREATED).

Response:

Status Code - 201 (CREATED)

Body -

{  
  "dish": {  
    "id": 1223,  
    "name": “Avocado Toast”,  
    "price": 8  
  }  
}

From this response, we see that the **dish** with **name** “Avocado Toast” and **price** 8 has been successfully created and added to the **dishes** resource.

#### Read

To read resources in a REST environment, we use the GET method. Reading a resource should never change any information - it should only retrieve it. If you call GET on the same information 10 times in a row, you should get the same response on the first call that you get on the last call.

GET can be used to read an entire list of items:

Request:

GET http://www.myrestaurant.com/dishes/

Response: Status Code - 200 (OK)

Body -

{  
  "dishes": [  
    {  
      "id": 1,  
      "name": “Spring Rolls”,  
      "price": 6  
    },  
    {  
      "id": 2,  
      "name": “Mozzarella Sticks”,  
      "price": 7  
    },  
    ...  
    {  
      "id": 1223,  
      "name": “Avocado Toast”,  
      "price": 8  
    },  
    {  
      "id": 1224,  
      "name": “Muesli and Yogurt”,  
      "price": 5  
    }  
  ]  
}

GET requests can also be used to read a specific item, when its **id** is specified in the request:

Request:

GET http://www.myrestaurant.com/dishes/1223

Response: Status Code - 200 (OK)

Body -

{  
  "id": 1223,  
  "name": “Avocado Toast”,  
  "price": 8  
}

After this request, no information has been changed in the database. The item with **id** 1223 has been retrieved from the **dishes** resource, and not modified. When there are no errors, GET will return the HTML or JSON of the desired resource, along with a 200 (OK) response code. If there is an error, it most often will return a 404 (NOT FOUND) response code.

#### Update

PUT is the HTTP method used for the CRUD operation, Update.

For example, if the price of Avocado Toast has gone up, we should go into the database and update that information. We can do this with a PUT request.

Request:

PUT http://www.myrestaurant.com/dishes/1223

Body -

{  
  "dish": {  
    "name": “Avocado Toast”,  
    "price": 10  
  }  
}

This request should change the item with **id** 1223 to have the attributes supplied in the request body. This **dish** with **id** 1223 should now still have the **name** “Avocado Toast”, but the **price** value should now be 10, whereas before it was 8.

Response: Status Code - 200 (OK)

Body -

{  
  "dish": {  
    "name": “Avocado Toast”,  
    "price": 10  
  }  
}

The response includes a Status Code of 200 (OK) to signify that the operation was successful. Optionally, the response could use a Status Code of 204 (NO CONTENT) and not include a response body. This decision depends on the context.

#### Delete

The CRUD operation Delete corresponds to the HTTP method DELETE. It is used to remove a resource from the system.

Let’s say that the world avocado shortage has reached a critical point, and we can no longer afford to serve this modern delicacy at all. We should go into the database and delete the item that corresponds to “Avocado Toast”, which we know has an **id** of 1223.

Request:

DELETE http://www.myrestaurant.com/dishes/1223

Such a call, if successful, returns a response code of 204 (NO CONTENT), with no response body. The **dishes** resource should no longer contain the **dish** object with **id** 1223.

Response: Status Code - 204 (NO CONTENT)

Body - None

Calling GET on the **dishes** resource after this DELETE call would return the original list of dishes with the **{"id": 1223, "name": “Avocado Toast”, "price": 10}** entry removed. All other **dish** objects in the **dishes** resource should remain unchanged. If we tried to call a GET on the item with **id** 1223, which we just deleted, we would receive a 404 (NOT FOUND) response code and the state of the system should remain unchanged.

Calling DELETE on a resource that does not exist should not change the state of the system. The call should return a 404 response code (NOT FOUND) and do nothing.

### CRUD Practice

The functions to Create, Read, Update, and Delete resources are fundamental components of a usable storage model. You have now seen a couple of examples for how the CRUD paradigm can help us design systems. Now, try to use CRUD to list out routes for a new example model. Imagine we are trying to design a system that keeps track of workout classes, including the name of each class, who teaches it, and the duration of the class. An example **class** object would look like:

{  
  "class": {  
    "id": 1      
    "name": “Pure Strength”,  
    “trainer”: “Bicep Bob”,  
    "duration": 1.5  
   }  
}

All of the classes are stored in a **classes** resource at **www.musclecademy.com/classes**.

For each CRUD operation, write out answers to the following questions:

* What routes would you need to implement to provide your workout class model with this CRUD functionality and what are their corresponding HTTP verbs?
* What effect would each route have on the database?
* What response body would each route return?
* What response code would each route return?

### CRUD Practice Answers

1) Create

**Route**: POST /classes

**Effect on Database**: Adds the class provided in the request body to the database

**Response Body**: **{ "class": The Newly-Created Class }**

**Success Response Code**: 201

2) Read (All Classes)

**Route**: GET /classes

**Effect on Database**: None

**Response Body**: **{ "classes": [ Array of All Saved Classess ] }**

**Success Response Code**: 200

3) Read (One Class)

**Route**: GET /classes/:id

**Effect on Database**: None

**Response Body**: **{ "class": The class with the specified ID }**

**Success Response Code**: 200

4) Update

**Route**: PUT /classes/:id

**Effect on Database**: Updates the class with the specified ID to have the class information provided in the request body

**Response Body**: **{ "class": The updated class now saved in the database }**

**Success Response Code**: 200

5) Delete

**Route**: DELETE /classes/:id

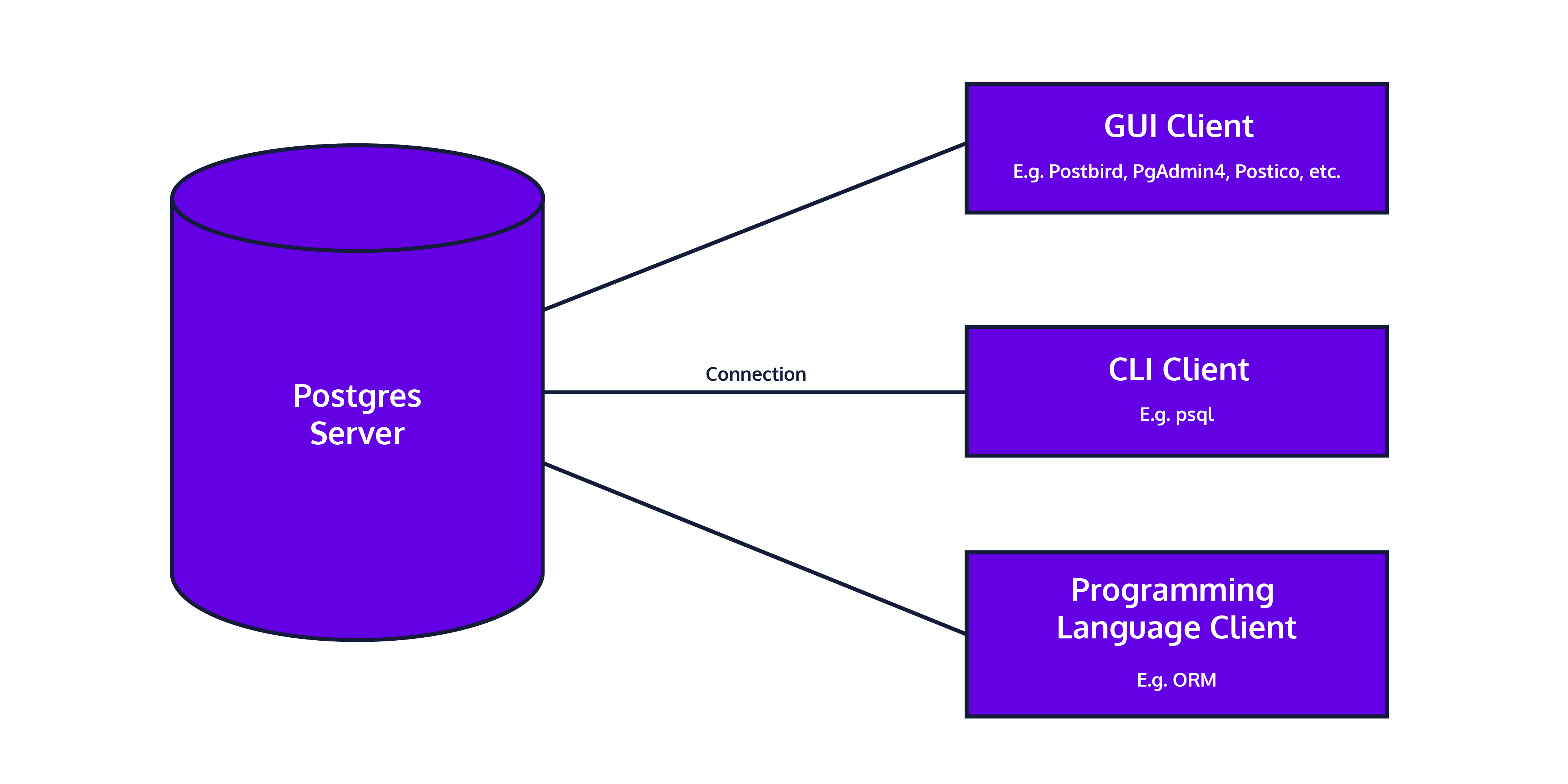
**Effect on Database**: Removes the class with the specified ID from the database

**Response Body**: None

**Success Response Code**: 204

## What is Postgres?

Postgres is “a powerful, open-source object-relational database system with over 30 years of active development that has earned it a strong reputation for reliability, feature robustness, and performance.” Postgres is currently used in production by many modern technology companies, be it small startups or large organizations such as Apple, Instagram, Twitch, and Reddit.

Postgres itself is a database “server.” There are several ways to connect to Postgres via “clients,” including [GUIs](https://en.wikipedia.org/wiki/Graphical_user_interface), [CLIs](https://en.wikipedia.org/wiki/Command-line_interface), and programming languages often via [ORMs](https://en.wikipedia.org/wiki/Object-relational_mapping). In order to run and use Postgres on your own computer, you will need to set up both a Postgres server and a client.

Postgres.app is a full-featured PostgreSQL installation packaged as a standard Mac app. This is helpful if you are looking for an all-in-one download option that has everything you need to get started with PostgreSQL, including popular extensions like PostGIS for geo data and plv8 for JavaScript.

pgAdmin is a popular open-source management tool for PostgreSQL. This is helpful if you are interested in using a graphical interface to help with the creation, maintenance, and use of database objects.

### A Note on Running SQL Locally

Before we get started, it’s important to discuss how running SQL on your own machine is slightly different than running SQL on Codecademy’s website. Consider an exercise on Codecademy that has two steps. The first step asks you to create a table, and the second step asks you to add a row to that table. To pass the first step, you might write something like this and then click run:

CREATE TABLE students (  
   name TEXT,   
   age INTEGER  
);

Then to pass the second step, you would add on to your file and hit run again, like so:

CREATE TABLE students (  
   name TEXT,   
   age INTEGER  
);  
   
INSERT INTO students (name, age)   
VALUES ('Zack Morris', 18);

Notice that technically you’ve now run the **CREATE TABLE** line twice — once the first time you hit the run button, and again when you hit the run button to pass the second checkpoint.

If you were running this code on your own computer, this would cause an error — the second time you run the **CREATE TABLE** line, you would see an error because the table was already created. When working on your own computer, it’s a better idea to run each new line separately, rather than adding a new line to the bottom of your file and running the entire file again.

If you were working through this example on your own computer, you should first run

CREATE TABLE students (  
   name TEXT,   
   age INTEGER  
);

and then run this line separately

INSERT INTO students (name, age)   
VALUES ('Zack Morris', 18);

That being said, sometimes you want to save your SQL statements in a single file for later use. For example, we have provided you with solution code in a **.sql** file. If you were to open your PostgreSQL client, and import that **.sql** file (usually done through the File menu of a client), you could run all of our solution code at once.

# CHAPTER: 29 Introduction: Designing Relational Databases

Which of the following would be a good choice as the primary key for a table?

 is completely valid for a primary key to be made up of multiple fields, as long as the combination uniquely identifies each record. This is known as a composite key. QuarterID and courseID is good choice for primary key.

Which of the following best describes the relationship between a table that stores information about movies and a table that stores information about actors and actresses?

That’s right! An actor or actress can star in multiple movies, and a movie can have multiple actors and actresses in it.Many – to - Many is best choice for this.

### **WHAT IS A DATABASE SCHEMA?:** a Database Schema describes the structure of a database. Database schemas generally contain information about table/column names, data types/constraints, relationships between tables, and user roles.

**Creating Your Tables:** A database table is made up of columns of information. Each column is assigned a name and data type. To create a table in PostgreSQL, we would use the following SQL syntax:

CREATE TABLE person (  
  first\_name varchar(15),  
  last\_name varchar(15),  
  age integer,  
  …  
  ssn char(9)  
);

In this example, the table name is person, and its column names include first\_name, last\_name and ssn. You can think of a column name representing a property, attribute or field in the table.

Each column name is associated with a column type which is a data type such as numeric, character, boolean or other interesting types. Here is a summary of common data types, what they represent, their sample values and how they display on Postgres:

| **Data Type** | **Representation** | **Value** | **Display** |
| --- | --- | --- | --- |
| integer | whole number | 617 | 617 |
| decimal | floating-point number | 26.17345 | 26.17345 |
| money | fixed floating-point number with 2 decimal places | 6.17 | $6.17 |
| boolean | logic | TRUE, FALSE | t, f |
| char(n) | fixed-length string removes trailing blanks | ‘123 ‘ | ‘123’ |
| varchar(n) | variable-length string | ‘123 ‘ | ‘123 ‘ |
| text | unlimited-length string | ‘123 ‘ |  |

Example: Create a book table with these columns:

* title for the book title, a varchar of 100 characters
* isbn for the book isbn, a varchar of 50 characters
* pages for the number of pages in the book, an integer
* price for the book price, of money type
* description for the book description, a varchar of 256 characters
* publisher for the book publisher name, a varchar of 100 characters

Checkpoint

Solution: CREATE TABLE book (

title varchar(100),

isbn varchar(50),

pages integer,

price money,

description varchar(256),

publisher varchar(100)

);

Table

### Query Results

### Database Schema

### book

|  |  |
| --- | --- |
| **name** | **type** |
| title | character varying |
| isbn | character varying |
| pages | integer |
| price | money |
| description | character varying |
| publisher | character varying |

|  |
| --- |
| **Rows: 0** |

**Querying Your Tables:** To insert data into a PostgreSQL table, use this syntax:

INSERT INTO table\_name VALUES (  
  column\_one\_value,  
  column\_two\_value,  
  …  
  column\_N\_value  
);

To query a table to return all the columns, type:

SELECT \* from table\_name;

the book table with the following data:

* title, 'Postgres for Beginners'
* isbn, '0-5980-6249-1'
* pages, 25
* price, 4.99
* description, 'Learn Postgres the Easy Way'
* publisher, 'Codecademy Publishing'

Then, add a query to the book table to validate all the data you entered based on the book title.

2: query the book table for the isbn '0-5980-6249-1'.

Did you notice that two books are returned based on the same isbn value? What would fix this problem so that only one unique row is returned per isbn value?

**Answer:**

INSERT INTO book VALUES (

  'Postgres for Beginners',

  '0-5980-6249-1',

  25,

  4.99,

  'Learn Postgres the Easy Way',

  'Codecademy Publishing'

);

SELECT \* FROM book WHERE title = 'Postgres for Beginners';

SELECT \* FROM book WHERE isbn = '0-5980-6249-1';

### Query Results

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **title** | **isbn** | **pages** | **price** | **description** | **publisher** |
| Postgres for Beginners | 0-5980-6249-1 | 25 | $4.99 | Learn Postgres the Easy Way | Codecademy Publishing |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **title** | **isbn** | **pages** | **price** | **description** | **publisher** |
| Postgres Made Easy | 0-5980-6249-1 | 30 | $9.99 | A great book for beginners to learn how to manage PostgreSQL | Codecademy Press |
| Postgres for Beginners | 0-5980-6249-1 | 25 | $4.99 | Learn Postgres the Easy Way | Codecademy Publishing |

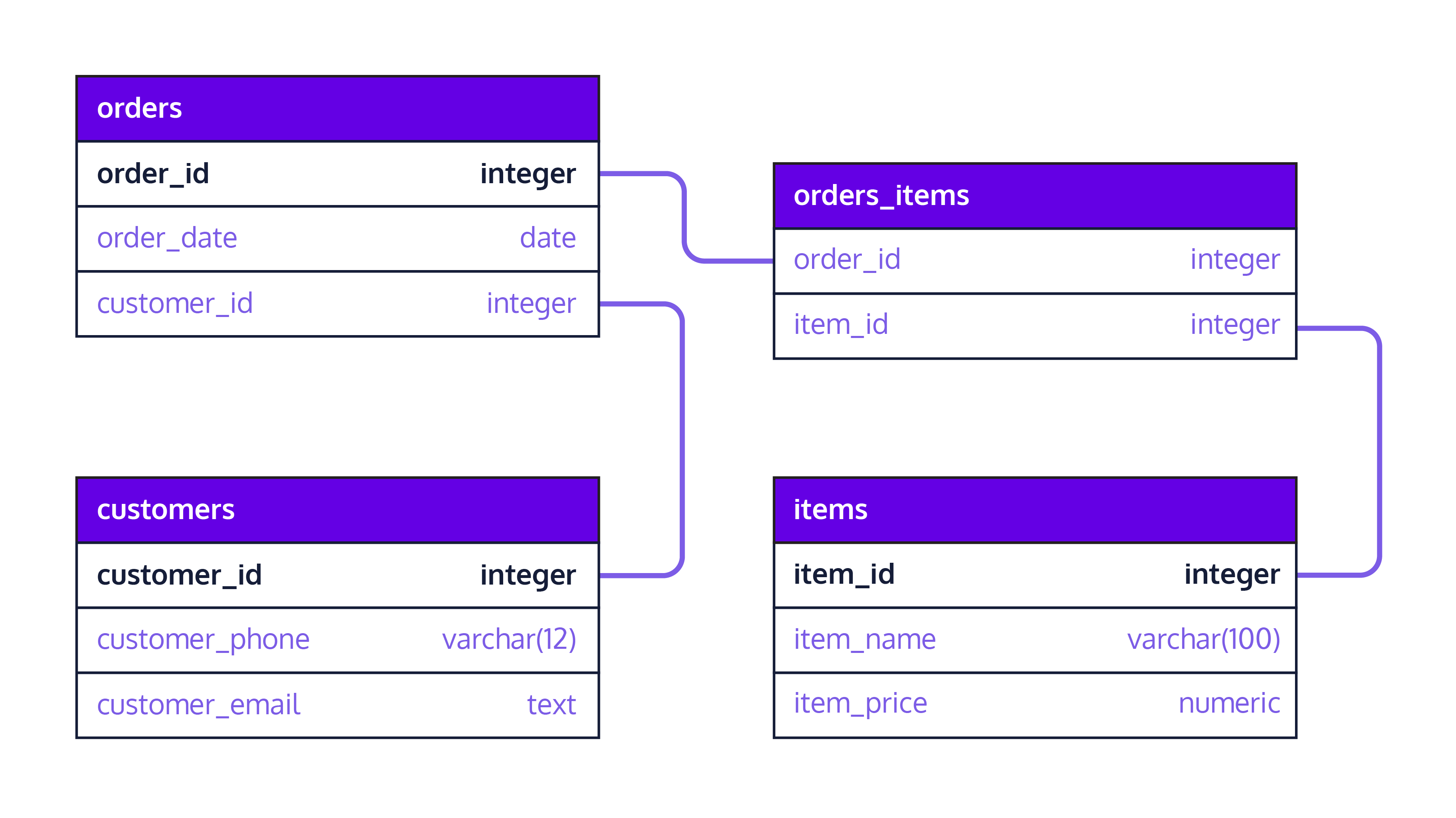
### PostgreSQL Naming Conventions

There are several naming conventions in PostgreSQL:

* Column names should be lower case with underscores between words (eg., birth\_date).
* The primary key of a table is often named id.

### Database Schema Design Tools

The image above shows an example database schema that was illustrated using a design tool. The schema contains four tables named orders, customers, orders\_items, and items, respectively. The lines in the visualization show how columns in different tables are related to each other.



### Database Tables

Database schemas describe the number of tables in a database and the data that they contain. Database tables should usually relate to a single construct described by an identifier called a primary key. For example, in an orders table where every order has a customer, it would be inefficient to write the complete details of each customer for every order they made. Instead, a separate customers table could contain that information.

### PostgreSQL Variable Types

When designing a database schema in PostgreSQL, every column must have a data type. For example, the table created in the example code has three columns with types integer, varchar, and boolean, respectively. This helps preserve data integrity over time by restricting the data that can be entered into the table.

CREATE TABLE people (  
    age integer,  
    name varchar,  
    is\_citizen boolean  
);

### Primary Keys

The primary key of a database table is a column or group of columns that can be used to uniquely identify every row of the table. For example, a table of students might have a primary key named student\_id, which contains unique ID numbers for each student.

### Composite Primary Keys

A primary key made up of multiple columns is called a composite primary key. Composite primary keys can be used when no single column uniquely identifies a row of a table, but multiple columns do. For example, the example table shown here could have a composite primary key made up of student\_id and course\_id.

| student\_id| course\_id | grade |  
|-----------|-----------|-------|  
| 1         | 503       | A     |  
| 1         | 401       | A-    |  
| 2         | 503       | B     |

### Foreign Keys

Database tables can have foreign key(s), which reference the primary key of another table. For example, an orders table may contain a column named customer\_id, which references an id column in a separate customers table. In this case, the customer\_id column in the orders table could be designated as a foreign key.

### Creating a Primary Key

In PostgreSQL, a column can be designated as a primary key using the PRIMARY KEY keyword. For example, the example code shows two CREATE TABLE statements. In the orders table, order\_id is designated as a primary key. In the orders\_items table, order\_id and item\_id are used to designate a composite primary key.

CREATE TABLE orders(  
order\_id integer PRIMARY KEY,  
order\_date date,  
);  
   
CREATE TABLE orders\_items(  
order\_id integer,  
order\_date date,  
item\_id integer,  
PRIMARY KEY (order\_id,item\_id)  
);

### Creating a Foreign Key

In PostgreSQL a foreign key is defined by using the REFERENCES keyword. For example, the code here shows two CREATE TABLE statements. The orders table contains a foreign key called customer\_id, which references the id column of the customers table.

CREATE TABLE orders (  
  order\_number integer,   
  customer\_id integer REFERENCES customers(id),  
);  
   
CREATE TABLE customers (  
  id integer PRIMARY KEY,  
  name text  
);

### One-to-One Database Relationships

In a relational database, two tables have a one-to-one relationship. For example, a person may only have one passport assigned to them. Conversely, a passport may only be issued to one person. A car may only have one vehicle identification number assigned to it and vice-versa. A driver may only have one driver’s license issued to them in their home state. Let’s elaborate on the last example further. Let’s say we have a driver table with the following columns:

* name
* address
* date\_of\_birth
* license\_id

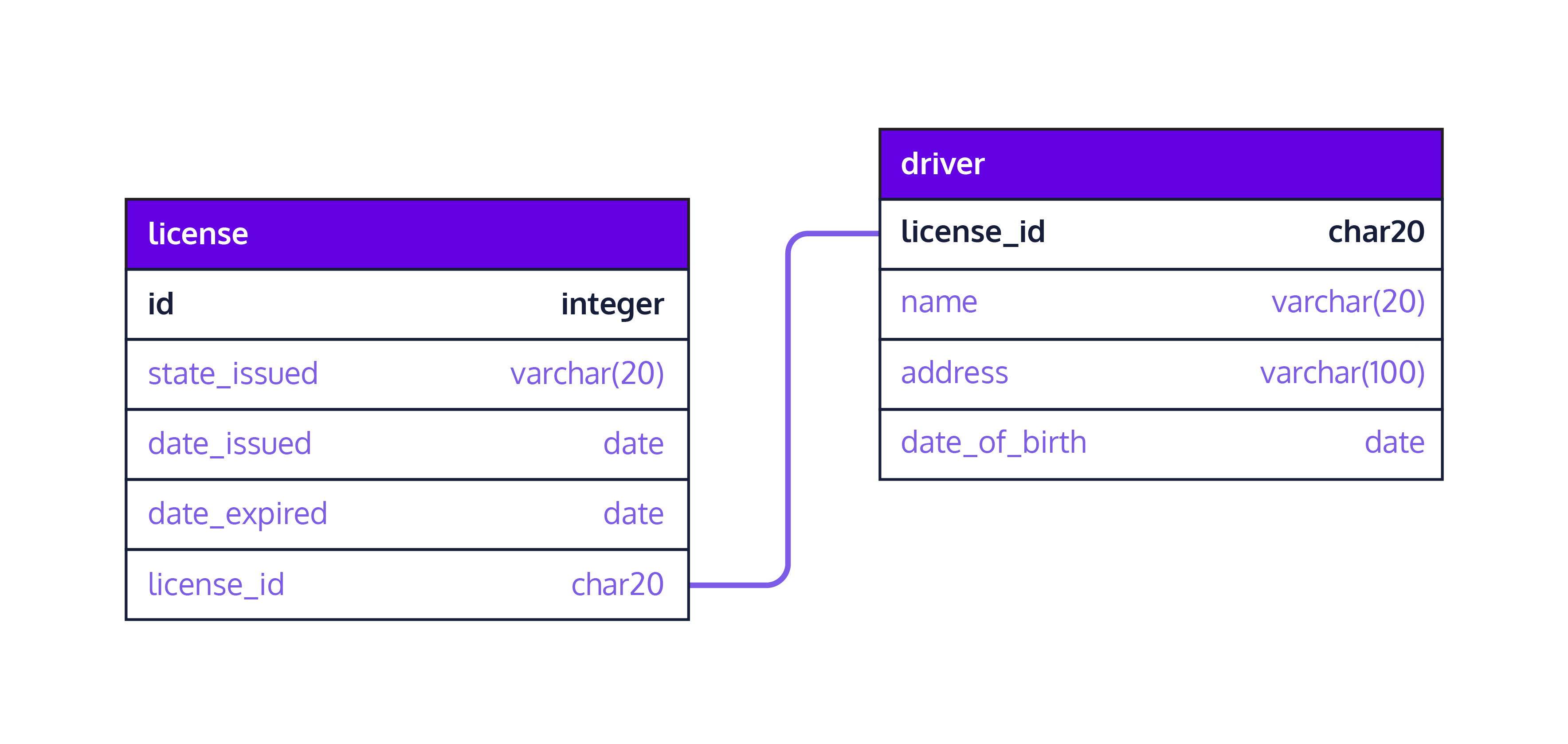
We also have a license table with the following columns:

* id
* state\_issued
* date\_issued
* date\_expired

In the driver table, the primary key that uniquely identifies a driver would be the license\_id. Similarly, the primary key that uniquely identifies a driver’s license in the license table would be the id itself. To establish a one-to-one relationship in PostgreSQL between these two tables, we need to designate a foreign key in one of the tables. We can pick the license\_id from driver to be the foreign key in the license table. However, doing this is not enough to ensure that duplicate rows will not exist in the license table.

To enforce a strictly one-to-one relationship in PostgreSQL, we need another keyword, UNIQUE. By appending this keyword to the declaration of the foreign key, we should be all set.

license\_id char(20) REFERENCES driver(license\_id) UNIQUE

The full PostgreSQL script for creating these two tables is as follows:

CREATE TABLE driver (  
    license\_id char(20) PRIMARY KEY,  
    name varchar(20),  
    address varchar(100),  
    date\_of\_birth date  
);        
   
CREATE TABLE license (  
    id integer PRIMARY KEY,  
    state\_issued varchar(20),  
    date\_issued date,  
    date\_expired  date,  
    license\_id char(20) REFERENCES driver(license\_id) UNIQUE  
);

### Many-to-One Database Relationships

In a relational database, two tables have a many-to-one relationship if each row in one table links to multiple rows of the other table. For example, a table of customers and a table of orders would have a many-to-one relationship if each customer can make multiple orders, but each order can only be associated with one customer.

### Many-to-Many Database Relationships

In a relational database, two tables have a many-to-many relationship if each row in one table can link to multiple rows in the other table, and vice versa. For example, a table of songs and a table of artists would likely have a many-to-many relationship because songs can have multiple artists and artists can have multiple songs.

### Implementing a Many-to-One Database Relationship

A many-to-one relationship can be implemented in PostgreSQL by creating a foreign key that references the primary key of another table. For example, the code here implements a many-to-one relationship between an orders and a customers table, where each customer can be associated with multiple orders.

CREATE TABLE orders (  
  order\_number integer,   
  customer\_id integer REFERENCES customers(id),  
);  
   
CREATE TABLE customers (  
  id integer PRIMARY KEY,  
  name text  
);

### Implementing a Many-to-Many Database Relationship

A many-to-many database relationship can be implemented in PostgreSQL using a third cross-reference table. This table should have two foreign keys referencing the primary keys of the related tables, as well as a composite primary key made up of the foreign key columns. For example, the code here implements a many-to-many relationship between a songs and an artists table.

To implement a many-to-many relationship in a relational database, we would create a third cross-reference table also known as a join table. It will have these two constraints:

* foreign keys referencing the primary keys of the two member tables.
* a composite primary key made up of the two foreign keys.

Let’s elaborate on this further with the recipe and ingredient many-to-many relationship. Let’s say a recipe table has the following columns:

* id (primary key)
* name
* serving\_size
* preparation\_time
* cook\_time

An ingredient table has the following columns:

* id (primary key)
* name
* amount

A third cross-reference table, recipes\_ingredients, will support the following columns:

* recipe\_id (foreign key referencing recipe table’s id)(primary key)
* ingredient\_id (foreign key referencing ingredient table’s id) (primary key)

Both recipe\_id and ingredient\_id also serve as a composite primary key for recipes\_ingredients.

Many-to-many relationship database schema
showing the relationship between recipes and ingredientsMany-to-many relationship database schema

Implementing a One-to-One Database Relationship

one-to-one relationship can be enforced in PostgreSQL by first creating a many-to-one relationship via a foreign key, then implementing a UNIQUE constraint on the foreign key. For example, the code here implements a one-to-one relationship between tables named employees and contact\_info.

CREATE TABLE employees (  
  id integer PRIMARY KEY,   
  name varchar(100)  
);  
   
CREATE TABLE contact\_info (  
  employee\_id integer REFERENCES employees(id) UNIQUE,  
  email text,  
  phone\_number varchar(9)  
);

**WHAT ARE DATABASE KEYS?: Suppose we have successfully created a database schema for a book inventory with three standalone tables - book, chapter and author. See image on the right pane.**

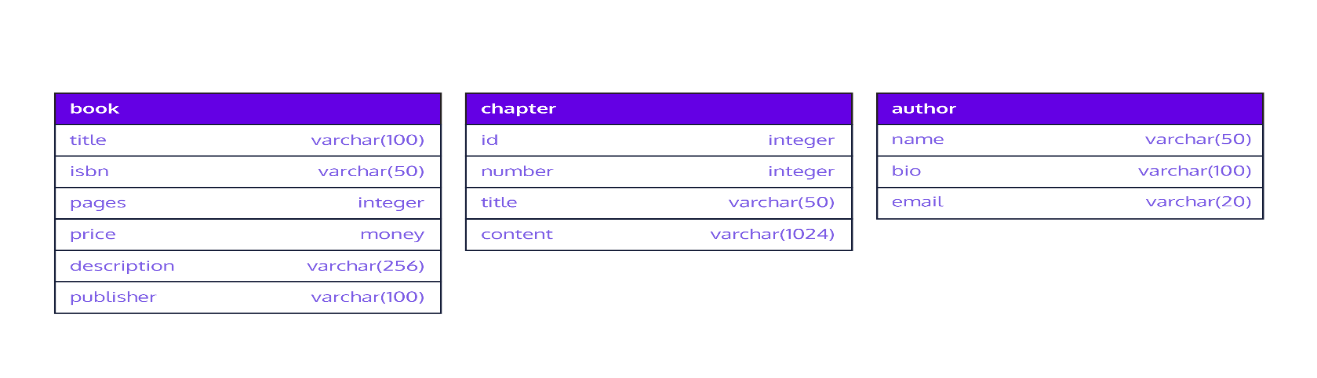
The book table has these columns:

* title for the book title, a varchar of 100 characters
* isbn for the book isbn, a varchar of 50 characters
* pages for the number of pages in the book, an integer
* price for the book price, a money type
* description for the book description, a varchar of 256 characters
* publisher for the book publisher name, a varchar of 100 characters

The chapter table has these columns:

* id for a unique chapter identifier, an integer
* number for chapter number, an integer
* title for the chapter title, a varchar of 50 characters
* content for the chapter content, a varchar of 1024 characters.

The author table has these columns:

* name for author name, a varchar of 50 characters
* bio for author bio, a varchar of 100 characters
* email for author email, a varchar of 20 c

**Primary Key**

A primary key is a designation that applies to a column or multiple columns of a table that uniquely identifies each row in the table. For example, a Social Security Number for an employee may serve as a primary key in an employee table with rows of employee data. Designating a primary key on a particular column in a table ensures that this column data is always unique and not null. For example, there may be multiple recipes of the same name, each with its own id but no two recipes should share the same id.

To designate a primary key in a table, type the PRIMARY KEY keyword in all caps next to the selected column when creating a table. For example, the following code designates the id column as the primary key for the recipe table.

CREATE TABLE recipe (  
  id integer PRIMARY KEY,  
  name varchar(20),  
  ...  
);

QUESTION: study the CREATE TABLE statement for the book table. Which column in the book table would you designate as the primary key?

In **script.sql**, edit the book table to designate the correct column as the primary key.

ANSWER:

CREATE TABLE book (

  title varchar(100),

  isbn varchar(50) PRIMARY KEY,//ISBN IS UNIQUE FOR EVERY BOOK.

  pages integer,

  price money,

  description varchar(256),

  publisher varchar(100)

);

QUESTION: study the CREATE TABLE statement for the chapter table. Think for a moment which column in this table would make an ideal primary key.

In **script.sql**, edit the chapter table to designate the correct column as the primary key.

Checkpoint

ANSWER:

CREATE TABLE chapter (

  id integer PRIMARY KEY,

  number integer,

  title varchar(50),

  content varchar(1024)

);

QUESTION: study the CREATE TABLE statement for the author table. Think for a moment which column in this table would make an ideal primary key.

In **script.sql**, edit the author table to designate the correct column as the primary key.

Checkpoint

ANSWER:

CREATE TABLE author (

  name varchar(50),

  bio varchar(100),

  email varchar(20) PRIMARY KEY

);

TABLE:

### Query Results

### Database Schema

### book

|  |  |
| --- | --- |
| **name** | **type** |
| title | character varying |
| isbn | character varying |
| pages | integer |
| price | money |
| description | character varying |
| publisher | character varying |

|  |
| --- |
| **Rows: 0** |

### chapter

|  |  |
| --- | --- |
| **name** | **type** |
| id | integer |
| number | integer |
| title | character varying |
| content | character varying |

|  |
| --- |
| **Rows: 0** |

### author

|  |  |
| --- | --- |
| **name** | **type** |
| name | character varying |
| bio | character varying |
| email | character varying |

|  |
| --- |
| **Rows: 0** |

#### **Information Schema**

As part of an international SQL standard, the [information schema](https://www.postgresql.org/docs/9.1/information-schema.html) is a database containing meta information about objects in the database including tables, columns and constraints. This schema provides users with read-only views of many topics of interest.

For example, to determine if a column has been designated correctly as a primary key, we can query a special view, key\_column\_usage, generated from this database. This view identifies all columns in the current database that are restricted by some constraint such as primary key or foreign key.

Suppose you would like to find out the constraints that have been placed on certain columns in a table, such as recipe, you would type the following query.

SELECT  
  constraint\_name, table\_name, column\_name  
FROM  
  information\_schema.key\_column\_usage  
WHERE  
  table\_name = 'recipe';

This should display the following output:

constraint\_name | table\_name | column\_name   
-----------------+------------+-------------  
 recipe\_pkey     | recipe     | id  
(1 row)

The constraint\_name value, such as recipe\_pkey, is generated by default to begin with a table name followed by the type of constraint. pkey refers to a primary key constraint, while fkey refers to a foreign key constraint.

**1.**

We have recreated the book, chapter and author tables with their corresponding primary keys that you previously defined in the last exercise.

In **script.sql**, write a query using the information\_schema.key\_column\_usage view to validate the existence of a primary key in the book table.

What is the name of the column that is the primary key?

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

In **script.sql**, query the information\_schema.key\_column\_usage view to validate the existence of a primary key in the chapter table.

What is the name of the column that has the primary key?

Checkpoint 3 Passed

**3.**

In **script.sql**, query the information\_schema.key\_column\_usage view to validate the existence of a primary key in the author table.

What is the name of the column that has the primary key?

Checkpoint

ANSWER:

SELECT

  constraint\_name, table\_name, column\_name

FROM

  information\_schema.key\_column\_usage

WHERE

  table\_name = 'author';

table

### Query Results

|  |  |  |
| --- | --- | --- |
| **constraint\_name** | **table\_name** | **column\_name** |
| chapter\_pkey | chapter | id |

### Database Schema

### book

|  |  |
| --- | --- |
| **name** | **type** |
| title | character varying |
| isbn | character varying |
| pages | integer |
| price | money |
| description | character varying |
| publisher | character varying |

|  |
| --- |
| **Rows: 0** |

### chapter

|  |  |
| --- | --- |
| **name** | **type** |
| id | integer |
| number | integer |
| title | character varying |
| content | character varying |

|  |
| --- |
| **Rows: 0** |

### author

|  |  |
| --- | --- |
| **name** | **type** |
| name | character varying |
| bio | character varying |
| email | character varying |

|  |
| --- |
| **Rows: 0** |

When we write book then table name show book but when we write author then automatically write author.

Composite Primary Key

Sometimes, none of the columns in a table can uniquely identify a record. When this happens, we can designate multiple columns in a table to serve as the primary key, also known as a composite primary key. For example, we have a table, popular\_books that contains the most popular books previewed and/or sold in a particular week.

popular\_books will have these columns:

* book\_title,
* author\_name,
* number\_sold
* number\_previewed

Since an author can have many books and a book can have many authors, there could be repeated listings of a particular book or author in the table.

For example, a listing of popular\_books sorted by book title may show the following:

      book\_title      | author\_name | number\_sold | number\_previewed   
----------------------+-------------+-------------+------------------  
 Postgres Made Easy   | Liz Key     |          33 |               50  
 Postgres Made Easy   | Tom Index   |          33 |               50  
 Beginner Postgres    | Tom Index   |          55 |               75  
 Postgres for Dummies | Liz Key     |          25 |               33

In the above example, the book title Postgres Made Easy is listed twice since it has two authors. If we list popular\_books by author name, we may find an author appearing twice such as the following:

author\_name |      book\_title        
-------------+----------------------  
 Liz Key     | Postgres Made Easy  
 Liz Key     | Postgres for Dummies  
 Tom Index   | Postgres Made Easy  
 Tom Index   | Beginner Postgres

As we see from above, neither book\_title nor author\_name can be a unique column. A composite primary key, however, can be derived from the combination of both book\_title and author\_name that would make a row unique.

To designate multiple columns as a composite primary key, use this syntax:

PRIMARY KEY (column\_one, column\_two)

For example, if we were to designate both recipe\_id and ingredient\_id as the composite primary key for the popular\_recipes table, we would write the CREATE TABLE statement for popular\_recipes as follows.

CREATE TABLE popular\_recipes (  
  recipe\_id varchar(20),  
  ingredient\_id varchar(20),  
  downloaded integer,  
  PRIMARY KEY (recipe\_id, ingredient\_id)  
);

### Instructions

**1.**

In **script.sql**, create a new table popular\_books with the following columns:

* book\_title, a varchar of 100 characters
* author\_name, a varchar of 50 characters
* number\_sold, an integer
* number\_previewed, an integer

and designate book\_title and author\_name as the composite primary key.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

In **script.sql**, write a query below the CREATE statement using the information\_schema.key\_column\_usage view to validate the existence of the composite primary key in popular\_books.

What are the column names that make up the composite primary key?

Checkpoint 3 Passed

CREATE TABLE popular\_books (

  book\_title varchar(100),

  author\_name varchar(50),

  number\_sold integer,

  number\_previewed integer,

  PRIMARY KEY (book\_title, author\_name)

);

SELECT

  constraint\_name, table\_name, column\_name

FROM

  information\_schema.key\_column\_usage

WHERE

  table\_name = 'popular\_books';

table;

### Query Results

|  |  |  |
| --- | --- | --- |
| **constraint\_name** | **table\_name** | **column\_name** |
| popular\_books\_pkey | popular\_books | book\_title |
| popular\_books\_pkey | popular\_books | author\_name |

### Database Schema

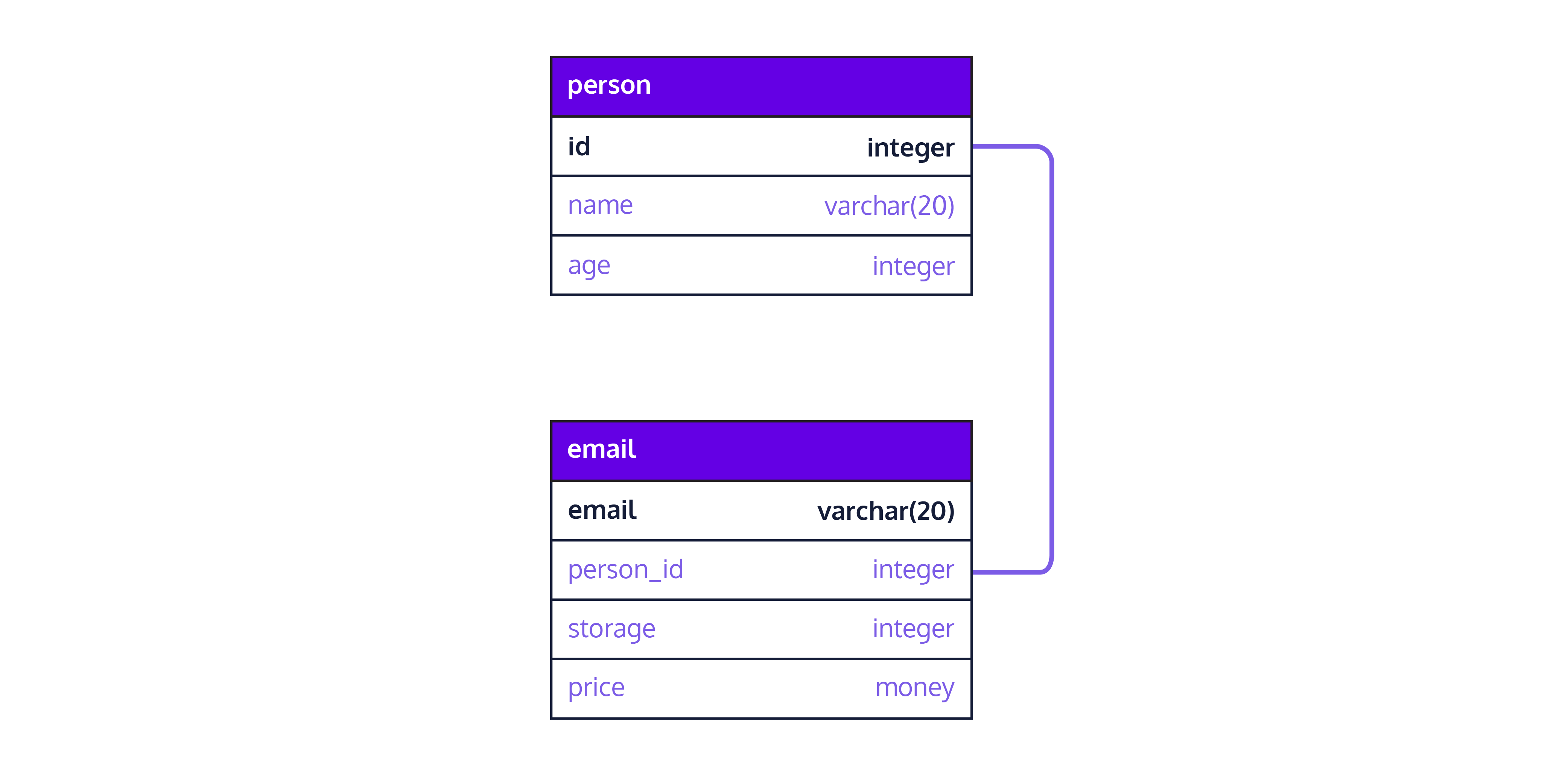
### popular\_books

|  |  |
| --- | --- |
| **name** | **type** |
| book\_title | character varying |
| author\_name | character varying |
| number\_sold | integer |
| number\_previewed | integer |

|  |
| --- |
| **Rows: 0** |

Foreign Key Part 1

In the illustration below, the foreign key is person\_id in the email table.



To designate a foreign key on a single column in PostgreSQL, we use the REFERENCES keyword:

CREATE TABLE person (  
  id integer PRIMARY KEY,  
  name varchar(20),  
  age integer  
);  
   
CREATE TABLE email (  
  email varchar(20) PRIMARY KEY,  
  person\_id integer REFERENCES person(id),  
  storage integer,  
  price money  
);

### Instructions

**1.**

Open **script.sql** and take a look at the book schema which you have created earlier. Can you guess how a book might be related to its chapters? If you were to say a book has multiple chapters, you would be correct.

In **script.sql**, add a column book\_isbn in the chapter table whose data type is the same as the isbn column in book. Then, designate that column to be a foreign key.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

In **script.sql** query the information\_schema.key\_column\_usage view to validate that you have designated book\_isbn as the foreign key in the chapter table.

Checkpoint 3 Passed

CREATE TABLE book (

  title varchar(100),

  isbn varchar(50) PRIMARY KEY,

  pages integer,

  price money,

  description varchar(256),

  publisher varchar(100)

);

CREATE TABLE chapter (

  id integer PRIMARY KEY,

  book\_isbn varchar(50) REFERENCES book(isbn),

  number integer,

  title varchar(50),

  content varchar(1024)

);

SELECT

  constraint\_name, table\_name, column\_name

FROM

  information\_schema.key\_column\_usage

WHERE

  table\_name = 'chapter';

table:

### Query Results

|  |  |  |
| --- | --- | --- |
| **constraint\_name** | **table\_name** | **column\_name** |
| chapter\_pkey | chapter | id |
| chapter\_book\_isbn\_fkey | chapter | book\_isbn |

### Database Schema

### book

|  |  |
| --- | --- |
| **name** | **type** |
| title | character varying |
| isbn | character varying |
| pages | integer |
| price | money |
| description | character varying |
| publisher | character varying |

|  |
| --- |
| **Rows: 0** |

### chapter

|  |  |
| --- | --- |
| **name** | **type** |
| id | integer |
| book\_isbn | character varying |
| number | integer |
| title | character varying |
| content | character varying |

|  |
| --- |
| **Rows: 0** |

Foreign Key Part 2

Now that you have related two tables together via a foreign key, you have ensured that you can correctly join the tables back together in a query.

For example, suppose that we want to join the person and email tables from the following schema back together:

We could use the following query to return a table of names and associated emails:

SELECT person.name AS name, email.email AS email  
FROM person, email  
WHERE person.id = email.person\_id;

### Instructions

**1.**

We have populated the database with sample book data. Open **script.sql** and write a query to display everything in the book table.

Checkpoint 2 Passed

**2.**

We have also populated the chapter table. In **script.sql**, write a query to display everything from the chapter table.

Checkpoint 3 Passed

**3.**

In **script.sql**, write a query to display book.title as book and chapter.title as chapters from both book and chapter tables.

Each row returned should show the names of the chapters and the name of the book they are in.

Checkpoint 4 Passed

SELECT \* FROM book;

SELECT \* FROM chapter;

SELECT book.title as book, chapter.title as chapters

FROM book

JOIN chapter

ON book.isbn = chapter.book\_isbn;

Table:

### Query Results

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **title** | **isbn** | **pages** | **price** | **description** | **publisher** |
| Postgres Made Easy | 0-5980-6249-1 | 130 | $9.99 | A great book for beginners to learn how to manage PostgreSQL | Codecademy Press |
| Postgres for Beginners | 0-2215-0418-4 | 255 | $7.99 | The best Postgres guide for beginners! | Codecademy Press |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **id** | **book\_isbn** | **number** | **title** | **content** |
| 1 | 0-5980-6249-1 | 1 | What is Postgres? | <h1>What is Postgres?</h1> <p>PostgreSQL, also known as Postgres, is a free and open-source relational database management system emphasizing extensibility and SQL compliance. It was originally named POSTGRES, referring to its origins as a successor to the Ingres database developed at the University of California, Berkeley. </p> |
| 2 | 0-5980-6249-1 | 2 | Database Schema | <h1>Database Schema</h1> <p>The database schema of a database is its structure described in a formal language supported by the database management system. The term "schema" refers to the organization of data as a blueprint of how the database is constructed.</p> |
| 3 | 0-5980-6249-1 | 3 | Database Keys | <h1>Database Key</h1> <p>A database key is an attribute or set of an attribute which helps you to identify a row(tuple) in a relation(table). They allow you to find the relation between two tables. Keys help you uniquely identify a row in a table by a combination of one or more columns in that table.</p> |
| 4 | 0-2215-0418-4 | 1 | Progress with Postgres | <h1>Introduction</h1> <p>PostgreSQL is a general purpose and object-relational database management system, the most advanced open source database system. PostgreSQL was developed based on POSTGRES 4.2 at Berkeley Computer Science Department, University of California.</p> |
| 5 | 0-2215-0418-4 | 2 | The Postgres Advantage | <h1>The Postgres Advantage</h1> <p>PostgreSQL was designed to run on UNIX-like platforms. However, PostgreSQL was then also designed to be portable so that it could run on various platforms such as Mac OS X, Solaris, and Windows.</p> <p>PostgreSQL is free and open source software. Its source code is available under PostgreSQL license, a liberal open source license. You are free to use, modify and distribute PostgreSQL in any form.</p><p> PostgreSQL requires very minimum maintained efforts because of its stability. Therefore, if you develop applications based on PostgreSQL, the total cost of ownership is low in comparison with other database management systems.</p> |

|  |  |
| --- | --- |
| **book** | **chapters** |
| Postgres Made Easy | What is Postgres? |
| Postgres Made Easy | Database Schema |
| Postgres Made Easy | Database Keys |
| Postgres for Beginners | Progress with Postgres |
| Postgres for Beginners | The Postgres Advantage |

### Database Schema

### book

|  |  |
| --- | --- |
| **name** | **type** |
| title | character varying |
| isbn | character varying |
| pages | integer |
| price | money |
| description | character varying |
| publisher | character varying |

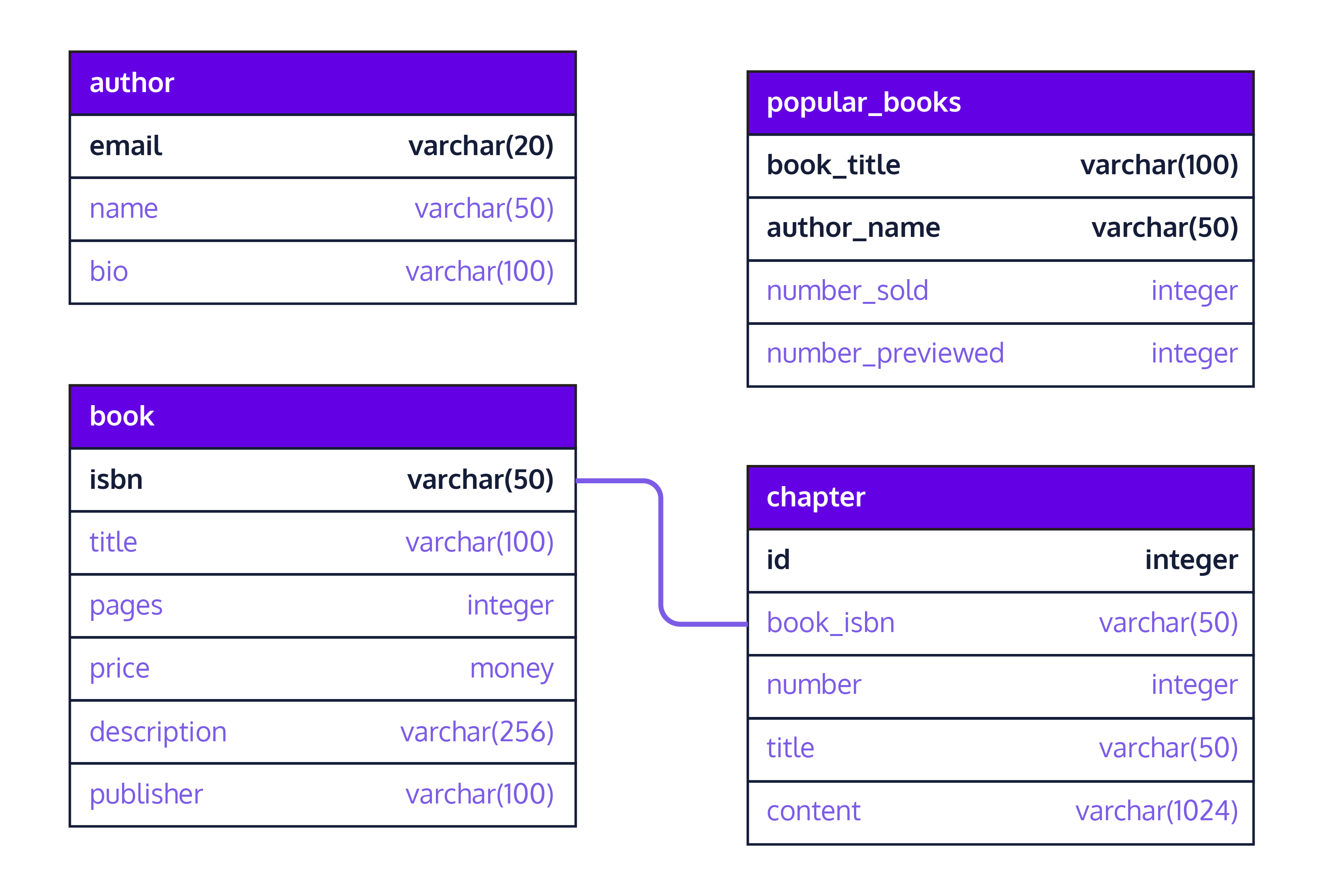
|  |
| --- |
| **Rows: 2** |

### chapter

|  |  |
| --- | --- |
| **name** | **type** |
| id | integer |
| book\_isbn | character varying |
| number | integer |
| title | character varying |
| content | character varying |

|  |
| --- |
| **Rows: 5** |

* You have created primary keys in the book schema and verified them.
* You have created a composite primary key in a new table, popular\_books, in the schema and verified it.
* You have created a foreign key in the chapter table in the schema and verified it.
* You have run queries from both book and chapter tables to validate the functions of primary and foreign keys.
* You have learned how to use the key\_column\_usage view from the information\_schema database to validate the existence of designated keys to columns.
* You have improved your overall book schema, uploaded it and verified it in the database.Image



One to many relationship

For example, consider a table where we want one person to be able to have many email addresses. However, if there is a primary key in the table, such as id, the following rows will be rejected by the database.

name   id (PK)     email         
Cody   2531       cody@yahoo.com   
Cody   2531       cody@google.com  
Cody   2531       cody@bing.com

To resolve this, we need to represent a one-to-many relationship with two tables - a parent and a child table. Analogous to a parent-child relationship where a parent can have multiple children, a parent table will house a primary key and the child table will house both primary and foreign keys. The foreign key binds the child table to the parent table.

The following illustration shows the one-to-many relationship between person and email tables.One-to-Many Relationship in Person to Email Database Schema


### Instructions

**1.**

We have pre-populated the book table with sample data. Write a query to select everything from the book table.

Checkpoint 2 Passed

**2.**

We have pre-populated the author table with sample data. Write a query to select everything from the author table.

Checkpoint 3 Passed

**3.**

Write statements to populate the books\_authors table to show the following relationships:

* 'Learn PostgreSQL Volume 1' is written by both 'James Key' and 'Clara Index'
* 'Learn PostgreSQL Volume 2' is written by 'Clara Index'

The primary keys for books are:

* '123457890' for 'Learn PostgreSQL Volume 1'
* '987654321' for 'Learn PostgreSQL Volume 2'

The primary keys for authors are:

* 'jkey@db.com' for 'James Key'
* 'cindex@db.com' for 'Clara Index'

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Write a query to show the one-to-many relationship between book and author. Display three columns using these aliases - book\_title, author\_name and book\_description.

You should expect 3 rows of results, in which one row might look like:

| **book\_title** | **author\_name** | **book\_description** |
| --- | --- | --- |
| Learn PostgreSQL Volume 1 | Clara Index | Manage database part one |

Checkpoint 5 Passed

Stuck? Get a hint

**5.**Write a query to show the one-to-many relationship between author and book. Display three columns as aliases - author\_name, author\_email and book\_title.

You should expect to see 3 rows of results and one of them might look like this:

author\_name author\_email book\_title

Clara Index [cindex@db.com](mailto:cindex@db.com) Learn PostgreSQL Volume 1

Answer: SELECT \* FROM book;

SELECT \* FROM author;

INSERT INTO books\_authors VALUES (

  '123457890',

  'jkey@db.com'

);

INSERT INTO books\_authors VALUES (

  '123457890',

  'cindex@db.com'

);

INSERT INTO books\_authors VALUES (

  '987654321',

  'cindex@db.com'

);

SELECT

    book.title AS book\_title,

    author.name AS author\_name,

    book.description AS book\_description

FROM

    book, author, books\_authors

WHERE

    book.isbn = books\_authors.book\_isbn

AND

    author.email = books\_authors.author\_email;

SELECT

    author.name AS author\_name,

    author.email AS author\_email,

    book.title AS book\_title

FROM

    book

JOIN

    books\_authors

ON

    book.isbn = books\_authors.book\_isbn

JOIN

    author

ON

    author.email = books\_authors.author\_email;

/\*\*/

Table:

### Query Results

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **title** | **isbn** | **pages** | **price** | **description** | **publisher** |
| Learn PostgreSQL Volume 1 | 123457890 | 100 | $2.99 | Manage database part one | Codecademy |
| Learn PostgreSQL Volume 2 | 987654321 | 200 | $3.99 | Manage database part two | Codecademy |

|  |  |  |
| --- | --- | --- |
| **name** | **bio** | **email** |
| James Key | Guru in database management with PostgreSQL | jkey@db.com |
| Clara Index | Guru in database management with PostgreSQL | cindex@db.com |

|  |  |  |
| --- | --- | --- |
| **book\_title** | **author\_name** | **book\_description** |
| Learn PostgreSQL Volume 1 | James Key | Manage database part one |
| Learn PostgreSQL Volume 1 | Clara Index | Manage database part one |
| Learn PostgreSQL Volume 2 | Clara Index | Manage database part two |

|  |  |  |
| --- | --- | --- |
| **author\_name** | **author\_email** | **book\_title** |
| James Key | jkey@db.com | Learn PostgreSQL Volume 1 |
| Clara Index | cindex@db.com | Learn PostgreSQL Volume 1 |
| Clara Index | cindex@db.com | Learn PostgreSQL Volume 2 |

### Database Schema

### book

|  |  |
| --- | --- |
| **name** | **type** |
| title | character varying |
| isbn | character varying |
| pages | integer |
| price | money |
| description | character varying |
| publisher | character varying |

|  |
| --- |
| **Rows: 2** |

### author

|  |  |
| --- | --- |
| **name** | **type** |
| name | character varying |
| bio | character varying |
| email | character varying |

|  |
| --- |
| **Rows: 2** |

### books\_authors

|  |  |
| --- | --- |
| **name** | **type** |
| book\_isbn | character varying |
| author\_email | character varying |

|  |
| --- |
| **Rows: 3** |

* You learned about one-to-one relationship, created a new table, book\_details to show its one-to-one correspondence with book and vice-versa, and imported dummy data and ran queries to validate the keys in the table.
* You learned about one-to-many relationship, created a new table, page to show a one-to-many relationship between chapter and page, imported sample data and ran queries to validate the relationship.
* You learned about many-to-many relationship, created a cross-reference table, books\_authors, to bind both book and author into a many-to-many relationship, populated the table and ran queries to validate the relationship.

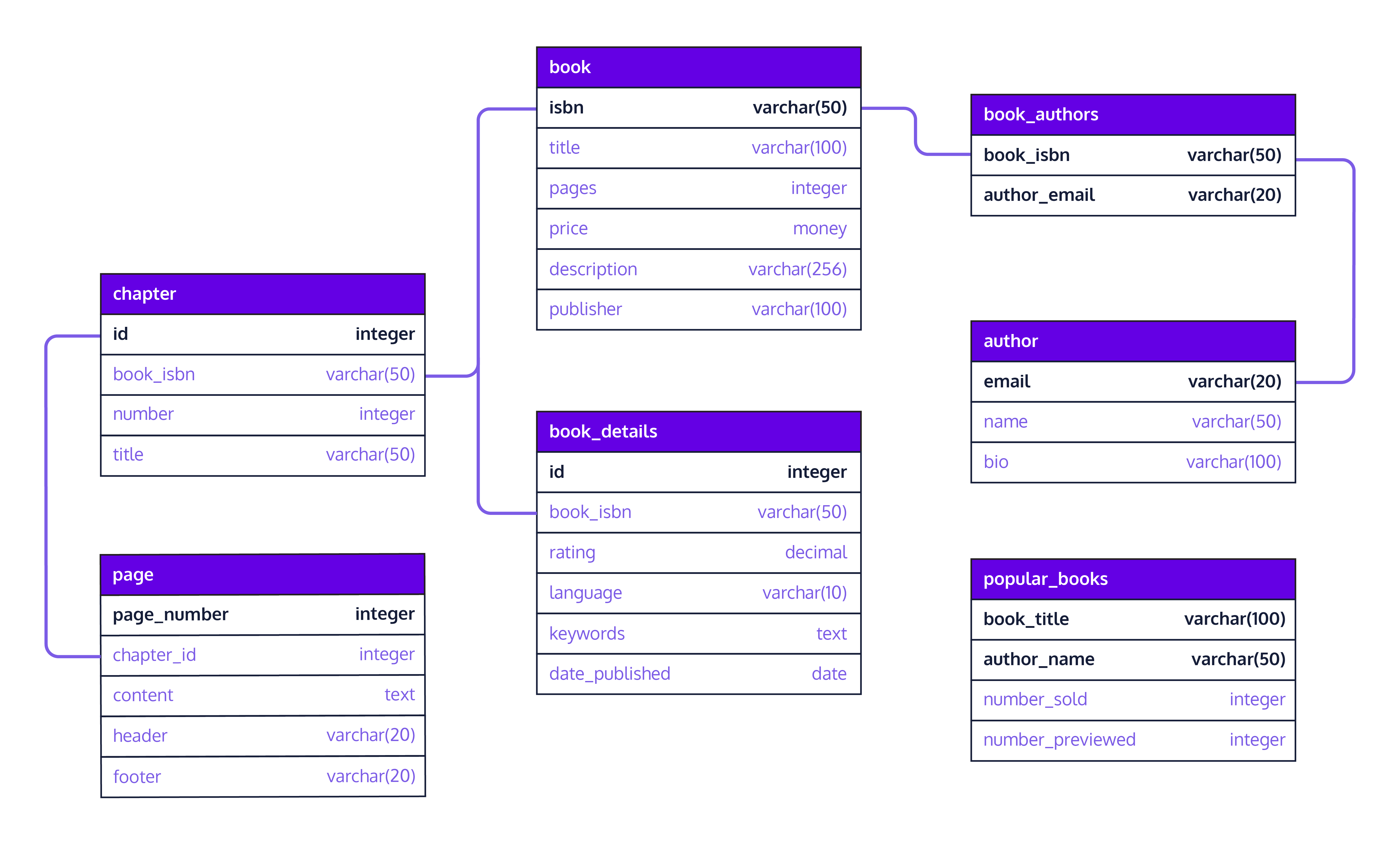
At this point, you have designed a solid book database schema that you can be proud of. If you were to import your schema into a database design software, you might have a schema diagram like the one on the right. Does it meet the requirements of a well-defined schema?

Are you able to do the following for your schema?

* Name the tables in your schema.
* Name the column names and types for each table in the schema.
* Name the constraints for each table.
* Name the relationships between tables.

Congratulations for a job well-done!!

## Image



ERDPlus is a web-based database modeling tool for creating Entity-Relationship Diagrams (ERDs), Relational Schemas (Relational Diagrams), and Star Schemas (Dimensional Models). This is helpful if you want to quickly and easily model your database and even directly export the SQL code for it.

**Duplicated Data**

Let’s now examine the college table from exercise 1 more closely. The primary key of this table is student\_id (a unique student identifier), and most of the columns describe characteristics of students, including their:

* name
* class year
* email
* major(s)
* advisor

This table also contains columns that do not describe students directly. For example, the columns advisor\_name, advisor\_department, advisor\_building, and advisor\_email describe further details about **advisors**; meanwhile, major\_1\_credits\_reqd and major\_2\_credits\_reqd describe further detail about **majors**.

When columns of a database table do not depend on (i.e., describe) the primary key, the same data may be duplicated in multiple places. For example, all students who share an advisor will have the same information listed in these four advisor-related columns. In future exercises, we will see how this can cause problems related to data modification. For now, let’s query the table to examine the duplication.

Remember that you can use a WHERE clause within a SELECT statement to return rows of a table where a particular condition is met. For example, the following query would return all sophomore student names and emails:

SELECT student\_name, student\_email  
FROM college  
WHERE student\_year = 'sophomore';

### Instructions

**1.**

The college table has been loaded for you. Professor Brunson is a popular advisor from the chemistry department. Use a SELECT statement and WHERE clause to select the advisor\_name, advisor\_department and advisor\_email columns (in that order) for all students with 'Brunson' listed as their advisor’s name. Inspect the repeated data.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Below your query from step 1, write another query to return major\_1 and major\_1\_credits\_reqd for all students with major\_1 = 'Computer Science'. Note how many times the same number of credits is recorded.

Answer:

SELECT advisor\_name, advisor\_department, advisor\_email

FROM college

WHERE advisor\_name = 'Brunson';

SELECT major\_1, major\_1\_credits\_reqd

FROM college

WHERE major\_1 = 'Computer Science';

Table:

### Query Results

|  |  |  |
| --- | --- | --- |
| **advisor\_name** | **advisor\_department** | **advisor\_email** |
| Brunson | Chemistry | Brunson47@college.edu |
| Brunson | Chemistry | Brunson47@college.edu |
| Brunson | Chemistry | Brunson47@college.edu |

Data Update Challenges

In the last exercise, we saw that the college table contains duplicated data. if we want to modify the duplicated information, because we’ll have to make the same updates in multiple locations.

For example, suppose that a faculty member named Professor 'Hill' changes their email address to 'hill@college.edu' and asks you to update the school database accordingly. Since multiple students may have Professor Hill as an advisor, you’ll need to update her email address in multiple rows of the table. To do this, your first instinct might be something like:

UPDATE college  
SET advisor\_email = 'hill@college.edu'  
WHERE advisor\_name = 'Hill';

Unfortunately, this could cause problems if there are multiple professors with the same name. For example, suppose we run the code above, then query the table for all rows where advisor\_name = 'Hill' and observe the following result:

| **advisor\_name** | **advisor\_department** | **advisor\_email** |
| --- | --- | --- |
| Hill | Biology | [hill@college.edu](mailto:hill@college.edu) |
| Hill | English | [hill@college.edu](mailto:hill@college.edu) |
| … | … | … |

Oh no! We accidentally modified two professors’ emails when we only intended to change one. If advisor information were housed in a separate table, we could have imposed a UNIQUE or PRIMARY KEY constraint on the advisor\_email column to make sure that no two advisors have the same emails recorded in our database; however, this was not possible in the current design, since we needed to store duplicate emails for all students with the same advisor. Our database is not well protected against modification errors!

### I

### Instructions

**1.**

Query the college table to print the advisor\_name, advisor\_department, and advisor\_email (in that order) for all students with Professor 'Sommer' listed as an advisor. Are there multiple advisors with this name?

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Update Prof. Sommer’s email to be 'sophie@college.edu'. Make sure to only update the email for Prof. 'Sommer' in the 'Statistics' department. Think about how this update is more complicated because there could be multiple advisors named 'Sommer'.

ANSWER:

SELECT advisor\_name,advisor\_department, advisor\_email

FROM college

WHERE advisor\_name='Sommer';

UPDATE college

SET advisor\_email = 'sophie@college.edu'

WHERE advisor\_name = 'Sommer' AND advisor\_department = 'Statistics';

TABLE:

### Query Results

|  |  |  |
| --- | --- | --- |
| **advisor\_name** | **advisor\_department** | **advisor\_email** |
| Sommer | History | Sommer3@college.edu |
| Sommer | History | Sommer3@college.edu |
| Sommer | History | Sommer3@college.edu |
| Sommer | Statistics | Sommer8@college.edu |
| Sommer | History | Sommer3@college.edu |
| Sommer | Statistics | Sommer8@college.edu |
| Sommer | Statistics | Sommer8@college.edu |

**Data Insertion Challenges:** we saw how data duplication can complicate database modification. Unfortunately, there is another potential problem that can arise when all columns in a table do not depend on the primary key: new data cannot be inserted into the table until a primary key is known.

Suppose, for example, that the school hires a new advisor in the Computer Science department. The advisor’s name is Professor Algorithm and his email is [algorithm1000@college.edu](mailto:algorithm1000@college.edu). Unfortunately, we cannot add a new row to this database table because Professor Algorithm has not been assigned to advise any students yet — and we need a primary key (associated with an individual student) in order to insert a new row of data.

we could theoretically come up with a fake student ID and leave non-relevant columns empty. For example, assuming that the database is set-up to assign NULL values by default to every non-key column, we could use the following code to insert Professor Algorithm’s information into the database:

INSERT INTO college   
(student\_id, advisor\_name, advisor\_department, advisor\_email)  
VALUES   
(-1, 'Algorithm', 'Computer Science', 'algorithm1000@college.edu');

**1.**

Some code has been provided for you in script.sql to attempt to insert a new row into the college table for an advisor with advisor\_name = 'Stern', advisor\_department = 'Biology', and advisor\_email = 'stern111@college.edu'. As is, this INSERT will throw an error (and not work) because we have not included a value in the primary key column. Press the Run button to inspect the error message.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Edit the INSERT statement from the previous step to include a student\_id of -1.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Below your INSERT statement, query the table to inspect this new row (student\_id = -1). IMPORTANT: you now have a student\_id listed in your database that doesn’t actually match with a student! This is a problem!

Checkpoint

ANSWER:

INSERT INTO college (student\_id, advisor\_name,advisor\_department, advisor\_email)

VALUES (-1, 'Stern', 'Biology', 'stern111@college.edu');

SELECT \*

FROM college

WHERE student\_id = -1;

TABLE:

### Query Results

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **student\_id** | **full\_name** | **student\_year** | **student\_email** | **major\_1** | **major\_1\_credits\_reqd** | **major\_2** | **major\_2\_credits\_reqd** | **advisor\_name** | **advisor\_department** | **advisor\_building** | **advisor\_email** |
| -1 |  |  |  |  |  |  |  | Stern | Biology |  | stern111@college.edu |

### Database Schema

### college

|  |  |
| --- | --- |
| **name** | **type** |
| student\_id | integer |
| full\_name | character varying |
| student\_year | character varying |
| student\_email | character varying |
| major\_1 | character varying |

**Search and Sort Efficiency:** example, suppose that your boss is interested in figuring out which majors are most popular. They ask you to produce a table of unique majors along with the number of students who have declared each one, sorted by popularity.

If all information about student majors were contained in a single column (eg., major\_1), this would be relatively simple:

SELECT major\_1, count(\*)  
FROM majors  
GROUP BY major\_1  
ORDER BY count DESC;

The first few rows returned by the query above would look something like:

| **count** | **major\_1** |
| --- | --- |
| 53 | History |
| 49 | Political Science |
| 49 | Computer Science |
| 49 | English |
| 47 | Spanish |
| 46 | Geology |

The problem is that these counts are incorrect — they completely ignore majors recorded in the major\_2 column! To remedy this, we’ll have to join the major\_1 and major\_2 columns together somehow, creating additional complexity.

### Instructions

**1.**

Let’s first combine all of the data found in the major\_1 and major\_2 columns. Note that major\_2 has many NULL values because some students only have one major. Fill in the blanks in the provided query to return the contents of major\_1 and NOT NULL contents of major\_2 in a single column called major. Note that this is way more complicated than it should be because of the poor table design!

SELECT major\_1 as major   
FROM college  
UNION ALL  
SELECT \_\_\_ as major  
FROM \_\_\_  
WHERE \_\_\_ IS NOT NULL;

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

EDIT your query from step 1: First, delete the ; at the end; then use the syntax indicated below to query the list of majors you produced in step 1 and return each major name alongside the number of students who have declared that major, ordered by popularity from most to least popular (you will need to use COUNT, GROUP BY, and ORDER BY to accomplish this). The syntax is:

WITH majors AS(  
\_\_query from step 1 goes here without ';' at the end\_\_  
)  
SELECT\_\_\_, count(\_)  
FROM majors  
GROUP BY \_\_\_  
ORDER BY \_\_\_ DESC;

Compare your result to the output from the narrative (which only looked at the major\_1 column). Once we include major\_2, is the most popular major still History? Or has it changed to something else?

Checkpoint

**ANSWER:** WITH majors AS(

SELECT major\_1 as major

FROM college

UNION ALL

SELECT major\_2 as major

FROM college

WHERE major\_2 IS NOT NULL

)

SELECT major, count(\*)

FROM majors

GROUP BY major

ORDER BY count DESC;

**TABLE:**

### Query Results

|  |  |
| --- | --- |
| **major** | **count** |
| Geology | 77 |
| Spanish | 75 |
| Computer Science | 74 |
| English | 74 |
| History | 74 |

**Restructuring the Advisor Columns:** First, we saw that every student with the same advisor has identical information recorded in all advisor-related columns in the college table. One way to address this is by moving the four advisor-related columns into their own table, with only one row per unique advisor. This helps ensure every table has its own purpose or concern.

To create a new table from an existing one, we can precede any query with CREATE TABLE new\_table\_name AS. For example, the following code selects the unique values of major\_1 and major\_1\_credits\_reqd from the original college table and inserts them into a newly created table called majors:

CREATE TABLE majors AS  
SELECT distinct major\_1, major\_1\_credits\_reqd  
FROM college;

We can also delete columns from our original table once we have moved them. For example, the following code drops the columns major\_1 and major\_1\_credits\_reqd from the college table:

ALTER TABLE college  
DROP COLUMN major\_1,   
DROP COLUMN major\_1\_credits\_reqd;

### Instructions

**1.**

Create a new table named advisors that contains the advisor\_email, advisor\_name, and advisor\_department columns (in that order) from the college table. Use the DISTINCT key word so that you keep one row per unique advisor (you can assume that each advisor has a unique email).

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Do not delete the CREATE TABLE statement from step 1. Below it, add an ALTER TABLE statement to delete the advisor\_name and advisor\_department columns from the original college table. Note that we need to keep advisor\_email (or some sort of advisor identifier) in the college table to ensure that we can still match advisors to students.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Do not delete the CREATE TABLE or ALTER TABLE statements from steps 1 and 2. Below them, query the new advisors table for all rows where 'Brunson' is listed as the advisor\_name. Note that this information is now condensed to a single row in our database!

Checkpoint 4 Passed

**ANSWER:**

CREATE TABLE advisors AS

SELECT DISTINCT advisor\_email, advisor\_name, advisor\_department

FROM college;

ALTER TABLE college

DROP COLUMN advisor\_name,

DROP COLUMN advisor\_department;

SELECT \*

FROM advisors

WHERE advisor\_name = 'Brunson';

**TABLE:**

### Query Results

|  |  |  |
| --- | --- | --- |
| **advisor\_email** | **advisor\_name** | **advisor\_department** |
| Brunson47@college.edu | Brunson | Chemistry |

### Database Schema

### college

|  |  |
| --- | --- |
| **name** | **type** |
| student\_id | integer |
| full\_name | character varying |
| student\_year | character varying |
| student\_email | character varying |
| major\_1 | character varying |
| major\_1\_credits\_reqd | integer |
| major\_2 | character varying |
| major\_2\_credits\_reqd | integer |
| advisor\_email | character varying |

|  |
| --- |
| **Rows: 1000** |

### advisors

|  |  |
| --- | --- |
| **name** | **type** |
| advisor\_email | character varying |
| advisor\_name | character varying |
| advisor\_department | character varying |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Restructuring the Major Columns  We saw in the previous exercises that there are two potential issues with the four major-related columns in the college table:   1. The two \_credits\_reqd columns describe the majors themselves, rather than the students, creating duplication 2. The repeating sets of columns make it challenging to search and sort our data by major.   To address these problems, we can extract the four major-related columns to create two new tables:   * A majors table listing each unique major and the number of credits required * A cross-reference table that matches students with majors. Note that we need two additional tables (as opposed to one) because the relationship between students and majors is many-to-many (multiple students can have the same major and each student can have multiple majors), while each major has only one value of credits\_reqd.   After creating these two new tables, student and major data will be stored in three smaller tables that are related as follows:  three table version of the college database with students, majors, and students_majors    The first few rows of these tables might look something like:   * students (student-specific data):  | **student\_id** | **full\_name** | **student\_year** | **student\_email** | | --- | --- | --- | --- | | 1 | Joshua Davis | senior | [Joshua1@college.edu](mailto:Joshua1@college.edu) | | 2 | Ricardo Sansburn | junior | [Ricardo2@college.edu](mailto:Ricardo2@college.edu) | | 3 | Waneesa el-Zakaria | freshman | [Waneesa3@college.edu](mailto:Waneesa3@college.edu) | | 4 | Selena Mendoza | freshman | [Selena4@college.edu](mailto:Selena4@college.edu) |      * majors (major-specific data):  | **major** | **credits\_reqd** | | --- | --- | | Statistics | 13 | | Data Science | 16 | | Art History | 15 | | Neuroscience | 16 |      * students\_majors (student-major pairs):  | **student\_id** | **major** | | --- | --- | | 1 | Chemistry | | 1 | Sociology | | 2 | Sociology | | 3 | Data Science |     If you look carefully at that last table, you’ll see that the student with student\_id = 1 (Joshua Davis) is listed in two rows of the students\_majors table: one row for each of his majors. Instructions **1.**  Create a table called majors that contains the major\_1 and major\_1\_credits\_reqd columns (in that order) from the college table. Use the DISTINCT key word to only include one row per unique major (note: for simplicity, we are assuming — correctly — that all possible majors are represented in the major\_1 column).  Checkpoint 2 Passed  Stuck? Get a hint  **2.**  Below your first CREATE TABLE statement from step 1: Copy, paste, and fill in the blanks in the following code to create a table called students\_majors containing all unique student-major pairs. To identify a unique student, use the student\_id column.  CREATE TABLE \_\_\_ AS SELECT major\_1 as major, \_\_\_  FROM \_\_\_ UNION ALL SELECT major\_2 as major, \_\_\_ FROM \_\_\_ WHERE major\_2 IS NOT NULL;  Checkpoint 3 Passed  Stuck? Get a hint  **3.**  Below the two CREATE TABLE statements from steps 1 and 2, write an ALTER TABLE statement to delete the major\_1, major\_1\_credits\_reqd, major\_2, and major\_2\_credits\_reqd columns from the original college table.  Checkpoint 4 Passed  Stuck? Get a hint  **4.**  Below all of your CREATE TABLE and ALTER TABLE statements from the previous steps, query the students\_majors table to print the first 10 rows (and all columns), ordered by student\_id. Does it look as expected?  Checkpoint 5 Passed  **Rows: 50** |

**ANSWER:**

CREATE TABLE majors AS

SELECT DISTINCT major\_1, major\_1\_credits\_reqd

FROM college;

CREATE TABLE students\_majors AS

SELECT major\_1 as major, student\_id

FROM college

UNION ALL

SELECT major\_2 as major, student\_id

FROM college

WHERE major\_2 IS NOT NULL;

ALTER TABLE college

DROP COLUMN major\_1,

DROP COLUMN major\_1\_credits\_reqd,

DROP COLUMN major\_2,

DROP COLUMN major\_2\_credits\_reqd;

SELECT \*

FROM students\_majors

ORDER BY student\_id

LIMIT 10;

**TABLE:**

### Query Results

|  |  |
| --- | --- |
| **major** | **student\_id** |
| Chemistry | 1 |
| Sociology | 1 |
| Sociology | 2 |
| Data Science | 3 |
| Environmental Science | 4 |
| Neuroscience | 4 |
| Mandarin | 5 |
| Chemistry | 5 |
| Neuroscience | 6 |
| English | 6 |

### Database Schema

### college

|  |  |
| --- | --- |
| **name** | **type** |
| student\_id | integer |
| full\_name | character varying |
| student\_year | character varying |
| student\_email | character varying |
| advisor\_email | character varying |

|  |
| --- |
| **Rows: 1000** |

### majors

|  |  |
| --- | --- |
| **name** | **type** |
| major\_1 | character varying |
| major\_1\_credits\_reqd | integer |

|  |
| --- |
| **Rows: 25** |

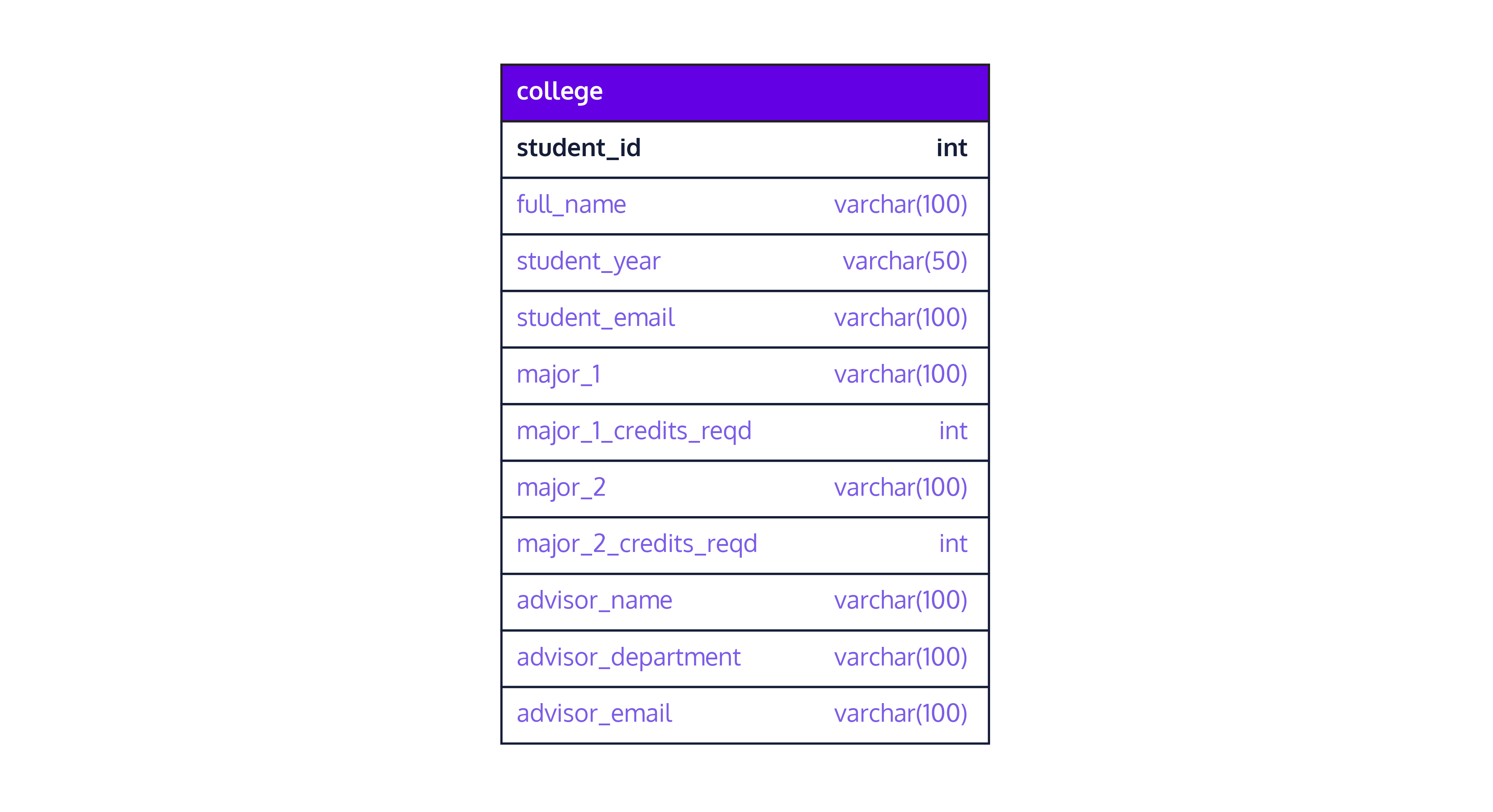
### students\_majors

|  |  |
| --- | --- |
| **name** | **type** |
| major | character varying |
| student\_id | integer |

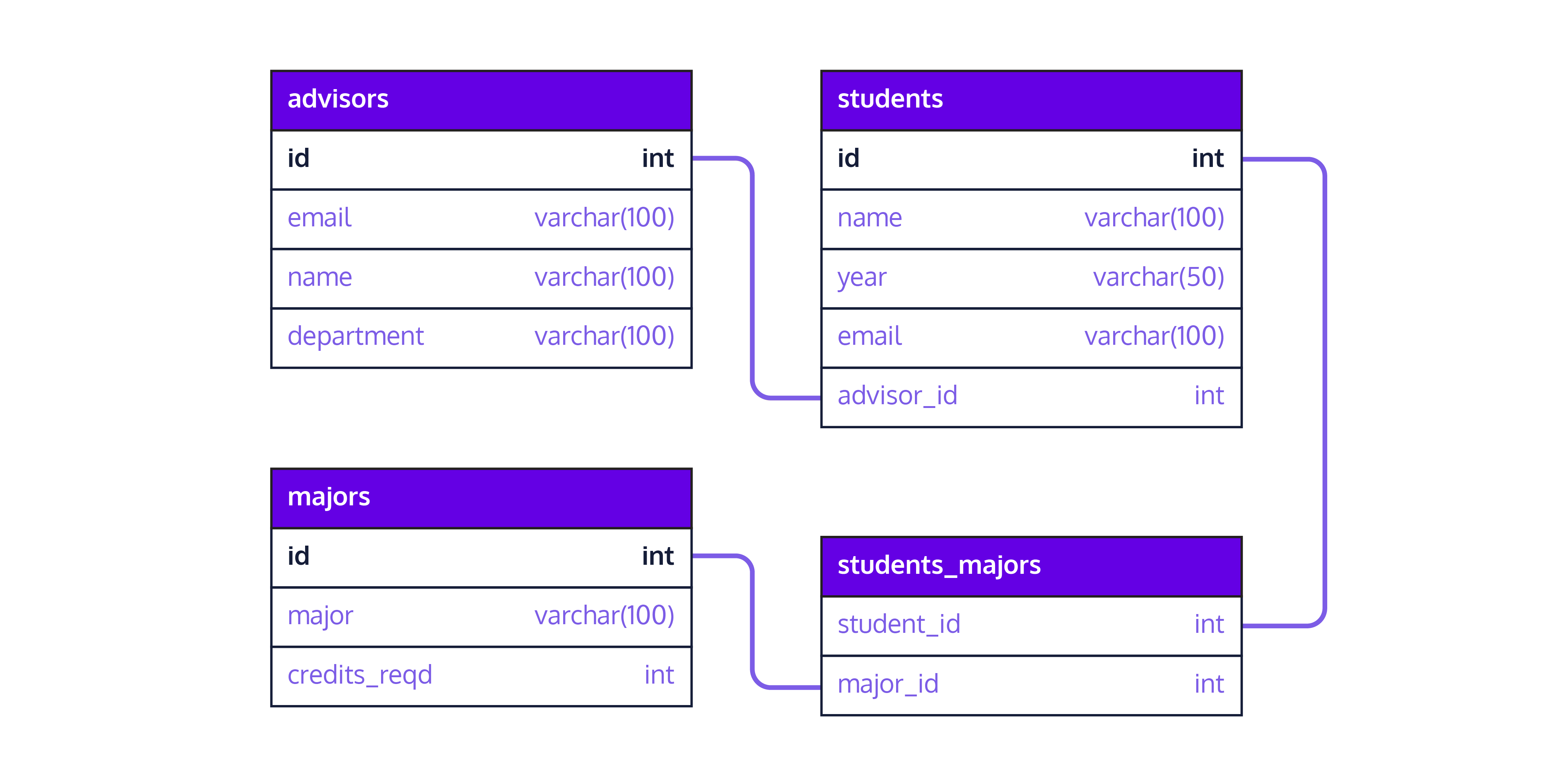
|  |
| --- |
| **Rows: 1600** |

**Creating Versus Modifying a Database Schema:** the two figures below show the original schema (with a single college table) compared to a normalized version with four tables.

non-normalized version:



compared to the normalized version:



In the previous exercises, we created tables similar to the ones in the four-table schema described above; however, there are some important differences. For example:

* The advisor\_name column is renamed name in the advisors table, because it is already clear that everything within that table describes advisors.
* The college table is renamed students because all non-student-related data has now been moved to other tables.
* The primary keys in the advisors, students, and majors tables are now unique integer values. While it is valid to use advisor\_email as a primary key in the advisors table, it can make things more difficult when we want to update or delete an email address.

In the four-table schema, we’ll also want to implement some constraints that were either not necessary or not possible in the original version. For example:

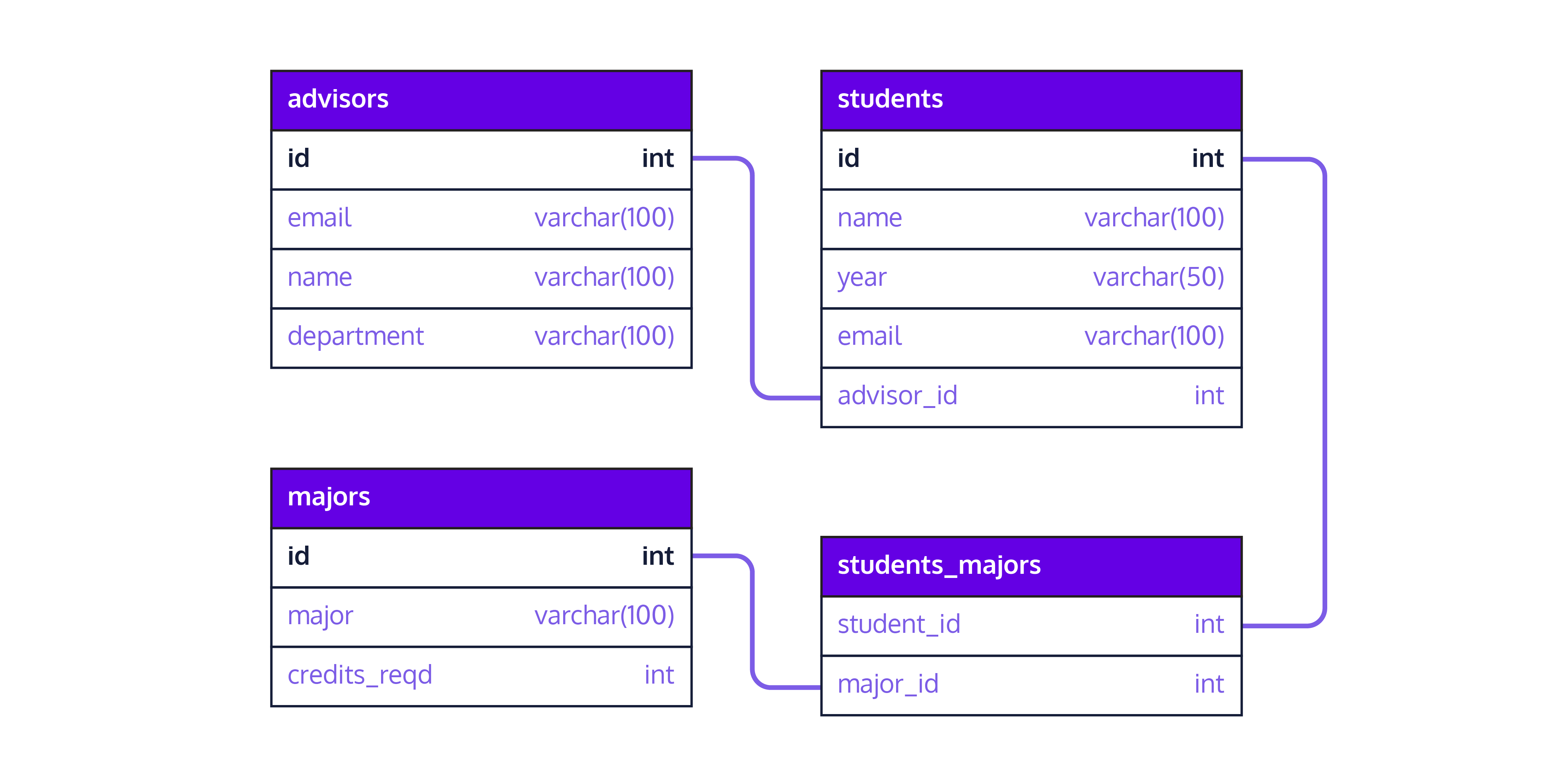
* We can now implement a UNIQUE constraint on the email column of the advisors table, ensuring that no two advisors have the same email listed.
* We can also enforce a foreign key constraint on the advisor\_id column of the students table so that each advisor\_id in that table matches an id in the advisors table.

In this exercise, we’ll create the four-table version of this schema using CREATE TABLE statements (instead of by extracting, deleting, and renaming columns from the original college table). Remember that the following syntax can be used to create a pair of related tables with primary and foreign key constraints. The first line creates a customers table with id as the primary key; the second creates an orders table with order\_id as the primary key and customer\_id as a foreign key referencing the id column in the customers table:

CREATE TABLE customers (  
  id serial PRIMARY KEY,  
  name text,  
  age integer  
);  
   
CREATE TABLE orders (  
  order\_id integer PRIMARY KEY,  
  customer\_id integer REFERENCES customers(id),  
  price money  
);

### Instructions

**1.**

Here is the four-table database schema that we want to create:

Create the advisors table from the schema diagram, with columns in the same order. Make sure to include any relevant primary and foreign key constraints and add a UNIQUE constraint to the email column.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Add a second CREATE TABLE statement (below the first) to create the students table from the schema diagram, with columns in the same order. Make sure to include any relevant primary and foreign key constraints and add a UNIQUE constraint to the email column.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Add a third CREATE TABLE statement (below the first two) to create the majors table from the schema diagram, with columns in the same order. Make sure to include any relevant primary and foreign key constraints.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Add a fourth CREATE TABLE statement (below the first three) to create the students\_majors table from the schema diagram, with columns in the same order. Make sure to include any relevant primary and foreign key constraints.

Checkpoint 5 Passed

**ANSWER:** CREATE TABLE advisors (

  "id" INTEGER PRIMARY KEY,

  "email" VARCHAR(100) UNIQUE,

  "name" VARCHAR(100),

  "department" VARCHAR(100)

);

CREATE TABLE students (

  "id" INTEGER PRIMARY KEY,

  "name" VARCHAR(100),

  "year" VARCHAR(50),

  "email" VARCHAR(100) UNIQUE,

  "advisor\_id" INTEGER REFERENCES advisors(id)

);

CREATE TABLE majors (

  "id" INTEGER PRIMARY KEY,

  "major" VARCHAR(100),

  "credits\_reqd" INTEGER

);

CREATE TABLE students\_majors (

  "student\_id" INTEGER REFERENCES students(id),

  "major\_id" INTEGER REFERENCES majors(id)

);

### **TABLE:** Query Results

### Database Schema

### advisors

|  |  |
| --- | --- |
| **name** | **type** |
| id | integer |
| email | character varying |
| name | character varying |
| department | character varying |

|  |
| --- |
| **Rows: 0** |

### students

|  |  |
| --- | --- |
| **name** | **type** |
| id | integer |
| name | character varying |
| year | character varying |
| email | character varying |
| advisor\_id | integer |

|  |
| --- |
| **Rows: 0** |

### majors

|  |  |
| --- | --- |
| **name** | **type** |
| id | integer |
| major | character varying |
| credits\_reqd | integer |

|  |
| --- |
| **Rows: 0** |

### students\_majors

|  |  |
| --- | --- |
| **name** | **type** |
| student\_id | integer |
| major\_id | integer |

|  |
| --- |
| **Rows: 0** |

Database Structure and Use: **1.**

The students\_majors and majors tables have been created for you. The students\_majors table contains columns student\_id and major\_id; the majors table contains columns id, major, and credits\_reqd

Fill in the blanks in the provided query to produce a list of majors and the number of students who have declared them, sorted by popularity.

SELECT \_\_\_, count(\*)  
FROM students\_majors, majors  
WHERE major\_id = id  
GROUP BY major  
ORDER BY \_\_\_ DESC;

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

An advisors table has been created for you with advisor email, name, and department. The email column is constrained as UNIQUE because we now expect no duplication in this column.

Below your query from step 1: copy, paste, and fill in the provided code to update the advisors table such that email = 'sophie@college.edu' in all rows where name = 'Sommer'. Unlike before, you should get an error because there are two professors with the name ‘Sommer’ and this update will therefore violate the UNIQUE constraint.

UPDATE \_\_\_  
SET \_\_\_ = \_\_\_  
WHERE name = 'Sommer';

Checkpoint

ANSWER:

SELECT major, count(\*)

FROM students\_majors, majors

WHERE major\_id = id

GROUP BY major

ORDER BY count DESC;

UPDATE advisors

SET email = 'sophie@college.edu'

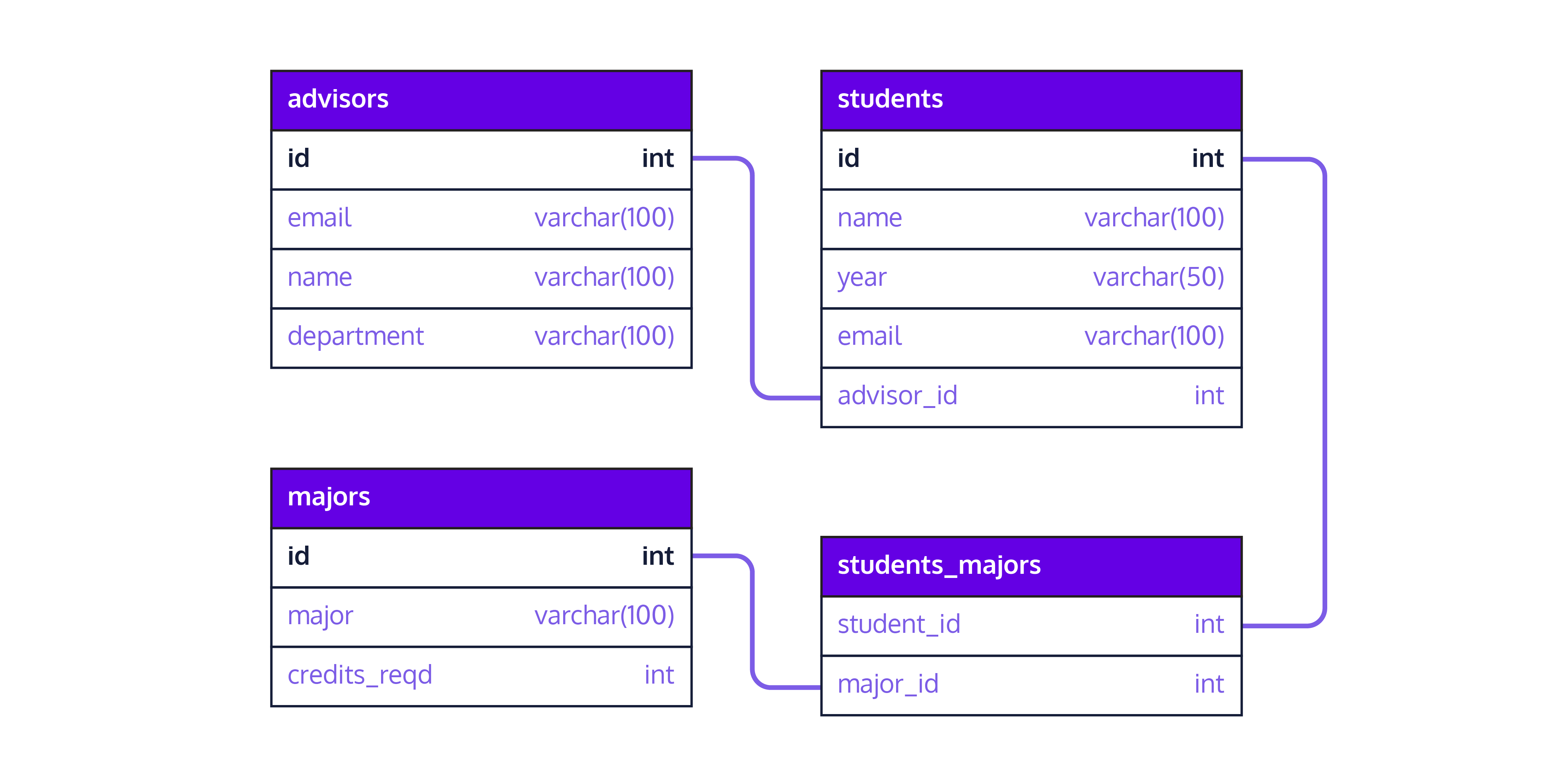
WHERE name = 'Sommer';

TABLE:

### Query Results

|  |  |
| --- | --- |
| **major** | **count** |
| Geology | 77 |
| Spanish | 75 |
| Computer Science | 74 |
| English | 74 |
| History | 74 |

Database Structure and Use (Continued): example, suppose that we’ve now fully redesigned our original database to have the following structure:



If we want to know how many students are advised by faculty in each department, we’ll have to join the students and advisors tables back together. To do this, we can select from both tables where the advisor\_id column from the students table equals the id column from the advisors table. The following query will do the trick:

SELECT students.id as student\_id, department as advisor\_department  
FROM students, advisors  
WHERE students.advisor\_id = advisors.id;

This gives us a table with student ids matched to advisor departments. The first few rows might look like this:

| **student\_id** | **advisor\_department** |
| --- | --- |
| 1 | Philosophy |
| 2 | Philosophy |
| 3 | Computer Science |

Next, we can alter this query slightly to count the number of students with advisors in each department. We can accomplish this using COUNT and GROUP BY as follows:

SELECT COUNT(students.id), advisors.department as advisor\_department  
FROM students, advisors  
WHERE students.advisor\_id = advisors.id  
GROUP BY advisor\_department;

This gives us what we want: a table showing the number of students advised by faculty in each department. Here are the first few rows:

| **count** | **advisor\_department** |
| --- | --- |
| 68 | Computer Science |
| 53 | Environmental Science |
| 24 | Chemistry |

That wasn’t too complicated - but note that we could have accomplished the same thing by querying the original college table without any joins! In the exercises below, you’ll walk through another example of a query that requires joining tables back together.

### Instructions

**1.**

The students, majors and students\_majors tables have been created for you as outlined in the schema from this exercise. Suppose that you are asked to email students a reminder about the number of credits required for their major(s).

To answer this question, you’ll need to join the students, students\_majors and majors tables back together. Fill in the code below to select the student\_id, and credits\_reqd columns from the appropriately joined table.

SELECT student\_id, \_\_\_  
FROM students, students\_majors, majors  
WHERE students.id = students\_majors.\_\_\_  
AND students\_majors.\_\_\_ = majors.id;

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Now, edit your query from step 1 to replace the credits\_reqd column with a column named total\_credits that contains the total number of credits required by each student (to accomplish this, you’ll need to sum the credits\_reqd values associated with each unique student\_id).

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Finally, edit your query from steps 1 and 2 to also return the email column from the students table as a third column named student\_email.

Note that, because you are grouping by student\_id, you will need to aggregate any other column you query; However, because the email column is unique for every student\_id, it doesn’t matter how you aggregate. We can, for example, use MIN(email) to get the first email associated with each student\_id(and this would yield the same results as MAX(email)).

Checkpoint

ANSWER: SELECT student\_id, SUM(credits\_reqd), MIN(email) as student\_email

FROM students, students\_majors, majors

WHERE students.id = students\_majors.student\_id

AND students\_majors.major\_id = majors.id

GROUP BY student\_id;

### Query Results

|  |  |  |
| --- | --- | --- |
| **student\_id** | **sum** | **student\_email** |
| 652 | 29 | Paul652@college.edu |
| 273 | 14 | Jazmine273@college.edu |
| 51 | 29 | Brittany51@college.edu |
| 951 | 16 | Jayse951@college.edu |
| 839 | 28 | Neil839@college.edu |
| 70 | 14 | Nawaar70@college.edu |

**Review**

* Repeating groups of columns and columns that are not dependent on the primary key of a table can cause problems related to:
  + Duplicated data
  + Data modification (updates/insertions)
  + Querying
* Database normalization is a process by which database tables are modified/restructured to address these problems
* Database design can be modified after data has been entered, but it is usually easiest to design a normalized schema up front
* It is important to keep database use in mind when designing an optimal database schema

In this final exercise, we’ve re-loaded the initial college table from exercise 1, as well as the 4 smaller tables from exercise 9 (students, students\_majors, majors and advisors). Below are some questions you might try to answer with both database structures to better understand how they differ.

### CHAPTER: 30 Advanced PostgreSQL

What is an Index?

An index is an organization of the data in a table to help with performance when searching and filtering records. A table can have zero, one, or many indexes. Let’s start by learning how to see what indexes already exist on a table. Say you want to see what indexes exist on your products table you would run the following query:

SELECT \*  
FROM pg\_Indexes  
WHERE tablename = 'products';

pg\_Indexes is a built-in view in PostgreSQL. Different database servers have different ways to see their indexes.

QUESTION: Lookup any indexes that already exist in the customers table. You will need to scroll down towards the bottom of the output. You should see another query with the columns schemaname, tablename, etc. These are the indexes currently associated with the table.

Pay particualar attention to tablename, indexname and indexdef. What do you think each of these represents?ANSWER:

Checkpoint

SELECT \*

FROM customers;

SELECT \*

FROM pg\_Indexes /\* pg \_indexes is neccesary to write\*/

WHERE tablename = 'customers';

TABLE:

### Query Results

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **customer\_id** | **first\_name** | **last\_name** | **email\_address** | **home\_phone** | **city** | **state\_name** | **years\_old** |
| 1 | Edward | Lewis | Edward.Lewis@example.com | 202-555-0264 | Pittsburgh | Pennsylvania | 82 |
| 2 | Frances | Campbell | Frances.Campbell@example.com | 202-555-0073 |  |  |  |

What is the benefit of an Index?

In a relational database like PostgreSQL, indexes are used to improve the speed of searching and filtering at the cost of slower inserts, updates, and deletes.

Impact of Indexes

To get insight into how PostgreSQL breaks down your statements into runnable parts, we can investigate the query plan by adding EXPLAIN ANALYZE before your query. Rather than returning the results of the query, it will return information *about* the query.

EXPLAIN ANALYZE SELECT \*  
FROM customers;

This would return the plan that the server will use to give you every row from every record from the customers table.

If you see “Seq Scan” this means that the system is scanning every record to find the specific records you are looking for. If you see “Index” (in our examples more specifically “Bitmap Index Scan”) you know that the server is taking advantage of an index to improve the speed of your search.

The other part to take note of is the “Planning time” and “Execution time”. The planning time is the amount of time the server spends deciding the best way to solve your query, should it use an index, or do a full scan of the table(s) for instance. The execution time is the amount of time the actual query takes to run after the server has decided on a plan of attack. You need to take both of these into consideration, and when examining your own indexes these are critical to understanding how effective your indexes are.

QUESTION: Let’s start with a search on a column without an index. Select all columns from the rows of the customers table where first\_name is 'David'. Before running your query, add EXPLAIN ANALYZE to the start of it.

**2.**The first\_name column doesn’t have an index on it, but the last\_name column does. Select all columns from the rows where last\_name is 'Jones'. Again, add EXPLAIN ANALYZE to the start of the query. Did the index help?

EXPLAIN ANALYZE SELECT \*

FROM customerS

WHERE first\_name = 'David';

EXPLAIN ANALYZE SELECT \*

FROM customers

WHERE last\_name = 'Jones';

### TABLE: Query Results

|  |
| --- |
| **QUERY PLAN** |
| Seq Scan on customers (cost=0.00..2614.00 rows=500 width=466) (actual time=0.025..9.470 rows=538 loops=1) |
| Filter: ((first\_name)::text = 'David'::text) |
| Rows Removed by Filter: 99462 |
| Planning time: 10.458 ms |
| Execution time: 9.514 ms |

|  |
| --- |
| **QUERY PLAN** |
| Bitmap Heap Scan on customers (cost=12.29..1003.86 rows=500 width=466) (actual time=0.267..0.859 rows=1008 loops=1) |
| Recheck Cond: ((last\_name)::text = 'Jones'::text) |
| Heap Blocks: exact=714 |
| -> Bitmap Index Scan on customers\_last\_name\_idx (cost=0.00..12.17 rows=500 width=0) (actual time=0.192..0.192 rows=1008 loops=1) |
| Index Cond: ((last\_name)::text = 'Jones'::text) |
| Planning time: 0.120 ms |
| Execution time: 0.924 ms |

### Database Schema

### customers

|  |  |
| --- | --- |
| **name** | **type** |
| customer\_id | integer |
| first\_name | character varying |
| last\_name | character varying |
| email\_address | character varying |
| home\_phone | character varying |
| city | character varying |
| state\_name | character varying |
| years\_old | integer |

|  |
| --- |
| **Rows: 100000** |

How to Build an Index

In PostgreSQL, the CREATE INDEX keywords can be used to create an index on a column of a table. Say you wanted to create an index called customers\_user\_name\_idx on the customers table on the user\_name column, this is how you would do that:

CREATE INDEX customers\_user\_name\_idx ON customers (user\_name);

**QUESTION:** Create an index called customers\_city\_idx on the customers table for the city column.

If you would like to confirm your index was successfully built you can check the table to see, recall this is done by running:

SELECT \*  
FROM pg\_Indexes  
WHERE tablename = 'customers';

Checkpoint

**ANSWER:**

CREATE INDEX customers\_city\_idx ON customers(city);

**TABLE:**

### Query Results

### Database Schema

### customers

|  |  |
| --- | --- |
| **name** | **type** |
| customer\_id | integer |
| first\_name | character varying |
| last\_name | character varying |
| email\_address | character varying |
| home\_phone | character varying |
| city | character varying |
| state\_name | character varying |
| years\_old | integer |

|  |
| --- |
| **Rows: 100** |

Index Filtering

Queries that filter data often use WHERE and ON clauses. If an index is created on the columns referenced in these clauses, the database server will examine the index to see if it will improve the speed of the query.

Say you were asked to get the number of orders placed by each person with the last name of 'Smith' or 'Jones', you could get that by running the following query.

SELECT  
    c.first\_name,  
    c.last\_name,  
    COUNT(o.order\_id) AS NumOforders  
FROM customers       AS c  
INNER JOIN orders    AS o      
ON o.customer\_id = c.customer\_id  
WHERE c.last\_name IN ('Smith', 'Jones')  
GROUP BY c.first\_name, c.last\_name;

(Note: if the codeblock is hard to read, drag the boundary of this pane to make it wider.)

In this script, the WHERE clause is filtering the possible customers by the last\_name. If there is an index on customers.last\_name the database server will use this to quickly find the specific customers to examine.

Another filter in this query is the INNER JOIN between orders and customers on the customer\_id. If there are indexes on these columns (one on orders.customer\_id and another for customers.customer\_id) they could also be searched faster using the respective indexes.

Filtering isn’t perfect and when you add on ANDs and ORs it can complicate the filtering. Often real-world queries can get very complicated joining multiple tables each with an ON clause which may or may not use an index and multiple possible clauses in the WHERE.

**QUESTION:**

In the example above, the WHERE clause is filtering the customers table based on the last\_name it is interested in finding. Let’s add an index to improve future searches on this Table/Column. Don’t forget the naming convention for indexes,

If you would like to confirm your index was successfully built you can check the table to see, recall this is done by running:

SELECT \*  
FROM pg\_Indexes  
WHERE tablename = 'customers';

ANSWER: all information including add,phone number, city, status,age

Checkpoint

CREATE INDEX customers\_last\_name\_idx ON customers(last\_name);

**TABLE:**

### Query Results

### Database Schema

### customers

|  |  |
| --- | --- |
| **name** | **type** |
| customer\_id | integer |
| first\_name | character varying |
| last\_name | character varying |
| Multicolumn Indexes  you can combine multiple columns together as a single index. When using multicolumn indexes, the search structure will be based on the values found in all of the columns.  For example, an index on First and Last Name might be a good idea if it is common to search by both together in your situation. Consider a table where the last names 'Smith' and 'Johnson' appear many times. Having another filter for the first name can help you find someone named 'Sarah Smith' much faster.  The index is built in the specific order listed at creation, so (last\_name, first\_name) is different from (first\_name, last\_name). Keep this in mind when you are building your indexes as the order will impact the efficiency of your searches.  Say you want to find 'David', 'Rachel', and 'Margaret' from the first\_name column with the last\_name of 'Smith'. If there is an index (last\_name, first\_name), the server would find everyone with the last name 'Smith' then in that much smaller group, find the specific first names you are searching for. If the index is (first\_name, last\_name) the server would go to each of the first\_name records you are interested in and then search for the last name 'Smith' within each one. In general, there isn’t a right or wrong order, it’s about what is appropriate for your setup and what you expect the index to be used for. If there is a good use for it, you could create both indexes as well! If both are present, when you run your script, the database server will determine which index to use based on your query. But remember, indexes take up space, so you shouldn’t always create every index you can think of.  Recall the way to create an index is:  CREATE INDEX <index\_name> ON <table\_name> (<column\_name>);  For a multicolumn index you only need to list out each of the columns in the order you wish them to be used. So if we wanted to create an index called customers\_last\_name\_first\_name\_idx for the customers table for the combination of last\_name and first\_name it would be written like this  CREATE INDEX customers\_last\_name\_first\_name\_idx ON customers (last\_name, first\_name);  In theory, you could list as many columns as your table has. We will discuss later why, in most cases, this would not be a good use of an index.  As a note, you might hear a multicolumn index referred to by other names as well, such as Composite or Compound. Instructions **1.**  Run an EXPLAIN ANALYZE SELECT for all columns where last\_name is 'Jones' and first\_name is 'David' (remember PostgreSQL is case sensitive). Note that we haven’t created an index yet. Let’s see how long this query takes without an index.  Checkpoint 2 Passed  Stuck? Get a hint  **2.**  Create a multicolumn index on customers for last\_name and first\_name (in that order) called customers\_last\_name\_first\_name\_idx.  Checkpoint 3 Passed  Stuck? Get a hint  **3.**  Run the EXPLAIN ANALYZE SELECT again for all columns where last\_name is 'Jones' and first\_name is 'David' (remember PostgreSQL is case sensitive) now that we have created the multicolumn index, compare the results of the two runs.  Checkpoint  Answer: EXPLAIN ANALYZE SELECT \*  FROM customers  WHERE last\_name = 'Jones' AND  first\_name = 'David';  CREATE INDEX  customers\_last\_name\_first\_name\_idx ON customers(last\_name, first\_name);  EXPLAIN ANALYZE SELECT \*  FROM customers  WHERE last\_name = 'Jones' AND  first\_name = 'David'; Query Results  |  | | --- | | **QUERY PLAN** | | Seq Scan on customers (cost=0.00..1691.36 rows=1 width=466) (actual time=1.119..13.168 rows=9 loops=1) | | Filter: (((last\_name)::text = 'Jones'::text) AND ((first\_name)::text = 'David'::text)) | | Rows Removed by Filter: 99991 | | Planning time: 11.007 ms | | Execution time: 13.193 ms |  |  | | --- | | **QUERY PLAN** | | Bitmap Heap Scan on customers (cost=4.44..12.24 rows=2 width=466) (actual time=0.035..0.044 rows=9 loops=1) | | Recheck Cond: (((last\_name)::text = 'Jones'::text) AND ((first\_name)::text = 'David'::text)) | | Heap Blocks: exact=9 | | -> Bitmap Index Scan on customers\_last\_name\_first\_name\_idx (cost=0.00..4.44 rows=2 width=0) (actual time=0.031..0.031 rows=9 loops=1) | | Index Cond: (((last\_name)::text = 'Jones'::text) AND ((first\_name)::text = 'David'::text)) | | Planning time: 0.118 ms | | Execution time: 0.063 ms |  Database Schemacustomers  |  |  | | --- | --- | | **name** | **type** | | customer\_id | integer | | first\_name | character varying | | last\_name | character varying | | email\_address | character varying | | home\_phone | character varying | | city | character varying |   Drop an Index  In PostgreSQL, the DROP INDEX command can be used to drop an existing index. We will soon go over why you might want to drop an index you have built, but for now, let’s learn the syntax to drop an index.  Say we want to drop the index customers\_city\_idx, Note that we pair the DROP statement with the optional IF EXISTS to protect from execution errors.  DROP INDEX IF EXISTS customers\_city\_idx;  We will get into some situations where you might want to drop an index you have already created in the following exercises. For now, though, you need to know how to drop them when appropriate. Instructions **1.**  Examine the current indexes on the customers table. As a reminder, this is how you examine the indexes on a table.  SELECT \* FROM pg\_Indexes WHERE tablename = '<table\_name>';  Checkpoint 2 Passed  Stuck? Get a hint  **2.**  Drop the index customers\_last\_name\_idx  Checkpoint 3 Passed  Stuck? Get a hint  **3.**  After dropping the index, query pg\_Indexes for the customers table again to see the change.  Checkpoint 4 Passed  Answer: SELECT \*  FROM pg\_indexes  WHERE tablename = 'customers';  DROP INDEX IF EXISTS customers\_last\_name\_idx;  SELECT \*  FROM pg\_indexes  WHERE tablename = 'customers'; Query Results  |  |  |  |  |  | | --- | --- | --- | --- | --- | | **schemaname** | **tablename** | **indexname** | **tablespace** | **indexdef** | | cc\_user | customers | customers\_pkey |  | CREATE UNIQUE INDEX customers\_pkey ON cc\_user.customers USING btree (customer\_id) | | cc\_user | customers | customers\_last\_name\_idx |  | CREATE INDEX customers\_last\_name\_idx ON cc\_user.customers USING btree (last\_name) |  |  |  |  |  |  | | --- | --- | --- | --- | --- | | **schemaname** | **tablename** | **indexname** | **tablespace** | **indexdef** | | cc\_user | customers | customers\_pkey |  | CREATE UNIQUE INDEX customers\_pkey ON cc\_user.customers USING btree (customer\_id) |  Database Schemacustomers  |  |  | | --- | --- | | **name** | **type** | | customer\_id | integer | | first\_name | character varying | | last\_name | character varying | | email\_address | character varying | | home\_phone | character varying | | city | character varying |   Why not Index every Column?  Indexes speed up searching and filtering, however, they slow down insert, update, and delete statements. If you have multiple indexes on a single table and you insert a record, you will need to update each index associated with the table. This can make indexes very costly.  Updates and deletes have similar drawbacks. When deleting a record that is associated with an index, it might be faster to find the record — by leveraging the index’s ability to search. However, once the record is found, removing or editing it will result in the same issue as inserting a new record.  Note that if you’re updating a non-indexed column, that update will be unaffected by the index. So if you are updating a non-indexed column while filtering by one with an index, an update statement can actually be faster with an index.  SELECT NOW();  \COPY customers FROM 'customers.txt' DELIMITER ',' CSV HEADER;  SELECT NOW();  DROP INDEX IF EXISTS customers\_bad\_idx;  Table: Query Results  |  | | --- | | **now** | | 2021-09-27 14:06:49.974738+00 |  |  | | --- | | **now** | | 2021-09-27 14:06:51.481952+00 |  Database Schemacustomers  |  |  | | --- | --- | | **name** | **type** | | customer\_id | integer | | first\_name | character varying | | last\_name | character varying | | email\_address | character varying |   Why not Index every Column (cont)?  If you wanted to examine the size of a table products you would run:  SELECT pg\_size\_pretty (pg\_total\_relation\_size('products')); Instructions **1.**  Check the size of the customers table before we create an index on it.  Checkpoint 2 Passed  Stuck? Get a hint  **2.**  Create an index using the last\_name column in the customers table again to see how it has been impacted by just a single new index, call it customers\_last\_name\_idx.  Checkpoint 3 Passed  Stuck? Get a hint  **3.**  Examine the size of the table after your index is created. Notice that this is the impact of a single simple index. You can imagine how multple complex indexes could swell a table size quickly.  Checkpoint  SELECT pg\_size\_pretty  (pg\_total\_relation\_size('customers'));  CREATE INDEX customers\_last\_name\_idx ON customers(last\_name);  SELECT pg\_size\_pretty  (pg\_total\_relation\_size('customers'));  Table: Query Results  |  | | --- | | **pg\_size\_pretty** | | 13 MB |  |  | | --- | | **pg\_size\_pretty** | | 15 MB |  Database Schemacustomers  |  |  | | --- | --- | | **name** | **type** | | customer\_id | integer | | first\_name | character varying | | last\_name | character varying |   When should I add an Index?  For instance, if you have the following index on the customers table  CREATE INDEX customers\_idx ON customers (last\_name);  Then you ran the following two queries  EXPLAIN ANALYZE VERBOSE SELECT \* FROM customers WHERE last\_name = 'Jones'   AND first\_name = 'David';   EXPLAIN ANALYZE VERBOSE SELECT \* FROM customers WHERE last\_name = 'Jones'   OR first\_name = 'David';  The first would run in ~1ms, the second would take ~33ms (in a local version of the customers table with 100k records). Because the first one uses the index it can jump right to just the records with the last\_name = 'Jones'. However, in the second with the OR, the index is useless since every record has to be searched anyway for any record with first\_name = 'David'. The index is ignored and the system looks at every record once, checking both conditions. Instructions **1.**  Here are two situations. Only one might benefit from an index. Pick the correct one, and build the index for it. Feel free to look at the hint to see an explanation of why one is correct and the other is not.  1) Your boss asks you to run a one time query to find out why your website slowed down for users so much last night. Your boss asks you to create an index on the logons table on the logon\_time to find out quickly how many users logged in last night. It should be called logons\_logon\_time\_idx.  2) Your shipping department needs a list each night of the orders for the day organized by the city for each customers order. It should be called customers\_city\_idx.  Checkpoint 2 Passed  Stuck? Get a hint  **2.**  This time build the correct multicolumn index on the customers table. You find that searches for customers by last\_name are running slowly. You know you need to find their first\_name, last\_name, and email\_address. Create the index that might help with this (not all possible combinations are listed and there are other valid ones, please select from the below list to help with the checking for accuracy).  1)  CREATE INDEX customers\_first\_name\_last\_name\_email\_address\_idx ON customers (first\_name, last\_name, email\_address);  2)  CREATE INDEX customers\_last\_name\_email\_address\_first\_name\_idx ON customers (last\_name, email\_address, first\_name);  3)  CREATE INDEX customers\_email\_address\_first\_name\_last\_name\_idx ON customers (email\_address, first\_name, last\_name);  Answer:  Checkpoint  CREATE INDEX customers\_city\_idx ON customers(city);  CREATE INDEX customers\_last\_name\_email\_address\_first\_name\_idx ON customers (last\_name, email\_address, first\_name);  Table: Query ResultsDatabase Schemacustomers  |  |  | | --- | --- | | **name** | **type** | | customer\_id | integer | | first\_name | character varying | | last\_name | character varying | | email\_address | character varying | | home\_phone | character varying | | city | character varying | | state\_name | character varying | | years\_old | integer |  |  | | --- | | **Rows: 100** |  logons  |  |  | | --- | --- | | **name** | **type** | | customer\_id | integer | | logon\_time | timestamp without time zone |  |  | | --- | | **Rows: 5** |   Summarized   * What an index is and how they function. * How to see what indexes exist on a table   SELECT \* FROM pg\_indexes WHERE tablename = '<table\_name>';   * EXPLAIN ANALYZE can be a powerful tool to see how your queries are impacted by an index. * How to build an index   CREATE INDEX <index\_name> ON <table\_name> (column\_name);   * Multicolumn indexes allow for more than one column to be used in combination as an index on a table   CREATE INDEX <index\_name> ON <table\_name> (<column\_name1>, <column\_name2>...);   * You can drop an index. This might be useful to do if you are modifying a large number of records on an indexed table.   DROP INDEX IF EXISTS <index\_name>;   * To see the size of a database table you can run the script   SELECT pg\_size\_pretty (pg\_total\_relation\_size('<table\_name>'));   * Some of the benefits and burdens of indexes:   + Increase in speed of searches/filtering   + Increase in storage space   + Increase in runtime for Insert/Update/Delete on impacted indexes.   Partial Index  A partial index allows for indexing on a subset of a table, allowing searches to be conducted on just this group of records in the table. So in our example, you would be searching an index of ~258 Thousand instead of 70+ Million. Let’s assume that in our example the users are stored in a users table and we want an index based on user\_name. If we know that all internal employees have an email\_address ending in '@wellsfargo.com', we would write the partial index like this:  CREATE INDEX users\_user\_name\_internal\_idx ON users (user\_name) WHERE email\_address LIKE '%@wellsfargo.com';  Notice that the filtering of the index does not have to be for a column that is part of your index. Instructions **1.**  To get a good feel for the power of partial indexes you need to be working with very large data sets and searching for a small part of those tables regularly, something we can not simulate well in this environment.  Let’s suppose the company you are working for wants to run regular advertising targeting your teenage (13-19) customers, sounds like this might be a good use of a partial index.  Run EXPLAIN ANALYZE on the customers table when searching for records where the years\_old is a teenager. There is already an index built for years\_old.  If you are not familiar with BETWEEN this can save you time. So if you write WHERE years\_old BETWEEN 13 AND 19 the database server will treat this just as if you had written ‘WHERE years\_old >= 13 AND years\_old <= 19`.  Checkpoint 2 Passed  Stuck? Get a hint  **2.**  Create a partial index on the customers table for the years\_old column where the range is 13-19. Call it customers\_years\_old\_teen\_idx.  Checkpoint 3 Passed  Stuck? Get a hint  **3.**  Create another EXPLAIN ANALYZE after you create your partial index, so you should have one before and another after your partial index so you can see the impact the index has.  Take note of the difference in both the planning and execution times.  Checkpoint  ANSWER:  EXPLAIN ANALYZE SELECT \*  FROM customers  WHERE years\_old >= 13 AND years\_old <= 19;  CREATE INDEX customers\_years\_old\_teen\_idx ON customers(years\_old)  WHERE years\_old >= 13 AND years\_old <= 19 ;  EXPLAIN ANALYZE SELECT \*  FROM customers  WHERE years\_old >= 13 AND years\_old <= 19 ; Query Results  |  | | --- | | **QUERY PLAN** | | Bitmap Heap Scan on customers (cost=13.42..1006.23 rows=500 width=466) (actual time=0.785..3.684 rows=6882 loops=1) | | Recheck Cond: ((years\_old >= 13) AND (years\_old <= 19)) | | Heap Blocks: exact=1358 | | -> Bitmap Index Scan on customers\_years\_old\_idx (cost=0.00..13.29 rows=500 width=0) (actual time=0.628..0.628 rows=6882 loops=1) | | Index Cond: ((years\_old >= 13) AND (years\_old <= 19)) | | Planning time: 11.023 ms | | Execution time: 3.987 ms |  |  | | --- | | **QUERY PLAN** | | Bitmap Heap Scan on customers (cost=10.91..1003.72 rows=500 width=466) (actual time=0.618..2.519 rows=6882 loops=1) | | Recheck Cond: ((years\_old >= 13) AND (years\_old <= 19)) | | Heap Blocks: exact=1358 | | -> Bitmap Index Scan on customers\_years\_old\_teen\_idx (cost=0.00..10.78 rows=500 width=0) (actual time=0.456..0.456 rows=6882 loops=1) | | Planning time: 0.217 ms | | Execution time: 2.799 ms |  Database Schemacustomers  |  |  | | --- | --- | | **name** | **type** | | customer\_id | integer | | first\_name | character varying | | last\_name | character varying |   Order By  To specify the order of an index, you can add on the order you want your index sorted in when you create the index. Say you have a logins table that tracks the user\_name and date\_time each time a login occurs. If you wanted to check to see who has been logging in recently to use your site you could run:  SELECT     user\_name,     date\_time FROM logins WHERE date\_time >= (NOW() - INTERVAL'1 month') ORDER BY date\_time DESC;  If you were running this query regularly you could improve the speed by creating your index like this:  CREATE INDEX logins\_date\_time\_idx ON logins (date\_time DESC, user\_name);  You could also use ASC to switch the direction. If your column contains NULLs you can also specify the order they appear by adding NULLS FIRST or NULLS LAST to fit your needs. By default, PostgreSQL orders indexes by ascending order with NULLs last  Say we are regularly being asked for customer email (email\_address) from specific states (state\_name). The results should be ordered by state\_name (descending) then email\_address (ascending). Let’s start with our standard multicolumn index with no ordering so we can compare with the ordering later to see the difference. Create the index called customers\_state\_name\_email\_address\_idx.  Checkpoint 2 Passed  Stuck? Get a hint  **2.**  Now let’s get a baseline for the time it takes to get the data we are after using a multicolumn index without our specific order on it. Do an EXPLAIN ANALYZE SELECT for the state\_name and email\_address on the customers table where the state\_name is either 'California' or 'Ohio' and remember to order the results by descending state\_name then email\_address.  Checkpoint 3 Passed  Stuck? Get a hint  **3.**  Now create the ordered multicolumn index for this scenario so we can compare. Remember, we want state\_name (descending) then email\_address (ascending). Create this index called customers\_state\_name\_email\_address\_ordered\_idx.  Checkpoint 4 Passed  Stuck? Get a hint  **4.**  Now let’s compare our runtime now that we have an order by on our multicolumn index that matches the output we desire. Add the same EXPLAIN ANALYZE you ran before below your ordered index and run it to see the change in run times. As a reminder, it should be an EXPLAIN ANALYZE SELECT for the state\_name and email\_address on the customers table where the state\_name is either 'California' or 'Ohio' and remember to order the results by descending state\_name then email\_address.  Do you expect this query to run faster or slower? Why?  Checkpoint 5 Passed  ANSWER:  CREATE INDEX customers\_state\_name\_email\_address\_idx ON customers(state\_name, email\_address);  EXPLAIN ANALYZE SELECT state\_name, email\_address  FROM customers  WHERE state\_name = 'California' OR state\_name = 'Ohio'  ORDER BY state\_name DESC, email\_address;  CREATE INDEX customers\_state\_name\_email\_address\_ordered\_idx ON customers(state\_name DESC, email\_address ASC);  EXPLAIN ANALYZE SELECT state\_name, email\_address  FROM customers  WHERE state\_name = 'California' OR state\_name = 'Ohio'  ORDER BY state\_name DESC, email\_address;  TABLE: Query Results  |  | | --- | | **QUERY PLAN** | | Sort (cost=3.51..3.51 rows=2 width=211) (actual time=0.041..0.043 rows=16 loops=1) | | Sort Key: state\_name DESC, email\_address | | Sort Method: quicksort Memory: 26kB | | -> Seq Scan on customers (cost=0.00..3.50 rows=2 width=211) (actual time=0.007..0.019 rows=16 loops=1) | | Filter: (((state\_name)::text = 'California'::text) OR ((state\_name)::text = 'Ohio'::text)) | | Rows Removed by Filter: 84 | | Planning time: 6.729 ms | | Execution time: 0.073 ms |  |  | | --- | | **QUERY PLAN** | | Sort (cost=3.51..3.51 rows=2 width=211) (actual time=0.028..0.030 rows=16 loops=1) | | Sort Key: state\_name DESC, email\_address | | Sort Method: quicksort Memory: 26kB | | -> Seq Scan on customers (cost=0.00..3.50 rows=2 width=211) (actual time=0.006..0.017 rows=16 loops=1) | | Filter: (((state\_name)::text = 'California'::text) OR ((state\_name)::text = 'Ohio'::text)) | | Rows Removed by Filter: 84 | | Planning time: 0.110 ms | | Execution time: 0.040 ms |  Database Schemacustomers  |  |  | | --- | --- | | **name** | **type** | | customer\_id | integer | | first\_name | character varying | | last\_name | character varying |   Primary Keys and Indexes  PostgreSQL automatically creates a unique index on any primary key you have in your tables. It will also do this for any column you define as having a unique constraint. A unique index, primary key, and unique constraint all reject any attempt to have two records in a table that would have the same value (multicolumns versions of these would reject any record where all the columns are equal).  Recall that to view any indexes already associated with a table, such as products, including the index automatically created by the system upon creation of a primary key, there is a system view you can query against.  SELECT \* FROM pg\_Indexes WHERE tablename = 'products';  As a note, the primary key index standard is to end in \_pkey instead of \_idx to identify it as a specific type of index. It is also the way the system names it when created automatically. Instructions **1.**  The primary key for the customers table has been removed by a careless developer, please recreate it on customer\_id.  Checkpoint 2 Passed  Stuck? Get a hint  **2.**  Examine the table structure to see that an index exists on the primary key you created.  ANSWER:  ALTER TABLE customers ADD PRIMARY KEY (customer\_id);  SELECT \*  FROM pg\_Indexes  WHERE tablename = 'customers';  TABLE: Query Results  |  |  |  |  |  | | --- | --- | --- | --- | --- | | **schemaname** | **tablename** | **indexname** | **tablespace** | **indexdef** | | cc\_user | customers | customers\_pkey |  | CREATE UNIQUE INDEX customers\_pkey ON cc\_user.customers USING btree (customer\_id) |  Database Schemacustomers  |  |  | | --- | --- | | **name** | **type** | | customer\_id | integer | | first\_name | character varying |   **Clustered Index:** A PostgreSQL database can have two types of indexes - clustered and non-clustered.  However, a table can only have one clustered index. This index physically changes the storage of the data in long term memory whereas a non-clustered index is a separate organization that references back to the original data.  To cluster your database table using an existing index (say products\_product\_name\_idx) on the products table you would use:  CLUSTER products USING products\_product\_name\_idx;  If you have already established what index should be clustered on you can simply tell the system which table to apply the cluster on.  CLUSTER products;  And if you want to cluster every table in your database that has an identified index to use you can simply call  CLUSTER; Instructions **1.**  Let’s say that the table we created a primary key on in the previous lesson (customers.last\_name) is most often searched by this column. Let’s create a clustered index on this column. The index customers\_last\_name\_idx already exists, please make it the clustered index.  Checkpoint 2 Passed  Stuck? Get a hint  **2.**  Let’s say that the customers table we just created a clustered index on has had a significant number of changes and now needs to be reclustered. Recluster this table.  Checkpoint  **ANSWER:**  CLUSTER customers USING customers\_last\_name\_idx;  CLUSTER customers;  /\*  in 1st line customers is table name and index is customer..... \_inx hai... USING is the keyword..  \*/  TABLE: Query ResultsDatabase Schemacustomers  |  |  | | --- | --- | | **name** | **type** | | customer\_id | integer | | first\_name | character varying | | last\_name | character varying |  Non-Clustered Indexes In PostgreSQL, a table can have multiple non-clustered indexes. These indexes create a key(s) and a pointer back to the table where the rest of the information can be found.  Checkpoint 3 Pas |  |

QUESTION: We are trying to keep an eye on our warehouses to ensure that they are near as many of our customers as possible and we have to check regularly as we gain and lose customers. Create an index for filtering on state\_name, call it customers\_state\_name\_idx. Remember, PostgreSQL defaults all indexes to non-clustered unless you specify one to be clustered. So you do not need to do anything special when creating your index for it to be non-clustered.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Let’s use our index now to find the last\_name of everyone in 'Texas' ordered by last\_name. Think about what information is found in the index and what information the database server uses the key/pointer combination to find the information in the table (answer in the hint).

In order to pass this step, have your query return last\_name followed by state\_name.

Checkpoint 3 Passed

ANSWER:

CREATE INDEX customers\_state\_name\_idx ON customers(state\_name);

SELECT last\_name, state\_name

FROM customers

WHERE state\_name = 'Texas'

ORDER BY last\_name;

TABLE:

### Query Results

|  |  |
| --- | --- |
| **last\_name** | **state\_name** |
| Campbell | Texas |
| Harris | Texas |

### Database Schema

### customers

|  |  |
| --- | --- |
| **name** | **type** |
| customer\_id | integer |
| first\_name | character varying |
| last\_name | character varying |

Index-Only Scans

CREATE INDEX customers\_idx ON customers (last\_name, first\_name);

This will improve the speed when searching for customers by last\_name and first\_name. What happens when we frequently want to know the customers email\_address as well? For each record found, it will use the index to find a pointer then look up the email\_address matched to that record found in the index to return the last\_name, first\_name, and email\_address. If you include the information that is regularly looked for, even if it isn’t used in the filtering, as part of the index, a secondary search can be avoided. So in this example, you could add email\_address as another column in the index to prevent the lookup step. Remember the order the columns are in when creating the index should be whatever is most useful for your particular situation for searches and filtering.

### Instructions

**1.**

There is already an index on last\_name. Let’s run a query with EXPLAIN ANALYZE looking for first\_name, last\_name, email\_address where last\_name = 'Smith'.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Create a new index based on the example index to prevent a secondary search when looking for first\_name, email\_address based on last\_name. Call it customers\_last\_name\_first\_name\_email\_address\_idx (put the coulmns in the order of the name).

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Create another EXPLAIN ANALYZE after you create your index, so you should have one before and another after your CREATE INDEX so you can see the impact the index has. The search should again be looking for first\_name, last\_name, email\_address where last\_name = 'Smith' and compare with results from step 1. You will notice the speed improvement as the system no longer has to go back to the actual table to find the information you are requesting.

Checkpoint 4 Passed

EXPLAIN ANALYZE SELECT \*

FROM customers

WHERE last\_name = 'Smith';

CREATE INDEX customers\_last\_name\_first\_name\_email\_address\_idx ON customers(last\_name, first\_name, email\_address);

EXPLAIN ANALYZE SELECT \*

FROM customers

WHERE last\_name = 'Smith';

TABLE:

### Query Results

|  |
| --- |
| **QUERY PLAN** |
| Bitmap Heap Scan on customers (cost=12.29..1003.86 rows=500 width=466) (actual time=0.367..1.799 rows=993 loops=1) |
| Recheck Cond: ((last\_name)::text = 'Smith'::text) |
| Heap Blocks: exact=713 |
| -> Bitmap Index Scan on customers\_idx (cost=0.00..12.17 rows=500 width=0) (actual time=0.282..0.282 rows=993 loops=1) |
| Index Cond: ((last\_name)::text = 'Smith'::text) |
| Planning time: 8.203 ms |
| Execution time: 1.891 ms |

|  |
| --- |
| **QUERY PLAN** |
| Bitmap Heap Scan on customers (cost=12.29..1003.86 rows=500 width=466) (actual time=0.173..0.830 rows=993 loops=1) |
| Recheck Cond: ((last\_name)::text = 'Smith'::text) |
| Heap Blocks: exact=713 |
| -> Bitmap Index Scan on customers\_idx (cost=0.00..12.17 rows=500 width=0) (actual time=0.097..0.098 rows=993 loops=1) |
| Index Cond: ((last\_name)::text = 'Smith'::text) |
| Planning time: 0.124 ms |
| Execution time: 0.884 ms |

### Database Schema

### customers

|  |  |
| --- | --- |
| **name** | **type** |
| customer\_id | integer |
| first\_name | character varying |
| last\_name | character varying |

Combining Indexes

* A single multicolumn index is faster (if ordered well) than combining indexes.
* A multicolumn index is less efficient than a single index in cases where a single index is needed.
* You could create all of them, then the server will try to use the best one in each case, but if they are all not used relatively often/equally then this is a misuse of indexes.

Take for example, searching for first\_name and last\_name in the customers table.

* If searches are most often for only one of the columns, that should be your index.
* If searches are most often last\_name and first\_name then you should have a multicolumn index.
* If the searches are frequent and evenly spread among; first\_name alone, last\_name alone, and the combination of the two, that is a situation where you would want to have all three indexes.

### Instructions

**1.**

Examine runtime of query for customers where last\_name = 'Jones' and first\_name = 'Steve' where the system is combining indexes. An index on first\_name and another on last\_name already existes.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Create an index for last\_name and first\_name on customers. Call it customers\_last\_name\_first\_name\_idx.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Examine runtime of query for customers where last\_name = 'Jones' and first\_name = 'Steve' and compare with the results of the first run with separate indexes were used. The specifics of what the EXPLAIN ANALYZE is showing you in the output window is beyond this lesson. But you will notice that the query takes fewer steps to get to the same answer. This is all having to do with having all the requested information in one place.

Checkpoint

EXPLAIN ANALYZE SELECT \*

FROM customers

WHERE last\_name = 'Jones'

  AND first\_name = 'Steve';

CREATE INDEX customers\_last\_name\_first\_name\_idx ON customers (last\_name, first\_name);

EXPLAIN ANALYZE SELECT \*

FROM customers

WHERE last\_name = 'Jones'

  AND first\_name = 'Steve';

TABLE:

### Query Results

|  |
| --- |
| **QUERY PLAN** |
| Bitmap Heap Scan on customers (cost=24.59..32.39 rows=2 width=466) (actual time=0.028..0.029 rows=0 loops=1) |
| Recheck Cond: (((first\_name)::text = 'Steve'::text) AND ((last\_name)::text = 'Jones'::text)) |
| -> BitmapAnd (cost=24.59..24.59 rows=2 width=0) (actual time=0.027..0.027 rows=0 loops=1) |
| -> Bitmap Index Scan on customers\_first\_name\_idx (cost=0.00..12.17 rows=500 width=0) (actual time=0.027..0.027 rows=0 loops=1) |
| Index Cond: ((first\_name)::text = 'Steve'::text) |
| -> Bitmap Index Scan on customers\_last\_name\_idx (cost=0.00..12.17 rows=500 width=0) (never executed) |
| Index Cond: ((last\_name)::text = 'Jones'::text) |
| Planning time: 0.317 ms |
| Execution time: 0.067 ms |

|  |
| --- |
| **QUERY PLAN** |
| Bitmap Heap Scan on customers (cost=4.44..12.24 rows=2 width=466) (actual time=0.036..0.037 rows=0 loops=1) |
| Recheck Cond: (((last\_name)::text = 'Jones'::text) AND ((first\_name)::text = 'Steve'::text)) |
| -> Bitmap Index Scan on customers\_last\_name\_first\_name\_idx (cost=0.00..4.44 rows=2 width=0) (actual time=0.035..0.035 rows=0 loops=1) |
| Index Cond: (((last\_name)::text = 'Jones'::text) AND ((first\_name)::text = 'Steve'::text)) |
| Planning time: 0.131 ms |
| Execution time: 0.057 ms |

### Database Schema

### customers

|  |  |
| --- | --- |
| **name** | **type** |
| customer\_id | integer |
| first\_name | character varying |
| last\_name | character varying |

Indexes Based On Expressions

For example, if you want to ensure the company\_name in a manufactures table is unique, you can add the UNIQUE option to make a unique index constraint on the results on your index. Any duplicate will then be rejected. Using UNIQUE here tells the system that your index also needs to be a constraint and only allow one record in the system that matches the criteria for your index. In other words, by creating an index with UNIQUE the system will automatically create the constraint to match the logic in the index at the same time. Just like the creation of a constraint, if you try to create an index in this way where the data already in the table does not pass, the system will reject your creation and notify you of the issue.

Let’s look at our UNIQUE example a bit more. In PostgreSQL, 'ExampleCompany' is NOT the same thing as 'examplecompany' even though we would probably want to reject this as a duplicate. You can add a function on your index to convert all your company\_name data to lower case by using LOWER. This ensures that 'ExampleCompany' would be considered the same as 'examplecompany'. This combination of the UNIQUE constraint and the use of the function LOWER would look like this:

CREATE UNIQUE INDEX unique\_manufacture\_company\_name\_idx ON manufacture(LOWER(company\_name));

These special indexes compound the pros and cons of indexes. Because the results of the expression are stored in the index, it saves the search function from having to perform it on every row on future searches. However, every change in the table data that impacts the index means it has to do the expression again, making Inserts and Updates more expensive on these indexes than a basic index. Be especially thoughtful about when to use indexes that use functions or expressions.

### Instructions

**1.**

Your site wants to ensure that all email addresses entered by users when creating a new account do not already exist. Create a unique index to quickly search and verify this, call it customers\_email\_address\_lower\_unique\_idx.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Let’s check to make sure the constraint and index are working correctly. Try to insert duplicate emails with different capitalization, such as [test@sample.com](mailto:test@sample.com) and [Test@sample.com](mailto:Test@sample.com), which we would expect to fail. In the table creation script, the first\_name and last\_name fields were declared as NOT NULL which means they are required to contain information so you will need to put something in for those as well.

Your insert script should look something like this:

INSERT INTO customers (first\_name, last\_name, email\_address)  
VALUES ('<first\_name1>', '<last\_name2>', '<email\_address>'),  
('<first\_name2>','<last\_name2>','<email\_address>');

the <first\_name1>, <last\_name1>, <first\_name2>, and <last\_name2> can be any valid string you like.

In order to test our index, the two <email\_address> fields should be the same string with different capitalization such as 'example@sample.com' and 'ExaMPle@SampLE.COM'. This will test the use of the LOWER function.

If you did this correctly, you should see an error message. Read this error message and think about why this error message is appearing.

ANSWER:

Checkpoint 3 Passed

CREATE UNIQUE INDEX customers\_email\_address\_lower\_unique\_idx ON customers(LOWER(email\_address));

INSERT INTO customers (first\_name, last\_name, email\_address)

VALUES ('me', 'here', 'test@sample.com'),

('john','smith','Test@sample.com');

TABLE:

### Query Results

### Database Schema

### customers

|  |  |
| --- | --- |
| **name** | **type** |
| customer\_id | integer |
| first\_name | character varying |
| last\_name | character varying |

* How to build a partial index

CREATE INDEX <index\_name> ON <table\_name> (<column\_name>)  
WHERE <condition>;

* How to improve the speed of ordering data
* The Relationship between:
  + Primary keys and unique indexes
  + Unique constraints and unique indexes.
* What a clustered index is and how to refresh one

CLUSTER <table\_name> USING <index\_name>;

* How non-clustered indexes work.
* Index-only scans
* Combining indexes vs multicolumn indexes
* Indexes based on expressions

CREATE INDEX <index\_name> ON <table\_name>(<EXP>(<column\_name>));

SOME TIPS REGARDING ABOUT SQL

## 1. Define business requirements first

We’ve covered [best practices to define business requirements for BI](https://www.sisense.com/blog/requirements-elicitation-enterprise-business-analytics/) elsewhere. Definitely make sure you’re applying those practices when optimizing [SQL queries](https://www.sisense.com/blog/sql-cheat-sheet-retrieving-column-description-sql-server/), including:

* **Identify relevant stakeholders.** Make sure all necessary parties are in the discussion to develop your query. When querying production databases, make sure the DBA team is included.
* **Focus on business outcomes.** Give the query a definite and unique purpose. Taxing the production database for exploratory or duplicative reports is an unnecessary risk.
* **Frame the discussion for optimal requirements.** Define the function and scope of the report by identifying its intended audience. This will focus the query on the tables with the correct level of detail.
* **Ask great questions.** Follow the 5 W’s: Who? What? Where? When? Why?
* **Write very specific requirements and confirm them with stakeholders.** The performance of the production database is too critical to have unclear or ambiguous requirements. Make sure the requirements are as specific as possible and confirm the requirements with all stakeholders before running the query.

## 2. SELECT fields instead of using SELECT \*

When running exploratory queries, many SQL developers use **SELECT \*** (read as “select all”) as a [shorthand to query](https://www.sisense.com/blog/sql-symbol-cheatsheet/) all available data from a table. However, if a table has many fields and many rows, this taxes database resources by querying a lot of unnecessary data.

Using the **SELECT** statement will point the database to querying only the data you need to meet the business requirements. Here’s an example where the business requirements request mailing addresses for customers.

**Inefficient:**

SELECT \*  
FROM Customers

This query may pull in other data also stored in the customer table, such as phone numbers, activity dates, and notes from sales and customer service.

**Efficient:**

SELECT FirstName, LastName, Address, City, State, Zip  
FROM Customers

This query is much cleaner and only pulls the required information for mailing addresses.

To keep an index of all tables and field names, run a query from a system table such as INFORMATION\_SCHEMA or ALL\_TAB\_COLUMNS (for MS SQL Server, read [this](https://www.mssqltips.com/sqlservertutorial/183/informationschemacolumns/)).

**Get started today with our free SQL starter kit:**

[Get Starter Kit](https://www.sisense.com/whitepapers/sql-analytics-best-practices-tips-and-tricks/)

## 3. Avoid SELECT DISTINCT

**SELECT DISTINCT** is a handy way to remove duplicates from a query. **SELECT DISTINCT** works by **GROUP**ing all fields in the query to create distinct results. To accomplish this goal however, a large amount of processing power is required. Additionally, data may be grouped [to the point of being inaccurate](https://www.sisense.com/blog/understanding-simpsons-paradox-to-avoid-faulty-conclusions/). To avoid using **SELECT DISTINCT**, select more fields to create unique results.

**Inefficient and inaccurate:**

SELECT DISTINCT FirstName, LastName, State  
FROM Customers

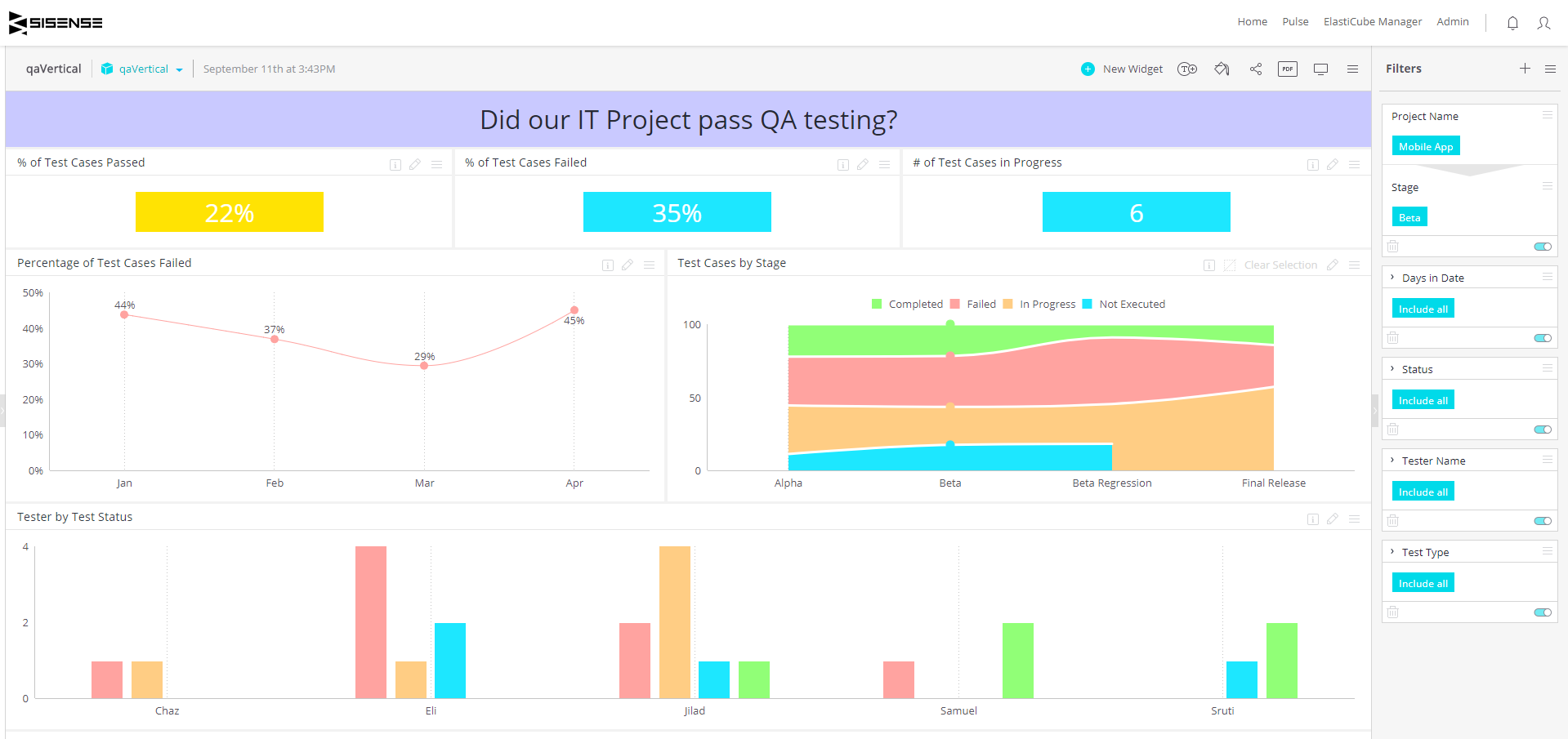
This query doesn’t account for multiple people in the same state having the same first and last name. Popular names such as David Smith or Diane Johnson will be grouped together, causing an inaccurate number of records. In larger databases, a large number of David Smiths and Diane Johnsons will cause this query to run slowly.

**Efficient and accurate:**

SELECT FirstName, LastName, Address, City, State, Zip  
FROM Customers

By adding more fields, unduplicated records were returned without using SELECT DISTINCT. The database does not have to group any fields, and the number of records is accurate.

**See Sisense in action:**

[[](https://www.sisense.com/blog/8-ways-fine-tune-sql-queries-production-databases/#lbf-3468)Explore Dashboard](https://www.sisense.com/blog/8-ways-fine-tune-sql-queries-production-databases/#lbf-3468)

## 4. Create joins with INNER JOIN (not WHERE)

Some SQL developers prefer to make joins with **WHERE** clauses, such as the following:

SELECT Customers.CustomerID, Customers.Name, Sales.LastSaleDate  
FROM Customers, Sales  
WHERE Customers.CustomerID = Sales.CustomerID

This type of join creates a Cartesian Join, also called a Cartesian Product or **CROSS JOIN**.

In a Cartesian Join, all possible combinations of the variables are created. In this example, if we had 1,000 customers with 1,000 total sales, the query would first generate 1,000,000 results, then filter for the 1,000 records where CustomerID is correctly joined. This is an inefficient use of database resources, as the database has done 100x more work than required. Cartesian Joins are especially problematic in large-scale databases, because a Cartesian Join of two large tables could create billions or trillions of results.

To prevent creating a Cartesian Join, use **INNER JOIN** instead:

SELECT Customers.CustomerID, Customers.Name, Sales.LastSaleDate  
FROM Customers  
   INNER JOIN Sales  
   ON Customers.CustomerID = Sales.CustomerID

The database would only generate the 1,000 desired records where CustomerID is equal.

Some DBMS systems are able to recognize **WHERE** joins and automatically run them as **INNER JOIN**s instead. In those DBMS systems, there will be no difference in performance between a **WHERE** join and **INNER JOIN**. However, **INNER JOIN** is recognized by all DBMS systems. Your [DBA](https://www.sisense.com/blog/the-6-functions-of-a-data-team/) will advise you as to which is best in your environment.

## 5. Use WHERE instead of HAVING to define filters

The goal of an efficient query is to pull only the required records from the database. Per the [SQL Order of Operations](https://www.sisense.com/blog/sql-query-order-of-operations/), **HAVING** statements are calculated after WHERE statements. If the intent is to filter a query based on conditions, a WHERE statement is more efficient.

For example, let’s assume 200 sales have been made in the year 2016, and we want to query for the number of sales per customer in 2016.

SELECT Customers.CustomerID, Customers.Name, Count(Sales.SalesID)  
FROM Customers  
   INNER JOIN Sales  
   ON Customers.CustomerID = Sales.CustomerID  
GROUP BY Customers.CustomerID, Customers.Name  
HAVING Sales.LastSaleDate BETWEEN #1/1/2016# AND #12/31/2016#

This query would pull 1,000 sales records from the Sales table, then filter for the 200 records generated in the year 2016, and finally count the records in the dataset.

In comparison, **WHERE** clauses limit the number of records pulled:

SELECT Customers.CustomerID, Customers.Name, Count(Sales.SalesID)  
FROM Customers  
  INNER JOIN Sales  
  ON Customers.CustomerID = Sales.CustomerID  
WHERE Sales.LastSaleDate BETWEEN #1/1/2016# AND #12/31/2016#  
GROUP BY Customers.CustomerID, Customers.Name

This query would pull the 200 records from the year 2016, and then count the records in the dataset. The first step in the **HAVING** clause has been completely eliminated.

**HAVING** should only be used when filtering on an aggregated field. In the query above, we could additionally filter for customers with greater than 5 sales using a HAVING statement.

SELECT Customers.CustomerID, Customers.Name, Count(Sales.SalesID)  
FROM Customers  
   INNER JOIN Sales  
   ON Customers.CustomerID = Sales.CustomerID  
WHERE Sales.LastSaleDate BETWEEN #1/1/2016# AND #12/31/2016#  
GROUP BY Customers.CustomerID, Customers.Name  
HAVING Count(Sales.SalesID) > 5

**>>Free Starter Kit:**[Get started today with our free SQL starter kit](https://www.sisense.com/whitepapers/sql-analytics-best-practices-tips-and-tricks/?gem)

## 6. Use wildcards at the end of a phrase only

When searching plaintext data, such as cities or names, wildcards create the widest search possible. However, the widest search is also the most inefficient search.

When a leading wildcard is used, especially in combination with an ending wildcard, the database is tasked with searching all records for a match anywhere within the selected field.

Consider this query to pull cities beginning with ‘Char’:

SELECT City FROM Customers  
WHERE City LIKE ‘%Char%’

This query will pull the expected results of **Char**leston, **Char**lotte, and **Char**lton. However, it will also pull unexpected results, such as Cape **Char**les, Crab Or**char**d, and **Rich**ardson.

A more efficient query would be:

SELECT City FROM Customers  
WHERE City LIKE ‘Char%’

This query will pull only the expected results of **Char**leston, **Char**lotte, and **Char**lton.

## 7. Use LIMIT to sample query results

Before running a query for the first time, ensure the results will be desirable and meaningful by using a **LIMIT** statement. (In some DBMS systems, the word TOP is used interchangeably with LIMIT.) The LIMIT statement returns only the number of records specified. Using a **LIMIT** statement prevents taxing the production database with a large query, only to find out the query needs editing or refinement.

In the 2016 sales query from above, we will examine a limit of 10 records:

SELECT Customers.CustomerID, Customers.Name, Count(Sales.SalesID)  
FROM Customers  
  INNER JOIN Sales  
  ON Customers.CustomerID = Sales.CustomerID  
WHERE Sales.LastSaleDate BETWEEN #1/1/2016# AND #12/31/2016#  
GROUP BY Customers.CustomerID, Customers.Name  
LIMIT 10

We can see by the sample whether we have a useable data set or not.

## 8. Run your query during off-peak hours

In order to minimize the impact of your analytical queries on the production database, talk to a DBA about scheduling the query to run at an off-peak time. The query should run when concurrent users are at their lowest number, which is typically the middle of the night (3 – 5 a.m.).

The more of the following criteria your query has, the more likely of a candidate it should be to run at night:

* Selecting from large tables (>1,000,000 records)
* Cartesian Joins or CROSS JOINs
* Looping statements
* SELECT DISTINCT statements
* Nested subqueries
* Wildcard searches in long text or memo fields
* Multiple schema queries

## Query confidently

With these tips in mind (plus some other [SQL tips and tricks](https://www.sisense.com/whitepapers/sql-analytics-best-practices-tips-and-tricks/) in your pocket) you should be able to build efficient, beautiful queries that will run smoothly and return the game-changing insights your team needs.

The purpose of benchmarking a database is not only to check capability of database, but also the behavior of a particular database against your application. Different hardwares provide different results based on the benchmarking plan that you set. It is very important to isolate the server (the actual one being benchmarked) from other elements like the servers driving the load, or the servers used to collect and store performance metrics. As part of the benchmarking exercise, you must get the application characteristics like a) Is the application is read or write intensive? or b) what is the read/write split (e.g. 80:20)? or c) How large is the dataset?, is the data and structure representative of the actual production database, etc.

[PostgreSQL](https://severalnines.com/product/clustercontrol/for_postgresql) is world's most advanced open source database. If any enterprise RDBMS customer wants to migrate their database to opensource, then PostgreSQL would be the first option to evaluate.

This post covers the following:

* How to benchmark PostgreSQL
* What are the key performance factors in PostgreSQL
* What are levers you can pull to increase performance
* What are performance pitfalls to avoid
* What are common mistakes people make?
* How do you know if your system is performing? What tools can you use?

## How to Benchmark PostgreSQL

The standard tool to benchmark PostgreSQL is [pgbench](https://www.postgresql.org/docs/devel/static/pgbench.html). By default, pgbench tests are based on TPC-B. It involves 5 SELECT, INSERT, and UPDATE commands per transaction. However, depending on your application behavior, you can write your own script files. Let us look into the default and some script oriented test results. We are going to use the latest version of PostgreSQL for these tests, which is PostgreSQL 10 at the time of writing. You can install it using [ClusterControl](https://severalnines.com/product/clustercontrol/for_postgresql), or using the instructions here: <https://www.openscg.com/bigsql/package-manager/>.

### Specs of machine

Version: RHEL 6 - 64 bit  
Memory : 4GB  
Processors: 4  
Storage: 50G  
PostgreSQL version: 10.0  
Database Size: 15G

Before you run benchmarking with pgbench tool, you would need to initialize it below command:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | -bash-4.1$ ./pgbench -i -p 5432 -d postgres  NOTICE:  table "pgbench\_history" does not exist, skipping  NOTICE:  table "pgbench\_tellers" does not exist, skipping  NOTICE:  table "pgbench\_accounts" does not exist, skipping  NOTICE:  table "pgbench\_branches" does not exist, skipping  creating tables…  100000 of 100000 tuples (100%) done (elapsed 0.18 s, remaining 0.00 s)  Vacuum…  set primary keys…  done. |

As shown in the NOTICE messages, it creates pgbench\_history, pgbench\_tellers, pgbench\_accounts, and pgbench\_branches tables to run the transactions for benchmarking.

Here is a simple test with 10 clients:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | -bash-4.1$ ./pgbench -c 10  starting vacuum...end.  transaction type: <builtin: TPC-B (sort of)>  scaling factor: 1  query mode: simple  number of clients: 10  number of threads: 1  number of transactions per client: 10  number of transactions actually processed: 100/100  latency average = 13.516 ms  tps = 739.865020 (including connections establishing)  tps = 760.775629 (excluding connections establishing) |

As you see, it ran with 10 clients and 10 transaction per client. It gave you 739 transactions/sec.It gave you 739 transactions/sec. If you want to run it for specific amount of time, you can use "-T" option. In general, a 15 mins or 30 mins run is sufficient.

As of now, we talked about how to run pgbench, however not about what should be options. Before you start the benchmarking, you should get proper details from application team on:

* What type of workload?
* How many concurrent sessions?
* What is the average result set of queries?
* What are the expected tps(transaction per sec)?

Here is an example for read-only work loads. You can use "-S" option to use only SELECTs which falls under read-only. Note that -n is to skip vacuuming on tables.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | -bash-4.1$ ./pgbench -c 100 -T 300 -S -n  transaction type: <builtin: select only>  scaling factor: 1000  query mode: simple  number of clients: 100  number of threads: 1  duration: 300 s  number of transactions actually processed: 15741  latency average = 1916.650 ms  tps = 52.174363 (including connections establishing)  tps = 52.174913 (excluding connections establishing)  -bash-4.1$ |

Latency here is the average elapsed transaction time of each statement executed by every client. It gives 52 tps with the hardware given. As this benchmark is for a read-only environment, let us try tweaking shared\_buffers and effective\_cache\_size parameters in postgresql.conf file and check the tps count. They are at default values in the above test, try increasing the values, and check the results.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | -bash-4.1$ ./pgbench -c 100 -T 300 -S -n  transaction type: <builtin: select only>  scaling factor: 1000  query mode: simple  number of clients: 100  number of threads: 1  duration: 300 s  number of transactions actually processed: 15215  latency average = 1984.255 ms  tps = 68.396758 (including connections establishing)  tps = 68.397322 (excluding connections establishing) |

Changing the parameters improved performance by 30%.

pgbench typically runs transactions on its own tables. If you have a workload of 50% reads and 50% writes (or a 60:40 environment), you can create a script file with a set of statements to achieve the expected workload.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | -bash-4.1$ cat /tmp/bench.sql  INSERT INTO test\_bench VALUES(1,'test');  INSERT INTO test\_bench VALUES(1,'test');  SELECT \* FROM test\_bench WHERE id=1;  SELECT \* FROM test\_bench WHERE id=2;  -bash-4.1$ ./pgbench -c 100 -T 300 -S -n -f /tmp/bench.sql  transaction type: multiple scripts  scaling factor: 1000  query mode: simple  number of clients: 100  number of threads: 1  duration: 300 s  number of transactions actually processed: 25436  latency average = 1183.093 ms  tps = 84.524217 (including connections establishing)  tps = 84.525206 (excluding connections establishing)  SQL script 1: <builtin: select only>   - weight: 1 (targets 50.0% of total)   - 12707 transactions (50.0% of total, tps = 42.225555)   - latency average = 914.240 ms   - latency stddev = 558.013 ms  SQL script 2: /tmp/bench.sql   - weight: 1 (targets 50.0% of total)   - 12729 transactions (50.0% of total, tps = 42.298662)   - latency average = 1446.721 ms   - latency stddev = 765.933 ms |

## What are the key Performance Factors in PostgreSQL

If we consider a real production environment, it is consolidated with different components at application level, hardware like CPU and memory, and the underlying operating system. We install PostgreSQL on top of the operating system to communicate with other components of the production environment. Every environment is different and overall performance will be degraded if it is not properly configured. In PostgreSQL, some queries run faster and some slow, however it depends on configuration that has been set. The goal of database performance optimization is to maximize the database throughput and minimize connections to achieve the largest possible throughput. Below are few key performance factors that affect the database:

* Workload
* Resource
* Optimization
* Contention

Workload consists of batch jobs, dynamic queries for online transactions, data analytics queries which are used for generating reports. Workload may be different during the period of the day, week or month, and depends on applications. Optimization of every database is unique. It can be database level configuration or query level optimization. We will be covering more about optimization in further sections of the post. Contention is the condition where two or more components of the workload are attempting to use a single resource in a conflicting way. As contention increases, throughput decreases.

## What are Tips and Best Practices

Here are few tips and best practices that you can follow to avoid performance issues:

* You can consider running maintenance activities like VACUUM and ANALYZE after a large modification in your database. This helps the planner to come up with the best plan to execute queries.
* Look for any need to index tables. It makes queries run much faster, rather than having to do full table scans.
* To make an index traversal much faster, you can use CREATE TABLE AS or CLUSTER commands to cluster rows with similar key values.
* When you see a performance problem, use the EXPLAIN command to look at the plan on how the optimizer has decided to execute your query.
* You can try changing the plans by influencing the optimizer by modifying query operators. For example, if you see a sequential scan for your query, you can disable seq scan using "SET ENABLE\_SEQSCAN TO OFF". There is no guarantee that the optimizer would not choose that operator if you disable it. The optimizer just considers the operator to be much more expensive. More details are here: <https://www.postgresql.org/docs/current/static/runtime-config-query.html>
* You can also try changing the costs parameters like CPU\_OPERATOR\_COST, CPU\_INDEX\_TUPLE\_COST, CPU\_TUPLE\_COST, RANDOM\_PAGE\_COST, and EFFECTIVE\_CACHE\_SIZE to influence the optimizer. More details are here: <https://www.postgresql.org/docs/current/static/runtime-config-query.html#RUNTIME-CONFIG-QUERY-CONSTANTS>
* Always filter data on the server rather than in client application. It will minimize the network traffic and gives better performance.
* To perform common operations, it is always recommended to use server-side procedures (triggers and functions). Server-side triggers or functions are parsed, planned, and optimized the first time they are used, not every time.

## What are Common Mistakes People Make

One of the common mistakes that people do is running the database server and database with default parameters. The PostgreSQL default configuration is tested in few environments, however not every application would find those values optimal. So you need to understand your application behavior and based on it, set your configuration parameters. You can use the pgTune tool to get values for your parameters based on the hardware that you are using. You can have a look at: <http://pgtune.leopard.in.ua/>. However, keep in mind that you will have to test your application with changes that you make, to see if there are any performance degradation with the changes.

Another thing to consider would be indexing the database. Indexes help to fetch the data faster, however more indexes create issues with loading the data. So always check if any unused indexes are there in the database, and get rid of those to reduce the maintenance of those indexes and improve loading of data.