# Data Cleaning Challenge: Handling missing values

**All days of the challange:**

* [Day 1: Handling missing values](https://www.kaggle.com/rtatman/data-cleaning-challenge-handling-missing-values)
* [Day 2: Scaling and normalization](https://www.kaggle.com/rtatman/data-cleaning-challenge-scale-and-normalize-data)
* [Day 3: Parsing dates](https://www.kaggle.com/rtatman/data-cleaning-challenge-parsing-dates/)
* [Day 4: Character encodings](https://www.kaggle.com/rtatman/data-cleaning-challenge-character-encodings/)
* [Day 5: Inconsistent Data Entry](https://www.kaggle.com/rtatman/data-cleaning-challenge-inconsistent-data-entry/)

Welcome to day 1 of the 5-Day Data Challenge! Today, we're going to be looking at how to deal with missing values. To get started, click the blue "Fork Notebook" button in the upper, right hand corner. This will create a private copy of this notebook that you can edit and play with. Once you're finished with the exercises, you can choose to make your notebook public to share with others. :)

**Your turn!** As we work through this notebook, you'll see some notebook cells (a block of either code or text) that has "Your Turn!" written in it. These are exercises for you to do to help cement your understanding of the concepts we're talking about. Once you've written the code to answer a specific question, you can run the code by clicking inside the cell (box with code in it) with the code you want to run and then hit CTRL + ENTER (CMD + ENTER on a Mac). You can also click in a cell and then click on the right "play" arrow to the left of the code. If you want to run all the code in your notebook, you can use the double, "fast forward" arrows at the bottom of the notebook editor.

Here's what we're going to do today:

* [Take a first look at the data](https://www.kaggle.com/code/rtatman/data-cleaning-challenge-handling-missing-values/notebook#Take-a-first-look-at-the-data)
* [See how many missing data points we have](https://www.kaggle.com/code/rtatman/data-cleaning-challenge-handling-missing-values/notebook#See-how-many-missing-data-points-we-have)
* [Figure out why the data is missing](https://www.kaggle.com/code/rtatman/data-cleaning-challenge-handling-missing-values/notebook#Figure-out-why-the-data-is-missing)
* [Drop missing values](https://www.kaggle.com/code/rtatman/data-cleaning-challenge-handling-missing-values/notebook#Drop-missing-values)
* [Filling in missing values](https://www.kaggle.com/code/rtatman/data-cleaning-challenge-handling-missing-values/notebook#Filling-in-missing-values)

Let's get started!

**Take a first look at the data**

The first thing we'll need to do is load in the libraries and datasets we'll be using. For today, I'll be using a dataset of events that occured in American Football games for demonstration, and you'll be using a dataset of building permits issued in San Francisco.

**Important!** Make sure you run this cell yourself or the rest of your code won't work!

# modules we'll use

import pandas as pd

import numpy as np

# read in all our data

nfl\_data = pd.read\_csv("../input/nflplaybyplay2009to2016/NFL Play by Play 2009-2017 (v4).csv")

sf\_permits = pd.read\_csv("../input/building-permit-applications-data/Building\_Permits.csv")

# set seed for reproducibility

np.random.seed(0)

/opt/conda/lib/python3.6/site-packages/IPython/core/interactiveshell.py:2698: DtypeWarning: Columns (25,51) have mixed types. Specify dtype option on import or set low\_memory=False.

interactivity=interactivity, compiler=compiler, result=result)

/opt/conda/lib/python3.6/site-packages/IPython/core/interactiveshell.py:2698: DtypeWarning: Columns (22,32) have mixed types. Specify dtype option on import or set low\_memory=False.

interactivity=interactivity, compiler=compiler, result=result)

The first thing I do when I get a new dataset is take a look at some of it. This lets me see that it all read in correctly and get an idea of what's going on with the data. In this case, I'm looking to see if I see any missing values, which will be reprsented with NaN or None.

# look at a few rows of the nfl\_data file. I can see a handful of missing data already!

nfl\_data.sample(5)

|  | **Date** | **GameID** | **Drive** | **qtr** | **down** | **time** | **TimeUnder** | **TimeSecs** | **PlayTimeDiff** | **SideofField** | **...** | **yacEPA** | **Home\_WP\_pre** | **Away\_WP\_pre** | **Home\_WP\_post** | **Away\_WP\_post** | **Win\_Prob** | **WPA** | **airWPA** | **yacWPA** | **Season** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **244485** | 2014-10-26 | 2014102607 | 18 | 3 | 1.0 | 00:39 | 1 | 939.0 | 12.0 | TB | ... | 1.240299 | 0.225647 | 0.774353 | 0.245582 | 0.754418 | 0.225647 | 0.019935 | -0.018156 | 0.038091 | 2014 |
| **115340** | 2011-11-20 | 2011112000 | 22 | 4 | 1.0 | 06:47 | 7 | 407.0 | 44.0 | OAK | ... | NaN | 0.056036 | 0.943964 | 0.042963 | 0.957037 | 0.943964 | 0.013073 | NaN | NaN | 2011 |
| **68357** | 2010-11-14 | 2010111401 | 8 | 2 | NaN | 00:23 | 1 | 1823.0 | 0.0 | CLE | ... | NaN | 0.365307 | 0.634693 | 0.384697 | 0.615303 | 0.634693 | -0.019390 | NaN | NaN | 2010 |
| **368377** | 2017-09-24 | 2017092405 | 24 | 4 | 1.0 | 08:48 | 9 | 528.0 | 8.0 | CLE | ... | 1.075660 | 0.935995 | 0.064005 | 0.921231 | 0.078769 | 0.064005 | 0.014764 | 0.003866 | 0.010899 | 2017 |
| **384684** | 2017-11-05 | 2017110505 | 11 | 2 | 1.0 | 09:15 | 10 | 2355.0 | 0.0 | DEN | ... | NaN | 0.928474 | 0.071526 | 0.934641 | 0.065359 | 0.071526 | -0.006166 | NaN | NaN | 2017 |

5 rows × 102 columns

Yep, it looks like there's some missing values. What about in the sf\_permits dataset?

# your turn! Look at a couple of rows from the sf\_permits dataset. Do you notice any missing data?

# your code goes here :)

**See how many missing data points we have**

Ok, now we know that we do have some missing values. Let's see how many we have in each column.

# get the number of missing data points per column

missing\_values\_count = nfl\_data.isnull().sum()

# look at the # of missing points in the first ten columns

missing\_values\_count[0:10]

Date 0

GameID 0

Drive 0

qtr 0

down 61154

time 224

TimeUnder 0

TimeSecs 224

PlayTimeDiff 444

SideofField 528

dtype: int64

That seems like a lot! It might be helpful to see what percentage of the values in our dataset were missing to give us a better sense of the scale of this problem:

# how many total missing values do we have?

total\_cells = np.product(nfl\_data.shape)

total\_missing = missing\_values\_count.sum()

# percent of data that is missing

(total\_missing/total\_cells) \* 100

24.87214126835169

Wow, almost a quarter of the cells in this dataset are empty! In the next step, we're going to take a closer look at some of the columns with missing values and try to figure out what might be going on with them.

# your turn! Find out what percent of the sf\_permits dataset is missing

**Figure out why the data is missing**

This is the point at which we get into the part of data science that I like to call "data intution", by which I mean "really looking at your data and trying to figure out why it is the way it is and how that will affect your analysis". It can be a frustrating part of data science, especially if you're newer to the field and don't have a lot of experience. For dealing with missing values, you'll need to use your intution to figure out why the value is missing. One of the most important question you can ask yourself to help figure this out is this:

**Is this value missing becuase it wasn't recorded or becuase it dosen't exist?**

If a value is missing becuase it doens't exist (like the height of the oldest child of someone who doesn't have any children) then it doesn't make sense to try and guess what it might be. These values you probalby do want to keep as NaN. On the other hand, if a value is missing becuase it wasn't recorded, then you can try to guess what it might have been based on the other values in that column and row. (This is called "imputation" and we'll learn how to do it next! :)

Let's work through an example. Looking at the number of missing values in the nfl\_data dataframe, I notice that the column TimesSec has a lot of missing values in it:

# look at the # of missing points in the first ten columns

missing\_values\_count[0:10]

Date 0

GameID 0

Drive 0

qtr 0

down 61154

time 224

TimeUnder 0

TimeSecs 224

PlayTimeDiff 444

SideofField 528

dtype: int64

By looking at [the documentation](https://www.kaggle.com/maxhorowitz/nflplaybyplay2009to2016), I can see that this column has information on the number of seconds left in the game when the play was made. This means that these values are probably missing because they were not recorded, rather than because they don't exist. So, it would make sense for us to try and guess what they should be rather than just leaving them as NA's.

On the other hand, there are other fields, like PenalizedTeam that also have lot of missing fields. In this case, though, the field is missing because if there was no penalty then it doesn't make sense to say *which* team was penalized. For this column, it would make more sense to either leave it empty or to add a third value like "neither" and use that to replace the NA's.

**Tip:** This is a great place to read over the dataset documentation if you haven't already! If you're working with a dataset that you've gotten from another person, you can also try reaching out to them to get more information.

If you're doing very careful data analysis, this is the point at which you'd look at each column individually to figure out the best strategy for filling those missing values. For the rest of this notebook, we'll cover some "quick and dirty" techniques that can help you with missing values but will probably also end up removing some useful information or adding some noise to your data.

**Your turn!**

* Look at the columns Street Number Suffix and Zipcode from the sf\_permits datasets. Both of these contain missing values. Which, if either, of these are missing because they don't exist? Which, if either, are missing because they weren't recorded?

**Drop missing values**

If you're in a hurry or don't have a reason to figure out why your values are missing, one option you have is to just remove any rows or columns that contain missing values. (Note: I don't generally recommend this approch for important projects! It's usually worth it to take the time to go through your data and really look at all the columns with missing values one-by-one to really get to know your dataset.)

If you're sure you want to drop rows with missing values, pandas does have a handy function, dropna() to help you do this. Let's try it out on our NFL dataset!

# remove all the rows that contain a missing value

nfl\_data.dropna()

|  | **Date** | **GameID** | **Drive** | **qtr** | **down** | **time** | **TimeUnder** | **TimeSecs** | **PlayTimeDiff** | **SideofField** | **...** | **yacEPA** | **Home\_WP\_pre** | **Away\_WP\_pre** | **Home\_WP\_post** | **Away\_WP\_post** | **Win\_Prob** | **WPA** | **airWPA** | **yacWPA** | **Season** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

0 rows × 102 columns

Oh dear, it looks like that's removed all our data! 😱 This is because every row in our dataset had at least one missing value. We might have better luck removing all the *columns* that have at least one missing value instead.

# remove all columns with at least one missing value

columns\_with\_na\_dropped = nfl\_data.dropna(axis=1)

columns\_with\_na\_dropped.head()

|  | **Date** | **GameID** | **Drive** | **qtr** | **TimeUnder** | **ydstogo** | **ydsnet** | **PlayAttempted** | **Yards.Gained** | **sp** | **...** | **Timeout\_Indicator** | **Timeout\_Team** | **posteam\_timeouts\_pre** | **HomeTimeouts\_Remaining\_Pre** | **AwayTimeouts\_Remaining\_Pre** | **HomeTimeouts\_Remaining\_Post** | **AwayTimeouts\_Remaining\_Post** | **ExPoint\_Prob** | **TwoPoint\_Prob** | **Season** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 2009-09-10 | 2009091000 | 1 | 1 | 15 | 0 | 0 | 1 | 39 | 0 | ... | 0 | None | 3 | 3 | 3 | 3 | 3 | 0.0 | 0.0 | 2009 |
| **1** | 2009-09-10 | 2009091000 | 1 | 1 | 15 | 10 | 5 | 1 | 5 | 0 | ... | 0 | None | 3 | 3 | 3 | 3 | 3 | 0.0 | 0.0 | 2009 |
| **2** | 2009-09-10 | 2009091000 | 1 | 1 | 15 | 5 | 2 | 1 | -3 | 0 | ... | 0 | None | 3 | 3 | 3 | 3 | 3 | 0.0 | 0.0 | 2009 |
| **3** | 2009-09-10 | 2009091000 | 1 | 1 | 14 | 8 | 2 | 1 | 0 | 0 | ... | 0 | None | 3 | 3 | 3 | 3 | 3 | 0.0 | 0.0 | 2009 |
| **4** | 2009-09-10 | 2009091000 | 1 | 1 | 14 | 8 | 2 | 1 | 0 | 0 | ... | 0 | None | 3 | 3 | 3 | 3 | 3 | 0.0 | 0.0 | 2009 |

5 rows × 41 columns

# just how much data did we lose?

print("Columns in original dataset: %d \n" % nfl\_data.shape[1])

print("Columns with na's dropped: %d" % columns\_with\_na\_dropped.shape[1])

Columns in original dataset: 102

Columns with na's dropped: 41

We've lost quite a bit of data, but at this point we have successfully removed all the NaN's from our data.

# Your turn! Try removing all the rows from the sf\_permits dataset that contain missing values. How many are left?

# Now try removing all the columns with empty values. Now how much of your data is left?

**Filling in missing values automatically**

Another option is to try and fill in the missing values. For this next bit, I'm getting a small sub-section of the NFL data so that it will print well.

# get a small subset of the NFL dataset

subset\_nfl\_data = nfl\_data.loc[:, 'EPA':'Season'].head()

subset\_nfl\_data

|  | **EPA** | **airEPA** | **yacEPA** | **Home\_WP\_pre** | **Away\_WP\_pre** | **Home\_WP\_post** | **Away\_WP\_post** | **Win\_Prob** | **WPA** | **airWPA** | **yacWPA** | **Season** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 2.014474 | NaN | NaN | 0.485675 | 0.514325 | 0.546433 | 0.453567 | 0.485675 | 0.060758 | NaN | NaN | 2009 |
| **1** | 0.077907 | -1.068169 | 1.146076 | 0.546433 | 0.453567 | 0.551088 | 0.448912 | 0.546433 | 0.004655 | -0.032244 | 0.036899 | 2009 |
| **2** | -1.402760 | NaN | NaN | 0.551088 | 0.448912 | 0.510793 | 0.489207 | 0.551088 | -0.040295 | NaN | NaN | 2009 |
| **3** | -1.712583 | 3.318841 | -5.031425 | 0.510793 | 0.489207 | 0.461217 | 0.538783 | 0.510793 | -0.049576 | 0.106663 | -0.156239 | 2009 |
| **4** | 2.097796 | NaN | NaN | 0.461217 | 0.538783 | 0.558929 | 0.441071 | 0.461217 | 0.097712 | NaN | NaN | 2009 |

We can use the Panda's fillna() function to fill in missing values in a dataframe for us. One option we have is to specify what we want the NaN values to be replaced with. Here, I'm saying that I would like to replace all the NaN values with 0.

# replace all NA's with 0

subset\_nfl\_data.fillna(0)

|  | **EPA** | **airEPA** | **yacEPA** | **Home\_WP\_pre** | **Away\_WP\_pre** | **Home\_WP\_post** | **Away\_WP\_post** | **Win\_Prob** | **WPA** | **airWPA** | **yacWPA** | **Season** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 2.014474 | 0.000000 | 0.000000 | 0.485675 | 0.514325 | 0.546433 | 0.453567 | 0.485675 | 0.060758 | 0.000000 | 0.000000 | 2009 |
| **1** | 0.077907 | -1.068169 | 1.146076 | 0.546433 | 0.453567 | 0.551088 | 0.448912 | 0.546433 | 0.004655 | -0.032244 | 0.036899 | 2009 |
| **2** | -1.402760 | 0.000000 | 0.000000 | 0.551088 | 0.448912 | 0.510793 | 0.489207 | 0.551088 | -0.040295 | 0.000000 | 0.000000 | 2009 |
| **3** | -1.712583 | 3.318841 | -5.031425 | 0.510793 | 0.489207 | 0.461217 | 0.538783 | 0.510793 | -0.049576 | 0.106663 | -0.156239 | 2009 |
| **4** | 2.097796 | 0.000000 | 0.000000 | 0.461217 | 0.538783 | 0.558929 | 0.441071 | 0.461217 | 0.097712 | 0.000000 | 0.000000 | 2009 |

I could also be a bit more savvy and replace missing values with whatever value comes directly after it in the same column. (This makes a lot of sense for datasets where the observations have some sort of logical order to them.)

# replace all NA's the value that comes directly after it in the same column,

# then replace all the reamining na's with 0

subset\_nfl\_data.fillna(method = 'bfill', axis=0).fillna(0)

|  | **EPA** | **airEPA** | **yacEPA** | **Home\_WP\_pre** | **Away\_WP\_pre** | **Home\_WP\_post** | **Away\_WP\_post** | **Win\_Prob** | **WPA** | **airWPA** | **yacWPA** | **Season** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 2.014474 | -1.068169 | 1.146076 | 0.485675 | 0.514325 | 0.546433 | 0.453567 | 0.485675 | 0.060758 | -0.032244 | 0.036899 | 2009 |
| **1** | 0.077907 | -1.068169 | 1.146076 | 0.546433 | 0.453567 | 0.551088 | 0.448912 | 0.546433 | 0.004655 | -0.032244 | 0.036899 | 2009 |
| **2** | -1.402760 | 3.318841 | -5.031425 | 0.551088 | 0.448912 | 0.510793 | 0.489207 | 0.551088 | -0.040295 | 0.106663 | -0.156239 | 2009 |
| **3** | -1.712583 | 3.318841 | -5.031425 | 0.510793 | 0.489207 | 0.461217 | 0.538783 | 0.510793 | -0.049576 | 0.106663 | -0.156239 | 2009 |
| **4** | 2.097796 | 0.000000 | 0.000000 | 0.461217 | 0.538783 | 0.558929 | 0.441071 | 0.461217 | 0.097712 | 0.000000 | 0.000000 | 2009 |

Filling in missing values is also known as "imputation", and you can find more exercises on it [in this lesson, also linked under the "More practice!" section](https://www.kaggle.com/dansbecker/handling-missing-values). First, however, why don't you try replacing some of the missing values in the sf\_permit dataset?

# Your turn! Try replacing all the NaN's in the sf\_permits data with the one that

# comes directly after it and then replacing any remaining NaN's with 0

And that's it for today! If you have any questions, be sure to post them in the comments below or [on the forums](https://www.kaggle.com/questions-and-answers).

Remember that your notebook is private by default, and in order to share it with other people or ask for help with it, you'll need to make it public. First, you'll need to save a version of your notebook that shows your current work by hitting the "Commit & Run" button. (Your work is saved automatically, but versioning your work lets you go back and look at what it was like at the point you saved it. It also let's you share a nice compiled notebook instead of just the raw code.) Then, once your notebook is finished running, you can go to the Settings tab in the panel to the left (you may have to expand it by hitting the [<] button next to the "Commit & Run" button) and setting the "Visibility" dropdown to "Public".

**More practice!**

If you're looking for more practice handling missing values, check out these extra-credit\* exercises:

* [Handling Missing Values](https://www.kaggle.com/dansbecker/handling-missing-values): In this notebook Dan shows you several approaches to imputing missing data using scikit-learn's imputer.
* Look back at the Zipcode column in the sf\_permits dataset, which has some missing values. How would you go about figuring out what the actual zipcode of each address should be? (You might try using another dataset. You can search for datasets about San Fransisco on the [Datasets listing](https://www.kaggle.com/datasets).)

\* no actual credit is given for completing the challenge, you just learn how to clean data real good :P

**SQL & Databases**

**Learning Objectives**

* Explore the landscape of SQL and databases and gain an understanding of how you will use them in your career as a data scientist
* Write best practice queries in SQL and be able to work with relational databases in Python

**Work to Complete**

In this unit, you'll:

* Practice writing SQL to solve business questions related to a country club case study

In this unit, you'll learn how to leverage Structured Query Language (SQL) to query relational database management systems. In other words, you'll use queries to understand the data contained in databases. Here's one example. Let's say you're asked to assess underperforming malls. One way to start your assessment would be to write a query to analyze daily foot traffic data stored in the malls' databases. Ready to learn more? Let's take a look.

**SQL & Databases Overview**

In this subunit, you’ll learn how to navigate through the wonderful world of databases using SQL. These tools are among the most powerful and impactful in the data scientist’s utility belt. First, we’ll give you the information you need to succeed then show you the current trends about popular databases, so you form a complete picture of the available tools. Lastly, we’ll show you a few videos about relational databases. Relational databases are the powerful technologies that enable our modern world: Larry Ellison, the man who popularized them, is the sixth wealthiest person alive. Our use of relational databased has shaped the field of data science.

1

**Introduction to SQL and Databases (8 pages)**

Open in new tab

Save



30 - 50 Minutes

11 Points

It’s time to meet your new sidekicks: SQL and Databases. These tools are vital for the completion of successful data science tasks.

In this article, you'll be introduced to:

* Relational databases
* Relational Database Management Systems
* Structured Query Language (SQL)
* The link between relational databases and set theory
* SQL data types
* Primary keys
* Not Only SQL (NoSQL): how it differs from SQL, its strengths, and weaknesses

2

**Introduction to SQL and Databases**

Here’s a quick quiz to help you process the information you just read. Best of luck! If you aren’t satisfied with your score, don’t worry! You can take the test as many times as you’d like throughout the course. If you want to improve your score, consider rereading the article.

Save

20 Minutes

4 Points

3

**Introduction to Entity-Relationship Diagrams (3 pages) Optional**

Open in new tab

Save



2 - 4 Minutes

This brief article introduces entity-relationship diagrams, immensely helpful tools that come in especially handy when working with databases. While this article is optional, we highly encourage you to at least skim through it now and use it as a reference when working with databases in the future.

4

**Database Trends**

Save



15 - 30 Minutes

6 Points

Let’s get meta for a minute. You've learned how important data and databases are, so what does the data about databases tell you? You can find out which technologies are most popular and who is using them. This article will tell you all you need to know about database trends across organizations. You'll learn:

* Which databases are the most popular
* What proportion of organizations uses more than one database
* The most popular combinations of databases
* The most time-consuming database tasks
* The critical metrics for capturing database performance

5

**Introduction to Relational Databases**

Save



5 - 10 Minutes

2 Points

It’s essential to learn how relational databases are rooted in relational algebra, and by extension, set theory. Check out this succinct video explanation from Prescott Computer Guy.

6

**Set Theory: The Method to Database Madness**

Save



15 - 30 Minutes

6 Points

Now that you understand relational algebra, sink your teeth into this article on set theory and how it lays the foundation that relational databases are built upon. Understanding these fundamentals will give you a better grasp of SQL joins, which can be a bit tricky. Not only are joins one of the most useful features of SQL, but they’re also one of the most tested skills in data science interviews.

Data Science Career Track  
Introduction to SQL and Databases  
Databases, DBMS, and SQL  
The best solution for storing large amounts of complex data is through the use of a database. In  
this article, we’re going to introduce you to key concepts of this technology, a foundation of our  
high tech world.  
On a basic level, a database is an organized collection of structured information collection of  
virtual tables that store information. This is often implemented as a collection of virtual tables  
that store information, combined with an account of how those tables relate to one another.  
Generally, when we use the word "database" we're referring to a type of software called a  
Database Management System or DBMS. Database Management Systems run on a dedicated  
machine - a server - or run on your own machine - a local instance.  
There are many different DBMS: MySQL, Microsoft SQL Server, and PostgreSQL to name a few.  
There’s no ‘best’ DBMS. Each has its own merits and downsides. Check out this optional link for  
a comparison of popular DBMS.  
Relational Databases, popularized by Oracle Corporation, are the standard database technology  
of the working world - 90% of Fortune 500 companies use Relational Databases in their daily  
operations. Relational Databases store data in a series of tables with links connecting data from  
one table with data in another. Because of their prevalence and ease of use, we’ll be using this  
technology to teach you about SQL and Data Science. The underlying theory behind Relational

Databases is rooted in relational algebra and set theory, we’ll explore this in a video later in the  
unit.  
SQL is a database-querying language: a language we can use both to manipulate the data within  
pre-existing data structures and to create data structures themselves. It has its own easy to use  
syntax. SQL is especially forgiving in its rules, making it a sweetly satisfying and immediately  
applicable language for the budding data scientist’s arsenal. The exact SQL syntax varies  
depending on the DBMS; what you write to query a MySQL database will be slightly different  
from PostgreSQL. However, the syntax is similar enough that once you learn one, you should be  
able to read the others, no matter the DBMS.  
Entities, Fields, and Records  
Every relational database comprises of one or more tables that contain data about a specific  
entity.  
An entity is a type of thing, not a thing itself. An entity would be the general class of things:  
Company, Tree, or State. An entity would not be Oracle, Redwood, or Virginia.  
For this example: suppose we are running a company that makes model vehicles like miniature  
Ferraris and Harley Davidsons. We want to store information about the different models we have  
in stock, so we build a general ‘Model’ entity table. That’s where we’ll store all the records of  
different models, and all the important attributes these models have. The entity is generally  
model vehicle, the record is for a specific model, and the fields are the qualities these specific  
models have.  
This is the products table, and it describes an entity: the products of our database. Each row is a  
record or specific example of that entity, and each column is a field or attribute. If there are no  
null values, each record has valid data within each cell for each column.  
productID productName productLine quantityInStock price  
S10\_1678 1969 Harley  
Davidson  
Motorcycles 7933 48.81

Ultimate  
Chopper  
S10\_1949 1952 Alpine  
Renault 1300  
Classic Cars 7305 98.58  
S10\_2016 1996 Moto  
Guzzi 1100i  
Motorcycles 6625 68.99  
For each model vehicle we sell, we’ll create new records, rows to store information fields about  
our model vehicles: what the name is, which type it is, how many we have in stock, etc. The table  
below contains brief definitions of each of the above terms.  
When we create our product table, we’ll list fields. Fields are properties we want to know about  
each specific product - the name, the category (for example, model motorcycle or boat), the  
quantity in stock, the price, etc. In the table, they will be the columns. Fields are also called  
attributes because each table column represents a feature of the entity.  
Term Definition  
Entity A type of thing  
Record An example of an entity  
Field An attribute of a record  
Data Types  
Every column has an associated data type. This feature of RDBMS improves data quality by  
preventing the accidental introduction of, for example, text into a numerical field. The advantage  
of this feature is that it improves data quality by ensuring all data is the same type. A weakness  
of giving a column an associated data type is that it is restrictive. For example, look at our  
productID from the model vehicle sample table. Let’s say a new manager decides to remove the  
prefix ‘S10\_’ from the productID column, in order to make productID an int instead of a varchar,  
there is no way to change the datatype of the column. They would have to create an entirely new  
column of a new data type.

SQL has standard data types, but every DBMS has its own implementation of these data types.  
Even if the names are the same, the data types can vary on details like storage size.  
Here are the common data types in SQL:  
Data Type Explanation Example  
char(n) A fixed-length string of  
alphanumeric text, that must be  
of length n  
area\_code CHAR(5)  
07837  
01653  
varchar(n) A variable-length string of  
alphanumeric text that can be  
any length up to a maximum  
length n  
email\_address VARCHAR(100)  
myemail@hotmail.com  
int An integer (whole number) view\_count INT  
1  
195  
float A number with floating point  
precision  
weight FLOAT  
0.18102010  
13.1213121  
boolean A boolean true/false value married BOOLEAN  
TRUE  
FALSE  
date A date date\_of\_birth DATE  
1990-03-04  
time A time time\_of\_birth TIME  
01:00:01  
timestamp A date and time date\_of\_birth TIMESTAMP  
1990-03-04  
01:00:01  
Primary keys  
Records require an identifier field - containing a unique value so we can differentiate records. It's  
possible that two rows end up with the same information - like two ‘John Smiths’ in a staff  
directory. We need a way of differentiating these records to avoid data corruption and  
confusion.  
Let’s look at our model vehicle dealership database. Say we have two products with the same  
name: a ‘1968 Dodge Charger’ with a 1:18 scale and another ‘1968 Dodge Charger’ with a 1:1  
scale. A unique identity field will work to separate these records.

When you create an entity table, you will define a primary key that should always be unique,  
non-null, and immutable. This means the primary key will never have a duplicate, it will contain  
real data, and no one will be able to change it. Let’s say our “1969 Harley Davidson Ultimate  
Chopper” comes in two scales, 1:1 and 1:18. In an unlikely scenario, both these models have the  
same price and stock, and we don’t have a column that keeps track of scale. A primary key will  
allow us to differentiate between these products when the fields to overlap. Let’s check out an  
example:  
productID productName productLine quantityInStock price  
S10\_1678 1969 Harley  
Davidson  
Ultimate  
Chopper  
Motorcycles 2000 91.02  
S10\_4698 1969 Harley  
Davidson  
Ultimate  
Chopper  
Motorcycles 2000 91.02  
When creating a table, we need to explicitly define the table, its columns, data types, keys, and  
any additional constraints before we can start work with it.  
We could’ve created the above table with a piece of SQL like this:  
CREATE TABLE products (  
productID int NOT NULL,  
productName varchar(70) NOT NULL,  
productLine varchar(50) NOT NULL,  
quantityInStock int NOT NULL,  
price decimal(10,2) NOT NULL,  
PRIMARY KEY (productID)  
);  
This code creates a table, gives each column a name, and defines the data type of each column.  
'NOT NULL' is a constraint that says each field must contain non-empty data. The primary key is  
marked as the productID field.  
This SQL is used as a DDL (Data-Definition Language), as we’re using SQL to make a data  
structure. SQL can also be used as a DML (Data Manipulation Language) when we update,  
retrieve, or delete information in our data structures with SQL queries.  
This query is the ‘Hello World!’ of SQL:

SELECT \*  
FROM products;  
It returns all columns and rows from the products table. You’ll learn more about how to  
construct SQL queries in the next subunit.  
A Brief Look At NoSQL  
NoSQL, which stands for ‘Not Only SQL,’ refers to databases that are not structured relationally.  
Let’s take a look at an illuminating example.  
Suppose we’re a hospital tracking medical information with a large relational database. We have  
a patients table with the fields: ID, Test1, Test2, Test3, DOB, and BloodType. Not all patients take  
all three tests: in fact, most patients have only take one of the three tests. Also, not all of the  
patients have their blood type recorded. Because of this, our patients table is sparse: it contains  
many null values. Suppose it looks like this:  
ID TEST1 TEST2 TEST3 DOB BTYPE  
1-06  
2732 80 95 1965 A-  
15 07  
2946 92 1965  
16 07  
3650 86 1965 O  
It’s a shame, but many relational databases contain tables just like this! Notice the poor data  
integrity: the values for the DOB column spill out over two rows, with the year on one row and  
the day and month below it! Patient 2732’s DOB, for example, is actually 1-06-1965.  
In many cases, we don't want a big, unwieldy relational database full of super sparse tables like  
this one. We’d prefer to arrange our data in a key-value table, as follows:  
ID KEY VALUE  
2732 TEST1 80

2732 TEST3 95  
2732 DOB 1-06-1965  
2732 BTYPE A-  
2946 TEST2 92  
2946 DOB 15-07-1965  
3650 TEST1 86  
3650 DOB 16-07-1965  
3650 BTYPE O  
Notice how we now have a table with no null values at all! We’ve replaced our confusing, sparse  
table with a long set of identifiable key-value pairs. Analytically, a table like this is better for  
storing extremely large datasets about our patients than a sparse relational table. Though we  
don’t know patient 2946’s blood type, we don’t need to put a NULL value into our table anymore.  
NoSQL works by replacing complex relational databases, in which the tables relate to one  
another in ways we define, by three-column tables of the sort exemplified above. These tables  
use the key-value (or associative array) data model. Such tables are great for storing enormous  
quantities of sparse data: where the number of attributes is very large, but the number of  
instances of those attributes is relatively low. With NoSQL, there’s no schema at all, which  
means that our data is susceptible to bad data entry.  
NoSQL has its strengths and weaknesses:  
Advantages of NoSQL Disadvantages of NoSQL  
Can make sense of big data, stored in large,  
sparse tables in relational databases.  
Simplistic data model. The relationships  
between the entities described by the data  
are obscured and may need painstaking user  
reconstruction.  
High performance and scalability. Data can  
be retrieved, deleted, and updated quickly,  
even with enormous datasets.  
Transaction management can be difficult  
depending on the DBMS  
Fast updates: to add a new entity attribute,  
just a new row.  
Mistakes are easily made and hard to track.  
Easily supports distributed data architectures, The application program manages the

where the data is stored in different physical  
locations  
validation of data and integrity constraints. If  
it crashes, the data can be corrupted.  
Wrapping Up  
We hope you’ve enjoyed this introductory article on databases, DBMS, SQL, and NoSQL. There’s  
quite a lot to learn about this interesting material! Don’t worry if you didn’t grasp absolutely  
everything. In the rest of the sub-unit, you’ll go through resources to cement your knowledge.  
Enjoy, and good luck!

3. What does SQL do when it's used as a Data Manipulation Language? (DML)

**Update, retrieve or delete information within a database.**

Create data structures to store data.

Write programs that connect to databases.

Write front-end applications.

. Which of the following is not a development environment for a DBMS?

MySQLWorkbench

PGAdmin

PHPMyAdmin

**PostgreSQL**

What is a developmental environment?

A development environment is **the collection of processes and tools that are used to develop the source code for a program or software product**. This involves the entire environment that supports the process end to end, including development, staging and production servers.

In a well-constructed table within a well-constructed relational database, what does that table represent?

An attribute of a type of thing we care about

An example of a type of thing we care about

**A type of thing we care about**

A selection of types of things we care about

Which of the following SQL Data Types is correctly described by the following: "A variable length string of alphanumeric text that can be any length up to a maximum length n"

**varchar(n**)

char(n)

float

timestamp

What are the three qualities that the primary key of any table ought to have?

Unique, non-null, numeric

Numeric, immutable, unique

**Non-null, unique, immutable**

Non-null, numeric, immutable

8 / 10

Suppose we're storing a table, called 'Customers', that records details about our customers. Which would be a good choice for a primary key?

lastName

telNo

**customerID**

homeAddress

Suppose that, in our library's database, the Books and Members tables relate as follows. A given book may be borrowed by no member, but if it is borrowed, it will be borrowed by just one member. How is this relation best characterised?

Optional 1-Many

Mandatory 1-1

Mandatory 1-Many

Optional 1-1

1. Which of the following is not a Database Management System?

MySQL

PostgreSQL

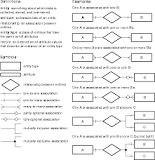
Microsoft SQL Server

NoSQL

What could a mandatory one-to-one relationship indicate?

A mandatory relationship indicates that **for every occurrence of entity A there must exist an entity B, and vice versa**. When specifying a relationship as being mandatory one-to-one, you are imposing requirements known as integrity constraints Opens a new window.Feb 5, 2007

What is one-to-many optional relationship?



A Customer has a one-to-many relationship with a Purchase Order because **a customer can place many orders, but a given purchase order can be placed by only one customer**. The relationship is optional because zero customers might place a given order (it might be placed by someone not previously defined as a customer).

2. One of the functions of Database Management Systems is to provide Transaction Management. What is Transaction Management?

Ensuring the transactions stored in the database have the right prices (e.g: $50)

**Ensuring that the transactions on the database leave it in a consistent state.**

Ensuring that the database's tables relate sensibly to one another.

Ensuring that you cannot add a row to any table without the right permissions.

One of the advantages of NoSQL is that it can make sense of a certain type of data that Relational database tables struggle to handle. What is this type of data?

Data stored in tables with lots of columns

**Sparse data: data stored in tables with lots of null values**

Data stored in tables that relate complexly to one another

Tables storing lots of numeric data

Data Science Career Track  
Introduction to Entity-Relationship Diagrams  
Entity-Relationship Diagrams  
One of the wonderful things about relational databases is that we can precisely specify the  
relationships between the constituent tables/entities to make our system crystal clear to us and  
its users. An entity-relationship diagram does this perfectly.  
Suppose we have a database containing tables for Mangers, Branches, Employees, and  
Locations, and these tables relate to one another in very specific ways:  
- A given employee may not work at any branch, but if they do, they’ll work at just one.  
- A given location may not have any branch at it, but if it does, it’ll have just one.  
- A given manager must manage at least one branch, and they may manage multiple  
branches;  
- A given branch may not be managed by any manager, but if it is, then it will be managed  
by just one.  
- A given branch may have no employees, but if it does, it may have multiple employees.  
- A given branch must have just one location.  
In fact, relational databases typically have schemas that constrain updates or changes to the  
database to ensure that such constraints are continuously met by future users.  
We can actually represent the relations between these tables with a diagram called an  
Entity-Relationship diagram. These are so neat. Check out our diagram:

To help us interpret this, we can use the following key:  
See how the diagram captures the descriptions of the specific relationships between the tables?  
For example, recall that a given employee may not work at any branch, but if they do, they’ll work  
at just one. This accounts for the Optional Multiplicity of One relationship on the far right of our  
diagram. And since a given branch may have no employees, but if it does, it may have multiple  
employees, we have an Optional Multiplicity of Many relationships just to the left of the previous  
one; second from the furthest right.

You won’t have to master Entity-Relationship diagrams in this course. They’re just a cool way to  
illustrate how relational databases can track complex relationships between the kinds of things  
we care about, where each kind of thing can be captured in a table.

# 2019 Database Trends – SQL vs. NoSQL, Top Databases, Single vs. Multiple Database Use

Posted: March 4, 2019

[Tweet](https://twitter.com/intent/tweet?text=2019+Database+Trends+-+SQL+vs.+NoSQL%2C+Top+Databases%2C+Single+vs.+Multiple+Database+Use&url=https%3A%2F%2Fscalegrid.io%2Fblog%2F2019-database-trends-sql-vs-nosql-top-databases-single-vs-multiple-database-use%2F)

[Share](https://www.facebook.com/share.php?u=https%3A%2F%2Fscalegrid.io%2Fblog%2F2019-database-trends-sql-vs-nosql-top-databases-single-vs-multiple-database-use%2F)

[Share](https://www.linkedin.com/cws/share?url=https%3A%2F%2Fscalegrid.io%2Fblog%2F2019-database-trends-sql-vs-nosql-top-databases-single-vs-multiple-database-use%2F)

Pin1

1 Shares

Wondering which databases are trending in 2019? We asked hundreds of developers, engineers, software architects, dev teams, and IT leaders at [DeveloperWeek](https://www.developerweek.com/) to discover the current NoSQL vs. SQL usage, most popular databases, important metrics to track, and their most time-consuming database management tasks. Get the latest insights on our supported databases [MySQL](https://scalegrid.io/mysql/), [MongoDB®](https://scalegrid.io/mongodb/), [PostgreSQL](https://scalegrid.io/postgresql/), ™" href="<https://scalegrid.io/redis/>" target="\_blank">Redis™\*, and many others to see which database management systems are most favored this year.

## SQL vs. NoSQL

As any database administrator knows, the first question you have to ask yourself is whether to use a SQL or NoSQL database for your application. What’s the difference between the two?

### SQL Databases

Also known as relational databases, define and manipulate data based on structured query language (SQL). These are most popularly used and useful for handling structured data that organizes elements of data and standardizes how they relate to one another and to different properties.

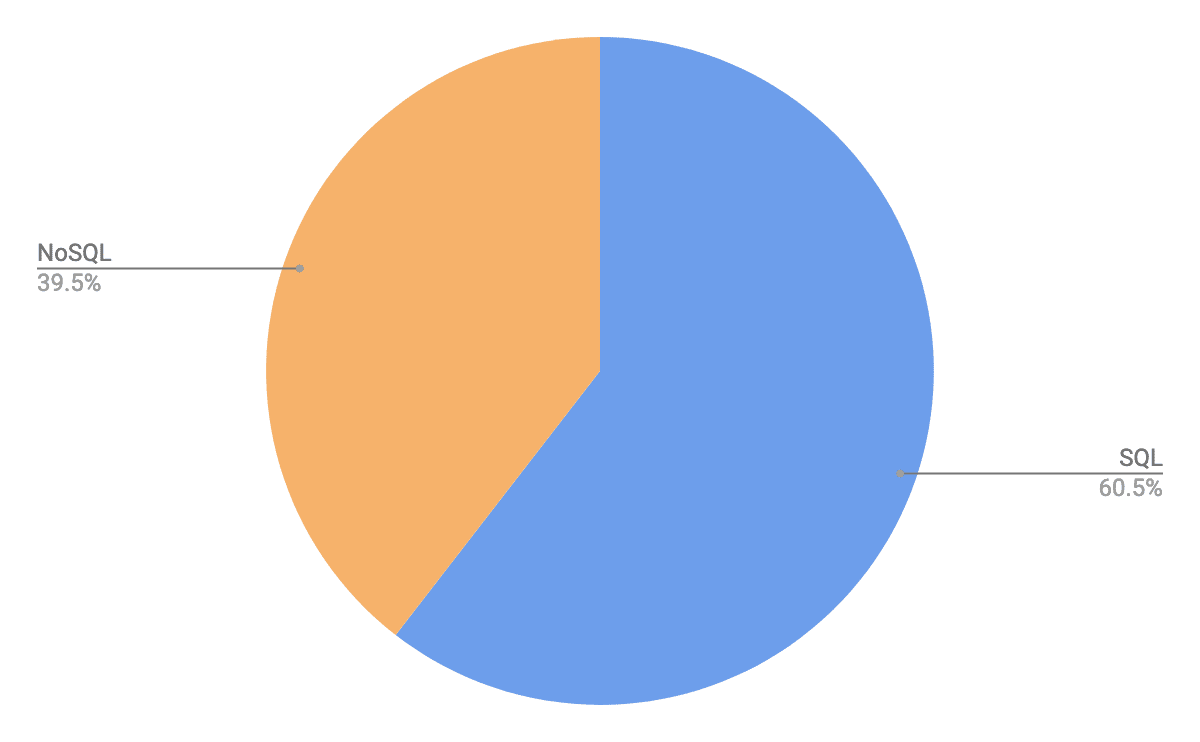
### NoSQL Databases

Also known as non-relational databases, allow you to store and retrieve unstructured data using a dynamic schema. NoSQL is popularly used for its flexible ability to create a unique structure, and can be document, graph, column, or even KeyValue organized as a data structure.

SQL has had a large lead over the non-relational alternatives for decades, but NoSQL is quickly closing the gap with popular databases such as MongoDB, Redis, and Cassandra. Though many organizations are choosing to migrate from legacy databases, such as Oracle, not all are moving to NoSQL way. Based on our findings, SQL still holds 60% with rising demand for systems such as PostgreSQL:

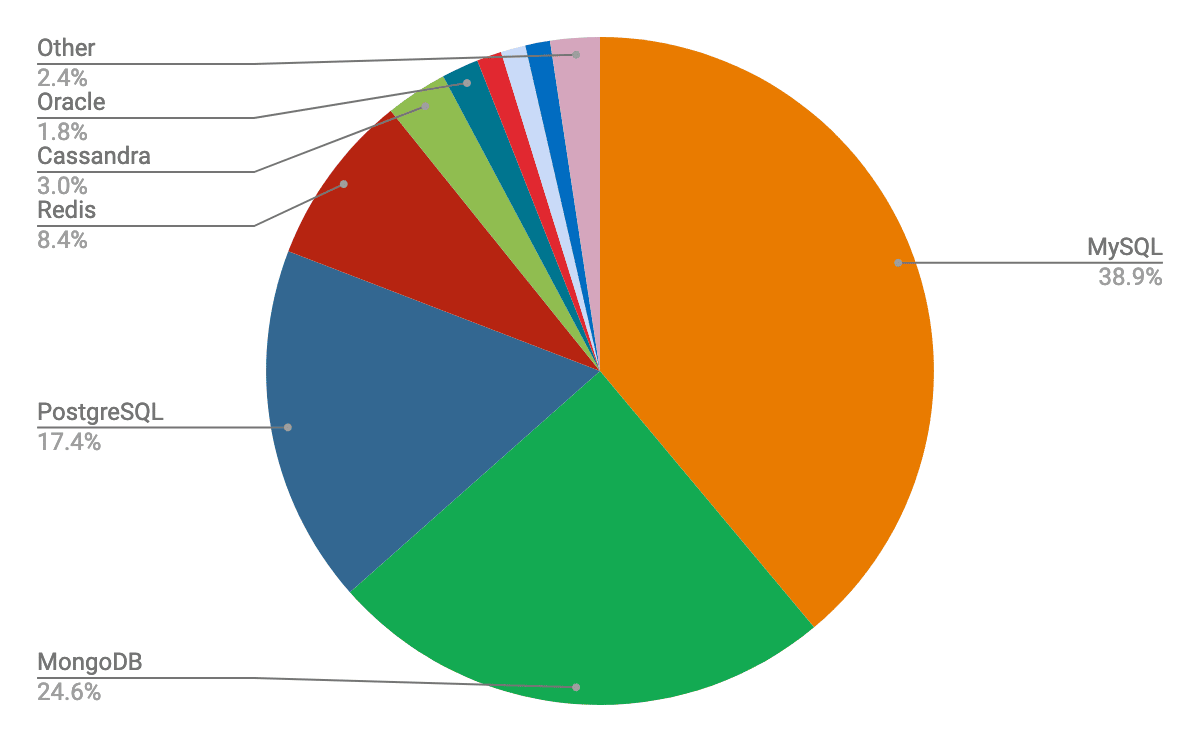
#### SQL Database Use: 60.48%

#### NoSQL Database Use: 39.52%

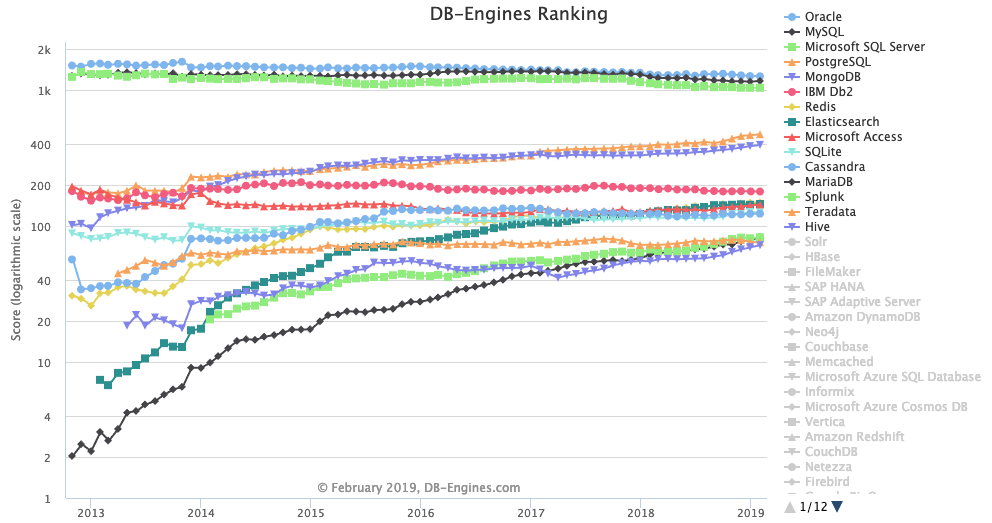
[](https://scalegrid.io/wp-content/uploads/2019/02/SQL-vs-NoSQL-Cloud-Database-Use-Stats.png)

## Most Popular Databases

So, which databases are most popular in 2019? Knowing that SQL was used by over 3/5 of respondents, you might assume Oracle stole the show. Guess again. MySQL dominated this report with 38.9% use, followed by MongoDB at 24.6%, PostgreSQL at 17.4%, Redis at 8.4%, and Cassandra at 3.0%. Oracle trailed behind at just 1.8% from these database reporters, and CouchDB, Berkeley DB, Microsoft SQL Server, Redshift, Firebase, Elasticsearch, and InfluxDB users combined our Other category at 2.4%.

[](https://scalegrid.io/wp-content/uploads/2019/02/Most-Popular-Databases-Used-MySQL-MongoDB-PostgreSQL-Redis-Cassandra-Oracle.png)

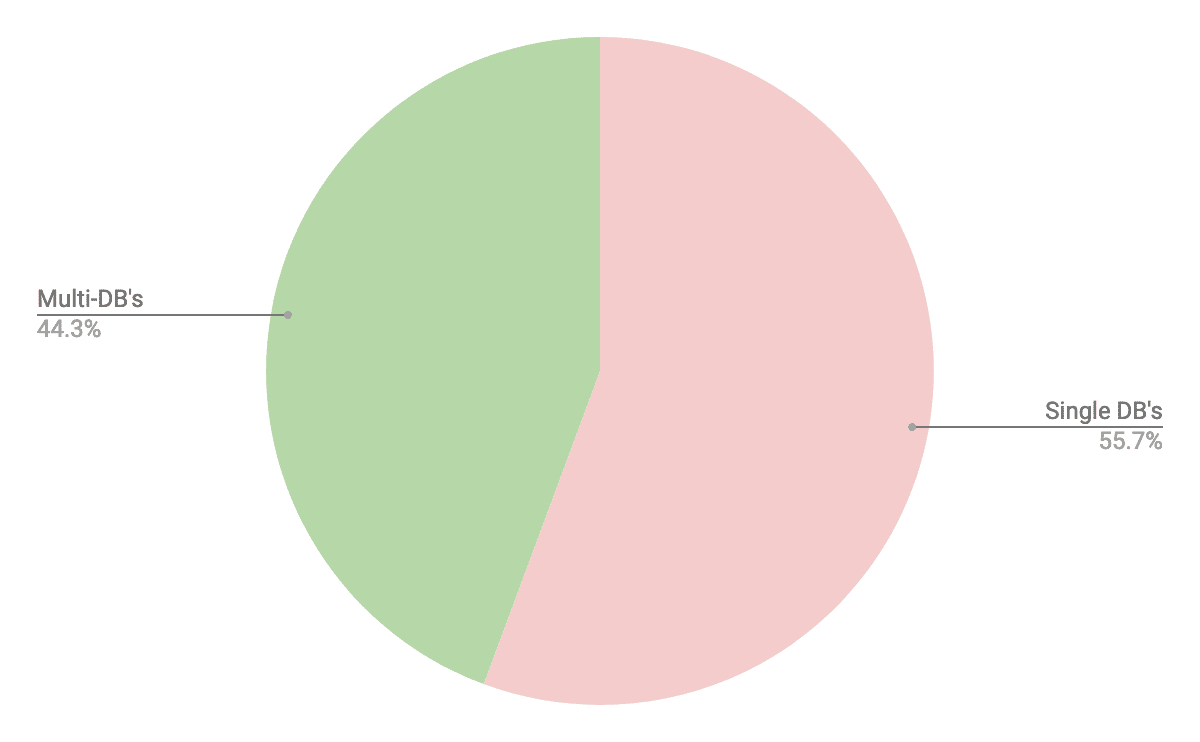
While these numbers might shock, there’s no mistaking the rise in popularity of MySQL, MongoDB, and PostgreSQL. So how does this survey compare to best-known source for database management system trends? [DB-Engines Ranking – Trend Popularity](https://db-engines.com/en/ranking_trend) report places these leaders in the top 5, but Oracle keeps hold at number one and Microsoft SQL Server at number 3.

[](https://scalegrid.io/wp-content/uploads/2019/02/DB-Engines-Ranking-Trend-Popularity-All-Databases-February-2019.png)

While we expected to see a much higher presence of Oracle database users, their representation was low at the world’s largest developer expo.

## Single Database vs. Multi-Database Use

Multi-database type use has exploded over the past decade, compared to the traditional strategy of throwing all of your eggs in one basket. How much so? Almost half of the organizations we spoke with actually use more than one type of database to power their applications than a single database! 44.3% reported using multiple databases, while 55.7% are operating with one:

[](https://scalegrid.io/wp-content/uploads/2019/02/Single-Database-vs-Multiple-Database-Use-Trends.png)

[2019 Database Trends – SQL vs. NoSQL, Top Databases, Single vs. Multiple Database UseClick To Tweet](https://twitter.com/share?text=2019+Database+Trends+%E2%80%93+SQL+vs.+NoSQL%2C+Top+Databases%2C+Single+vs.+Multiple+Database+Use+%7C+%40scalegridio&url=https://scalegrid.io/blog/2019-database-trends-sql-vs-nosql-top-databases-single-vs-multiple-database-use/)

### SQL & NoSQL Multiple Database Combinations

So, knowing that almost half of our respondents are combining multiple databases to support their products, what types of database management systems are they using together? This one is less of a shocker, 75.6% of multiple database type use is made up of a combination of both SQL and NoSQL databases. This solidifies that case that, for many organizations, one size does not fit all. While you might have a preference over SQL vs. NoSQL, there’s no denying the fact that they both offer clear advantages of the other. Instead of limiting your organization to one database type, evolve (or develop) your data strategy for compatibility so that these powerful database management systems can complement each other and fill the gaps in your data needs!



[Write](https://medium.com/m/signin?operation=register&redirect=https%3A%2F%2Fmedium.com%2Fnew-story&source=---two_column_layout_nav-----------------------new_post_sidenav-----------)

[Sign In](https://medium.com/m/signin?operation=login&redirect=https%3A%2F%2Fmedium.com%2Fbasecs%2Fset-theory-the-method-to-database-madness-5ec4b4f05d79&source=post_page---two_column_layout_nav-----------------------global_nav-----------)

[[basecs](https://medium.com/basecs?source=post_page-----5ec4b4f05d79--------------------------------)](https://medium.com/basecs?source=post_page-----5ec4b4f05d79--------------------------------)

Published in

[basecs](https://medium.com/basecs?source=post_page-----5ec4b4f05d79--------------------------------)

[[](https://medium.com/@vaidehijoshi?source=post_page-----5ec4b4f05d79--------------------------------)](https://medium.com/@vaidehijoshi?source=post_page-----5ec4b4f05d79--------------------------------)

[Vaidehi Joshi](https://medium.com/@vaidehijoshi?source=post_page-----5ec4b4f05d79--------------------------------)

May 1, 2017

·

11 min read

·

**Set Theory: the Method To Database Madness**

Now that we’re finally a third of the way through this series, things are finally starting to come together. Sure, we know about quite a few different data structures, how they work, which ones are fast, and how certain ones are more helpful for solving specific problems than others.

But there’s almost no point to knowing any of that if we don’t have a sense of how they’re actually used in real life. It’s like learning geometry: it probably all seemed pointless until one day, you woke up and realized that you were actually going to have to calculate the square footage of a room because you wanted to re-carpet the floor! (Okay, I’ve never actually had to do that, but I can imagine that geometry as a concept would generally be pretty helpful if I ever have to.)

A lot of things are going to come together today, because we’re going to learn about a data structure that is almost dogmatic in its theory, but incredibly ubiquitous in practice. In fact, you’ve probably already worked with this structure in some shape or form, and you were likely introduced to it in your middle school math class.

So, what data structure am I talking about? Why, I’m referring to the all-powerful **set**, of course!

**No fear set theory**

Before we get into the actual implementation of sets, we first need to understand where on earth they come from. This means it’s time for us to dive into theory — set theory! But, fear not: there’s a good chance you’ve used set theory in some capacity or another. In fact, you probably know set theory by another name: **a Venn diagram**. The Venn diagram was actually only incorporated into the “set theory curriculum” in the 1960’s because it was such an effective way of illustrating simple relationships between sets.

Okay, now that we’re sure that set theory won’t be that scary, we’d better figure out what it is, exactly! A *set* is actually a mathematical concept, and the way that we relate sets to one another is referred to as **set theory**.

A **set** is nothing more than an unordered collection of elements with absolutely no duplicates.

There are three important pieces to that definition: ***unordered***, ***elements***, and ***no duplicates***. Actually, those three words encompass pretty much the entire definition of a set; if we can remember that, we’ll basically know everything about how sets work.

But we’ll come back to why that’s important in a bit. First, let’s look at some sets in action. We know that sets are well-represented by Venn diagrams, so for our example, we’ll look at two sets: a set of some of the books I’ve read, and a set of some of the books I own.

Diagram, text

Description automatically generated

Books I’ve read vs. books I own

Since we’re familiar with the concept of [Venn diagrams](https://en.wikipedia.org/wiki/Venn_diagram), we know that the center section in green (where the two sets intersect) represents books that I have both read *and* books that I own. We also know that the two sets drawn above exist within the larger group of all the books in the world!

The Venn diagram is a good introduction to set theory, because it makes the next part a lot easier to explain. Imagine that we wanted to represent these two sets of data in some sort of structure. Well, we already know that we need to divide them into two groups: *books I’ve read*, and *books I own*. To make it a little easier, we’ll call *books I’ve read* as **set X**, and *books I own* as **set Y**. Reconfigured into data structures, those two sets in the Venn diagram could also be rewritten to look like this:

Text

Description automatically generated

Sets are a collection of distinct, unique objects that never contain duplicated values.

We’ll notice that both set X and set Y look a little bit like objects or hashes: the elements don’t have indexes or any sort of order. They also don’t have any repeated values, which is part of what makes them a set. Remember that a set is a collection of unique, unordered objects, which means that we’ll never find duplicated values within a set.

**Pain-free (set) operations**

So, what can we do with these sets now that we have them written in data structure format? Well, we can perform some operations on them! The two most important operations that are performed on sets are **intersections** and **unions**.

Text, letter

Description automatically generated

Basic operations on sets

The intersection of two sets is often denoted in shorthand like this: ***X ∩ Y***. The intersection represents where two sets — you guessed it — *intersect*! In other words, it yields all of the elements that exist within *both* of the sets. In our example, the intersection of set X and set Y are all of the elements that exist in both of them. A good way keyword to remember how intersections work is the word **and**: the elements that exists in both X and Y. In this case, that would mean “Code Complete” and “Milk & Honey”. Even though they exist in both sets, since sets can only ever contain *unique* values, we don’t repeat them; each of these book titles exists only once in the set.

The union of two sets is often denoted in shorthand like this: ***X ∪ Y***. The union represents the entirety of two sets, or the two sets when they’ve been *united* together. In other words, it yields all of the elements that exist in *either* of the two sets. A good way keyword to remember how intersections work is the word **or**: the elements that exists in both X or Y. In this case, that would mean all of the eight book titles! The important thing to remember is that even though “Code Complete” and “Milk & Honey” exist in both sets, they can only ever appear once in the union of set X and set Y, since sets can only have distinct values and can never contain duplicates.

If we were to apply intersections and unions to our Venn diagram from earlier, our diagrams would look like this:

Diagram

Description automatically generated

Intersections vs. unions

Okay, time to complicate things a little bit more! Intersections and unions are great, but they’re only scratching the surface of set theory. For our purposes, we’ll need to be familiar with some other operations as well. As it turns out, there are two operations that turn up quite a bit in computer science: **set differences** and **relative complements**. We’ll learn how they both play a role in the next section, but first, let’s figure out how they work!

Diagram, text

Description automatically generated

Set differences and relative complements

Set differences are how we can figure out the difference between two sets. In other words, we can determine what a set looks like without any of the elements that are contained in the *other* set. Another way to write this is **X — Y**. In the example shown here, the difference between set X and set Y results in all of the elements that exist in set X but do not exist in set Y, or the letters *C, Z,* and *W*.

Relative complements are basically the opposite of set differences. For example, the relative complement of Y as compared to X will return all of the elements in set Y that don’t appear in set X. We can denote the relative complement by using the short hand **Y ∖ X**, which is actually results in the exact same returned set as **Y— X**. In our example, the set Y is smaller than the set X. In our example, the only thing that exists in Y that doesn’t exist in X is the number *2*.

Effectively, we’re simply subtracting set X *from* set Y, and answering the question: *what exists in Y that doesn’t exist in X*?

You might have noticed that in some of the examples, we’re dealing with strings, and in others, the elements are letters, and sometimes even numbers. This brings up an important point: sets can contain literally any kind of element or object! You can think of them as hashes in that way: they can hold any item, as long as it only occurs once within the set.

Alright, let’s look at one last operation — the most complicated one of them all. But we can handle it!

Sometimes, when we have two sets, we might want to find the opposite of the intersection of the two sets. In other words, I might want to find all of the books that I own, and all of the books that I have read, but *none* that intersect between the two. What would we call that subset? And how would we find it?

Well, the proper term for what we’re looking for in this case is something called the **symmetric difference** of our two sets, which is also sometimes referred to as the **disjunctive union**. The symmetric difference yields all of the elements that exist within either of the two sets, but *do not* exist at the intersection (X ∩ Y) of them.

Let’s look at an example, which should help clarify what I mean:

Text, letter

Description automatically generated

Symmetric differences of sets

In the example above, the symmetric difference is basically the same as find the relative complement of set X and set Y. If we super mathematical about it, finding the symmetric difference is the same as finding the union of relative complements of set X and of set Y. We could write that out as: **X △ Y= (X ∖ Y) ∪ (Y ∖ X)**.

But don’t let that confuse you!

All we really need to do in order to find the symmetric difference of two sets is ask ourselves: *what elements exist in set X that don’t exist in set Y, and which elements exist in Y that don’t exist in X?* In other words: which elements are unique to each set, and don’t occur within both of them?

In the example above, the numbers *1, 2,* and *3* occur within both sets. However, the letters *A, B, C* and *X, Y, Z* occur only within set X or set Y, and are therefore the symmetric difference of the two sets.

Okay, that was a *lot* of theoretical stuff. Let’s see this theory play out in practice, shall we?

**Sets all around us**

Hopefully by now, you’re wondering what the point of learning sets is. I don’t blame you: it’s a good question! And it’s finally time to answer it.

Guess what? Sets are *everywhere*. They’re actual data structures that we can use whenever we want in [Java](https://docs.oracle.com/javase/7/docs/api/java/util/Set.html), [Python](https://docs.python.org/2/library/sets.html), [Ruby](http://ruby-doc.org/stdlib-2.4.1/libdoc/set/rdoc/Set.html), and even [JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set)! You might even be able to guess some of the methods that each of these languages allows us to perform on sets.

Let’s take a quick look, using JavaScript as an example:

Obviously, some of the method names will change from language to language. For example, Ruby’s [implementation](http://ruby-doc.org/stdlib-2.4.1/libdoc/set/rdoc/Set.html#method-i-include-3F) of has is called include?, but the idea is fairly similar from one language to another. Python’s version of sets actually allows you to [call methods](https://docs.python.org/2/library/sets.html) like intersection, union, and symmetric\_difference!

But, what good are sets, anyways? I mean, we can use them in all of these languages, but when are they useful?

Text, letter

Description automatically generated

Time-complexity of set operations

Well, for one thing, they can be pretty time-efficient.

Remember all of those complicated operations like intersection, union, and difference? Well, guess what? The amount of time it takes for us to run any of those complex operations depends purely on the length of the two sets. This is because in order for us to find the intersection, union, or difference/complement of these two sets, we have to effectively traverse through the entire length of the two sets being compared. Usually, even giant sets still won’t take that much time to traverse.

But what about basic operations? What about adding an element to one of those sets, removing it, or even finding an element within it? Well, all of those operations take **constant time**, or *O(1)*. This can be incredibly powerful, and often means that a set might be a better structure than a dictionary or a hash!

But, wait a second: how is it possible that all of these set operations are so fast?! How does that even happen? Well, as it turns out, many sets are actually implemented by [hash tables](https://medium.com/basecs/taking-hash-tables-off-the-shelf-139cbf4752f0) under the hood! (See, I promised you it was all going to come together!) We already know about hash tables, but why do they make for good skeletons for set implementations?

A picture containing text, whiteboard

Description automatically generated

Hash tables are often used to implement sets!

Well, there are a few good reasons: **first**, given what we know about hash tables, they’ll always have unique keys for each element. This is great for sets, since sets can only have unique values in them. **Second**, in hash tables, order doesn’t really matter, just as how order doesn’t matter in a set. **Finally**, hash tables provide a *O(1)* constant access time, which is what ideal for basic operations performed on a set.

Alright, so hash tables make for good sets. And sets are data structures that most languages give us for free. But when I started this post, I told you that sets were *everywhere*, right? I feel like I should probably let you in on a little secret that’s going to (hopefully) blow your mind:

Relational databases are based almost entirely upon set theory.

In fact, if you’ve ever worked with or queried a database, or had to write SQL, you’re probably familiar with the idea of finding records at the intersection of a table.

Diagram

Description automatically generated

Relational databases are based entirely upon set theory

This is nothing more than an abstraction of the Venn diagram version of sets that we started off this post with! In fact, even the most complicated SQL statements are nothing more than operations on sets.

Diagram

Description automatically generated

<http://stackoverflow.com/questions/406294/left-join-vs-left-outer-join-in-sql-server>

A SQL INNER JOIN is just the *intersection* of two sets.

Finding the LEFT JOIN of two tables is nothing more than finding the *set difference* or the *relative complement* of the two tables.

A SQL query that calls for a FULL OUTER JOIN is merely returning the *union* of two sets.

And that super complicated FULL OUTER JOIN where one keys on both tables is NULL? (Also known as the bane of my existence when it comes to writing SQL statements?) That’s just the *symmetric difference* or *disjunctive union* of two tables.

How amazing is that?! All of that seemingly boring set theory, when put into practice, makes databases seem like the *coolest things ever*. And that, my friend, is a truly a feat in and of itself!

**Resources**

Set theory is fairly widespread in various parts of computer science, from its use in relational databases, to a data structure that exists in various languages, including Python, Ruby, JavaScript, and Java. There are plenty of good resources out there to help you get a better understanding of set theory; here are a few to get you started!

1. [LEFT JOIN vs. LEFT OUTER JOIN in SQL Server](http://stackoverflow.com/questions/406294/left-join-vs-left-outer-join-in-sql-server), StackOverflow
2. [Set Theory](https://mariadb.com/kb/en/sql-99/set-theory/), MariaDB
3. [Complexity of Python Operations](https://www.ics.uci.edu/~pattis/ICS-33/lectures/complexitypython.txt), Professor Richard E. Pattis
4. [Set Theory — Design of Computer Programs](https://www.youtube.com/watch?v=x7ARbhM7kuw), Udacity
5. [Set Theory | Introduction](http://quiz.geeksforgeeks.org/set-theory/), GeeksforGeeks

scroll icon

**SQL Case Study - Country Club**

In this subunit, you’ll bring together all your new knowledge to answer business questions with high-level SQL queries. In the process, you’ll be querying an external MySQL database through the PHPMyAdmin interface, and you’ll use a local SQLite instance on your computer. You’ll use one of these two methods of doing database work every day in your career, so understanding both is vital to your success as a data scientist.

1

**Working with Relational Databases in Python**

Save



1 - 1.5 Hours

22 Points

Chapter Three of this DataCamp course will give you a primer on how to work with relational databases in Python, a practical professional skill that you'll draw on while working on the next case study. Typically, you'll work with databases in one of two ways: you'll either make a local instance of the database on your computer or you'll connect to an external database. You'll do both in the case study. Chapter Three of this course will walk you through how to do that.

**Please note that only Chapter Three of this resource is mandatory.**

2

**Introduction to PHP MyAdmin**

Open in new tab

Save



2 - 4 Minutes

1 Points

For the coming case study, you'll be using PHPMyAdmin: an open-source administration tool for MySQL and MariaDB. This much loved and commonly used interface takes a little getting used to. The aim of this video is to help you become familiar with PHPMyAdmin before you tackle the case study.

3

**Case Study - Country Club**

Save

3 - 6 Hours

82 Points

In this case study, you'll use MySQL, PHPMyAdmin, Juptyer Notebook, and SQLite to tackle a series of challenges on a database containing information about a country club. For this case study, you'll see what a database looks like on a nice interface like PHPMyAdmin, and connect to a local instance of the database using Python and SQLite. Data scientists use these methodologies to do work on a daily basis. You'll retrieve particular pieces of information and translate requests into SQL queries. Make use of the DataCamp courses from the previous subunits to succeed in this case study!

There are two difficulty tiers for this case study. The only difference between them is the amount of guidance we give in setting up the SQLite connection in Python. We always encourage students to challenge themselves, but it's also totally fine if you want more guidance while working on complex assignments like this one.

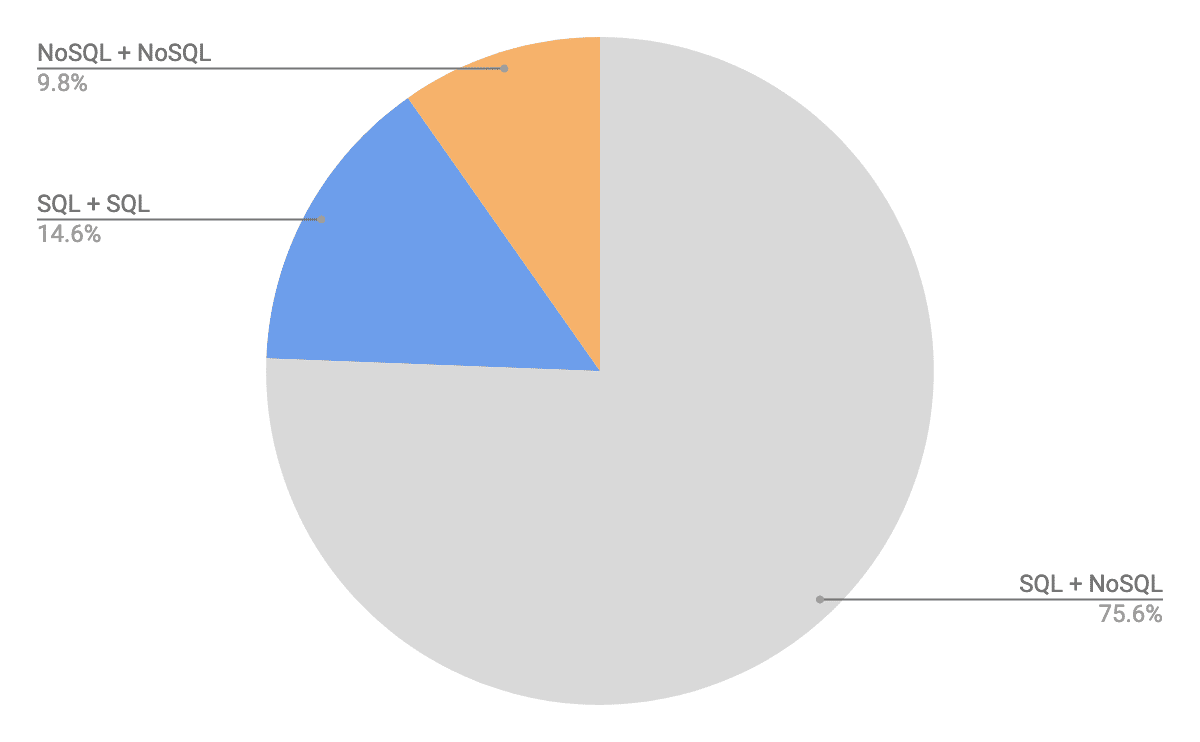
* [**Tier Two**](https://www.springboard.com/archeio/download/18a1b2578e084d51999efa6a9fd2144a/)**:** If you feel confident tackling the challenge of exploring data from PHPMyAdmin and establishing a connection to a local SQLite database, pick Tier Two.
* [**Tier One**](https://www.springboard.com/archeio/download/1bf0319530ab4b599eb7973508728921/): If you want some helpful pointers during that process, go for Tier One. The SQL queries will be the same for both tiers. In this project folder, you will also find two resources that will guide you through the setup of SQLite.

Once you’ve decided on your difficulty level, paste the relevant SQLFiles folder into your working directory on PHPMyAdmin. Open the SQLTasks file containing the instructions for your case study. Happy querying!

#### SQL + NoSQL Database Use: 75.6%

#### SQL + SQL Database Use: 14.6%

#### NoSQL + NoSQL Database Use: 9.8%

[](https://scalegrid.io/wp-content/uploads/2019/02/Multiple-Database-Use-Combinations-SQL-and-NoSQL.png)

### Most Popular Multiple Database Type Combinations

If you’re a single database type user who’s considering adding another database type to your mix, this section might be of high interest – which databases, SQL and NoSQL alike, are most popularly used together.

The clear winner with over 1/3 of multiple database type use is the combination of MySQL and MongoDB. While MongoDB is often considered an alternative to MySQL, the two databases do work well together when properly designed. The second most popular combination was MySQL and PostgreSQL together. These two SQL databases are clear competitors, but can be jointly used to store different data sets. As you can see in the above section graph, the 9.76% representation of MySQL and PostgreSQL comprises a large majority of the SQL + SQL use in multiple databases.

#### MySQL + MongoDB: 34.15%

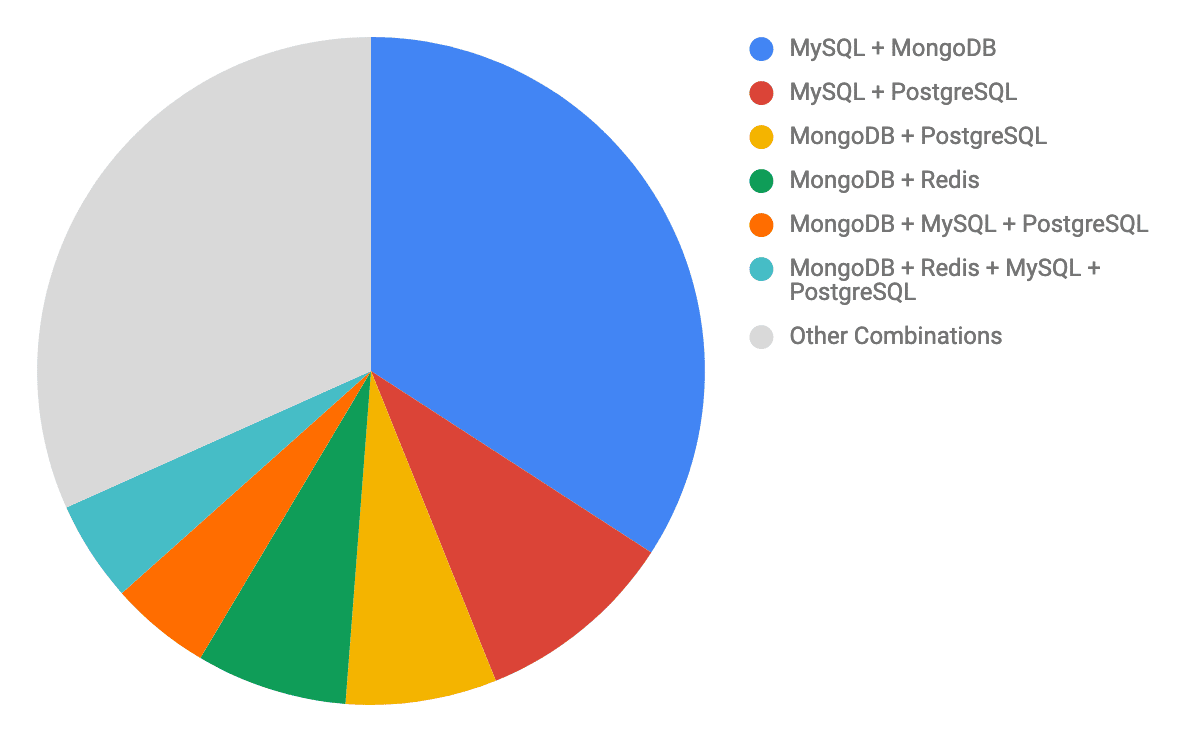
#### MySQL + PostgreSQL: 9.76%

#### MongoDB + PostgreSQL: 7.32%

#### MongoDB + Redis: 7.32%

#### MySQL + MongoDB + PostgreSQL: 4.88%

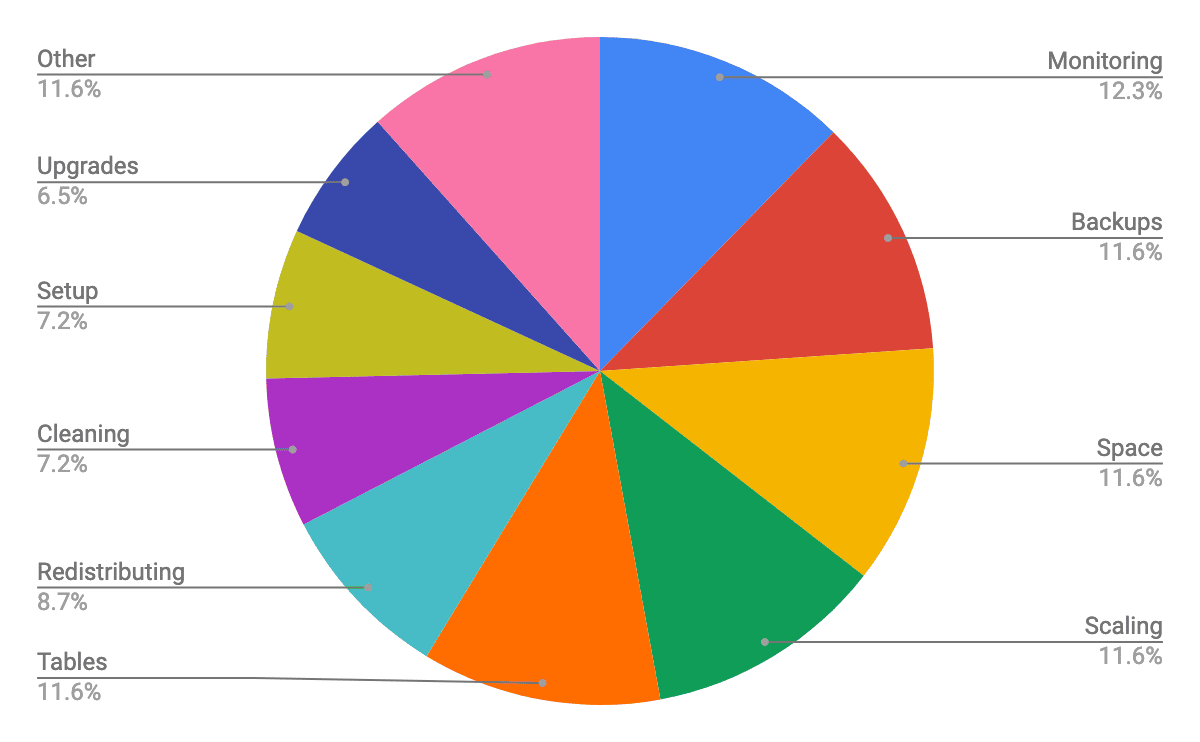
#### MySQL + MongoDB + PostgreSQL + Redis: 4.88%

[](https://scalegrid.io/wp-content/uploads/2019/02/Most-Popular-Multiple-Database-Type-Use-Combinations.png)

## Most Time-Consuming Database Management Task

So, now that we know which database management systems, types, and use combinations are most popular, let’s take a look at what’s eating up our time on the database management front. As anyone who’s managed a database before knows, there are countless tasks involved with maintaining a healthy production deployment. So, we were not surprised to see such a diverse response in our most time-consuming database management task question.

Monitoring came in at number one with 12.6% from our respondents, barely breaking ahead of backups, managing disk space, scaling, and joining tables who all tied for number two with 11.6% each. Standalone at number three was maintaining and redistributing changes between views and stored programs at 8.7%, and again a tie at number 4 with 7.2% for each cleaning and database setup. Upgrades came in at number five with 6.5%, and a dozen other tasks made up the 11.6% Other category, including migrations, queries, comparing, tuning, and replication.

[](https://scalegrid.io/wp-content/uploads/2019/02/Most-Time-Consuming-Database-Management-Task-Chart.png)

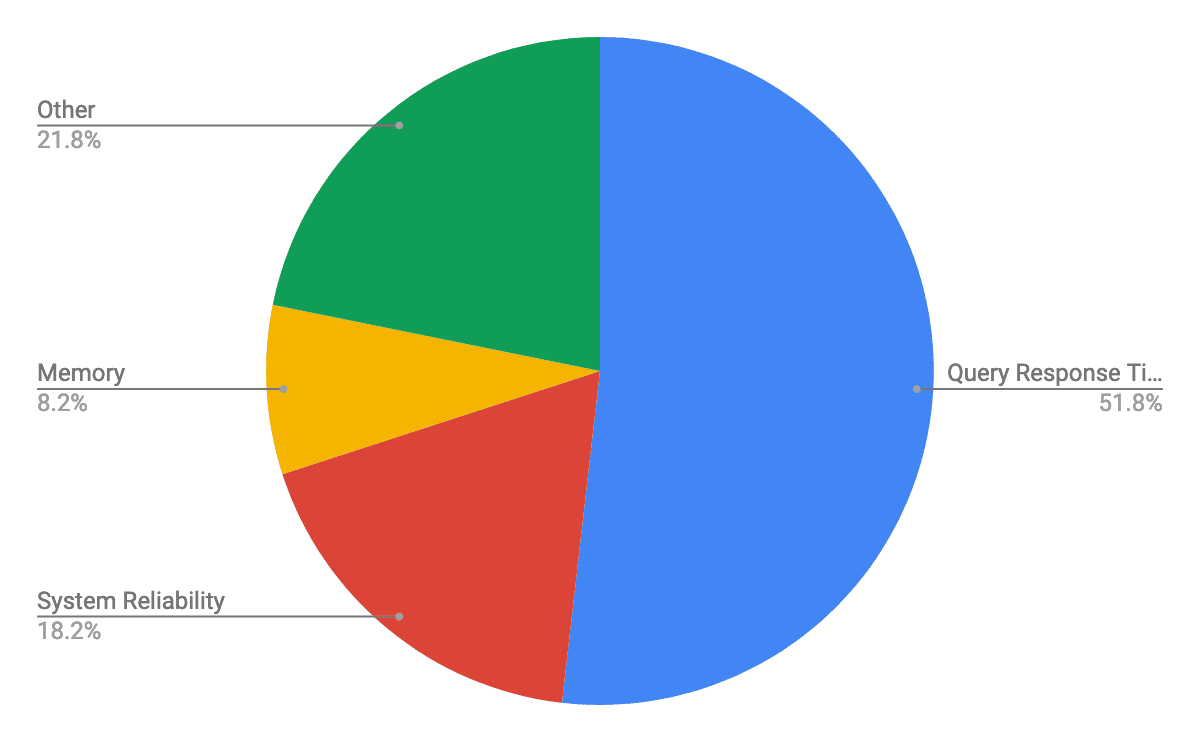
## Most Important Metric Tracked For Database Performance

While we saw a wide variety of responses for the most important database management task, the most important metric to track for performance had three significant leaders.

Query response time was not only the most tracked metric, but also the majority with 51.8% of responses! We expected this to lead as it came in at 30.8% from a [Most Time-Consuming PostgreSQL Management Task](https://scalegrid.io/blog/latest-postgresql-trends-most-time-consuming-tasks-important-metrics-to-track/) report we compiled in October of 2018, but significantly increased when we expanded this question to all database management systems. Query speed is an extremely important metric to track on a continuous basis so you can identify slow-running queries that could be affecting your application performance. Many DBA’s use a [Slow Query Analyzer](https://help.scalegrid.io/docs/mysql-monitoring-slow-query-analyzer) tool to identify problem queries, see which sort of query it is associated with, understand their queries by time range, and find the top queries causing read-load in your system to identify those queries that are not indexed.

Coming in at number two was reliability with 18.2% from our respondents. Needless to say, while outages are less common than slow queries, if your databases goes down, it will have the most serious impact on your performance. That’s why it’s critically important to implement a [high availability framework](https://scalegrid.io/blog/mysql-high-availability-framework-explained-part-1/) for your production deployments to keep your databases online if there’s an outage in one of your datacenters.

Memory then came in at number three with 8.2% of responses. The more memory you have available, the better your database should perform. Both understanding and [monitoring memory usage](https://scalegrid.io/blog/tracking-mongodb-memory-usage/) should be high on your list, as insufficient or exhausted memory will cause your database to read and write data to your disk which is dramatically slower.

[](https://scalegrid.io/wp-content/uploads/2019/02/Most-Important-Metric-Tracked-For-Database-Performance-Queries-Reliability-Memory.png)

Thanks to the hundreds of participants who contributed to the cloud database trends report at DeveloperWeek 2019! We are excited to share these insights, and hope to hear your thoughts below in our comments.

**SQL Case Study - Country Club**

In this subunit, you’ll bring together all your new knowledge to answer business questions with high-level SQL queries. In the process, you’ll be querying an external MySQL database through the PHPMyAdmin interface, and you’ll use a local SQLite instance on your computer. You’ll use one of these two methods of doing database work every day in your career, so understanding both is vital to your success as a data scientist.

1

**Working with Relational Databases in Python**

Save



1 - 1.5 Hours

22 Points

Chapter Three of this DataCamp course will give you a primer on how to work with relational databases in Python, a practical professional skill that you'll draw on while working on the next case study. Typically, you'll work with databases in one of two ways: you'll either make a local instance of the database on your computer or you'll connect to an external database. You'll do both in the case study. Chapter Three of this course will walk you through how to do that.

**Please note that only Chapter Three of this resource is mandatory.**

2

**Introduction to PHP MyAdmin**

Open in new tab

Save



2 - 4 Minutes

1 Points

For the coming case study, you'll be using PHPMyAdmin: an open-source administration tool for MySQL and MariaDB. This much loved and commonly used interface takes a little getting used to. The aim of this video is to help you become familiar with PHPMyAdmin before you tackle the case study.

3

**Case Study - Country Club**

Save

3 - 6 Hours

82 Points

In this case study, you'll use MySQL, PHPMyAdmin, Juptyer Notebook, and SQLite to tackle a series of challenges on a database containing information about a country club. For this case study, you'll see what a database looks like on a nice interface like PHPMyAdmin, and connect to a local instance of the database using Python and SQLite. Data scientists use these methodologies to do work on a daily basis. You'll retrieve particular pieces of information and translate requests into SQL queries. Make use of the DataCamp courses from the previous subunits to succeed in this case study!

There are two difficulty tiers for this case study. The only difference between them is the amount of guidance we give in setting up the SQLite connection in Python. We always encourage students to challenge themselves, but it's also totally fine if you want more guidance while working on complex assignments like this one.

* [**Tier Two**](https://www.springboard.com/archeio/download/18a1b2578e084d51999efa6a9fd2144a/)**:** If you feel confident tackling the challenge of exploring data from PHPMyAdmin and establishing a connection to a local SQLite database, pick Tier Two.
* [**Tier One**](https://www.springboard.com/archeio/download/1bf0319530ab4b599eb7973508728921/): If you want some helpful pointers during that process, go for Tier One. The SQL queries will be the same for both tiers. In this project folder, you will also find two resources that will guide you through the setup of SQLite.

Once you’ve decided on your difficulty level, paste the relevant SQLFiles folder into your working directory on PHPMyAdmin. Open the SQLTasks file containing the instructions for your case study. Happy querying!

**Creating a database engine in Python**

**50 XP**

**1. Creating a database engine in Python**

All right, we're back! What we really want to do is to get data out of our databases using SQL, or the Structured Query Language. But before we get to that, we're going to need to figure out how to connect to a database.

**2. Creating a database engine**

We'll use an SQLite database as an example because SQLite is fast and simple while still containing enough functionality to introduce you to all the necessary concepts of querying a database. There are times when you would prefer to use PostgreSQL or MySQL, but for our purposes here, an Introduction to Interacting with Relational Databases in Python, a SQLite database is perfect. We'll once again look at the Northwind database. There are many packages we could use to access an SQLite database such as sqlite3 and SQLAlchemy. We'll use SQLAlchemy as it works with many other Relational Database Management Systems, such as Postgres and MySQL. So without further ado, to connect to 'Northwind dot sqlite', we need to import the relevant funtion create\_engine from the package SQLAlchemy. We then use the function create\_engine to fire up an SQL engine that will communicate our queries to the database. The only required argument of create\_engine is a string that indicates the type of database you're connecting to and the name of the database. Next, In order to query the database, we need to connect to the engine to do so.

**3. Getting table names**

But before we do this, we would like to know the names of the tables it contains. To do this, apply the method table\_names to the object engine. This will return a list of the table names that you can then print the console. Now it's your turn to do the same: fire up the database engine and print the table names!

**4. Let's practice!**

After this, I'll be back to show you how to connect to the engine, query your DBs and then you'll get loads of practice writing your own queries to import data from relational databases!

### Course Description



Much of the world's raw data—from electronic medical records to customer transaction histories—lives in organized collections of tables called relational databases. Being able to wrangle and extract data from these databases using SQL is an essential skill within the data industry and in increasing demand.   
  
In this two-hour introduction to SQL, you'll get to know the theory and the practice through bite-sized videos and interactive exercises where you can put your new-found skills to the test.   
  
SQL is an essential language for building and maintaining relational databases, which opens the door to a range of careers in the data industry and beyond. You’ll start this course by covering data organization, tables, and best practices for database construction.   
  
The second half of this course looks at creating SQL queries for selecting data that you need from your database. You’ll have the chance to practice your querying skills before moving on to customizing and saving your results.   
  
PostgreSQL and SQL Server are two of the most popular SQL flavors. You’ll finish off this course by looking at the differences, benefits, and applications of each. By the end of the course you’ll have some hands-on experience in learning SQL and the grounding to start applying it on projects or continue your learning in a more specialized direction.

Read More

1. 1

#### Relational Databases

0%

Before writing any SQL queries, it’s important to understand the underlying data. In this chapter, we’ll discover the role of SQL in creating and querying relational databases. Using a database for a local library, we will explore database and table organization, data types and storage, and best practices for database construction.

[View Chapter Details](https://www.datacamp.com/courses/introduction-to-sql?embedded=true)

  2

#### Querying

0%

Learn your first SQL keywords for selecting relevant data from database tables! After practicing querying skills in a database of books, you’ll customize query results using aliasing and save them as views so they can be shared. Finally, you’ll explore the differences between SQL flavors and databases such as PostgreSQL and SQL Server.

[View Chapter Details](https://www.datacamp.com/courses/introduction-to-sql?embedded=true)

**Daily XP0**

**Databases**

**50 XP**

**1. Databases**

Hello and welcome! My name is Izzy Weber, and I will be your SQL coach.

**2. Course goals**

We have two main goals in this course. In Chapter One, we will get to know databases, which store and organize data electronically. We'll discuss how databases and the data they store are structured. This context will prepare us for our second goal: to extract data from databases using SQL code in Chapter Two. Let's dive in!

**3. Introducing databases**

A database stores data. Let's imagine that we are in charge of storing and organizing data for a library. We might set up a database that holds information such as the data pictured here on patrons, books, and checkouts. This information is housed in objects called tables, with data organized into rows and columns. This database contains a patrons table, a books table, and a checkouts table.

**4. Introducing databases**

A closer look at the patrons table shows that it stores various data about our library's patrons, like library card number, name, the year the patron became a library member, and the total overdue fines the patron owes our library.

**5. Relational databases**

A relational database defines relationships between tables of data inside the database. For example, each of our library patrons might each be associated with several checkouts. Through these relationships, we can draw conclusions about data housed in separate tables in the same database, and answer questions such as "Which books did James check out during 2022?" or "Which books are checked out most often?"

**6. Database advantages**

These tables might look similar to the way data is organized in spreadsheet applications such as Excel or Google Sheets, but databases are far more powerful than spreadsheets. Databases can store much more data, and storage is more secure due to encryption.

**7. Database advantages**

Possibly the biggest advantage of a database is that many users can write queries to gather insights from the data at the same time. When a database is queried, the data stored inside the database does not change: rather, the database information is accessed and presented according to instructions in the query. Which leads us to the star of this show:

**8. SQL**

SQL! SQL, or S-Q-L, is short for Structured Query Language. It is the most widely used programming language for creating, querying, and updating relational databases. Once we are familiar with the data we have and which table it is stored on, we can use SQL to begin writing queries to answer questions about our library -- more on that in Chapter Two.

**9. Let's practice!**

Alright, let's flex this newfound database knowledge in some exercises!

# What are the advantages of databases?

Imagine you are part of a discussion at work about whether or not to create a database. You've learned about several advantages of storing data in a database rather than other traditional formats like spreadsheets.

See if you can remember what they are by selecting all of the advantages.

##### Answer the question

**50XP**

#### Possible Answers

**More storage**

press1

**Many people can use at once**

press2

**Can be secured with encryption**

press3

Can easily see all data at once access it only by queries

press4

Fast and easy setup only professional can set it up

press5

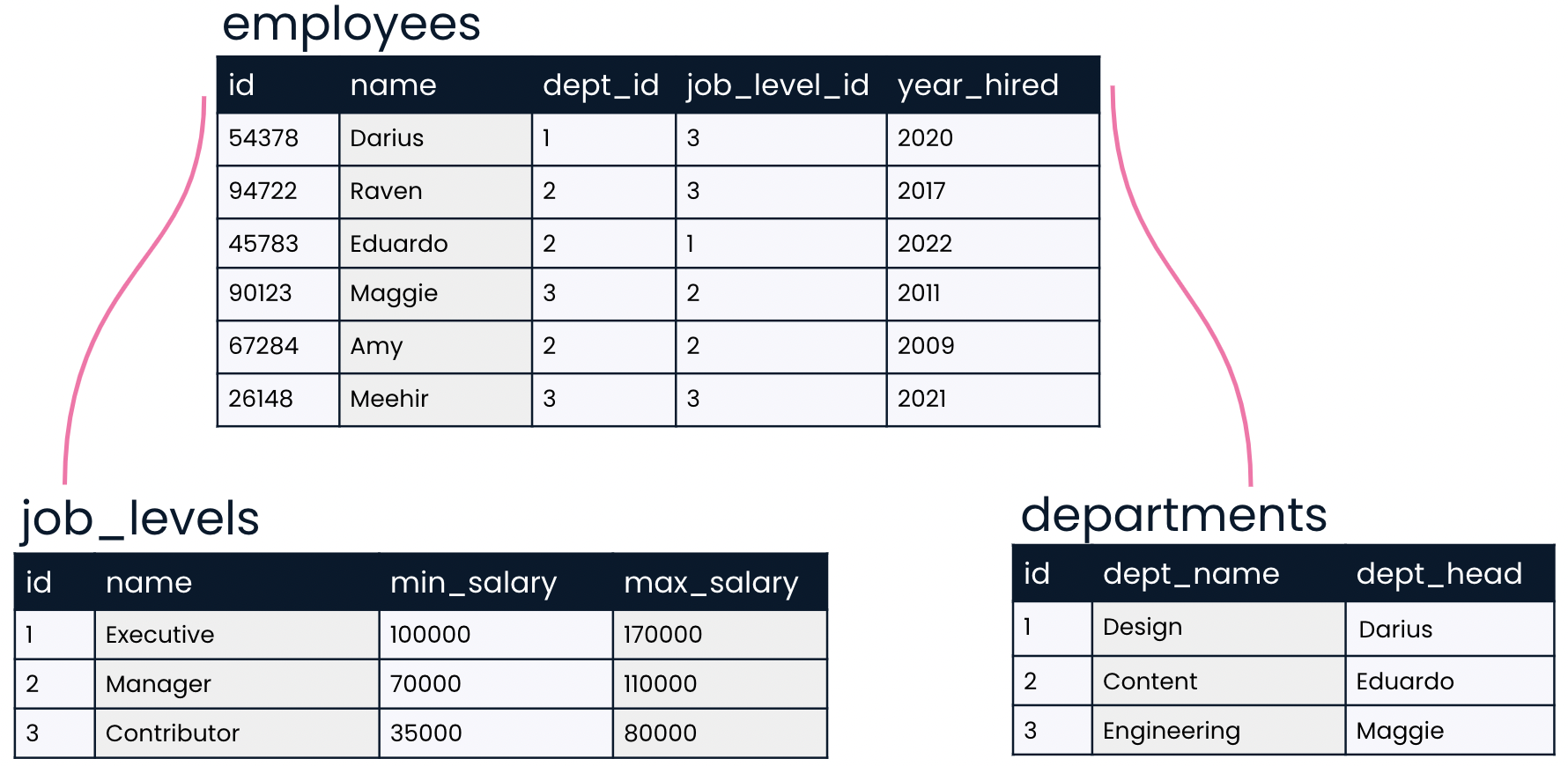
+50 XP

Well done! There are many advantages of databases, which is why much of the world's data is stored in them.

**Daily XP100**

# Data organization

If you'd like to use SQL to gain insights from data, understanding the organization of a database is an important first step. Take a look at the database below. Which of the following statements correctly describes its organization?



##### Answer the question

**50XP**

#### Possible Answers

* 

This is a table containing three relational databases: employees, job\_levels, and departments.

press1

* 

**This is a relational database containing three tables: employees, job\_levels, and departments.**

press2

* 

This is a database, but it is not relational, because no relationship exists between job levels and departments.

press3

* 

This is not a database because there is no SQL code shown.

Well done! This is an example of how data in databases is organized into many tables.

**Daily XP200**

# Picking a unique ID

You've learned that a unique identifier is a unique value that identifies a record so that it can be distinguished from other records in the same table.

Let's take a closer look at the employees table. Which of the fields do you think is best suited to be a unique identifier?

Table

Description automatically generated

##### Answer the question

**50XP**

#### Possible Answers

* 

name

press1

* 

dept\_id

press2

* 

year\_hired

press3

* 

**Id**

press4

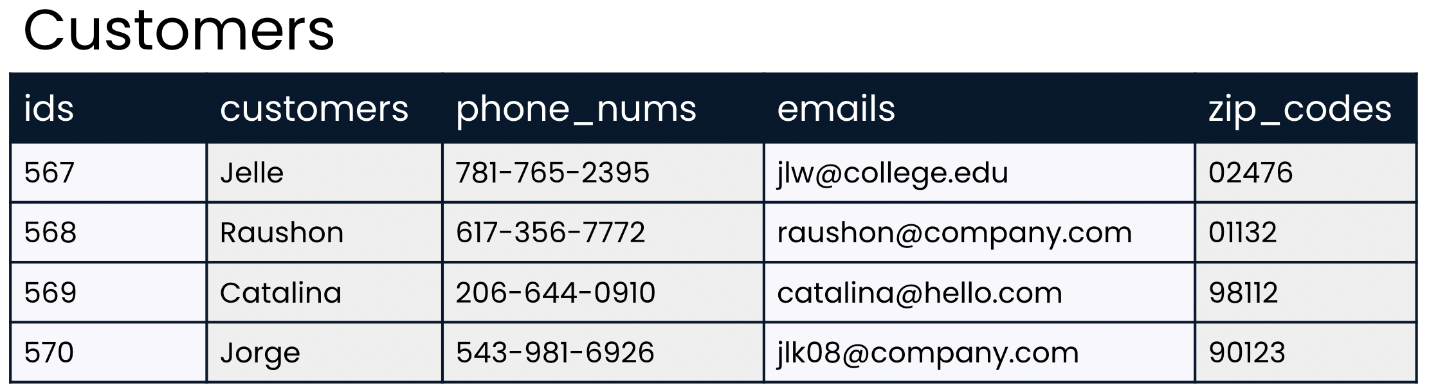
Yes, that's right! id is unique across all current employees, and we can expect that new employees will receive unique ids as well.

**Daily XP250**

##### Exercise

#### Setting the table in style

Imagine that you are designing a database and the following table has been suggested. Your task is to provide feedback on how this table could be improved. Use the skills you learned in the last video to critique it!



##### Instructions

**100XP**

* Move each card to the proper bucket based on whether or not it is a good suggestion for improvement.
* +100 XP
* Well done on improving that table! Getting naming conventions right makes life much easier when we begin writing SQL queries, so it's important to set ourselves up for success

**Daily XP350**

##### Exercise

##### Exercise

# Our very own table

We've set up a database inside this course, and the books table is available in the exercise. You'll use SQL to query this table in the next chapter, but for now, it's time to explore what data books holds!

Your task is to choose the option below that best describes the information contained in books. There's no need to do any coding in this exercise; you can answer this question by looking at the books table in the console next to the words "query result." Because some book titles are long, you may need to scroll to the right in order to see all the information that the books table contains.

##### Instructions

**50 XP**

##### Possible Answers

* 

books contains records for id, title, author, year, and genre.

* ****

**books contains fields for id, title, author, year, and genre.**

* 

books contains records for title, author, year, and genre. id is a unique identifier but not a record.

* 

books contains fields for title, author, year, and genre. id is a unique identifier but not a field.

Great work! Each record contains information about a book, organized into id, title, author, year, and genre fields.

**Daily XP400**

**Data**

**50 XP**

**1. Data**

Welcome to the final part of the databases chapter! This lesson will focus on the data inside a database as well as its storage.

**2. SQL data types**

When a table is created, a data type must be indicated for each field. The data type is chosen based on the type of data that the field will hold - a number, text, or a date for example. We use data types for several reasons. First, different types of data are stored differently and take up different amounts of storage space. Second, some operations only apply to certain data types. It makes sense to multiply a number by another number, but it does not make sense to multiply text by other text for example.

**3. Strings**

In programming, a "string" refers to a sequence of characters such as letters or punctuation. On the patrons table, the data in the names field is made up of strings, such as "Maham" and "James". SQL has several different data types that can hold strings. Some string data types can only hold short strings, such as a string up to 250 characters. Storing short strings in a small data type like this saves storage space. SQL's VARCHAR data type is more flexible and can store small or large strings - up to tens of thousands of characters! Because of its flexibility, VARCHAR is very commonly used for storing strings.

**4. Integers**

Integer data types store whole numbers, such as the years in the member\_year column of the patrons table. Just as with strings, SQL offers a few different data types for storing integers, depending on how big the numbers we'd like to store are. INT, a common SQL integer data type, can store numbers from less than negative two billion to more than positive two billion!

**5. Floats**

Float data types store numbers that include a fractional part, such as the 2-point-05 dollars that one patron, Jasmin, owes in fines. Just as we might expect, SQL also offers several float data types depending on how many digits the numbers in the field are expected to be. The NUMERIC data type can store floats which have up to 38 digits total - including those before and after the decimal point.

**6. Schemas**

Now that we're familiar with data types, we can look at a database schema. Schemas are often referred to as "blueprints" of databases. A schema shows a database's design, such as what tables are included in the database and any relationships between its tables. A schema also lets the reader know what data type each field can hold. The schema for our library database shows the VARCHAR data type is used for strings like book title, author, and genre. We can also see that the patrons table is related to the checkouts table, but not the books table.

**7. Database storage**

Finally, let's discuss storage. The information we find in a database table is physically stored on the hard disk of a server. Servers are centralized computers that perform services via requests made over a network. In our case, the service performed is data access, but servers are also used to access websites or files stored on the server. Any computer can be a server if it is set up to provide a service, even a laptop! However, servers are generally very powerful and large machines, because they are best equipped to handle a high volume of requests and data.

**8. Let's practice!**

Okay, let's get some practice with data!

# At your service

Now that you know more about how data is stored, it's time to test those skills!

Select the statement about database storage that is false.

##### Answer the question

**50XP**

#### Possible Answers

* 

Servers can be used for storing website information as well as databases.

press1

* 

A server can handle requests from many computers at once.

press2

* 

**Servers are usually personal computers such as laptops.**

press3

* 

Data from a database is physically stored on a server.

press4

+50 XP

That's right! While it is technically possible for a laptop to be a server, laptops aren't well suited to the job since they are not powerful enough to handle many requests at once and don't have as much storage capacity as larger computers.

# Finding data types

Imagine that you are starting a new job and have just started getting to know your new employer's database. You know that it's important to know the data type—such as VARCHAR, INT, or NUMERIC—corresponding to each field in a table. Where could you find this information?

##### Answer the question

**50XP**

#### Possible Answers

* 

You can find this information by looking at each table in the database.

press1

* 

You can find this information by looking at a diagram of relationships between tables.

press2

* 

You can find this information by looking at the values in each field for each table.

press3

* 

**You can find this information by looking at a database schema.**

press4

+50 XP

Awesome work. Database schemas show data types for each field in all tables, and they also show relationships between tables. Looking at a schema is an excellent way to get to know a new database!

**Daily XP550**

##### Exercise

#### Choice of type

You've learned that when a table is created, a data type must be indicated for each field. Choosing the correct data type allows the data to be stored correctly and makes certain operations associated with that data type available. For example, mathematical operations can be performed on NUMERIC and INT data types, but not on VARCHAR data. Thus, it makes sense to store numerical values as NUMERIC or INT so that you can perform math operations on them if needed.

In this exercise, you'll practice selecting the proper data type for your data!

##### Instructions

**100XP**

* Drag the field description to the bucket indicating the best data type to use for that field.
* +100 XP
* Congratulations! Now that you know what data types to expect in each field, we're ready to query!
* **Daily XP650**
* **Introducing queries**
* **50 XP**
* **1. Introducing queries**
* Welcome back. Now that we understand how data is organized in databases, we can begin drawing insights using SQL queries!
* **2. What is SQL useful for?**
* Recall from the last chapter that SQL is used to answer questions both within and across relational database tables. In the library database, we might use SQL to find which books James checked out from the library in 2022. In an HR database, we could query salaries for employees in Marketing and Accounting to determine whether pay across departments is comparable.
* **3. Best for large datasets**
* In many organizations, SQL is used as a complement to other tools such as spreadsheet applications. If the data we're interested in can fit in a spreadsheet and does not have many relationships to other data of interest, we can analyze it in a spreadsheet. But for sprawling and diverse data such as the data related to a retail platform, organizing the data in a database is best. Then, we use SQL queries to uncover trends in website traffic, customer reviews, and product sales. Which products had the highest sales last week? Which products get the worst review scores from customers? How did website traffic change when a feature was introduced? SQL shines when an organization has lots of data with complex relationships.
* **4. Keywords**
* Let's write our first SQL code! To do that, we will need to learn a few keywords. Keywords are reserved words used to indicate what operation we'd like our code to perform. The two most common keywords are SELECT and FROM. Perhaps we'd like a list of every patron our library has. The SELECT keyword indicates which fields should be selected - in this case, the name field. The FROM keyword indicates the table in which these fields are located - in this case, the patrons table.
* **5. Our first query**
* Let's put these parts together. Here's how the query should be written. The SELECT statement appears first, followed on the next line by the FROM statement. It's best practice to end the query with a semicolon to indicate that the query is complete. We also capitalize keywords while keeping table and field names all lowercase. Now let's take a look at the results of our query, often called a result set. The result set lists all patron names, just as we had hoped. Note that we have not changed our database by writing this query. The tables, including the patrons table, are exactly the same as before we wrote the query. In order to share our results, we can save the SQL code we have written so that our collaborators can use it to query the database themselves. We'll cover saving queries in a later lesson.
* **6. Selecting multiple fields**
* To select multiple fields, we can list multiple field names after the SELECT keyword, separated by commas. For example, to select card number and name, we'd list both field names in the order we'd like them to appear in our result set. Notice that this does not have to match the order the fields are presented in the table: listing name before card\_num means that name appears first in the results.
* **7. Selecting multiple fields**
* As you might expect, we can select three fields such as name, card\_num, and total\_fine by listing all three field names after the SELECT keyword and separating them with commas.
* **8. Selecting all fields**
* What if we'd like to select all four fields in the patrons table? We could list out the four field names after the SELECT statement, but there's an even easier way: we can tell SQL to select all fields using an asterisk in place of the four field names.
* **9. Let's practice!**
* Let's get some hands-on experience writing our own queries! In this chapter, you'll work with the books table introduced in the previous chapter, which lists all 350 books that our little library owns.

**Daily XP700**

# SQL strengths

Which of the below scenarios describes a situation in which using SQL would be useful?

##### Answer the question

**50XP**

#### Possible Answers

* 

All data needed to answer the business question is presented in a spreadsheet, and no complicated relationships exist between different data points.

press1

* 

**Large amounts of data about many different but related areas of a business are housed in a relational database.**

press2

* 

The data needed to answer the business question doesn't exist yet.

press3

Exactly! When we've got this type of vast and interrelated data, SQL is excellent for targeting and returning only the data which is relevant to our business question.

+100 XP

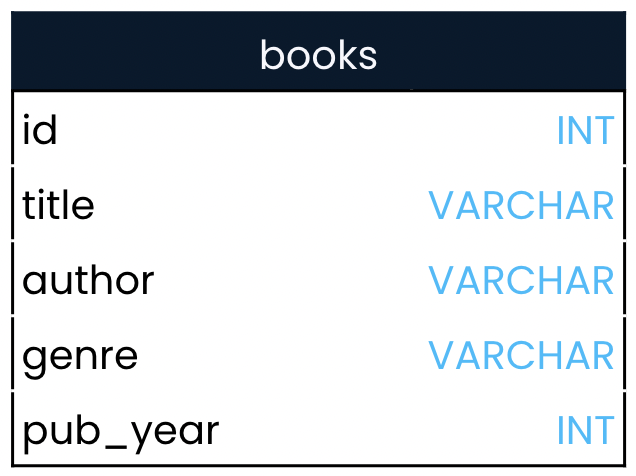
Great style! Your coworkers will thank you when you share clean and readable queries with them.

# Querying the books table

You're ready to practice writing your first SQL queries using the SELECT and FROM keywords. Recall from the video that SELECT is used to choose the fields that will be included in the result set, while FROM is used to pick the table in which the fields are listed.

Feel free to explore books in the exercise. Let's zoom in on this table in the database schema to see the fields and data types it contains.

Your task in this exercise is to practice selecting fields from books.



* [1](javascript:void(0))
  + Use SQL to return a result set of all book titles included in the books table.

 [2](javascript:void(0))

* Select both the title and author fields from books.

 [3](javascript:void(0))

* Select all fields from the books table.

Well done! You've mastered your first two SQL keywords. The more keywords you know, the more complex SQL queries you'll be able to write!

- Select all fields from the books table

SELECT \*

FROM books;

**Daily XP950**

**Writing queries**

**50 XP**

**1. Writing queries**

It's time to level up on our SQL queries by learning a few more commonly used keywords. Let's dive in.

**2. Aliasing**

Sometimes it can be helpful to rename columns in our result set, whether for clarity or brevity. We can do this using aliasing. Perhaps we'd like to select the name and hire year for each record in the employees table. We could alias the name column as first\_name in the query by adding the AS keyword to indicate an alias of first\_name after selecting the name field. The result set now has first\_name rather than name as the column header. The alias only applies to the result of this particular query; in other words, the field name in the employees table itself is still name rather than first\_name.

**3. Selecting distinct records**

Some SQL questions require a way to return a list of unique values. Let's imagine that we are interested in getting a list of years in which we hired our current employees. If we select the year\_hired field from the employees table, the result set shows several years listed twice, which isn't what we are looking for. To get a list of years with no repeat values, we can add the DISTINCT keyword before the year\_hired field name in the SELECT statement. Now, we can see that all of our employees were hired in just four different years.

**4. DISTINCT with multiple fields**

It's possible to return the unique combinations of multiple field values by listing multiple fields after the DISTINCT keyword. Take a look at the employees table. Perhaps we'd like to know the years that different departments hired employees. We could use this SQL query to look at this information, selecting the dept\_id and year\_hired from the employees table. Looking at the results, we see that department three hired two employees in 2021.

**5. DISTINCT with multiple fields**

To avoid repeating this information, we could add the DISTINCT keyword before the fields to select. Notice that the department id and year\_hired fields still have repeat values individually, but none of the records are the same: they are all unique combinations of the two fields.

**6. Views**

Finally, let's discuss saving SQL result sets. In SQL, a view refers to a table that is the result of a saved SQL SELECT statement. Views are considered virtual tables, which means that the data a view contains is not generally stored in the database. Rather, it is the query code that is stored for future use. A benefit of this is that whenever the view is accessed, it automatically updates the query results to account for any updates to the underlying database. To create a view, we'll add a line of code before the SELECT statement: CREATE VIEW, then the name we'd like for the new view, then the AS keyword to assign the results of the query to the new view name. Here, we create a view called employee\_hire\_years by assigning the results of a query selecting three fields from the employees table to a new view. There is no result set when creating a view.

**7. Using views**

Once a view is created, however, we can query it just as we would a normal table by selecting FROM the view.

**8. Let's practice!**

Time to practice refining and saving queries with these new keyword

**Daily XP1000**

##### Exercise

##### Exercise

# Making queries DISTINCT

You've learned that the DISTINCT keyword can be used to return unique values in a field. In this exercise, you'll use this understanding to find out more about the books table!

There are 350 books in the books table, representing all of the books that our local library has available for checkout. But how many different authors are represented in these 350 books? The answer is surely less than 350. For example, J.K. Rowling wrote all seven Harry Potter books, so if our library has all Harry Potter books, seven books will be written by J.K Rowling. There are likely many more repeat authors!

##### Instructions 1/2

**50 XP**

* [1](javascript:void(0))
  + Write SQL code that returns a result set with just one column listing the unique authors in the books table.

 [2](javascript:void(0))

* Update the code to return the unique author and genre combinations in the books table.

ou've passed this exercise with DISTINCTion! Notice that you found 247 unique authors in the books table overall but 249 unique combinations of authors and genres. This means there are one or two authors who have written books in multiple genres!

-- Select unique authors and genre combinations from the books table

SELECT DISTINCT author, genre

FROM books;

**Daily XP1100**

##### Exercise

##### Exercise

# Aliasing

While the default column names in a SQL result set come from the fields they are created from, you've learned that aliasing can be used to rename these result set columns. This can be helpful for clarifying the intent or contents of the column.

Your task in this exercise is to incorporate an alias into one of the SQL queries that you worked with in the previous exercise!

##### Instructions

**100 XP**

* Add an alias to the SQL query to rename the author column to unique\_author in the result set.
* -- Alias author so that it becomes unique\_author
* SELECT DISTINCT author AS unique\_author
* FROM books;

It's AS easy AS that! Great work. The alias you just implemented makes it clear that only unique authors are listed in the results and that there are no duplicates. This is clear even to someone who is reading only the result set and does not know the SQL code behind the results

**Daily XP1200**

##### Exercise

##### Exercise

# VIEWing your query

You've worked hard to create the below SQL query:

SELECT DISTINCT author AS unique\_author

FROM books;

What if you'd like to be able to refer to it later, or allow others to access and use the results? The best way to do this is by creating a view. Recall that a view is a virtual table: it's very similar to a real table, but rather than the data itself being stored, the query code is stored for later use.

##### Instructions 1/2

**50 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* Add a single line of code that saves the results of the written query as a view called library\_authors.
* -- Save the results of this query as a view called library\_authors
* \_\_\_
* SELECT DISTINCT author AS unique\_author
* FROM books;
* -- Save the results of this query as a view called library\_authors
* CREATE VIEW library\_authors AS
* SELECT DISTINCT author AS unique\_author
* FROM books;

-- Your code to create the view:

CREATE VIEW library\_authors AS

SELECT DISTINCT author AS unique\_author

FROM books;

-- Select all columns from library\_authors

SELECT \*

FROM library\_authors;

Amazing! As your SQL queries become long and complex, you'll want to be able to save your queries for referencing later. Views can also be useful when the information contained in a database table isn't quite what you need. You can create your own custom view with exactly the information you are looking for, without needing to edit the database itself, which you may not have permission to do. Creating views is a valuable skill to have, and you've mastered it!

**Daily XP1300**

**SQL flavors**

**50 XP**

**1. SQL flavors**

Our last topic in this short course is SQL flavors.

**2. SQL flavors**

SQL has a few different versions, or flavors. Some are free, while others have customer support and are made to complement major databases such as Microsoft's SQL Server or Oracle Database, which are used by many companies. All SQL flavors are used with table-based relational databases like the ones we've seen, and the vast majority of keywords are shared between them! In fact, all SQL flavors must follow universal standards set by the International Organization for Standards and the American National Standards Institute. Only additional features on top of these standards result in different SQL flavors.

1. 1 Table flatlay photo created by freepik www.freepik.com

**3. Two popular SQL flavors**

Let's take a look at two of the most popular SQL flavors. PostgreSQL is a free and open-source relational database system which was originally created at the University of California, Berkeley, and was sponsored by America's famous Defense Advanced Research Projects Agency, or DARPA. DARPA also sponsored research leading to creating the internet, the computer mouse, and GPS! The name "PostgreSQL" is used to refer to both the database system itself as well as the SQL flavor used with it. SQL Server is also a relational database system which comes in both free and enterprise versions. It was created by Microsoft, so it pairs well with other Microsoft products. T-SQL is Microsoft's proprietary flavor of SQL, used with SQL Server databases.

**4. Comparing PostgreSQL and SQL Server**

Think of SQL flavors as dialects of the same language. If Claudia speaks American English, she will have no trouble understanding people on a trip to London, even though most people in London speak British English and there are some small differences. Here's an example of a small difference between SQL Server and PostgreSQL: when we want to limit the number of records returned, we use the LIMIT keyword in PostgreSQL. Here, we limit the number of employee names and ids selected to only the first two records. The exact same results are achieved in SQL Server using the TOP keyword instead of LIMIT. Notice that this keyword is the only difference between the two queries! Limiting results is useful when testing code, since many result sets can have thousands of results! It's best to write and test code using just a few results before removing the LIMIT for the final query.

**5. Choosing a flavor**

New SQL learners may wonder which flavor they should learn. This may be an easy decision if a learner knows that her employer uses Microsoft's SQL Server, for example. Or it might be a hard one for a job seeker or student who doesn't know what database management system a future employer might use. Don't worry too much about what flavor to learn. As we've seen, the differences are small. A PostgreSQL wizard can become a SQL Server wizard by learning a handful of different keywords!

**6. Let's practice!**

Now that we've sampled a few SQL flavors, let's practice!

**Daily XP1350**

##### Exercise

#### Comparing flavors

The video introduced several differences between SQL Server and PostgreSQL. These are just two of many relational database systems available, but the differences are fairly representative of the types of differences you'll see between other SQL database systems and flavors as well.

##### Instructions

**100XP**

* Drag the statement to the database management system that it correctly describes.

**Daily XP1450**

##### Exercise

#### Comparing flavors

The video introduced several differences between SQL Server and PostgreSQL. These are just two of many relational database systems available, but the differences are fairly representative of the types of differences you'll see between other SQL database systems and flavors as well.

##### Instructions

**100XP**

* Drag the statement to the database management system that it correctly describes.

+100 XP

Excellent work! Both SQL Server and PostgreSQL are very popular SQL flavors. As your SQL journey continues, you're sure to see their names pop up!

# Limiting results

Let's take a look at a few of the genres represented in our library's books.

Recall that limiting results is useful when testing code since result sets can have thousands of results! Queries are often written with a LIMIT of just a few records to test out code before selecting thousands of results from the database.

Let's practice with LIMIT!

##### Instructions

**100 XP**

* Using PostgreSQL, select the genre field from the books table; limit the number of results to 10.
* -- Select the first 10 genres from books using PostgreSQL
* SELECT genre
* FROM books
* LIMIT 10;

here's no LIMIT to your SQL skills! Great work. You can see from this exercise how it's nice to work with small result sets by limiting the number of results.

**Daily XP1550**

# Translating between flavors

In the previous exercise, you wrote the following code using PostgreSQL:

SELECT genre

FROM books

LIMIT 10;

The database in this course is a PostgreSQL database, so you won't be able to run SQL Server code in any of the exercises. What if you did want to update the above query to work with SQL Server, though? How would you do that?

##### Answer the question

**50XP**

#### Possible Answers

* 

Replace FROM with TABLE

press1

* 

Replace SELECT, FROM, and LIMIT with the corresponding SQL Server keywords

press2

* 

**Replace LIMIT with TOP**

press3

* 

Replace LIMIT with TOP and remove the ; at the end of the query

press4

+50 XP

Looks like you've got the flavor of things! Great work. Only a few keyword differences exist between PostgreSQL and SQL Server—most syntax is the same!