Creating and Connecting to Databases

Contents

- 6.1. Introduction: What is a Database?
- 6.2. Types of Databases
- 6.3. Working With Databases in Python

Table of Contents

Creating and Connecting to Databases

- Introduction: What is a Database?
  - Querying a Database vs. Using an API
  - Atomicity, Consistency, Isolation, Durability (ACID)
  - A Brief History of Databases
- Types of Databases
  - Navigational Databases
  - Relational Databases
    - Relational Database Normalization
    - Entity-Relationship Diagrams
    - Relational Database Management Systems (RDBMS)
  - NoSQL Databases
    - Comparing NoSQL Databases to Relational Databases
    - Key-Value Stores
    - Document Stores
    - Wide-Column Stores
    - Graph Databases
- Working With Databases in Python
  - Installing Database Systems Using Docker
  - Example: Wine Reviews
  - Using SQLite
  - Using MySQL
    - Connecting to the MySQL Server and Creating a New Database
    - Connecting to an Existing Database on MySQL
  - Using PostgreSQL
    - Connecting to the PostgreSQL Server and Creating a New Database

- Connecting to an Existing Database on PostgreSQL
- Using MongoDB

6.1. Introduction: What is a Database?

A database is an organized collection of many data records, whether those records are tables, JSON, or another format. If a single data record is a book, then a database is a library: the library contains many books, which are organized according to some system such as alphabetical order or the Dewey Decimal System. Databases, if they are carefully organized and well-maintained, allow big organizations to keep track of huge amounts data and access specific data quickly. Databases also make security easier: by storing all the data together inside a database, it is also easier to use encryption, credentials, and other security measures to control who has access to what data inside the database. Databases can exist locally, on your own computer's hard drive for example, or on remote servers that anyone with an internet connection and the right credentials can access.

6.1.1. Querying a Database vs. Using an API

As a data scientist, you will frequently have to issue requests to remote databases to get the data you need. These requests are called **queries**. We've already discussed sending queries to a database in Chapter 4 through APIs, but here we will focus on situations in which no API is available. Remember that an API is code that takes a request as input, translates that request to language the database can understand, and returns the requested data subject to the API's restrictions and security measures. A REST API requires client-server separation, and that allows database managers to make changes to a database without requiring front end users to change their code for issuing a query because the API does that translation on behalf of the user. So if a remote database has an API, the database managers intend for users to make queries through that API, and not to query the database directly.

If there is no API, users will have to pay close attention to how the database is organized to query the database correctly. The specific way in which a database is organized is called the **schema**. If the database managers change the schema for any reason, users will need to change their queries to work with the new schema.

Both APIs and direct queries can include security measures, but APIs allow more nuanced security. For

example, an API can be used to keep track of how much data is being requested and how many calls have been issued and block users that exceed pre-set limits. A database without an API can require a username and password to grant or deny access to the data, but cannot in general place restrictions on the extent of use the way APIs can.

6.1.2. Atomicity, Consistency, Isolation, Durability (ACID)

The purpose of a database is to store data and make sure that data can always be transferred to approved applications. A transfer of data is called a **transaction**, and a transaction is considered valid if it meets four criteria - atomicity, consistency, isolation, and durability - that were first described by Theo Haerder and Andreas Reuter in 1983:

- Atomicity: one transaction might require many actions to be run in sequence. For example, making a credit card purchase involves five steps:
    1. The customer swipes the card and approves the purchase
    2. The vendor's card reader communicates the payment information to the vendor's bank
    3. The vendor's bank communicates with the credit card issuer
    4. The issuer either declines or approves and pays the request and sends this information to the bank
    5. The bank communicates the result to the vendor

  Atomicity says that all of these steps should be considered part of a single transaction. The transaction is successful if and only if all five steps are successful, and the transaction fails if any of these steps fail. In the case of a credit card purchase, that means that if the credit card issuer pays the bank, but the bank fails to transfer that payment to the vendor, then the entire transaction fails and the issuer's payment to the bank is cancelled: that prevents situations where people disagree about who did or did not pay.

- Consistency: the result of a transaction is not allowed to invalidate the data that already exists in the database. If there is an error that causes an invalid datapoint to be transmitted to the database, the entire transaction is cancelled. Consistency also means that no data should contradict other data. For example, if a customer's account is debited by some amount, it might take some time for the transaction to be communicated to both the bank and the credit card company. A transaction is not considered complete until the record of the customer's account balance is updated in every system. It might take longer to complete a transaction this way, but this step ensures that there's no possibility of problems due to different systems having different balances for the same account.
- Isolation: if there are many transactions issued to the same database in a short period of time, one transaction must be fully completed before the next transaction is processed. That ensures that the information available in the database for each transaction is the most up-to-date.
- Durability: once a transaction has been made, the record of that transaction cannot be erased, even if the system experiences a hardware failure such as a power outage.

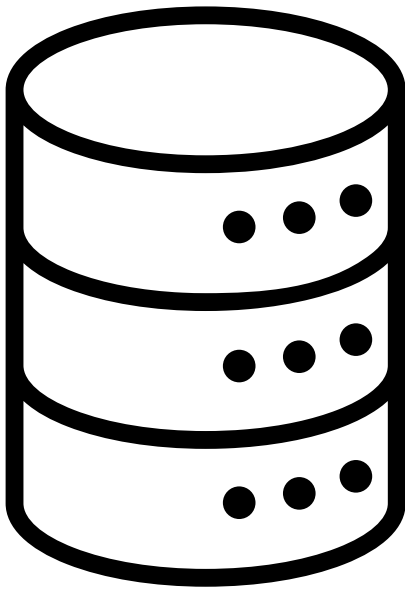
The ACID criteria are standards that well-maintained databases need to achieve, regardless of how the data are organized, stored, and used.

6.1.3. A Brief History of Databases

According to Kristi L. Berg, Tom Seymour, and Richa Goel (2013, p. 29):

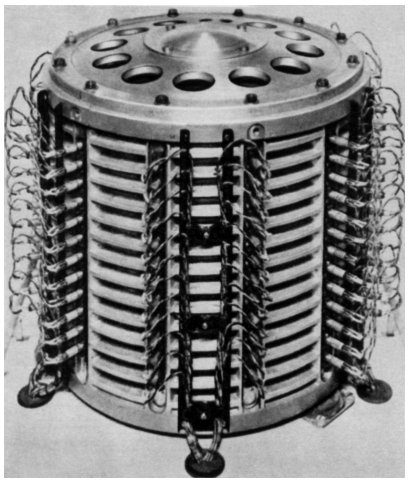
> The origins of the database go back to libraries, governmental, business and medical records before the computers were invented… . Once people realized they needed to have the means to store data and maintain the data files for later retrieval, they were trying to find ways to store, index, and retrieve data. With the emergence of computers, the world of the database changed rapidly, making it an easy, cost effective, and less space-consuming task to collect and maintain the database.

While physical database systems date back to ancient times, electronic databases were invented in the 1960s to work with the first computers capable of using disk storage. The first databases emphasized storage over the content of the data. The focus was to find sufficient physicial storage for a large number of records with the same fields and different values for those fields. As the early concerns revolved around physical storage, the icon for a database resembles physical disk storage. You've probably seen some representation of this icon to represent a database:



Source: Wikimedia Commons

To be specific, this standard database icon is a picture of a drum memory data storage device, which was invented in 1932 and predates disk drives. Here's a picture of a drum memory that would have been used in the 1950s:



Source: Wikipedia

From the 1960s onward, new database types were developed to match the needs of businesses as data and databases gained increased use in industry and government. In the 1970s the invention of the relational database structure allowed users to query a database for records with specific content. Relational databases also enabled more stringent data validation standards - certain fields could only be

populated by values with specific data types - and allowed database managers to add new features without having to go back and revise every record in the database. In the 1980s and 1990s, the relational framework was extended to work with object-oriented programming languages and with web-based applications such as APIs. More recent innovations in database structures respond to the greater need for databases to store massive amounts of data, the need to use distributed storage and cloud computing, and the proliferation of data in which records exist in an interconnected network or come from real-time streaming. The following table summarizes these innovations:

Timeframe	Database Need	New Type of Database
1960s	Store data and page through records	Navigational
1970s-1980s	Search/filter based on the content of records; add new features without having to revise all records in the database	Relational
1980s	Work with object oriented programming languages	Object Oriented
1990s	Use internet connectivity to create client-server systems and APIs	Web-Enabled Relational
2000s	Huge data storage memory requirements; need for flexible schema	NoSQL Databases
2010s	Host databases on distributed systems using cloud computing	Databases with cloud functionality
2010s-2020s	Store data in which records link to each other in a network	Graph databases
2010s-2020s	Store data in which records come from a streaming time series	Time series databases

A **database management system** (DBMS) is software that allows users to create, revise, or delete records in a database, generate security protocols, or issue queries to obtain a selection of the data. Every innovation in database structure and functionality brought about the creation of new DMBSs. People sometimes muddle the terminology and refer to a database, its schema, and a DBMS interchangably: it's not uncommon to hear people speak about a "MySQL database" when it would be more accurate to describe "a database with a relational schema managed using the MySQL DBMS". In addition, there has been a huge proliferation in DBMSs, and people can spend a lot of time debating competiting DBMSs that are only marginally different from each other. That can be frustrating to new coders who are trying to find the "right" or "best" DBMS, and many people have had the experience of finally mastering one DBMS only to be told that its been replaced by a new one. It is important to understand your options when it comes to choosing a database type and a DBMS, but it is also important to understand which distinctions are major and which are minor.

6.2. Types of Databases

As described in the previous section, electronic databases have evolved a great deal since the 1960s to increase the functionality of working with data, to handle new kinds of data, and manage bigger and bigger amounts of data. If you intend to create a database, the first decision you need to make is what type of

database you are going to create. If you are going to work with an existing database, knowing what type of database you are working with will guide you to the correct methods for issuing queries to this database.

To illustrate the differences between types of database, consider the following (fake) data, based on data on books checked out from the Jefferson-Madison Regional Library (https://www.jmrl.org/), a network of public libraries that serves Charlottesville, Virginia and the surrounding region. In this data table, rows represent individual books checked out by particular patrons, every library patron has a unique library card number, every book has a title and at least one author, the checkout dates are recorded but the return dates are `NULL` if the book is still checked out, and the librarian who conducted the transaction is included along with library branch where they work:

CardNum	Patron	Title	ISBN-10	Authorship	Checkout	Return	Librari
18473	Ernesto Hanna	A Brief History of Time	0553380168	Stephen Hawking	8/25/24	9/3/24	José María Chantal
29001	Lakshmi Euripides	Love in the Time of Cholera	0307389731	Gabriel Garcia Marquez	11/23/24	12/14/24	José María Chantal
29001	Lakshmi Euripides	Good Omens	0060853980	[Neil Gaiman, Terry Pratchett]	1/7/25	2/3/25	Antony Yusif
29001	Lakshmi Euripides	The Master and Margarita	0143108271	Mikhail Bulgakov	4/2/25	NULL	Antony Yusif
73498	Pia Galchobhar	Freakonomics: A Rogue Economist Explores the Hidden Side of Everything	006073132X	[Steven Levitt, Stephen J. Dubner]	3/2/25	3/25/25	Dominic Mansoo
73498	Pia Galchobhar	Moneyball: The Art of Winning an Unfair Game	0393324818	Michael Lewis	3/24/25	NULL	Antony Yusif

This table contains all of the data. However, this representation of the data is not the way any of the commonly used database schemas organize the data.

6.2.1. Navigational Databases

Early databases treated records as equivalent units, stored in sequential order, like pages in a book. Patient records at a hospital, for example, could all be stored in a database, and an individual patient's

data could be extracted by flipping through the records until the patient's data appears. Navigational databases also allow records to link to other records based on common features, such as patients who see the same doctor.

If the library data is stored in a navigational database, one record might appear on a librarian's computer screen like this:

Card No.: 29001

Patron: Lakshmi Euripides

Checkout:

Title: Love in the Time of Cholera

Author: Gabriel Garcia Marquez

Checkout: 11/23/19

Return: 12/14/19

Librarian: José María Chantal

Branch: Charlottesville

Previous record

Next record

Previous record by patron

Next record by patron

Previous record at branch

Next record at branch

Notice that only one record appears on the screen, but there are links to the previous and next records in the database, along with the previous or next records for this patron or that occurred at this branch.

The major drawback of navigational database systems is that there is no way to perform a search other than moving through the links, which is time consumming and computationally inefficient. Navigational databases cannot filter data based on the content of a record, so there's no way to see all the times a book by Gabriel Garcia Marquez was checked out, for example. While the early history of databases was dominated by navigational types, and while this type of database still gets used in some applications, most databases no longer use this structure.

6.2.2. Relational Databases

The relational database structure was first proposed by E. F. Codd in 1970. He describes the motivation for a relational database model as follows (p. 377):

> The relational view (or model) of data … appears to be superior in several respects to the [navigational] model presently in vogue for non-inferential systems. It provides a means of describing data with its natural structure only - that is, without superimposing any additional structure for machine representation purposes. Accordingly, it provides a basis for a high level data language which will yield maximal independence between programs on the one hand and machine representation and organization of data on the other.

In other words, relational databases do not have to rely on a pre-set ordering or indexing of the records. There should be no intrinsic meaning to the order of the rows or the order of the columns. That way, to find specific data, a user only needs to search for that data and does not need to lookup the location of the

data in an index. The ability to find records based only on the data present within the record is the key purpose of the structured query language (SQL) which is still the predominant way to work with relational databases. SQL is the “high level data language” that Codd had in mind when he formulated the relational database structure.

Codd also describes a further advantage of the relational schema: “it forms a sound basis for treating derivability, redundancy, and consistency of relations” (p. 377):

- Derivability refers to an ability to use a set of operations to pinpoint a desired selection of the data; that is, we can use queries that operate on the quantities in the data to extract exactly the data we need. If we wanted data on all books checked out from the Crozet branch in February that had more than one author, we can specify a query to return that dataset.
- Redundancy refers to the fact that some of the information in a data table is implied by other features, and therefore takes up more space than it needs to. For example, in the library data shown above, it’s redundant to list “Charlottesville” over and over for every row in which the librarian is Antony Yusif because each librarian works at only one branch. If we know the librarian, we can lookup the branch where that librarian works in a separate table, and separating the data into two tables will reduce the overall amount of memory needed to store the data.
- Consistency refers to the possibility of contradictions within the data. If someone updates the data one record at a time, then it is possible for the person entering the data to make a mistake. For example, I could create another row in the library data that lists Antony Yusif with the Crozet branch, which is a mistake because Antony Yusif works at the Charlottesville branch. Either someone needs to catch and fix this mistake, or else the data now contain contradictory information. The relational database, however, lists librarians with their branches in another table, so there’s no opportunity for me to mistakenly input the wrong branch: instead the correct branch is automatically drawn from this second table.

6.2.2.1. Relational Database Normalization

In order for a relational database to provide derivability, redundancy, and consistency, it must be in **normal form**. Database normalization is a series of rules that a collection of data tables within a relational database must follow in order to solve the problems that Codd identified. There are many, progressive versions of normal forms: each version adds more stringent rules, but solves more problems. Here we will discuss three different organization standards called first, second, and third normal form.

6.2.2.1.1. First Normal Form

To be in first normal form (1NF), data must meet three criteria:

1. Every table must have a **primary key**.

A primary key is one column or a combination of several columns that contain unique identifying values for each row. (A combination of columns that comprise the primary key is sometimes called a superkey.) This column or combination of columns is called a **foreign key** when used in other tables.

2. The values inside every cell in every table must be **atomic**.

That means these values cannot be lists, tuples, dictionaries, tables, or dataframes. Put another way:

every individual piece of data must get its own cell. But what constitutes an “individual piece of data”? The answer depends on the context of the data and the purpose of the database. Take for example the `Patron` column in the library data, which contains the patrons’ first and last names. Is this column non-atomic? It depends on whether we have a purpose in considering the patrons’ first and last names separately. Suppose we run an automated email client that extracts the first names from the database and sends emails to each patron beginning “Dear *firstname*,”, then we should break the `Patron` field into first and last names. But if we have no reason to consider the first and last names separately, then breaking this column up is just an unnecessary complication.

Atomization is necessary in order for the data to be searchable. If we want to find all books by Stephen Hawking, for example, it helps a great deal for each author name to be contained in separate cells.

3. There are no **repeating groups**.

This point is one of the most confusing and contentious in the discourse on relational databases because different sources define the notion of “repeating groups” in different ways. Generally speaking, repeating groups are “a repeating set of columns … containing similar kinds of values in a table.” But that’s really vague and it can be hard to know from one case to the next whether columns are similar enough to be considered repeating. A better way to think about this rule is as follows: repeated groups are almost always problems that arise immediately after splitting up non-atomic data. After atomization, answer the following two questions:

i. Do we have to arbitrarily put numbers or ordering language into the new columns’ names?

ii. Does atomizing data and creating new columns generate new missing values? (Please note that the second question does not include missing values that already existed in the data prior to atomization.)

If the answer to either question is yes, then the data have repeating groups. So to create a 1NF database, it is necessary to atomize data in a way that does not create a repeating groups problem. To see an example of how to proceed, consider again the example of data on transactions from the library:

For this library example, let’s make a few simplifying assumptions:

- No one takes out a book more than once on any one day
- No two authors share the same name (This is unrealistic because there are other authors named Dan Brown, for example. If we wanted to dispel this assumption, we would have to use a unique ID of some kind for authors.)
- Every librarian works at only one library branch, and no two librarians share the same name.

The library data passes criterion 1 for 1NF because `CardNum`, `ISBN-10` (a unique identifying code for books), and `Checkout` together comprise the primary key: these columns uniquely identify the rows because one person checks out a specific book only once on a particular day. But the data fail on criterion 2 because `Authorship` is not atomic: some of the books have multiple authors, and in these cases the authors are stored in lists. If we break `Authorship` up into different columns, like this,

Title	ISBN-10	First Author	Second Author
A Brief History of Time	0553380168	Stephen Hawking	NULL
Love in the Time of Cholera	0307389731	Gabriel Garcia Marquez	NULL
Good Omens	0060853980	Neil Gaiman	Terry Pratchett
The Master and Margarita	0143108271	Mikhail Bulgakov	NULL
Freakonomics: A Rogue Economist Explores the Hidden Side of Everything	006073132X	Steven Levitt	Stephen J. Dubner
Moneyball: The Art of Winning an Unfair Game	0393324818	Michael Lewis	NULL

we would also fail to satisfy criterion 3 because we created a repeating groups problem. We had to include ordering language in names for the columns to store the authors' names: we could have equivalently listed Stephen J. Dubner as the first author of *Freakonomics* and Steven Levitt as the second author, so this placement is arbitrary. In addition, because most of the books do not have a second author, we created a lot of missing values in the `Second Author` column that did not exist in the data before. If the data had included even one book with 3, 4, or more authors, then we would have to create even more author columns with even more missing data. This organization also complicates any search we might want to do on authors: instead of searching for books in which Stephen J. Dubner is an author, we have to do a more complicated search for books in which Stephen J. Dubner is first author, or second author, or third author, and so on.

So, in general, splitting non-atomic data into separate columns is not the solution to creating a 1NF database. Instead, a better approach is to create a second table with each value of the non-atomic column on a new row, along with the columns the non-atomic columns depend on (please see the next section for a more formal definition of what it means for one column to "depend on" another); in this case, authorship depends on book, so we should bring the `ISBN-10` values into the table as well.

In the theoretical literature on relational databases, tables are called **entities** and the columns of each entity are called **attributes**. Entities have names, and we generally name the entities after the unit of observation that defines the rows in the table. We create a new entity to store the authorship data and we call it *AUTHORSHIP*:

AUTHORSHIP

ISBN-10	Author
0553380168	Stephen Hawking
0307389731	Gabriel Garcia Marquez
0060853980	Neil Gaiman
0060853980	Terry Pratchett
0143108271	Mikhail Bulgakov
006073132X	Steven Levitt
006073132X	Stephen J. Dubner
0393324818	Michael Lewis

In this entity, `ISBN-10` and `Author` together comprise the primary key. Unlike the table with columns for each author, there are no new missing values here. Also, searching for specific authors is much more straightforward now as it requires matching to just the one Author attribute.

To meet the rules of 1NF, we also remove the authors from the original table, which we can call the *TRANSACTIONS* entity:

TRANSACTIONS

CardNum	Patron	Title	ISBN-10	Checkout	Return	Librarian	Branch
18473	Ernesto Hanna	A Brief History of Time	0553380168	8/25/24	9/3/24	José María Chantal	Charlottes
29001	Lakshmi Euripides	Love in the Time of Cholera	0307389731	11/23/24	12/14/24	José María Chantal	Charlottes
29001	Lakshmi Euripides	Good Omens	0060853980	1/7/25	2/3/25	Antony Yusif	Charlottes
29001	Lakshmi Euripides	The Master and Margarita	0143108271	4/2/25	NULL	Antony Yusif	Charlottes
73498	Pia Galchobhar	Freakonomics: A Rogue Economist Explores the Hidden Side of Everything	006073132X	3/2/25	3/25/25	Dominicus Mansoor	Crozet
73498	Pia Galchobhar	Moneyball: The Art of Winning an Unfair Game	0393324818	3/24/25	NULL	Antony Yusif	Charlottes

Together, the *AUTHORSHIP* and *TRANSACTIONS* entities are a 1NF database.

6.2.2.1.2. Functional Dependence

Second and third normal forms are based on the idea of **functionally dependent** attributes in a entity. Understanding this concept is the key to understanding database normalization.

One attribute (Y) is functionally dependent on another (X) if the value of X implies a specific value for Y. Stated another way, each value of X appears alongside one and only one value of Y. That is pretty abstract, so here are some guidelines that help me understand the notion of functional dependence:

- This use of “function” is the exact same as the concept of a function from algebra and pre-calculus. A correspondence $f(x) = y$ is a function if each value of x has only one associated value of y. The reverse is not necessary true, however: one value of y may have multiple values of x that map to it.
- The X attribute will either be a primary key, or something that should be a primary key in another table.

For example, `CardNum` and `Patron` are two attributes in the *TRANSACTIONS* entity. We say that `Patron` (Y) depends on `CardNum` (X) because one library card number has exactly one patron (with one name) associated with it. The reverse is not true: it is possible for two different patrons to have the same name (the same value of `Patron`) but for each to have different library card numbers.

In addition to `CardNum` and `Patron`, the library data also contains the following functionally dependent relationships:

- `Title` is functionally dependent on `ISBN-10` because one book has one title, but the reverse is not true because there are situations in which different books may have the same title.
- `Author` is functionally dependent on `ISBN-10` because one book has one author (or one set of authors), but the reverse is not true because that author (or set of authors) may have more than one book.
- `Return` is functionally dependent on the transaction, defined as the combination of `CardNum`, `ISBN-10`, and `Checkout` because one transaction of a checked-out library book has one return date for the book (though that date might be NULL or “not yet returned”). But the transaction does not depend on the return date because many books are returned each day.
- `Librarian` and `Branch` are both functionally dependent on the transaction (`CardNum`, `ISBN-10`, and `Checkout`) because one librarian processes the transaction at one branch, but the reverse is not true because one librarian processes many transactions and many transactions happen at one branch.
- Finally, `Branch` is functionally dependent on `Librarian` because (in this example) each librarian works at only one branch, but one branch employs many librarians.

6.2.2.1.3. Second Normal Form

Second normal form (2NF) solves more of the problems of Codd identified. When data are organized in second normal form, it helps enforce data consistency by making it harder for new data to be entered that contradicts existing data.

For data to qualify as being 2NF, it must meet the following criteria:

1. The data must meet all the criteria to be 1NF,
2. Wikipedia lists the second crition for 2NF as:

> It does not have any non-prime attribute that is functionally dependent on any proper subset of any candidate key of the relation. A non-prime attribute of a relation is an attribute that is not a part of any candidate key of the relation.

This one is tricky and takes some work to wrap your mind around, so let’s discuss what this second criterion means in practical terms. A **candidate key** is the same thing as a primary key - it is one column or a combination of columns whose values uniquely identify the rows of a data table. A **non-prime attribute** is a column that is not the primary key and is not part of a collection of columns that together construct a primary key. In the *TRANSACTIONS* entity in the library database `CardNum`, `ISBN-10`, and `Checkout` are prime attributes because they comprise the primary key, and `Patron`, `Title`, `Return`, `Librarian`, and `Branch` are non-prime attributes.

The condition for the data to be 2NF says that all the non-prime attributes need to depend on ALL of the attributes within the primary key, and not just SOME or ONE of them. As we saw in the previous section, of the non-prime attributes, `Return`, `Librarian`, and `Branch` depend on the entire key. But `Patron` depends only on `CardNum` and `Title` depends only on `ISBN-10`, so the data are not in 2NF.

There are two ways to reformat data so that it meets the rules of 2NF.

1. In general, data in 1NF will only fail to conform to 2NF if several columns together comprise the primary key. If the primary key is only one column, then it is not possible for any non-prime attribute to depend on only part of the primary key. The simplest way to revise our data so that it is 2NF is to create a new column (maybe called `TransactionID` in this case) that alone serves as the primary key. While this approach satisfies the rules for 2NF, it does not remove the the functional dependencies, it just moves them to relationships between non-prime attributes and that creates more problems for 3NF.
2. A better approach is to create new tables for each subset of the prime attributes that has dependent attributes, and to move those attributes to these new tables. For example, the library data has attributes that depend only on `CardNum` and attributes that only depend on `ISBN-10`. So we can create two new tables, one for patrons and one for books, that contain `CardNum` and `ISBN-10` as primary keys respectively along with all attributes that depend on these keys. The Patrons table is

PATRONS

CardNum	Patron
18473	Ernesto Hanna
29001	Lakshmi Euripides
73498	Pia Galchobhar

Note that for the Patrons table, only three rows are needed for the three patrons in the data, as opposed to the six rows in the Transactions table. We are now storing less data without losing any information. We also now have a natural location for additional information about the patrons, if we choose to collect it. We left `CardNum` in the original Transactions table because this column is being depended on by `Patron`, and it becomes a **foreign key** which is a column in a table which is a primary key in a different table in the database.

BOOKS

ISBN-10	Title
0553380168	A Brief History of Time
0307389731	Love in the Time of Cholera
0060853980	Good Omens
0143108271	The Master and Margarita
006073132X	Freakonomics: A Rogue Economist Explores the Hidden Side of Everything
0393324818	Moneyball: The Art of Winning an Unfair Game

Like with the Patrons table, the Books table gives us a place to store new information about books that is

not relevant to specific transactions. We continue to store the authors in the separate Authorship table to account for the fact that different books have different numbers of authors.

We remove the columns that depended only on `CardNum` or `ISBN-10` from the Transactions table, which is now:

TRANSACTIONS

CardNum	ISBN-10	Checkout	Return	Librarian	Branch
18473	0553380168	8/25/24	9/3/24	José María Chantal	Charlottesville
29001	0307389731	11/23/24	12/14/24	José María Chantal	Charlottesville
29001	0060853980	1/7/25	2/3/25	Antony Yusif	Charlottesville
29001	0143108271	4/2/25	NULL	Antony Yusif	Charlottesville
73498	006073132X	3/2/25	3/25/25	Dominicus Mansoor	Crozet
73498	0393324818	3/24/25	NULL	Antony Yusif	Charlottesville

6.2.2.1.4. Third Normal Form

The purpose of third normal form (3NF) is to eliminate the possibility of accidentally invalidating the data by changing some non-prime attributes without also changing the non-prime attributes they are functionally dependent on.

The criteria for a database to qualify as being in 3NF are:

1. All of the criteria necessary for the database to be in 2NF.
2. "Every non-prime attribute … is non-transitively dependent on every [attribute]"

A simpler and more intuitive way to state the second condition is:

2. No non-prime attribute has a functional dependency on another non-prime attribute.

Functional dependence between non-prime attributes is called **transitive dependence** because there are two ways that a non-prime attribute (X) can be functionally dependent on the primary key (Z). X can be directly dependent on Z, or indirectly dependent through a functional dependence on another non-prime attribute (Y) that directly depends on Z. If an attribute depends on another in this indirect way, that is a transistive dependency. This condition can only violated if *non-prime attributes depend on one another*.

Our library database is not in 3NF because there is a functional dependency among the non-prime attributes in the TRANSACTIONS table. `Branch` depends on `Librarian` because each librarian works at only one branch, so changing the branch must imply that there is a different librarian who handled the transaction.

To convert a database that is in 2NF to 3NF, first find all the non-prime attributes (X) that depend on another non-prime attribute (Y) in the same table. Then remove the X attributes and create a new entity

table for each Y that contains the X that depends on that Y. In this case, X is Librarian and Y is Branch, so we create a new entity for Librarians that contains Branch as a feature:

LIBRARIANS

Librarian	Branch
José María Chantal	Charlottesville
Antony Yusif	Charlottesville
Dominicus Mansoor	Crozet

We also remove Branch from the Transactions table:

TRANSACTIONS

CardNum	ISBN-10	Checkout	Return	Librarian
18473	0553380168	8/25/24	9/3/24	José María Chantal
29001	0307389731	11/23/24	12/14/24	José María Chantal
29001	0060853980	1/7/25	2/3/25	Antony Yusif
29001	0143108271	4/2/25	NULL	Antony Yusif
73498	006073132X	3/2/25	3/25/25	Dominicus Mansoor
73498	0393324818	3/24/25	NULL	Antony Yusif

The rest of the 3NF database is:
$*PATRONS*$

CardNum	Patron
18473	Ernesto Hanna
29001	Lakshmi Euripides
73498	Pia Galchobhar

$$BOOKS$$

ISBN-10	Title
0553380168	A Brief History of Time
0307389731	Love in the Time of Cholera
0060853980	Good Omens
0143108271	The Master and Margarita
006073132X	Freakonomics: A Rogue Economist Explores the Hidden Side of Everything
0393324818	Moneyball: The Art of Winning an Unfair Game

AUTHORSHIP

ISBN-10	Author
0553380168	Stephen Hawking
0307389731	Gabriel Garcia Marquez
0060853980	Neil Gaiman
0060853980	Terry Pratchett
0143108271	Mikhail Bulgakov
006073132X	Steven Levitt
006073132X	Stephen J. Dubner
0393324818	Michael Lewis

It can be difficult to remember the specific criteria that need to be met in order for a database to be in 1NF, 2NF, and 3NF. It helps to remember that for 1NF every data table must include a primary key (even if several columns comprise it), for 2NF every non-prime attribute must depend on the entire primary key, and for 3NF the non-prime attributes must not depend on anything other than the primary key. If it helps you, these three rules are summarized in the following sentence:

> Give me the key (1NF), the whole key (2NF), and nothing but the key (3NF), so help me Codd.

There are many versions of normalization above and beyond 3NF. They are mainly designed to prevent rare anomalies in the data.

6.2.2.2. Entity-Relationship Diagrams

The way that a database is organized with different tables, each with specific columns of particular data

types, in a way that satisfies the normalization rules described above is called the **database schema**.

The best way to communicate the schema of a database is through a visualization called an **entity-relationship (ER) diagram**. ER diagrams show the different tables (entities) that exist in the database, how they relate to one another and are connected to each other through primary and foreign keys, and they illustrate where and how every feature in the data is stored.

There are three types of ER diagram: the **conceptual model**, the **logical model**, and the **physical model**. These models differ only in the amount of information each one includes in the ER diagram. The conceptual model contains the least amount of information, and generally just shows the entities and the connections between them. The logical model includes the information from the conceptual model, adds the attributes within each entity, and denotes the primary and foreign keys in each entity. The physical model includes all of the information contained in the logical model and adds information about how the data will exist on a computer system, such as the data type for each attribute.

The problem with ER diagrams, however, is that there are many competing standards for constructing one and there is a great deal of confused language and contradictory terminology surrounding them. The first standard, now usually called **Chen's notation**, was first described by Peter Pin-Shan Chen in 1976, and uses a flow-chart like system that includes rectangles, ovals, diamonds, and connecting lines to illustrate a database. Today the most commonly used standard, however, is to use **information engineering (IE) notation**, also called **crow's feet notation**. IE notation was first described by Gordon C. Everest in 1976, and represents entities as rectangles that contain lists of the attributes inside the entity, with lines connecting the rectangles to show the relationships between entities. Other standard notations for ER diagrams include Unified Modeling Language (UML) notation, Barker's notation for Oracle systems, Arrow notation, and Integration DEFinition for Information Modeling (IDEF1X) notation. The IE standard is now the predominant standard in industry.

A key concept for ER diagrams is whether two entities can be joined in a way that is **one-to-one**, **many-to-one**, **one-to-many**, or **many-to-many**. These designations refer to the number of rows in one table that match to one row in the other table. To determine the kind of matching, consider two entities X and Y, and complete the following sentences by filling out the segments inside the curly braces:

> One row in {Y} matches {one or many} rows in {X}

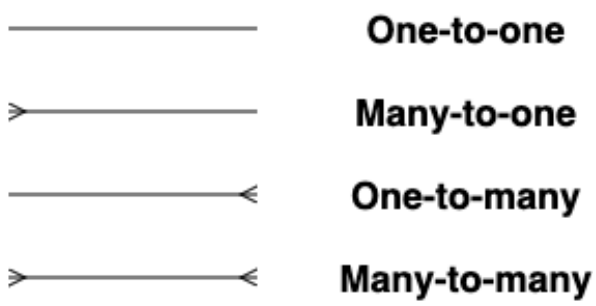
> One row in {X} matches {one or many} rows in {Y}

The relationship between X and Y is then one-to-one, many-to-one, one-to-many, or many-to-many based on whether we choose one or many in each of the two sets of curly braces.

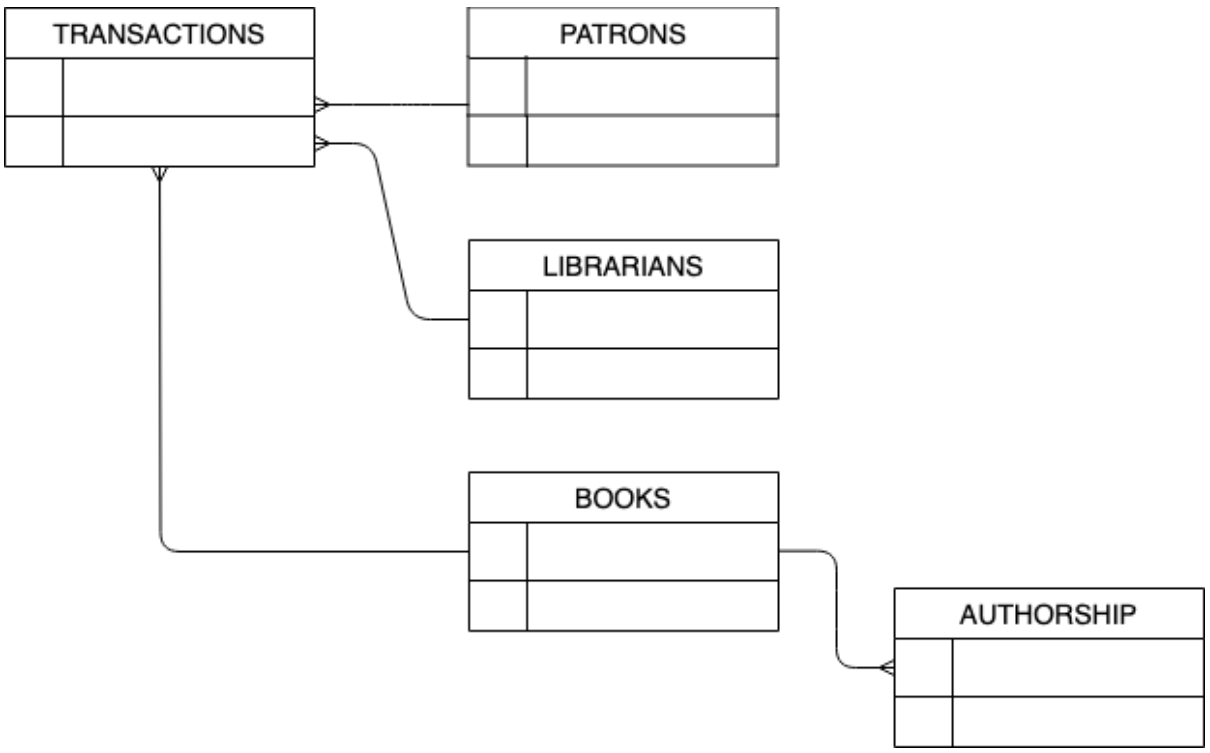
Consider for example the Transactions and Patrons tables in the library database example. One row in Patrons matches MANY rows in Transactions, because one patron may have multiple transactions. Going in the other direction, one row in Transactions matches ONE row in Patrons because each patron has a single row in the Patrons table. So this relationship is a many-to-one relationship.

IE diagrams show one box for every entity in the database along with the name of the entity, and also draws lines between entities that are connected via a shared attribute. There are different types of lines to connects entities depending on whether the entities have a one-to-one, many-to-one, one-to-many, or

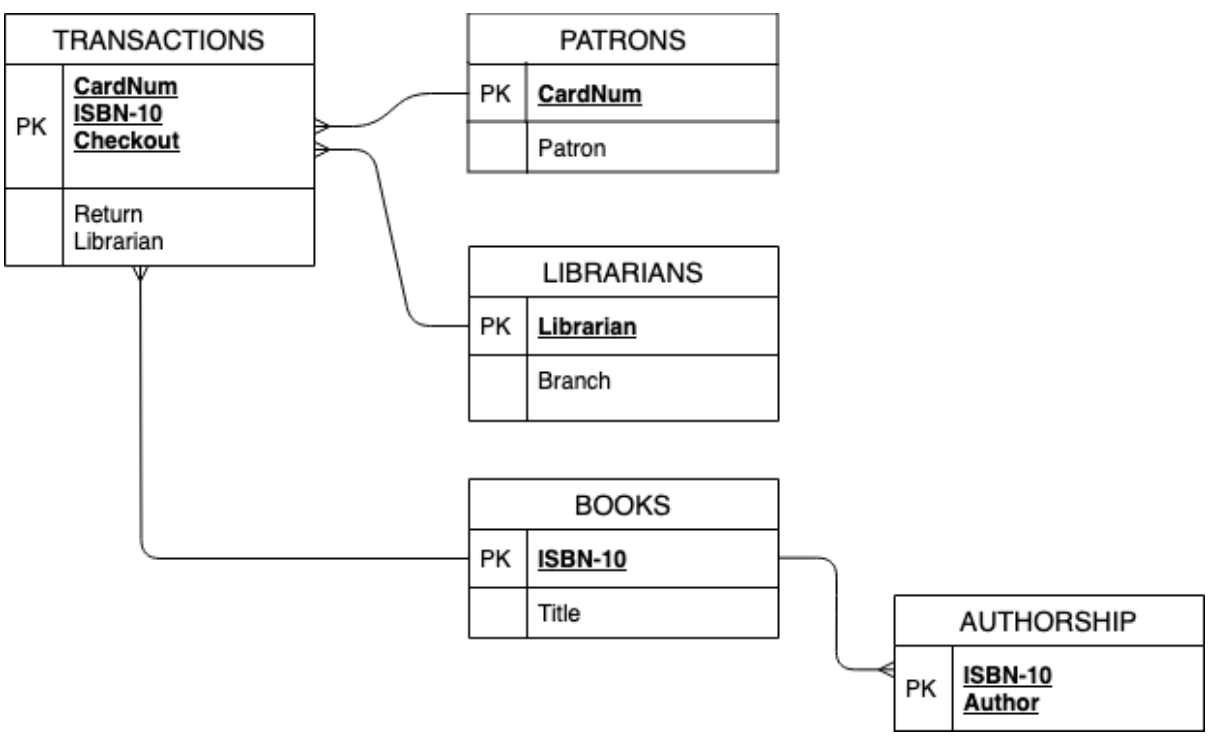
many-to-many relationship. The basic notation for connecting lines is



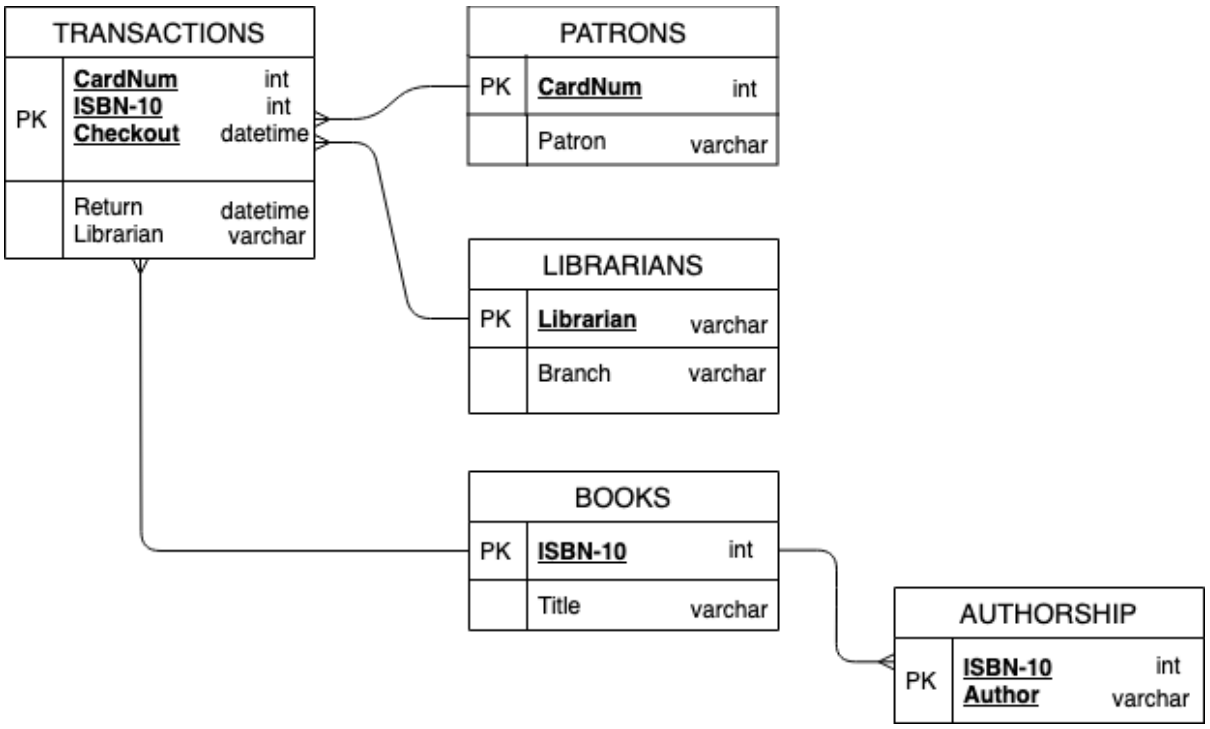
This notation is sometimes called “crow’s feet”, describing the appearance of the parts of these lines that indicate “many.” A conceptual ER diagram in IE notation creates these boxes and connects them, like this:



The logical model in IE notation includes the attributes contained within each entity as a list underneath the title of each box. The first attribute should be the primary key. By default, draw.io places a horizontal divide after the first item in the list to emphasize that this first item is the primary key. To the left, there’s room to write “PK” for further clarification about which attribute is the primary key. (Codes other than PK are allowed here for more complex identification relationships.) Optionally, the lines can be drawn from a primary key directly to the foreign key it is matched to. But in this case, these matches are implied by the fact that the matching attributes have the same name in both tables.



The physical model contains all of the information that exists in the logical model and adds information that is important for the physical storage of the entities. In general, that means that we list the data type next to each attribute, like this:



There are several software packages that create ER diagrams directly from code, often using the database markup language (DBML). One software package that can be used for free is dbdocs.io.

6.2.2.3. Relational Database Management Systems (RDBMS)

Once a database is up and running, it's time to choose software to handle the four CRUD operations: creating, reading, updating, and deleting records in the database. Issuing a query to the database is the same thing as performing a CRUD operation in which a selection of records is read and arranged in a table that is output by the operation. The software to perform CRUD operations on a relational database is called a relational database management system (RDBMS).

If your purpose is to access data stored in an existing database maintained by someone else, an RDBMS has already been chosen to manage how the data can be accessed. All you need to do in that case is learn which RDBMS has been implemented and use an interface for that DBMS to access the data. In general, one database can only work with one DBMS, although some DBMSs like PostgreSQL have built-in foreign wrappers to access data from databases connected to other DBMSs.

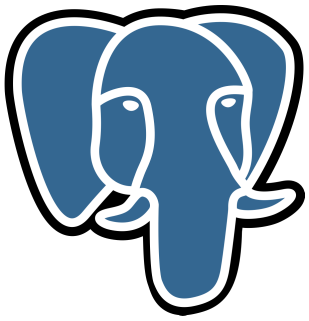
If, however, you are building a new database, the choice of RDBMS is up to you, and this choice can be very confusing. Wikipedia lists several dozen distinct DBMSs for relational databases, and the these RDBMSs mostly overlap in their functionality. For example, all of them implement queries using a coding protocol called the **structured query language** (SQL), which is pronounced either by saying the letters "S-Q-L" or by saying "sequel". SQL is such a universal standard for working with relational databases that the phrases "relational database" and "SQL database" are synonymous, and non-relational databases are commonly called "NoSQL". SQL is a programming langauge that is separate from Python, R, or another environment, but modern RDMBSs make it easy to embed SQL code into standard Python, R, or another coding language's syntax. Although the vast majority of RDBMSs use SQL, many RDBMSs use their own implementations of SQL which can vary in small but important ways from system to system. Learning the basics of SQL code is an important topic in and of itself, and it is the focus of chapter 7.

Despite the fact that RDBMSs are very similar, the question of which RDBMS to use comes up again, and

again, and again. Given all this confusion, how should you make this decision? First and foremost, it's important to understand that any advice on this question you will find online is a matter of opinion: there's no objectively best RDBMS, so before taking any advice from a blog or Stack Overflow post, please consider the biases of the post's author. Some DBMSs are proprietary, for example, and "advice" articles from these sources might be thinly veiled sales pitches. That said, there are important distinctions between RDBMSs.

The biggest distinction is whether the RDBMS is proprietary or open source. This choice is similar to proprietary vs. open source choices in other domains, such as statistical computing, in which privately-owned options like SAS, SPSS, and Stata exist along with open source alternatives like R and Python. Proprietary database management software, like Microsoft SQL Server, Oracle, and IBM DB2 offer a lot of features, including cloud computing and storage. Paid licenses for these options also usually come with dedicated support resources, and that makes sense for large enterprise databases that do not want to depend on volunteers to build the framework of the DBMS and will not consult crowd-sourced forums for troubleshooting. But these options can be expensive. Alternatively, open source RDBMSs are free, and are covered under licenses like the GNU General Public License, which allows users the freedom to run, study, share, and modify the software. It can be argued that open source software cannot be trusted, as it is maintained by volunteers who lack the accountability of paid workers at a large firm like Oracle, Microsoft, or IBM. But there's a compelling argument that open-source software is even more trustworthy than proprietary software because open-source projects generally have a lot more people checking the source code for errors and because software that does not allow users to view the source code can encourage bad, secretive behaviors from the software. As of September 2024, open source RDBMSs and proprietary RDBMSs were about equally popular.

Python and Jupyter Lab are both open source, and many of the most important tools in data science are open source. So I recommend also using open source options for database management. The three most widely-used open source RDBMSs are

MySQL		https://www.mysql.com/
SQLite		https://www.sqlite.org/index.html
PostgreSQL		https://www.postgresql.org/

The following discussion draws heavily on the excellent blog post by "ostezer" and Mark Drake that carefully compares the SQlite, MySQL, and PostgreSQL RDBMSs. For more detailed explanations of the

differences between the three systems, read their post.

MySQL (pronounced “my sequel”), SQLite (pronounced “sequel light”), and PostgreSQL (pronounced “post-gress”) all have strengths and weaknesses relative to each other, and none of them are unambiguously better than the others. The best approach is to understand what is distinctive about each RDBMS and to choose the system that best suits your particular needs. There are several areas in which these three RDBMSs differ:

1. **Local or networked connectivity**: Does the RDBMS work over networked connections, such as cloud storage, or does it only work on databases stored on a local harddrive? If it works over a network, does it depend on a single server? If so, what steps does it take to achieve ACID compliance even when that server goes down for some reason?
2. **Size and speed of the software**: How much space does the RDBMS itself take up in local or remote memory? How quickly will the RDBMS process a query?
3. **Security**: What procedures are in place to limit which users can access the data?
4. **Popularity**: More popular DMBSs will have the bigger communities of developers. Those communities will in general produce better documentation for the software, and will develop more third party extensions for the software.
5. **Functionality and SQL compliance**: Functionality refers to the set of tasks that a particular DBMS is able to run, and some DBMSs have abilities that other DBMSs do not. Functionality is in part determined by how much a DBMS complies with the entire SQL coding standard. SQL is a language that exists outside the control of any particular DBMS and the full set of SQL commands and syntax is maintained by the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO). Although most RDMSs use SQL, not every RDBMS implements everything from standard SQL. Some RDBMSs, however, provide new functions that are outside standard SQL.

The following table describes the strengths and weaknesses of open-source RDBMSs with regard to the considerations listed above:

Consideration	MySQL	SQLite	PostgreSQL
Connectivity	MySQL is fully equipped for networks, and it a popular choice for web-applications. However, it can violate ACID protocols when many users are issuing read and write commands at the same time.	Unlike MySQL and PostgreSQL, SQLite does not run a server process, which means that it does not work over networked connections. It works strictly on a local machine. It's a good choice for organizing data for the use of one person, or to support an application that runs locally, but it is not a viable option for sharing data.	PostgreSQL is fuly capable of networked connections and employs advanced techniques to ensure transactions comply with the ACID standards.
Size and speed	MySQL has a reputation for being the fastest open source RDBMS.	SQLite is designed to be as lightweight as possible, taking up minimal space in memory. But the space available for storing data is limited by the space available on the local harddrive.	PostgreSQL is generally slower than MySQL (though the differences are often slight or negligable), but it efficiently uses multiple CPUs to handle queries faster.
Security	MySQL allows database owners to manage the credentials and privileges of individual database users.	SQLite is designed for one person to access the data on local storage, so there's no mechanism for allowing different users to have different credentials for accessing the data.	PostgreSQL has all of the security features that MySQL has, and exceeds them in some cases, although there is debate about whether PostgreSQL or MySQL is more secure.

Consideration	MySQL	SQLite	PostgreSQL
Popularity as of September 2024	MySQL is the most popular open source RDBMS, and is the second most popular RDBMS overall.	SQLite is the 6th most popular open source RDBMS and 10th most popular overall. But it is a popular addition to application software that is deployed on local systems, such as web browsers.	PostgreSQL is the 2nd most popular open source RDBMS and 4th most popular RDBMS overall. It is also growing in popularity more quickly than MySQL or SQLite. There are many third party extensions, just not as many as are available for MySQL right now. While MySQL and SQLite have proprietary elements that sell extensions to the open source base software, PostgreSQL is fully open source.
Functionality	MySQL does not implement the full set of SQL functions in order to boost the speed of queries. Specifically, it cannot perform full joins. But it does have a big community of developers that create extensions.	SQLite implements most of SQL, generally following the functionality of PostgreSQL. SQLite handles fewer data types than MySQL and PostgreSQL.	PostgreSQL implements most of SQL, more than MySQL does, and is designed to work well with object oriented programming languages. It handles more data types than MySQL, including JSON records.

To summarize, SQLite only works on local systems, although it has many advantages for applications that do not require a network connection: it is lightweight and easy to use and initialize. MySQL and PostgreSQL both work well for allowing users to access the data remotely, and can easily set up credentials for individual users to selectively access parts of the database. MySQL is the most popular open source option, and has been for a while, and is still the fastest option although it is missing some of the functionality that is present in universal SQL. PostgreSQL has functionality that is as close as possible to the complete SQL standard. Also, while MySQL remains open source, its development company is now owned by Oracle, which has been spinning off various proprietary extensions to MySQL. In contrast, PostgreSQL and its extensions all remain open source.

6.2.3. NoSQL Databases

The term “NoSQL” refers to a database that is not organized according to a relational schema, and therefore cannot use any DBMS that uses SQL. A better name for these schema is “non-relational”. NoSQL databases are a recent innovation that have gotten a lot of hype over the last ten years, and it is described in various places as “modern” and “next generation”. Although the language surrounding NoSQL has only existed since the 2000s, the concept of non-relational databases is as old as databases

themselves, as the original navigational schema are be considered versions of NoSQL today.

In addition to the key-value store, document store, wide-column store, and graph versions of NoSQL databases described below, there are specialized NoSQL DBMSs for time series data (influxDB), object oriented data stores (InterSystems Caché), search engines (Elasticsearch), Triplestores, Resource Description Framework (RDF), or XML Stores (MarkLogic), Multivalue Systems (Adabas), Event Stores (Event Store), and Content Stores (Apache Jackrabbit).

6.2.3.1. Comparing NoSQL Databases to Relational Databases

Some people argue that NoSQL is a replacement for relational databases, but the reality is that NoSQL and relational databases are well-suited for different purposes and both are likely to remain in the standard toolkit of professionals who work with data for a long time.

NoSQL differs from relational databases in three ways:

1. Unlike relational databases, a NoSQL database allows for a **flexible schema**.

The term “flexible schema” does not have a specific definition, but it is generally understood to mean a relaxation of the restrictions that relational databases require. For example, when describing DynamoDB, a NoSQL DBMS owned by Amazon, Amazon’s Chief Technology Officer Werner Vogels wrote:

> Amazon DynamoDB is an extremely flexible system that does not force its users into a particular data model … . DynamoDB tables do not have a fixed schema but instead allow each data item to have any number of attributes, including multi-valued attributes.

In other words, when we input a record into a NoSQL database, the attributes in the record, the number of attributes, and the data type of each attribute can change from record to record. We alter the structure of the data to match the specific needs of each record. With NoSQL databases we do not have to follow the rules of any normal form, so multi-valued attributes and non-prime and transitive dependencies are fine.

Relational databases in contrast are very strictly organized. That makes the database easier to use once it is constructed because the ER diagram illustrates where all of the data are stored, and normalization goes a long way towards preventing errors. But in order for a relational database to be worthwhile, there’s a lot of work that has to go in to the design and building of the database, and the data that goes into the database must fit within the tightly controlled schema. For example, the library database described earlier is in 3NF. It’s easy to add new transactions, patrons, books, authors, or librarians to the data. It’s also easy to add new attributes, just so long as they include a key that matches with a primary key in one of the attributes. But it is much more difficult to alter the schema to express the situation in which librarians can work at more than one library branch, or to add items to be checked out that aren’t books, or to handle a patron whose library card has expired and is requesting a new card. All of these transactions are records that the library would want to keep track of, and there is nowhere in the relational database as it is currently organized to store these data. We can reorganize the database, but that would be a lot of work if we already have a great deal of data stored. We would also find ourselves in the same position of being unable to process new data every time the library provides a new class of items or another novel situation occurs that the library would want to keep track of.

An alternative strategy is to store data in a NoSQL database. We can input a record of a patron using a public computer for 30 minutes into a NoSQL database in JSON format:

```
{cardNum=18473,
Patron="Ernesto Hanna",
Service="Public computer use",
Duration=30,
Date="04/24/2520"}
```

In practice, relational databases require a lot of work when building the database, but all that work makes it easier to query and work with data once the database has been built. In contrast, NoSQL databases require much less work at the outset because every record may follow its own schema, but that flexibility results in more work being necessary to extract a selection of data and to prepare it to be analyzed.

2. NoSQL databases are easier than relational databases to store on **distributed systems** by using **sharding**.

A distributed system is network of many separate but connected computers or data servers. Sharding is the act of breaking a database into many distinct parts by storing subsets of the database's records in separate, smaller databases, called shards. Once a database has been sharded, it can be stored on a distributed system: instead of placing the entire database into one massive storage device (increasing the capacity of one storage device is called **vertical scaling**), we break the database into shards and store each shard on a different storage device (increasing the number of storage devices is called **horizontal scaling**).

It is easier to shard a NoSQL database than it is to shard a relational database. A NoSQL database is a container for many records and does not use different data tables, and that's why it is straightforward to move single NoSQL records in their entirety to new storage. In contrast, a relational database has separate data tables for each entity, and the concept of a record is much fuzzier: given one library transaction, for example, data about that transaction comes from separate tables for the patron, the book, the book's authors, and the librarian. It's not possible to operate on a record in its entirety without first joining all of these tables, and even then, there's ambiguity about what counts as a record (is it a transaction? a patron? a book? an author? or all of the above?). The puzzle of how exactly to shard a relational database makes horizontal scaling much more difficult for relational databases. For startups and other organizations that cannot invest in the infrastructure to store massive databases in a single storage device, building a NoSQL database and sharding is an economical option.

3. Unlike relational databases, NoSQL databases do not always follow the [ACID](#acid) standards. Instead, they follow the **BASE** standards.

ACID governs how layers of systems involved with one transaction communicate. One of the ACID standards is consistency, which requires that every system has the exact same data. Consistency takes time, as all of the systems need to be updated before a transaction is considered complete. We wait several moments for credit card transactions to process as the various banks and credit card companies sync their data, for example.

But consistency is sometimes too much of a burden on a data sharing system. Take for example a cloud storage system like Dropbox: when you save a file in the Dropbox folder on your computer, it may take several moments for the file to upload to the Dropbox server and several more moments for the server to sync your data with the other devices that are connected to your Dropbox account. During these

moments, the data in your local folder differ from the data on the server, so this transaction does not satisfy the ACID requirements. While Dropbox is syncing your files, you are still able to use those files on your computer despite the fact that those files may be different from the ones on the server. This process works because a Dropbox user is confident that the files will all sync up *eventually*, once Dropbox processes the changes.

The way Dropbox syncs files does not accord with ACID, but it does meet a different set of standards called called BASE (get it? ACID vs BASE), which are:

- **Basic Availability**: The system should be usable at any given point in time, regardless of how much progress the system has made in syncing data across servers. For Dropbox, that means that a user can still access their files even if they haven't yet been uploaded to the Dropbox server or synced to other devices.
- **Soft-state**: Because the system is constantly updating and syncing data, at any point in time we do not know the true state of the system.
- **Eventual consistency**: The system will, given enough time, sync all data to be the same across all the servers connected to the database.

One advantage of relaxing ACID in favor of BASE is speed. Without the requirement that all data on the system must sync up in real time, individual transactions can proceed more quickly. That's a big advantage for organizations whose operations depend on quick responses. The disadvantage is that we cannot be certain without the assurances of ACID that the data we use are in fact correct. In some cases working with out-of-date data can be illegal or can lead to unacceptable business risks, but in other situations there can be a lot of tolerance for some incorrect data. Whether or not a database needs to follow the ACID protocols depends on what we need the database to do.

NoSQL databases often follow the BASE standards, and not ACID, because BASE makes more sense in a distributed system. When a database is sharded, it might exist across hundreds or thousands of servers, and updating data across all of those servers can take a significant amount of time. Waiting to achieve total data consistency before using the data to do something can take a prohibitive amount of time. The BASE standards are much more forgiving for applications that need to work faster than the speed at which data updates.

6.2.3.2. Key-Value Stores

The three most common types of NoSQL database are **key-value stores** (or key-value databases), **document stores** (or document-oriented databases), and **wide-column stores** (or column-based stores or column-oriented databases). Each format involves three types of data:

1. A key that uniquely identifies each record and is queryable,
2. A document (or array, or dictionary) that contains all of the data for a record,
3. And metadata that describes the relationships between documents.

A key-value store is the simplest way to organize data in a NoSQL database. Every record is stored as a tuple of length 2: the first item is a key, and the second value is a string that contains all of the data for that key. For example, the library data can be stored in key-value format as follows:

Key	Value
1	[18473, Ernesto Hanna, A Brief History of Time, Stephen Hawking, 8/25/24, 9/3/24, José María Chantal, Charlottesville]
2	[29001, Lakshmi Euripides, Love in the Time of Cholera, Gabriel Garcia Marquez, 11/23/24, 12/14/24, José María Chantal, Charlottesville]
3	[29001, Lakshmi Euripides, Anne of Green Gables, L.M. Montgomery, 1/7/25, 2/3/25, Antony Yusif, Charlottesville]
4	[29001, Lakshmi Euripides, The Master and Margarita, Mikhail Bulgakov, 4/2/25, Antony Yusif, Charlottesville]
5	[73498, Pia Galchobhar, Freakonomics: A Rogue Economist Explores the Hidden Side of Everything, Steven Levitt, Stephen J. Dubner, 3/2/25, 3/25/25, Dominicus Mansoor, Crozet]
6	[73498, Pia Galchobhar, Moneyball: The Art of Winning an Unfair Game, Michael Lewis, 3/24/25, NULL, Antony Yusif, Charlottesville]

In the key-value NoSQL structure, the key is **visible** to the DBMS, which means that we can perform searches or queries for particular keys, but the rest of the data are **opaque**, which means that they are only recognized by the DBMS as a single sting, and we cannot query or search for values of these attributes. Because the DBMS is agnostic about the content of the string, it is possible to put any kind of formatting into the Value field: it can organize the data in a list or array, in a JSON dictionary, or in another structure. It can contain column names, or not. The key-value DBMS can only import this string into another environment, like Python, and it is up to code in the import environment to parse the formatting. That can be extremely tricky because each Value field can contain different attributes and data types. In this example the 4th record is lacking a return date, and the 5th record has two authors: when we read the data, we need to use our knowledge of the data to catch and account for these discrepancies. The advantage of a key-value database is the speed with which it processes queries about a specific key, but the disadvantage is that is cannot query data beyond the key, and the data it returns might be in need of a lot of cleaning. As of April 2020, the most popular key-value store DBMS is Redis, which is open source.

6.2.3.3. Document Stores

A document store is similar to a key-value store, with two important differences:

1. The Value field in a document store must be formatted as JSON (or as XML for an XML-oriented database),
2. And unlike a key-value database, it is possible to query data based on the fields within the JSON records.

The first transaction in the library data, for example, is stored in the document store like this:

```
{transactionID=1,
cardNum=18473,
Patron="Ernesto Hanna",
Book=["A Brief History of Time", "Stephen Hawking"],
Checkout="8/25/24",
Return="9/3/24",
Librarian="José María Chantal",
Branch="Charlottesville"}
```

One JSON record in a document store is called, fittingly, a **document**. Documents contain the key and also the entirety of the Value field, so like a key-value store, it is possible to query the keys. But it is also possible to query any of the fields within this JSON. We could, for example, search for records in which the `Patron` field exists and is equal to `"Ernesto Hanna"`.

Document stores have more functionality than key-value databases. In addition to the ability to issue more complex queries, document stores can be converted to **search engine databases** with software that implements a search engine to extract data from the documents. In addition, it is possible to store metadata in a JSON record along with the regular data to organize documents or to create an extra layer of security for particular documents. The disadvantage of a document store relative to a key-value store is that the extra functionality results in bigger DBMSs and slower responses, in general. While document stores allow for flexible schema that cannot fit within a relational framework, queries return lists of JSON-formatted records which need to be parsed by code in an external environment.

The most popular document store as of April 2020 is MongoDB, which is also the most popular DBMS among all NoSQL options, and the 5th most popular DBMS overall.

6.2.3.4. Wide-Column Stores

A wide column store, like a relational database, contains tables. However unlike a relational database, every table has only one row, and each of these tables might have different columns.

Two of the records in the library data look like this in wide-column format:

$$TransactionID = 4$$

CardNum	Patron	BookTitle	Author	Checkout	Librarian	Branch
29001	Lakshmi Euripides	The Master and Margarita	Mikhail Bulgakov	4/2/25	NULL	Antony Yusif

$$TransactionID = 5$$

CardNum	Patron	BookTitle	Author	SecondAuthor	Checkout	Return	Librarian
73498	Pia Galchobhar	Freakonomics: A Rogue Economist Explores the Hidden Side of Everything	Steven Levitt	Stephen J. Dubner	3/2/25	3/25/25	Dominicus Mansoor

Like key-value and document stores, wide-column stores have records with a unique key and a document that contains the data where the schema is flexible. In this case, the table associated with `TransactionID=4` lacks a return date, and the table associated with `TransactionID=5` adds a second author column. Wide-column stores are very similar to document stores in that it is possible to issue a query based on any of the columns. It is also possible to include metadata by clustering columns under shared tags called **supercolumns**. For example, `BookTitle`, `Author`, and `SecondAuthor` may all be clustered within the supercolumn `Book`. These supercolumn families follow exactly the same logic as fields nested within other fields in a JSON tree.

Generally speaking wide-column stores offer the same functionality as document stores, but operate on tables instead of on JSON records. The most popular column-oriented DBMS as of April 2020 is Cassandra.

6.2.3.5. Graph Databases

Graph databases describe an entirely different paradigm of data models. The key-value, document, and wide-column stores work with *independent* records. For these systems to work, no record should contain attributes that point to other records in the data. That way, it's not a problem to treat records separately and store them on different servers. Relational databases on the other hand contain dependencies because the different entities in the data are related to one another. However these relationships exist between columns stored in different tables, not between the records themselves.

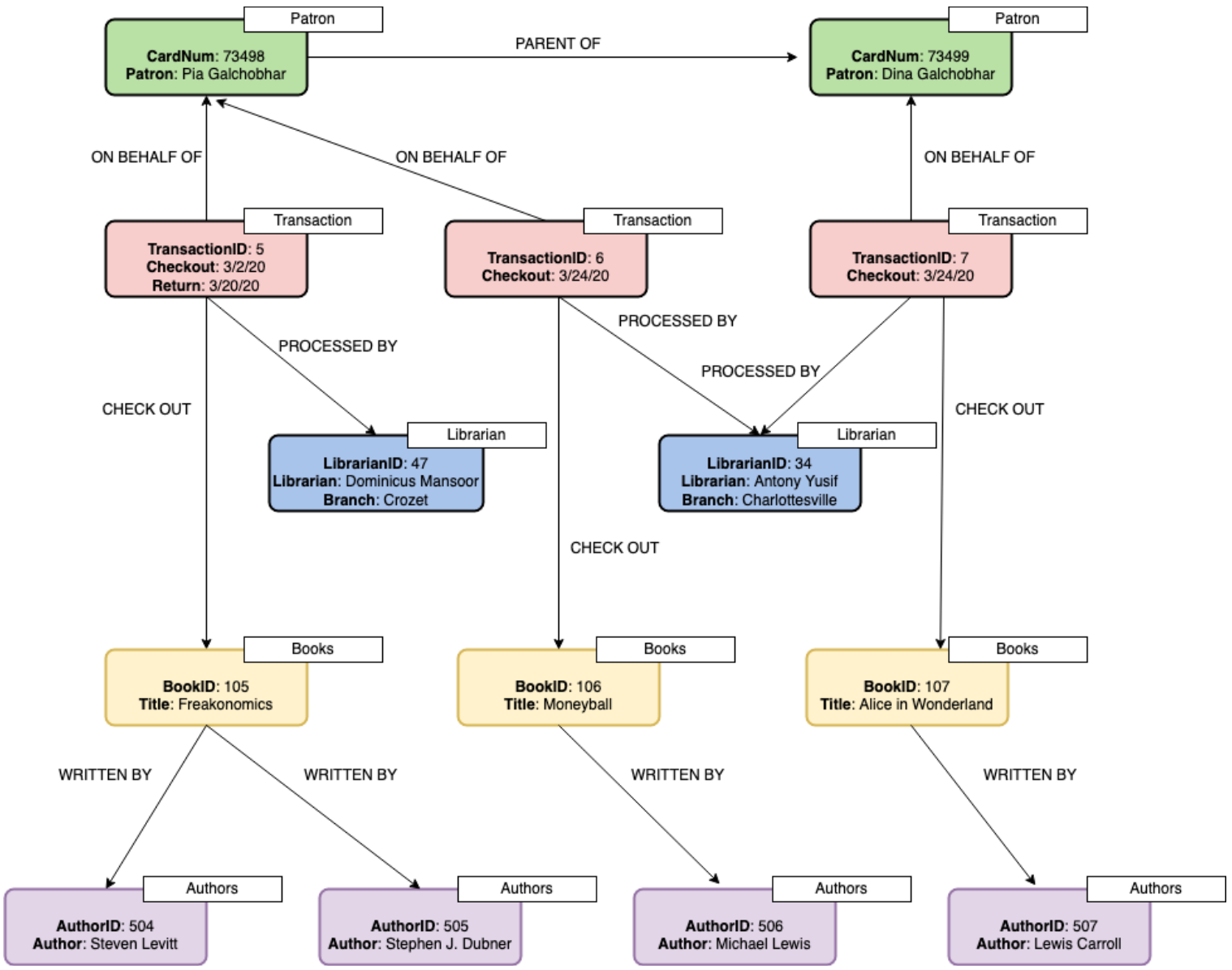
If the records in a database are directly related to each other, a graph database provides a way to store specific data about these relationships. The graph database schema combines a relational and wide-column framework. First, each record is stored in a table with one row, as would be the case in a wide-column store. Next the tables are broken into separate entities, as we would do in a relational schema. That is, attributes that would continue to exist in the same entity in a relational database also exist in the same entity in a graph database.

Every entity for every record in a graph database is called a **node**. Every node has at least one **label**, the name of the entity, attached to it. In the library data, we have nodes for transaction, patron, librarian, book, and author. The attributes within one node are called **properties**. Nodes are connected to other nodes through **relationships**, which are represented by an arrow that is directed from one node to another. The node the arrow starts from is called the **source node**, and the relationship for this node is called an **outgoing relationship**; the node the arrow points to is called the **target node**, and the relationship for this node is called an **incoming relationship**. Every arrow is labeled with text describing the **relationship type**.

It can be confusing to know which node is the source and which is the target and what direction the arrow should point, but it is easier to remember that these symbols should follow natural language. We usually say the target node first and the source node second. For example, in the library data, *transactions are conducted on behalf of patrons*. In this case:

- TRANSACTION is a source node,
- PATRON is a target node,
- and the relationship points from TRANSACTION to PATRON and is labeled “on behalf of”.

Breaking a database into nodes for individual records and entities adds complication. But it does provide the advantage of allowing relationships to exist between records in addition to entities. For example, suppose that the library has a program in which parents can apply for library cards for their children. The library might want to know the parent of a child in their patrons table. Suppose that library patron Pia Galchobhar checks out “Freakonomics: A Rogue Economist Explores the Hidden Side of Everything” by Steven Levitt and Stephen J. Dubner on March 2, 2020 from the Crozet branch, where she is helped by librarian Dominicus Mansoor, and returns it on March 20. Four days later she visits the Charlottesville branch with her daughter Dina where librarian Antony Yusif checks out “Moneyball: The Art of Winning an Unfair Game” by Michael Lewis for her, and “Alice in Wonderland” by Lewis Carroll for Dina. These data have the following graph representation:



The arrow that points from the patron node for Pia Galchobhar to the patron node for Dina tells us that Pia is Dina’s parent. All of the other information, however, is also possible to represent in a relational database with an ER diagram. So unless there is compelling information regarding the relationship between records, a relational database is a more elegant and compact representation of the data. As of April 2020, the most popular graph database is neo4j, which has a great deal of functionality to store graph structured data and

to query based on nodes, relationships, labels, or properties. neo4j is also compliant with the ACID standards.

6.3. Working With Databases in Python

There are three challenges that working with databases in Python entails:

- Installing the database systems, which are external to Python
- Installing and importing the Python packages that we need to connect to these database systems and to issue queries to them
- And setting up a quick way to send commands via SQL or another query language to these systems from Python, getting data back in the form of a pandas dataframe

There are many approaches to solving these challenges. I will describe some methods to solve them quickly that are applicable on Mac, Windows, and Linux systems both locally and on the cloud.

6.3.1. Installing Database Systems Using Docker

Only a few database systems, such as SQLite, operate on files only. SQLite exists as a Python package, and databases are stored as files on the disk. These files work just like any other data file we worked with in the prior chapters.

All other database systems set up a separate server that must run either on your own computer's working memory or on a virtual machine on a cloud system like Amazon Web Services. Those servers usually take the form of separate software that must be installed and run on a machine. Database systems are notoriously difficult to install because the number of configuration parameters and security layers that must be managed is hard to deal with.

The easiest way to install a database system on your machine is to use a **container**. A container runs on a system, such as your computer, and simulates an entirely different computer. Specific kinds of software and configurations can exist inside the container without existing outside the container elsewhere on the system. For example, you can use a container to run Windows on a Mac, or vice versa, or Linux on any system. One major use of containers is to package and distribute software when the software has difficult configuration and storage requirements, as is the case with database systems. The major software for building and managing containers is called Docker, and images (zipped directories that build specific containers) can be stored and shared for free via a website called Docker Hub. There are Docker images available for free that deploy MySQL, PostgreSQL, MongoDB, and many other database systems. These images have set up everything the database needs to function inside the container.

This section will cover the main steps needed to use Docker to run MySQL, PostgreSQL, and Mongo. In practice you will probably not need all three of these systems, in which case you can follow the steps but skip the lines of code that reference a system you are not using. If you need a different database system, look on Docker Hub for the official image for that system, and follow these steps using whatever ENV variables, ports, data directories, and specific configurations the Docker Hub page indicates. For a longer and more detailed discussion of Docker as it relates to a data science workflow, see the Appendix.

Step 1: Install or update the following Python packages:

```
mysql-connector-python
psycopg[binary,pool]
pymongo
sqlalchemy
python-dotenv
```

Please note, there are packages named `mysql-connector` and `psycopg2`, but we specifically need `mysql-connector-python` and `psycopg[binary,pool]`. Also, while we need to install `python-dotenv`, we import this package as `dotenv`.

Step 2: Create an ".env" file to store your passwords, API keys, and other sensitive information. Follow the steps to create a .env file discussed in Chapter 4.

Step 3: Open the `.env` file. Inside the .env file, paste the following

```
POSTGRES_PASSWORD=password1
MONGO_INITDB_ROOT_PASSWORD=password2
MYSQL_ROOT_PASSWORD=password3
mongo_init_db=mongodb
MYSQL_DATABASE=db
MONGO_INITDB_ROOT_USERNAME=mongo
```

On the first three lines, change `password1`, `password2`, and `password3` to anything you like (but please do not use the @ symbol as that causes problems for some packages). Leave the 4th, 5th, and 6th lines alone. Then SAVE the .env file.

Step 4: If you have not yet done so, download and install the Docker desktop client for your operating system. Once the Docker desktop is installed, open it on your local computer.

Step 5: Open a new text file and save it as "compose.yaml". Then paste the following code into it and save the file:

```
services:

  mysql:
    image: mysql:latest
    env_file:
      - .env
    ports:
      - "3306:3306"
    volumes:
      - mysqldata:/var/lib/mysql

  postgres:
    image: postgres:latest
    env_file:
      - .env
    ports:
      - "5432:5432"
    volumes:
      - postgresdata:/var/lib/postgresql/data


  mongo:
    image: mongo:latest
    env_file:
      - .env
    ports:
      - "27017:27017"
    volumes:
      - mongodata:/data/db

volumes:
  mysqldata:
  postgresdata:
  mongodata:
```

Step 6: In the terminal, use `cd` to navigate to the folder where you saved the compose.yaml file, and type

```
docker compose up
```

If successful, you will see a large amount of output from JupyterLab, MySQL, PostgreSQL, and Mongo. If you see a “docker daemon” error, then double-check that the Docker desktop client is running.

Step 7: To test whether MySQL is running properly, run the following code inside your Python script or notebook. If this code runs without error, you are set up to start working with MySQL via Python:

```
import mysql.connector
import dotenv
import os
dotenv.load_dotenv()
MYSQL_ROOT_PASSWORD = os.getenv('MYSQL_ROOT_PASSWORD')
dbserver = mysql.connector.connect(
    user='root',
    password=MYSQL_ROOT_PASSWORD,
    host='localhost',
    port='3306')
```

```
---------------------------------------------------------------------------
MySQLInterfaceError                       Traceback (most recent call last)
File ~/.pyenv/versions/3.12.5/lib/python3.12/site-packages/mysql/connector/connection_ce
    333 try:
--> 334     self._cmysql.connect(**cnx_kwargs)
    335     self._cmysql.converter_str_fallback = self._converter_str_fallback

MySQLInterfaceError: Can't connect to MySQL server on 'localhost:3306' (61)

The above exception was the direct cause of the following exception:

DatabaseError                             Traceback (most recent call last)
Cell In[1], line 6
      4 dotenv.load_dotenv()
      5 MYSQL_ROOT_PASSWORD = os.getenv('MYSQL_ROOT_PASSWORD')
----> 6 dbserver = mysql.connector.connect(
      7     user='root',
      8     password=MYSQL_ROOT_PASSWORD,
      9     host='localhost',
     10     port='3306')

File ~/.pyenv/versions/3.12.5/lib/python3.12/site-packages/mysql/connector/pooling.py:32
    319         raise ImportError(ERROR_NO_CEXT)
    321 if CMySQLConnection and not use_pure:
--> 322     return CMySQLConnection(*args, **kwargs)
    323 return MySQLConnection(*args, **kwargs)

File ~/.pyenv/versions/3.12.5/lib/python3.12/site-packages/mysql/connector/connection_ce
    149 if kwargs:
    150     try:
--> 151         self.connect(**kwargs)
    152     except Exception:
    153         self.close()

File ~/.pyenv/versions/3.12.5/lib/python3.12/site-packages/mysql/connector/abstracts.py:
   1396     self.config(**kwargs)
   1398 self.disconnect()
-> 1399 self._open_connection()
   1401 charset, collation = (
   1402     kwargs.pop("charset", None),
   1403     kwargs.pop("collation", None),
   1404 )
   1405 if charset or collation:

File ~/.pyenv/versions/3.12.5/lib/python3.12/site-packages/mysql/connector/connection_ce
    337         self.converter.str_fallback = self._converter_str_fallback
    338 except MySQLInterfaceError as err:
--> 339     raise get_mysql_exception(
    340         msg=err.msg, errno=err.errno, sqlstate=err.sqlstate
    341     ) from err
    343 self._do_handshake()
    345 if (
    346     not self._ssl_disabled
    347     and hasattr(self._cmysql, "get_ssl_cipher")
   (...)
    352
    353     # `get_ssl_cipher()` returns the name of the cipher being used.

DatabaseError: 2003 (HY000): Can't connect to MySQL server on 'localhost:3306' (61)
```

To test whether PostgreSQL is running properly, run the following code inside your Python script or notebook. If this code runs without error, you are set up to start working with PostgreSQL via Python:

```
import psycopg
import dotenv
import os
dotenv.load_dotenv()
POSTGRES_PASSWORD = os.getenv('POSTGRES_PASSWORD')
dbserver = psycopg.connect(
    user='postgres',
    password=POSTGRES_PASSWORD,
    host='localhost',
    port = '5432'
)
```

To test whether MongoDB is running properly, run the following code inside your Python script or notebook. If this code runs without error, you are set up to start working with MongoDB via Python:

```
import pymongo
import dotenv
import os
dotenv.load_dotenv()
MONGO_INITDB_ROOT_USERNAME = os.getenv('MONGO_INITDB_ROOT_USERNAME')
MONGO_INITDB_ROOT_PASSWORD = os.getenv('MONGO_INITDB_ROOT_PASSWORD')
myclient = pymongo.MongoClient(f"mongodb://{MONGO_INITDB_ROOT_USERNAME}:{MONGO_INITDB_RO
myclient.list_databases()
```

```
<pymongo.command_cursor.CommandCursor at 0x1119e8950>
```

Step 8: You are now set up to work in Python in a way that stores data in these databases and issues queries to extract data from the databases.

Step 9: After you are done working with these databases, return to the terminal where you previously typed `docker compose up`. Press CONTROL + C to stop the container. Then type

```
docker compose down
```

to remove the extra volumes and networks Docker had built and free up that space on your computer.

6.3.2. Example: Wine Reviews

To demonstrate how to work with a database in Python, we will use Kaggle data on over 100,000 wine reviews that appeared in Wine Enthusiast Magazine, which were compiled as a web scraping project by Zack Thoutt:

```
import numpy as np
import pandas as pd
total = pd.read_csv("https://github.com/jkropko/DS-6001/raw/master/localdata/winemag.csv
total
```

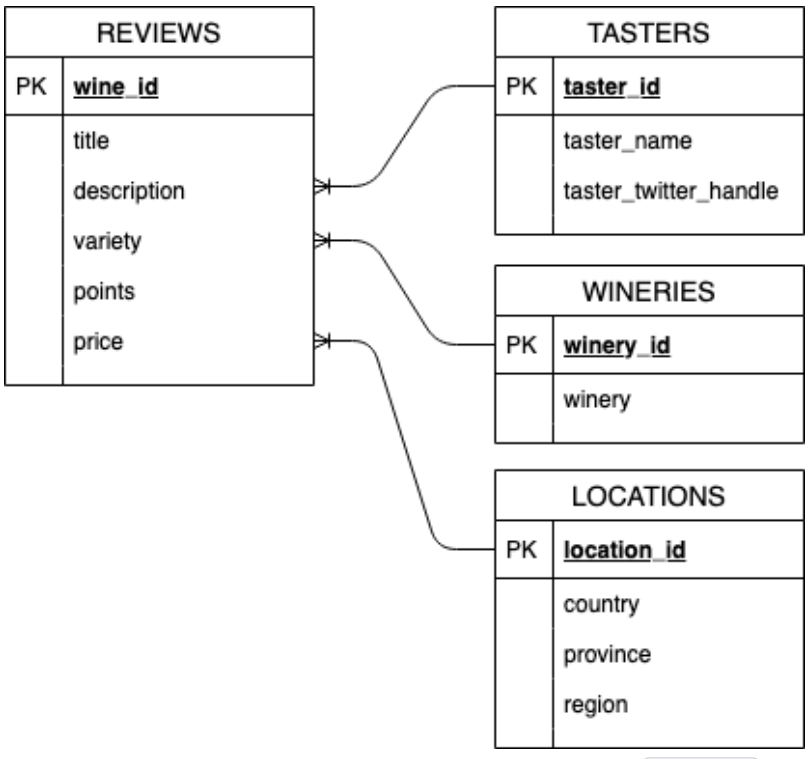
	wine_id	country	description	points	price	province	region	taster_name	t
0	0	Italy	Aromas include tropical fruit, broom, brimston...	87	NaN	Sicily & Sardinia	Etna	Kerin O’Keefe	
1	1	Portugal	This is ripe and fruity, a wine that is smooth...	87	15.0	Douro	NaN	Roger Voss	
2	2	US	Tart and snappy, the flavors of lime flesh and...	87	14.0	Oregon	Willamette Valley	Paul Gregutt	
3	3	US	Pineapple rind, lemon pith and orange blossom ...	87	13.0	Michigan	Lake Michigan Shore	Alexander Peartree	
4	4	US	Much like the regular bottling from 2012, this...	87	65.0	Oregon	Willamette Valley	Paul Gregutt	
...	...	...	...	...	...	...	...	...	
103722	129966	Germany	Notes of honeysuckle and cantaloupe sweeten th...	90	28.0	Mosel	NaN	Anna Lee C. Iijima	
103723	129967	US	Citation is given as much as a decade of bottl...	90	75.0	Oregon	Oregon	Paul Gregutt	
103724	129968	France	Well-drained gravel soil gives this wine its c...	90	30.0	Alsace	Alsace	Roger Voss	
103725	129969	France	A dry style of Pinot Gris, this is crisp with ...	90	32.0	Alsace	Alsace	Roger Voss	
103726	129970	France	Big, rich and off-dry, this is powered by inte...	90	21.0	Alsace	Alsace	Roger Voss	

103727 rows × 12 columns

The features in this dataframe are:

- `wine_id`: a primary key identifying one wine
- `country`: the country of origin for the wine
- `description`: the text of the wine review written by the Wine Enthusiast critic
- `points`: the score given to the wine by the critic on a scale from 0 to 100 (although the lowest rated wine in the data is still rated 80, so even bad wine is pretty good!)
- `price`: the price in U.S. dollars
- `province`: the province or state within the country in which the wine was produced
- `region`: the region within the province or state in which the wine was produced (for example, Central Virginia as opposed to NOVA or the Tidewater)
- `taster_name`: the name of the Wine Enthusiast critic
- `taster_twitter_handle`: the critic's Twitter handle
- `title`: the official name and vintage of the wine
- `variety`: the type of grapes that were used to make the wine
- `winery`: the name of the winery that produced the wine

If we want to store these data in a relational database, we need to reorganize the data into separate tables to fit with the rules of database normalization. I create four tables for four separate entities: REVIEWS for the reviews of individual wines, TASTERS for information about the critics, WINERIES for information about the wineries, and LOCATION for the details on where the wine was produced. All relationships are many-to-one: many wines are reviewed by the same critic, many wines are produced by the same winery, and many wines are produced in the same location, but every wine comes from one winery at one location and is reviewed by one critic. The logical entity-relationship diagram for the database is:



I did the work to generate these tables using `pandas`, and I saved each entity as a separate CSV file:

```
reviews = pd.read_csv("https://github.com/jkropko/DS-6001/raw/master/localdata/wines_rev
tasters = pd.read_csv("https://github.com/jkropko/DS-6001/raw/master/localdata/wines_tas
wineries = pd.read_csv("https://github.com/jkropko/DS-6001/raw/master/localdata/wines_wi
locations = pd.read_csv("https://github.com/jkropko/DS-6001/raw/master/localdata/wines_
```

Once you create a database with a particular DBMS, the easiest workflow is to continue to use that DBMS as implemented in Python, and not to switch DBMSs. SQLite, MySQL, and PostgreSQL all have specific

strengths for working with relational databases and MongoDB is a popular choice for document stores, but there are always other DBMS options. Use the DBMS that you like best, or the DBMS that matches the preferences of other people in your organization.

6.3.3. Using SQLite

SQLite differs from MySQL, PostgreSQL, and Mongo in that it stores an entire database in a single file saved onto a computer's hard storage. The `sqlite3` package for Python combines the local installation of SQLite with the Python functionality for working with SQLite. Unlike other DBMSs, there is no need to separately install and run SQLite via Docker or any other method. We can just use `sqlite3`, which should be pre-installed on your version of Python:

```
import sqlite3
```

As SQLite works with files, it is important to first check that your code is set to the proper default directory for saving and loading files. You can do that with `os.getcwd()`. If this is not the folder you want to use to save your database file, you can change this directory by typing `os.chdir(`*path to folder*`)`.

The `.connect()` method in the `sqlite3` package takes a filename as its argument, and creates this file as a new (empty) database if it doesn't exist. If the file exists, it creates a connection to that database so that we can submit SQL code to query that database. Database files usually have the extension ".db".

```
wine_db = sqlite3.connect("winereviews.db")
```

The database now exists as "winereviews.db" in your local storage and as `wine_db` in the Python environment. But the database is currently empty. To add data frames to serve as entities in the database, use the `.to_sql()` method, which operates on the four `pandas` data frames we created above:

```
reviews.to_sql('reviews', wine_db, index=False, chunksize=1000, if_exists='replace')
tasters.to_sql('tasters', wine_db, index=False, chunksize=1000, if_exists='replace')
wineries.to_sql('wineries', wine_db, index=False, chunksize=1000, if_exists='replace')
locations.to_sql('locations', wine_db, index=False, chunksize=1000, if_exists='replace')
```

```
1488
```

The first argument of `.to_sql()` names the entity, the second argument specifies the database in which these entities should be placed, the third argument switches off the default behavior of uploading the dataframe's row indices to the database as an additional column in each table, `chunksize` breaks the data transfer into smaller parts to avoid overwhelming the database server (that's more of a problem for MySQL and PostgreSQL), and the last argument tells the method to overwrite each entity if a table with the given name already exists in the database. If instead you would like to add additional data to existing tables in the database, you can write `if_exists='append'` here. The `wine_db` database now contains four entities, named `reviews`, `tasters`, `wineries`, and `locations`.

Now that the database exists and is populated with the data, we can begin to issue queries. Queries employ SQL code, which we will discuss in detail in chapter 7. To issue queries, use the

`pd.read_sql_query()` method: all you need to do is pass the SQL query as a string as the first argument and the name of the database as the second argument, and the function returns the data frame we want. Although any string will work for the SQL code, I like using the triple quote syntax to allow for a multiline string. For example, to extract the entire “reviews” table, we can type:

```
myquery = '''
SELECT *
FROM reviews
'''
df = pd.read_sql_query(myquery, wine_db)
df
```

	wine_id	title	variety	description	points	price	taster_id	wine
0	0	Nicosia 2013 Vulkà Bianco (Etna)	White Blend	Aromas include tropical fruit, broom, brimston...	87	NaN	1	
1	1	Quinta dos Avidagos 2011 Avidagos Red (Douro)	Portuguese Red	This is ripe and fruity, a wine that is smooth...	87	15.0	2	
2	2	Rainstorm 2013 Pinot Gris (Willamette Valley)	Pinot Gris	Tart and snappy, the flavors of lime flesh and...	87	14.0	3	
3	3	St. Julian 2013 Reserve Late Harvest Riesling ...	Riesling	Pineapple rind, lemon pith and orange blossom ...	87	13.0	4	
4	4	Sweet Cheeks 2012 Vintner's Reserve Wild Child...	Pinot Noir	Much like the regular bottling from 2012, this...	87	65.0	3	
...	...	...	...	...	...	...	...	
103722	129966	Dr. H. Thanisch (Erben Müller-Burggraef) 2013 ...	Riesling	Notes of honeysuckle and cantaloupe sweeten th...	90	28.0	6	
103723	129967	Citation 2004 Pinot Noir (Oregon)	Pinot Noir	Citation is given as much as a decade of bottl...	90	75.0	3	
103724	129968	Domaine Gresser 2013 Kritt Gewurztraminer (Als...	Gewürztraminer	Well-drained gravel soil gives this wine its c...	90	30.0	2	
103725	129969	Domaine Marcel Deiss 2012 Pinot Gris (Alsace)	Pinot Gris	A dry style of Pinot Gris, this is crisp with ...	90	32.0	2	
103726	129970	Domaine Schoffit 2012 Lieu-dit Harth Cuvée Car...	Gewürztraminer	Big, rich and off-dry, this is powered by inte...	90	21.0	2	

103727 rows × 9 columns

One useful feature of SQlite is that it automatically generates an entity named `sqlite_master` that contains metadata about the other tables in the database. For example, to see a list of all of the entity

names in the database, we run the following query:

```
myquery = '''
SELECT name
FROM sqlite_master
WHERE type='table'
ORDER BY name
'''
pd.read_sql_query(myquery, wine_db)
```

	name
0	locations
1	reviews
2	tasters
3	wineries

As we work with the database, the `.commit()` method makes any changes we've made to the database viewable by any other users of the database, and the `.close()` method prevents any additional changes without first issuing a call to `.connect()` again:

```
wine_db.commit()
wine_db.close()
```

These two methods are mostly important for people who are adding, updating, or deleting data from the database and those who work with collaborators.

6.3.4. Using MySQL

First, follow the instructions in Installing Database Systems Using Docker to ensure that MySQL is running via a container on your system. Then in your script or notebook, import this module:

```
import mysql.connector
```

One important difference between SQLite and MySQL is that MySQL always requires a password to access a database, and SQLite does not. Because SQLite is designed to work only on local storage, the idea of a password is considered redundant because a user would have already entered a password to access their personal operating system. The following examples require a password - but because I do not want to display my password in this notebook, I will use the same method we used in Chapter 4 to load the password chosen in Step 3 in the Installing Database Systems Using Docker section of this chapter:

```
import dotenv
from sqlalchemy import create_engine
dotenv.load_dotenv()
mysqlpassword = os.getenv('MYSQL_ROOT_PASSWORD')
```

There are two ways to connect to MySQL via Python, and both are needed:

- We can connect to the **MySQL server**: the instance of MySQL running on the computer (via a Docker container). Connecting to the server is important for things like creating new databases on this server, adding or removing users who can access these databases, and viewing metadata about these databases.
- Or we can connect to one of the **databases** that currently exists on the MySQL server. Connecting to a database is the proper way of using SQL to pull data from a database.

When we first create a database, we should do so using a connection to the server. But after creating the database, most of the work we do in Python involving that database will be done by connecting directly to the database.

6.3.4.1. Connecting to the MySQL Server and Creating a New Database

We can connect to the local MySQL server by typing

```
dbserver = mysql.connector.connect(
    user='root',
    password=MYSQL_ROOT_PASSWORD,
    host='localhost',
    port='3306')
```

Here we specify “root” as the username because we are acting as the root user, or administrator, of the database server. In other circumstances, if you are not the administrator, you will be assigned a different user name that can be specified under `user`. Because we are running the DBMS ourselves `password` is the `MYSQL_ROOT_PASSWORD` we chose in the .env file when we lauched MySQL via Docker. `host='localhost'` refers to the fact that MySQL is running on our local computer, and not on a remote or cloud system. If the database were running instead on a remote system, we would place that computer’s IP address here. `port` is the local port on this computer that we specified in the `compose.yaml` file.

To work with this server we create a cursor which uses the `.execute()` method to understand SQL code:

```
cursor = dbserver.cursor()
```

Note: do not run the following code to access an existing database. This is for creating a new database from scratch, or for deleting an existing database and starting one from scratch to replace it. To create a new database within your MySQL server, type

```
try:
    cursor.execute("CREATE DATABASE winedb")
except:
    cursor.execute("DROP DATABASE winedb")
    cursor.execute("CREATE DATABASE winedb")
```

where `winedb` is the name of the new (empty) database on your server. If there is already a database named `winedb` on this server, `cursor.execute("CREATE DATABASE winedb")` will yield an error. The `try:` and `except:` syntax is called a **try-catch block**: it executes the code under `try:`, and if there is an error, executes the code under `except:` instead. In this case, if the database already exists, the

`except` block drops the database then creates a new (empty) version of that database.

To see all of the databases that currently exist on the server, type

```
cursor.execute("SHOW DATABASES")
databases = cursor.fetchall()
databases
```

```
[('information_schema',),
 ('mysql',),
 ('performance_schema',),
 ('sys',),
 ('winedb',)]
```

6.3.4.2. Connecting to an Existing Database on MySQL

Before we can place data into the empty wine database, we have to connect to this database. For that we can use the same syntax that we used to connect to the MySQL server, but a simpler approach is to use the `sqlalchemy` package to simplify the process for storing data in and extracting data from a database.

`sqlalchemy` is widely used, and works with many different DBMSs in Python. To see a full list, see this discussion in the sqlalchemy documentation.

The idea is that we can combine all of the information for accessing the database including usernames, passwords, hosts, ports, and database names into a single string called an **engine**. We can then pass this engine to various methods, including many `pandas` methods, to work with exactly the right database. First we import the `create_engine()` method from `sqlalchemy`:

```
from sqlalchemy import create_engine
```

Engines are easier to understand if we use a formatted string, like this:

```
dbms = 'mysql'
package = 'mysqlconnector'
user = 'root'
password = MYSQL_ROOT_PASSWORD
host = 'localhost'
port = '3306'
db = 'winedb'

engine = create_engine(f"{dbms}+{package}://{user}:{password}@{host}:{port}/{db}")
engine
```

```
Engine(mysql+mysqlconnector://root:***@localhost:3306/winedb)
```

This engine allows us to use the `.to_sql()` method from `pandas` to place the four data frames into the wine database. First we set a name for each entity, then we pass the `sqlalchemy` engine, then we specify that the entity should be overwritten with new data if it already exists (we could also specify that the data is appended to the old, or that the operation provides an error if the entity already exists):

```
reviews.to_sql('reviews', con = engine, index=False, chunksize=1000, if_exists = 'replac
tasters.to_sql('tasters', con = engine, index=False, chunksize=1000, if_exists = 'replac
wineries.to_sql('wineries', con = engine, index=False, chunksize=1000, if_exists = 'repl
locations.to_sql('locations', con = engine, index=False, chunksize=1000, if_exists = 're
```

```
1488
```

Now all four tables are entities within the `winedb` database.

To query the database, we can use the `pd.read_sql_query()` function by passing the `sqlalchemy` engine to this function:

```
myquery = '''
SELECT *
FROM reviews
'''
pd.read_sql_query(myquery, con=engine)
```

	wine_id	title	variety	description	points	price	taster_id	wine
0	0	Nicosia 2013 Vulkà Bianco (Etna)	White Blend	Aromas include tropical fruit, broom, brimston...	87	NaN	1	
1	1	Quinta dos Avidagos 2011 Avidagos Red (Douro)	Portuguese Red	This is ripe and fruity, a wine that is smooth...	87	15.0	2	
2	2	Rainstorm 2013 Pinot Gris (Willamette Valley)	Pinot Gris	Tart and snappy, the flavors of lime flesh and...	87	14.0	3	
3	3	St. Julian 2013 Reserve Late Harvest Riesling ...	Riesling	Pineapple rind, lemon pith and orange blossom ...	87	13.0	4	
4	4	Sweet Cheeks 2012 Vintner's Reserve Wild Child...	Pinot Noir	Much like the regular bottling from 2012, this...	87	65.0	3	
...	...	...	...	...	...	...	...	
103722	129966	Dr. H. Thanisch (Erben Müller-Burggraef) 2013 ...	Riesling	Notes of honeysuckle and cantaloupe sweeten th...	90	28.0	6	
103723	129967	Citation 2004 Pinot Noir (Oregon)	Pinot Noir	Citation is given as much as a decade of bottl...	90	75.0	3	
103724	129968	Domaine Gresser 2013 Kritt Gewurztraminer (Als...	Gewürztraminer	Well-drained gravel soil gives this wine its c...	90	30.0	2	
103725	129969	Domaine Marcel Deiss 2012 Pinot Gris (Alsace)	Pinot Gris	A dry style of Pinot Gris, this is crisp with ...	90	32.0	2	
103726	129970	Domaine Schoffit 2012 Lieu-dit Harth Cuvée Car...	Gewürztraminer	Big, rich and off-dry, this is powered by inte...	90	21.0	2	

103727 rows × 9 columns

Before ending our work with this MySQL database, we use the `.commit()` method to save and make any changes we made in the MySQL server accessible to other users:

```
dbserver.commit()
```

And we close the connection to the server to prevent any other applications from interfacing with this server:

```
dbserver.close()
```

6.3.5. Using PostgreSQL

PostgreSQL and MySQL are accessed in Python in very similar ways. The following discussion mirrors the discussion for MySQL above, with only a few small changes as needed.

First, follow the instructions in Installing Database Systems Using Docker to ensure that PostgreSQL is running via a container on your system. Then in your script or notebook, import this module:

```
import psycopg
```

Like MySQL, PostgreSQL always requires a password to access a database, and we specified a password in the .env file in the same directory as the `compose.yaml` file, so that our local installation of the PostgreSQL server via Docker recognizes this password. We first load the password from the .env file:

```
postgres_password = os.getenv('POSTGRES_PASSWORD')
```

As with MySQL, there are two ways to connect to PostgreSQL via Python, and both are needed:

- We can connect to the **PostgreSQL server**: the instance of PostgreSQL running on the computer (via a Docker container). Connecting to the server is important for things like creating new databases on this server, adding or removing users who can access these databases, and viewing metadata about these databases.
- Or we can connect to one of the **databases** that currently exists on the PostgreSQL server. Connecting to a database is the proper way of using SQL to pull data from a database.

When we first create a database, we should do so using a connection to the server. But after creating the database, most of the work we do in Python involving that database will be done by connecting directly to the database.

6.3.5.1. Connecting to the PostgreSQL Server and Creating a New Database

We can connect to the local PostgreSQL server by typing

```
dbserver = psycopg.connect(
    user='postgres',
    password=POSTGRES_PASSWORD,
    host='localhost',
    port = '5432'
)
dbserver.autocommit = True
```

Here we specify “postgres” as the username because we are acting as the superuser, or administrator, of the database server and “postgres” is the default name for the PostgreSQL superuser with `password` set to `POSTGRES_PASSWORD`. `host='localhost'` refers to the fact that PostgreSQL is running on our local computer, and not on a remote or cloud system. If the database were running instead on a remote system, we would place that computer’s IP address here. `port` is the local port on this computer that we specified in the `compose.yaml` file (PostgreSQL runs on port 5432 by default). `dbserver.autocommit = True` instructs PostgreSQL to run every SQL query individually, rather than abiding by the default behavior of taking all SQL commands as part of one transaction and running them all together when the user employs the `.commit()` method.

To work with this server we create a cursor which uses the `.execute()` method to understand SQL code:

```
cursor = dbserver.cursor()
```

Note: do not run the following code to access an existing database. This is for creating a new database from scratch, or for deleting an existing database and starting one from scratch to replace it. To create a new database within the PostgreSQL server, type

```
try:
    cursor.execute("CREATE DATABASE winedb")
except:
    cursor.execute("DROP DATABASE winedb")
    cursor.execute("CREATE DATABASE winedb")
```

where `winedb` is the name of the new (empty) database on your server. If there is already a database named `winedb` on this server, `cursor.execute("CREATE DATABASE winedb")` will yield an error. The `try:` and `except:` syntax (a try-catch block) executes the code under `try:`, and if there is an error, executes the code under `except:` instead. In this case, if the database already exists, the `except` block drops the database then creates a new (empty) version of that database.

To see all of the databases that currently exist on the server, type

```
cursor.execute("SELECT datname FROM pg_database")
databases = cursor.fetchall()
databases
```

```
[('postgres',), ('winedb',), ('template1',), ('template0',)]
```

6.3.5.2. Connecting to an Existing Database on PostgreSQL

Before we can place data into the empty wine database, we have to connect to this database. For that, the

simplest approach is to use the `create_engine()` method from the `sqlalchemy` package to simplify the process for storing data in and extracting data from a database. We combine all of the information for accessing the database including usernames, passwords, hosts, ports, and database names into a single string called an **engine**. We can then pass this engine to various methods, including many `pandas` methods, to work with exactly the right database.

Engines are easier to understand if we use a formatted string, like this:

```
dbms = 'postgresql'
package = 'psycopg'
user = 'postgres'
password = POSTGRES_PASSWORD
host = 'localhost'
port = '5432'
db = 'winedb'

engine = create_engine(f"{dbms}+{package}://{user}:{password}@{host}:{port}/{db}")
engine
```

```
Engine(postgresql+psycopg://postgres:***@localhost:5432/winedb)
```

This engine allows us to use the `.to_sql()` method from `pandas` to place the four data frames into the wine database. First we set a name for each entity, then we pass the `sqlalchemy` engine, then we specify that the entity should be overwritten with new data if it already exists (we could also specify that the data is appended to the old, or that the operation provides an error if the entity already exists):

```
reviews.to_sql('reviews', con = engine, index=False, chunksize=1000, if_exists = 'replac
tasters.to_sql('tasters', con = engine, index=False, chunksize=1000, if_exists = 'replac
wineries.to_sql('wineries', con = engine, index=False, chunksize=1000, if_exists = 'repl
locations.to_sql('locations', con = engine, index=False, chunksize=1000, if_exists = 're
```

```
-2
```

Now all four tables are entities within the `winedb` database.

To query the database, we can use the `pd.read_sql_query()` function by passing the `sqlalchemy` engine to this function:

```
myquery = '''
SELECT *
FROM reviews
'''
pd.read_sql_query(myquery, con=engine)
```

	wine_id	title	variety	description	points	price	taster_id	wine
0	0	Nicosia 2013 Vulkà Bianco (Etna)	White Blend	Aromas include tropical fruit, broom, brimston...	87	NaN	1	
1	1	Quinta dos Avidagos 2011 Avidagos Red (Douro)	Portuguese Red	This is ripe and fruity, a wine that is smooth...	87	15.0	2	
2	2	Rainstorm 2013 Pinot Gris (Willamette Valley)	Pinot Gris	Tart and snappy, the flavors of lime flesh and...	87	14.0	3	
3	3	St. Julian 2013 Reserve Late Harvest Riesling ...	Riesling	Pineapple rind, lemon pith and orange blossom ...	87	13.0	4	
4	4	Sweet Cheeks 2012 Vintner's Reserve Wild Child...	Pinot Noir	Much like the regular bottling from 2012, this...	87	65.0	3	
...	...	...	...	...	...	...	...	...
103722	129966	Dr. H. Thanisch (Erben Müller-Burggraef) 2013 ...	Riesling	Notes of honeysuckle and cantaloupe sweeten th...	90	28.0	6	
103723	129967	Citation 2004 Pinot Noir (Oregon)	Pinot Noir	Citation is given as much as a decade of bottl...	90	75.0	3	
103724	129968	Domaine Gresser 2013 Kritt Gewurztraminer (Als...	Gewürztraminer	Well-drained gravel soil gives this wine its c...	90	30.0	2	
103725	129969	Domaine Marcel Deiss 2012 Pinot Gris (Alsace)	Pinot Gris	A dry style of Pinot Gris, this is crisp with ...	90	32.0	2	
103726	129970	Domaine Schoffit 2012 Lieu-dit Harth Cuvée Car...	Gewürztraminer	Big, rich and off-dry, this is powered by inte...	90	21.0	2	

103727 rows × 9 columns

Finally we close the connection to the server to prevent any other applications from interfacing with this server:

```
dbserver.close()
```

6.3.6. Using MongoDB

MongoDB works with document stores, a type of NoSQL database that stores records as documents, often in JSON format. As with MySQL and PostgreSQL, the first step is to install the external MongoDB software on to your computer via Docker. The version we will use is the Community Edition, which is open source. First, follow the instructions in Installing Database Systems Using Docker to ensure that Mongo is running via a container on your system.

For this example, we'll use a JSON version of the wine database. The following code converts the total wine reviews dataset to JSON format and registers it as JSON formatted data (in this case, a list containing dictionaries for each wine):

```
import json
wine_json_text = total.to_json(orient="records")
wine_json = json.loads(wine_json_text)
```

The first record in the JSON data is

```
wine_json[0]
```

```
{'wine_id': 0,
 'country': 'Italy',
 'description': "Aromas include tropical fruit, broom, brimstone and dried herb. The pal
 'points': 87,
 'price': None,
 'province': 'Sicily & Sardinia',
 'region': 'Etna',
 'taster_name': 'Kerin O'Keefe',
 'taster_twitter_handle': '@kerinokeefe',
 'title': 'Nicosia 2013 Vulkà Bianco  (Etna)',
 'variety': 'White Blend',
 'winery': 'Nicosia'}
```

To work with MongoDB in Python, we use the `pymongo` library:

```
import pymongo
```

To access the MongoDB server, use the `.MongoClient()` method. The address inside the following call to `.MongoClient()` works similarly to a `sqlalchemy` engine by combining the host and port of the server with the username and password for accessing this server:

```
mongo_user = os.getenv('MONGO_INITDB_ROOT_USERNAME')
mongo_password = os.getenv('MONGO_INITDB_ROOT_PASSWORD')
host = 'localhost'
port = 27017
myclient = pymongo.MongoClient(f"mongodb://{mongo_user}:{mongo_password}@{host}:{port}/"
```

Unlike MySQL and PostgreSQL, the same connection represented with `myclient` can be used to

communicate with both the Mongo server and databases on that server in a straightforward way.

To create a database on this server, we pass a string element to `myclient` named after the database we want to create. Here we create a database called “winedb” on the MongoDB server, and we refer to this database as `winedb` in the Python environment. If “winedb” already exists, this code establishes a connection to the existing database without overwriting it:

```
winedb = myclient["winedb"]
```

Another difference between Mongo and relational DBMSs like MySQL and PostgreSQL is that Mongo adds an additional layer of storage. MySQL and PostreSQL have two layers: a server that contains many databases, and databases that contain data. With Mongo, however, while the server still contains many databases, one database can contain many collections of documents. Here we set up a new collection named “winecollection” inside the “winedb” database on the MongoDB server, and we refer to this collection as `winecollection` in the Python environment.

For the purposes of this example, I start with an empty collection. **Note: do not run the following code to access an existing collection. This is for creating a new collection from scratch, or for deleting an existing collection and starting one from scratch to replace it.**

```
collist = winedb.list_collection_names()
if "winecollection" in collist:
  winedb.winecollection.drop()
```

The following code accesses “winecollection” if it exists, or creates an empty collection named “winecollection” if it does not yet exist:

```
winecollection = winedb["winecollection"]
```

It’s time to put some documents into this document store. To put one document into a collection, use the `.insert_one()` method which operates on the Python variable that refers to the collection we are editing. Here we put the first wine in the data into “winecollection”:

```
onewine = winecollection.insert_one(wine_json[0])
```

When we operate on a MongoDB database by creating, reading, updating, or deleting records, and set this operation equal to a Python variable (`onewine` in this case), that variable acts as the cursor for the database.

When we put documents into a MongoDB document store, MongoDB places a unique primary key onto the document. To see the primary key for the record we just placed into the database, call the `.inserted_id` attribute of the cursor:

```
onewine.inserted_id
```

```
ObjectId('673ca6e778ef897a253fb2c2')
```

To query the database, write the query in JSON format, not in SQL format. For example, to find the wine whose title is “Nicosia 2013 Vulkà Bianco (Etna)”, we write:

```
myquery = { 'title': 'Nicosia 2013 Vulkà Bianco  (Etna)'}
```

To execute this query, specify the collection and apply the `.find()` method, passing the query to `.find()`. To see the query results, use a loop across elements of the cursor (`mywine` in this case) and print the elements. Here, because we are issuing a query for the only record in the database, we see one result:

```
mywine = winecollection.find(myquery)
for x in mywine:
    print(x)
```

```
{'_id': ObjectId('673ca6e778ef897a253fb2c2'), 'wine_id': 0, 'country': 'Italy', 'descrip
```

To delete documents, query the documents you want to delete and pass this query to the `.delete_one()` method:

```
myquery = { 'title': 'Nicosia 2013 Vulkà Bianco  (Etna)'}
winecollection.delete_one(myquery)
```

```
DeleteResult({'n': 1, 'ok': 1.0}, acknowledged=True)
```

Now “winecollection” is empty again.

Next, let’s insert all of the records into “winecollection”. In this case, we use the `.insert_many()` method applied to `winecollection` and pass a list of individual JSON formatted records:

```
allwine = winecollection.insert_many(wine_json)
```

To see the number of documents in a collection, use the `.count_documents()` method on the collection. Passing an empty set `{}` counts all documents, and passing a JSON query counts the number of documents that match that query:

```
winecollection.count_documents({})
```

```
103727
```

Of all of these wines, we can use a query to identify all of the reviews of wines from Virginia. We can further narrow that search down to only the Cabernet Sauvignons from Virginia. The query that identifies these documents is:

```
myquery = {'province': 'Virginia',
           'variety': 'Cabernet Sauvignon'}
vawine = winecollection.find(myquery)
winecollection.count_documents(myquery)
```

```
13
```

There are 13 Cabernet Sauvignons from Virginia in the data. To collect the data from these records and output a JSON file with only these wines, we use the `dumps` and `loads` functions from the `bson.json_util` module. Do not type `pip install bson`: the `bson` library is installed along with `pymongo` with matching versions. Installing `bson` separately acquires the old version of `bson`, which can clash with `pymongo`.

`bson.json_util` is similar to the `json` library, but it contains extra functionality to work with the `pymongo` curser to extract the data.

```
from bson.json_util import dumps, loads
```

To assist in issuing queries to the Mongo collection, we can use the following helper function:

```
def mongo_query(collection, row_query={}, col_query={}):
    find = collection.find(row_query, col_query)
    find_dump = dumps(find)
    find_loads = loads(find_dump)
    return find_loads
```

This function passes a query to the Mongo database that uses a dictionary to filter the rows and another dictionary to filter the columns. The result of the query is formatted as a list of dictionaries. We will discuss the Mongo query language in greater detail in the next chapter.

For example, to see all of the wines that are Cabernet Sauvignons from Virginia, we can type:

```
results =mongo_query(winecollection,
            row_query={'province': 'Virginia','variety': 'Cabernet Sauvignon'},
            col_query={})
results
```

```
[{'_id': ObjectId('673ca6e778ef897a253fd8db'),
  'wine_id': 12218,
  'country': 'US',
  'description': 'Cherry jam notes are bright on the nose, showing sawdust and menthol t
  'points': 86,
  'price': 25.0,
  'province': 'Virginia',
  'region': 'Virginia',
  'taster_name': 'Alexander Peartree',
  'taster_twitter_handle': None,
  'title': 'The Boneyard 2012 Cabernet Sauvignon (Virginia)',
  'variety': 'Cabernet Sauvignon',
  'winery': 'The Boneyard'},
 {'_id': ObjectId('673ca6e778ef897a253fe4b4'),
  'wine_id': 16002,
  'country': 'US',
  'description': 'This Cabernet Sauvignon smells ripe, with cherry cola, red currant and
  'points': 86,
  'price': 50.0,
  'province': 'Virginia',
  'region': 'Virginia',
  'taster_name': 'Carrie Dykes',
  'taster_twitter_handle': None,
  'title': 'Gray Ghost 2014 Reserve Cabernet Sauvignon (Virginia)',
  'variety': 'Cabernet Sauvignon',
  'winery': 'Gray Ghost'},
 {'_id': ObjectId('673ca6e778ef897a253fe4b5'),
  'wine_id': 16003,
  'country': 'US',
  'description': 'This wine has upfront aromas of black cherries, dried herbs and baking
  'points': 86,
  'price': 28.0,
  'province': 'Virginia',
  'region': 'Virginia',
  'taster_name': 'Carrie Dykes',
  'taster_twitter_handle': None,
  'title': 'Gray Ghost 2015 Unfiltered Cabernet Sauvignon (Virginia)',
  'variety': 'Cabernet Sauvignon',
  'winery': 'Gray Ghost'},
 {'_id': ObjectId('673ca6e778ef897a253ffb8a'),
  'wine_id': 23457,
  'country': 'US',
  'description': "Currant, cola, spice and cherry lead this layered Cab from Delfosse. I
  'points': 86,
  'price': 24.0,
  'province': 'Virginia',
  'region': 'Monticello',
  'taster_name': 'Susan Kostrzewa',
  'taster_twitter_handle': '@suskostrzewa',
  'title': 'Delfosse 2007 Cabernet Sauvignon (Monticello)',
  'variety': 'Cabernet Sauvignon',
  'winery': 'Delfosse'},
 {'_id': ObjectId('673ca6e778ef897a25400611'),
  'wine_id': 26887,
  'country': 'US',
  'description': "The nose on this wine is quite interesting, with a ripe cranberry scer
  'points': 81,
  'price': 43.0,
  'province': 'Virginia',
  'region': 'Virginia',
  'taster_name': 'Carrie Dykes',
  'taster_twitter_handle': None,
  'title': 'Winery at La Grange 2012 Cabernet Sauvignon (Virginia)',
  'variety': 'Cabernet Sauvignon',
  'winery': 'Winery at La Grange'},
 {'_id': ObjectId('673ca6e778ef897a25403530'),
  'wine_id': 41932,
  'country': 'US',
  'description': 'An oaky aroma of buttered toast is elevated over ripe cherry and crush
  'points': 87,
```

```
 'price': 26.0,
 'province': 'Virginia',
 'region': 'Virginia',
 'taster_name': 'Alexander Peartree',
 'taster_twitter_handle': None,
 'title': 'Annefield Vineyards 2012 Cabernet Sauvignon (Virginia)',
 'variety': 'Cabernet Sauvignon',
 'winery': 'Annefield Vineyards'},
{'_id': ObjectId('673ca6e778ef897a25403909'),
 'wine_id': 43247,
 'country': 'US',
 'description': 'Lush black plum, bramble and black pepper mingle with cedar and spicy
 'points': 87,
 'price': 34.0,
 'province': 'Virginia',
 'region': 'Middleburg',
 'taster_name': 'Carrie Dykes',
 'taster_twitter_handle': None,
 'title': 'Casanel 2014 Estate Cabernet Sauvignon (Middleburg)',
 'variety': 'Cabernet Sauvignon',
 'winery': 'Casanel'},
{'_id': ObjectId('673ca6e878ef897a254076c4'),
 'wine_id': 63335,
 'country': 'US',
 'description': 'Cherry candies, grapefruit and watermelon grace the nose. The light-bo
 'points': 84,
 'price': 19.0,
 'province': 'Virginia',
 'region': 'Shenandoah Valley',
 'taster_name': 'Alexander Peartree',
 'taster_twitter_handle': None,
 'title': 'Shenandoah Vineyards 2014 Rosé of Cabernet Sauvignon (Shenandoah Valley)',
 'variety': 'Cabernet Sauvignon',
 'winery': 'Shenandoah Vineyards'},
{'_id': ObjectId('673ca6e878ef897a254081d2'),
 'wine_id': 66833,
 'country': 'US',
 'description': 'Lush blackberry, vanilla and baking spices entice the olfactory. These
 'points': 87,
 'price': 28.0,
 'province': 'Virginia',
 'region': 'Middleburg',
 'taster_name': 'Carrie Dykes',
 'taster_twitter_handle': None,
 'title': 'Casanel 2013 Cabernet Sauvignon (Middleburg)',
 'variety': 'Cabernet Sauvignon',
 'winery': 'Casanel'},
{'_id': ObjectId('673ca6e878ef897a2540ae15'),
 'wine_id': 80812,
 'country': 'US',
 'description': 'Ripe and rich on the nose with a harmonious black fruit and sweet vani
 'points': 86,
 'price': 19.0,
 'province': 'Virginia',
 'region': 'Virginia',
 'taster_name': 'Anna Lee C. Iijima',
 'taster_twitter_handle': None,
 'title': 'Veramar 2008 Estate Club Cabernet Sauvignon (Virginia)',
 'variety': 'Cabernet Sauvignon',
 'winery': 'Veramar'},
{'_id': ObjectId('673ca6e878ef897a2540c9a3'),
 'wine_id': 89662,
 'country': 'US',
 'description': 'This wine has intense aromas of blueberry pie and vanilla, along with
 'points': 84,
 'price': 27.0,
 'province': 'Virginia',
 'region': 'Virginia',
 'taster_name': 'Carrie Dykes',
 'taster_twitter_handle': None,
 'title': 'Narmada 2014 Cabernet Sauvignon (Virginia)',
```

```
 'variety': 'Cabernet Sauvignon',
 'winery': 'Narmada'},
{'_id': ObjectId('673ca6e878ef897a25411844'),
 'wine_id': 114775,
 'country': 'US',
 'description': 'With notes of green pepper, clove and ripe plum, this wine is super he
 'points': 85,
 'price': 34.0,
 'province': 'Virginia',
 'region': 'Virginia',
 'taster_name': 'Carrie Dykes',
 'taster_twitter_handle': None,
 'title': 'James River 2010 Cabernet Sauvignon (Virginia)',
 'variety': 'Cabernet Sauvignon',
 'winery': 'James River'},
{'_id': ObjectId('673ca6e878ef897a25413abb'),
 'wine_id': 125676,
 'country': 'US',
 'description': 'Cherry jam notes are bright on the nose, showing sawdust and menthol t
 'points': 86,
 'price': 25.0,
 'province': 'Virginia',
 'region': 'Virginia',
 'taster_name': 'Alexander Peartree',
 'taster_twitter_handle': None,
 'title': 'The Boneyard 2012 Cabernet Sauvignon (Virginia)',
 'variety': 'Cabernet Sauvignon',
 'winery': 'The Boneyard'}]
```


A list of dictionaries can be converted to a pandas dataframe with the `pd.DataFrame.from_records()` method:

```
pd.DataFrame.from_records(results)
```

	_id	wine_id	country	description	points	price	province	
0	673ca6e778ef897a253fd8db	12218	US	Cherry jam notes are bright on the nose, showi...	86	25.0	Virginia	
1	673ca6e778ef897a253fe4b4	16002	US	This Cabernet Sauvignon smells ripe, with cher...	86	50.0	Virginia	
2	673ca6e778ef897a253fe4b5	16003	US	This wine has upfront aromas of black cherries...	86	28.0	Virginia	
3	673ca6e778ef897a253ffb8a	23457	US	Currant, cola, spice and cherry lead this laye...	86	24.0	Virginia	M
4	673ca6e778ef897a25400611	26887	US	The nose on this wine is quite interesting, wi...	81	43.0	Virginia	
5	673ca6e778ef897a25403530	41932	US	An oaky aroma of buttered toast is elevated ov...	87	26.0	Virginia	
6	673ca6e778ef897a25403909	43247	US	Lush black plum, bramble and black pepper ming...	87	34.0	Virginia	Mi
7	673ca6e878ef897a254076c4	63335	US	Cherry candies, grapefruit and watermelon grac...	84	19.0	Virginia	She
8	673ca6e878ef897a254081d2	66833	US	Lush blackberry, vanilla and baking spices ent...	87	28.0	Virginia	Mi
9	673ca6e878ef897a2540ae15	80812	US	Ripe and rich on the nose with a harmonious	86	19.0	Virginia	

	_id	wine_id	country	description	points	price	province
				bl...			
10	673ca6e878ef897a2540c9a3	89662	US	This wine has intense aromas of blueberry pie ...	84	27.0	Virginia
11	673ca6e878ef897a25411844	114775	US	With notes of green pepper, clove and ripe plu...	85	34.0	Virginia
12	673ca6e878ef897a25413abb	125676	US	Cherry jam notes are bright on the nose, showi...	86	25.0	Virginia

`pymongo` does not require that we commit the changes, but we do end our session by closing the connection to the MongoDB server:

```
myclient.close()
```