Lesson 1

Exploring Relational Databases and SQL

SQL is used to access data. Before jumping into SQL and how it is used, it is important to step back and consider how information that you will access has been stored. In this chapter, you'll dive into the topic of data and databases to set the foundation for then accessing the information. You will get a high-level look at databases in general and at relational databases specifically.

Learning Objectives

By the end of this lesson, you will be able to:

- Describe what a relational database is, how it works, and how it differs from a database management system (DBMS)
- Define database tables, relations, columns, attributes, rows, records, tuples, and data types
- Identify the ACID properties
- Know about entity integrity and uniqueness using keys
- Discuss database backup strategies

SAVING DATA

To be useful, software systems must *remember*. If your character started at the beginning (level 0) every time you fired up a video game, or your online banking app reset your balance to $0 when you logged off, or your phone forgot your contacts when it rebooted, you wouldn't use them. To remember, applications must save data in a way that allows ready access to that data when needed.

There are a few options for saving data.

- Write text or bytes directly into a file
- Store data in a relational database
- Store data in a nonrelational database

The first option of writing directly into a file can be cumbersome. In this case, the file typically is expected to be local (on the same computer as the program that is accessing the file). This means there is a high risk of losing data if something happens to that computer.

In a software environment, databases (relational or not) are preferable to files because they can store data separately from the application itself, often on a completely separate server. While this might slow down access to the data to a small degree, the fact that they are separate means that multiple applications can access the same database and that changes to the data in the database are immediately accessible to any application that uses that data.

Nonrelational databases are becoming more common today, but relational databases are standard across many industries as a way of storing data in a predictable and reliable way that allows applications to easily retrieve that data as needed.

WHAT IS A DATABASE?

In real life, most people work with databases every day, often without realizing it. Most computer applications depend on some kind of data access to work correctly. Any advanced system with a goal of identifying objects and performing specific actions on those objects (such as an employee list, an inventory, or a course roster) depends on a database. A **database** is a structured representation of data that can be read from and written to, and a database is often stored separately from any application that uses the data.

Database Uses

Modern computer applications rely heavily on databases, even when the program in question isn't designed to help users manage data. Computer games rely on databases to keep track of characters, character attributes, items that each character can use during gameplay, and even locations within the game. A learning management system (LMS) uses databases to keep track of learners, instructors, content, grades, attendance, and communication between users.

As a concrete example, consider a modern smartphone. The phone itself has a database that stores connection information, OS version, model number, serial number, and similar data about the device itself.

A smartphone also has a variety of applications on it, many of which have their own databases. Common examples include a contact list, a calendar, email apps, photo galleries, social networking apps, and shopping apps. While these apps store data for internal use, the user can grant permission for some apps to access data stored in other apps. For example, a calendar app may be connected to the contact app's database so that the user can easily add appointments with specific people to their calendar, while Facebook can access photos stored on the phone so that the user can share the photos with friends.

In none of these cases, though, does the user have direct access to the database itself. Instead, the application's front end (the part the user interacts with) includes tools that allow the user to create and retrieve data, update existing data, and even delete data that the user no longer needs. The software developer must incorporate the database into the application in a way that allows the application to access and manage the data.

Data vs. Information

When talking about databases, data and information are invariably mentioned. The terms *data* and *information* are often used interchangeably in casual speech, but from a software perspective, there is a very clear difference between the two. Specifically:

- The term **data** refers to individual, raw facts. In many cases, individual pieces of data are meaningless on their own.
- When we process data, the result is **information**. Unlike the raw data, information is useful and normally corresponds to the end user's specific needs.

As an example, consider a piece of data like *smith*. On its own, this is meaningless. While you might first think it is someone's last name, there isn't enough information here to tell you exactly whose name it is. It could also be an occupation rather than a name.

In a specific context, however, this piece of data can be combined with other data to give you useful information. As part of a course roster, for example, it could be combined with a first name to reference a specific student. In a job application or online profile, it could reference the person's work experience, with a completely different name.

Structured vs. Unstructured

A database can contain data that is **structured** or **unstructured**. Modern database software programs hosting databases can usually handle both structured and unstructured data, but it is still good to understand the difference.

In a database with structured data, which we will call a *structured database*, the data is organized in a specific pattern. This makes it easy to control what data is available and where to find specific pieces of data. In a structured database, the developer can limit what kinds of data are stored in the database to improve **data integrity** and reduce the amount of redundant data. This comes as a trade-off in that creating new data and accessing stored data are relatively slow compared to creating and accessing data in an unstructured database. Structured databases are best for datasets that contain predictable types of data, such as bank accounts, personnel records, and inventories.

NOTE *Data integrity* refers to the reliability and accuracy of the data. A *dataset* is a collection of related information that is composed of separate elements but can be manipulated as a group.

A database with unstructured data typically does contain some amount of structure, but without the strict controls inherent to a structured database. Unstructured databases are typically a little faster than structured databases, but they are also prone to duplicate or redundant data. Unstructured databases are often found in applications that have unpredictable or irregular kinds of data, such as social media posts, online product reviews, and similar user-generated content.

NOTE In this lesson, we will look only at structured databases, specifically **relational databases**.

Database vs. DBMS

As mentioned earlier, a database is a representation of data that can be read from and written to and is often stored separately from any application that uses the data. The fact that the database is separate from an application means that it can be made available to multiple applications, such as allowing access to a contact database from a calendar app or posting photos from an image database using a social media app. While any application that uses a specific database must know how to access the data, the data itself is simply a pool that any authorized app can pull from.

A **database management system (DBMS)** is a software system that manages databases. The DBMS executes commands, provides security, enables network access, and provides admin tools for database administrators (DBAs) to work with database files.

A subset of DBMSs includes relational database management systems (RDBMSs) that are designed specifically to work with relational databases. There are many options for RDBMSs, including MySQL, PostgreSQL, Microsoft SQL Server, Oracle Database, and DB2. The choice of DBMS determines some factors of how the data itself is organized, but in large part, all RDBMSs do the same thing.

While a specific RDBMS will be used in this book, keep in mind that all RDBMSs do essentially the same things in the same way. If you understand how one RDMBS works, you can easily transfer that knowledge to a different RDBMS.

RELATIONAL DATABASE CONCEPTS

Relational databases are highly structured in that they organize data into one or more **tables** or **relations**, where each table represents a logical group of data. In day-to-day professional work, we usually say *table*. *Relation* is the formal, academic term, which you may run into if you read about databases in other contexts. Relation is also the basis for the term **relational database**. At a more abstract level, the term **entity** is also used to refer to a table, especially during the design phase of a database and before the database is built on a server.

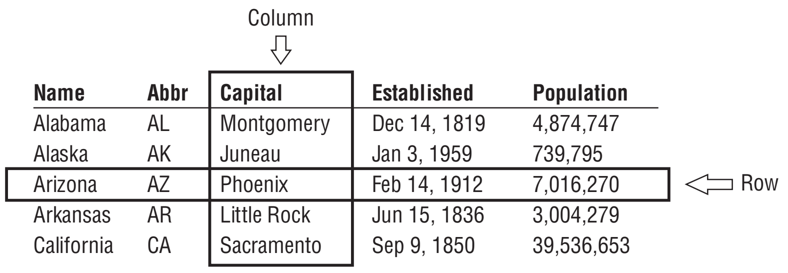
NOTE The relational database model was first proposed by Edgar F. Codd in 1970, with the main goal of reducing duplicate data in a database and thereby making it easier to retrieve and manage specific data.

A table can be imagined as a two-dimensional grid of cells.

- A **row** in a table is a horizontal group of cells, one cell high. It holds facts about one discrete thing represented by the table. That thing could be anything such as a person, a credit card transaction, or a professional sports mascot.
- A **column** in the table is a vertical strip of cells, one cell wide. Every cell in the column holds the same type of data, but each cell is a fact about a different thing. For example, if the table includes data about people, then the table could include separate columns for name, phone number, and address.
- A **cell** represents the intersection of a row and a column. Each cell contains a single piece of data.

The following is a concrete example:

Name	Abbr	Capital	Established	Population
Alabama	AL	Montgomery	Dec 14, 1819	4,874,747
Alaska	AK	Juneau	Jan 3, 1959	739,795
Arizona	AZ	Phoenix	Feb 14, 1912	7,016,270
Arkansas	AR	Little Rock	Jun 15, 1836	3,004,279
California	CA	Sacramento	Sep 9, 1850	39,536,653



Name	Abbr	Capital	Established	Population
Alabama	AL	Montgomery	Dec 14, 1819	4,874,747
Alaska	AK	Juneau	Jan 3, 1959	739,795
Arizona	AZ	Phoenix	Feb 14, 1912	7,016,270
Arkansas	AR	Little Rock	Jun 15, 1836	3,004,279
California	CA	Sacramento	Sep 9, 1850	39,536,653

In the data in this table, each **row** represents one state. Rows are also known as **records** or **tuples**. The term **record** is common, while **tuple** is an academic term. A **database**

record is a single row in a table in the database, and each row in a table is considered a separate object from the other rows in the same table.

Each **column** represents a fact about a state: its name, abbreviation, capital, date established, and current population. There's a subtle nuance here. The term *column* refers to a strip of vertical cells, but it also refers to a definition: an overall name (*Abbr, Capital, Population*) and restrictions on the size, shape, and type of data allowed as values in the column. In fact, when a developer says *column*, they're usually talking about the definition, rather than the cells themselves. To reduce confusion, we may call the value of a record's column a **field**. Academically, column definitions are also called **attributes**.

NOTE The size, shape, and type of data allowed in a column will be discussed in more detail in Part 2, "Applying SQL." As an example, a Population field might be defined as one that can hold a whole number (the type) that is between one and four billion (the size and shape).

ACID COMPLIANCE

Relational databases provide rich and powerful ways to model our data, and it doesn't stop there. A relational database's data structures and algorithms also provide behavior guarantees. They can't guarantee an action will always work, but they can guarantee the state of a database after an action succeeds or fails. They also guarantee predictable behavior when multiple users are interacting with a database. There are many types of guarantees.

The ACID properties are four of the most important guarantees. ACID is an acronym for the following:

- Atomicity
- Consistency
- Isolation
- Durability

Before jumping into ACID, you need to understand database transactions. A relational database allows the following actions:

- Read existing data
- Insert new data
- Update existing data
- Delete existing data
- Add or alter schema (tables and relationships)

A **transaction** is a set of one or more actions that represents a single, logical unit of work. Say you want to reserve three rooms at a hotel, and those room reservations are stored in at least three rows in a database. In most circumstances, you don't want to book *any* rooms if one or more room reservations fail—it's all or nothing. That makes your three-room reservation a transaction. It's a single unit of work that should succeed or fail as a unit.

If you purchase a concert ticket for an in-demand concert, the software system must first find an available ticket and then put it on hold until you can provide payment. That's a transaction. If it wasn't, the system might find an available ticket only to have another person purchase it before you. Without a transaction, the ticket could be purchased twice, or the system might waste time presenting tickets that are no longer for sale. Imagine selling tickets to a show that's predicted to sell out in 10 minutes without software that can handle transactions.

ACID Properties

As mentioned, ACID is an acronym for atomicity, consistency, isolation, and durability. The ACID properties are not required. You can run a database without them; however, for some situations, running without ACID is risky. In situations involving things, such as banking, medical records, and real-time decision-making, bad things can and will happen to your application if you ignore ACID.

Additionally, you never know what will happen when using a software. The network can fail, the operating system can crash, or another user can alter data that you're using. Given enough time, something *will* fail.

ACID-compliant databases are designed to withstand unexpected failures without corrupting your data. Let's look at each of the elements of ACID.

Atomicity

A transaction is atomic if it follows the "all-or-nothing" rule. If one action in the transaction fails, then the entire transaction fails. An atomic transaction never partially succeeds.

Imagine a scenario where you write a new row of data to a table with 10 columns. On the eighth column write, a power failure occurs, and your server immediately shuts down. If the database supports atomicity, it will notice the unfinished transaction and restore the data to its pre-transaction state when it comes back online.

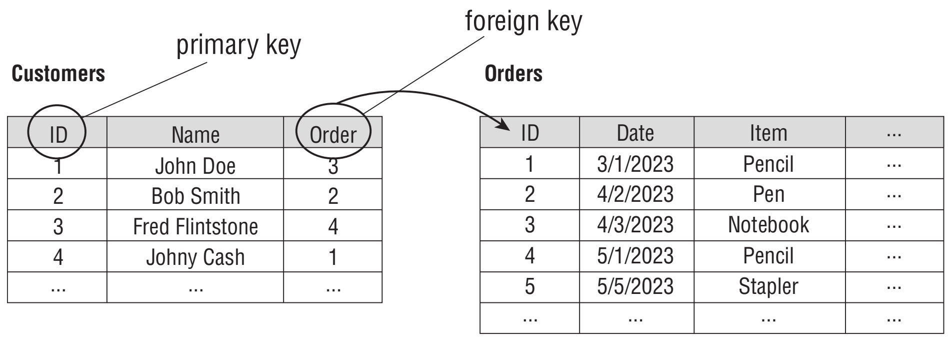
Consistency

A transaction is consistent if it can move the database from only one valid state to another valid state. This means that the data processed during the operation is in a consistent state when the transaction starts and when the transaction ends as well. For example, when we transfer money from one account to another, consistency means that the sum of both accounts before the transfer and after it will be the same. If Account A has $200 and Account B has $100, the sum of both accounts would be $300. If $100 is transferred from Account A to Account B, then Account A would have $100, and Account B would have $200. The sum of both accounts is still $300. Consistency has been maintained.

A consistent database also enforces constraints on the types and sizes of data that are allowed. For example, the balance of an account is expected to be a numeric value. A birthdate is expected to be a date value. The database will maintain consistency by ensuring that the type of the data and size of the data are maintained.

Consistency also enforces primary and foreign key relationships. A primary key is a unique value assigned to each row of a table. For example, in a table containing bank account information, the account number would likely be unique and thus could be a primary key. In regard to consistency, the system will never allow a duplicate primary key in a table to occur, and it will require that each record have a primary key value.

A foreign key is a value within the row of a table that is used to connect or is related to another table. For example, a store can have a table of customers, and each customer can have multiple orders. An order ID can be stored in the customer row that can then connect to a table of orders, as shown in Figure 1.1.



ID	Name	Order
1	John Doe	3
2	Bob Smith	2
3	Fred Flintstone	4
4	Johny Cash	1
...	...	...

ID	Date	Item	...
1	3/1/2023	Pencil	...
2	4/2/2023	Pen	...
3	4/3/2023	Notebook	...
4	5/1/2023	Pencil	...
5	5/5/2023	Stapler	...
...	...	...	...

Figure 1.1 Customer orders

For foreign keys, most DBMS systems by default will not allow you to **orphan** a row, where the value used as a foreign key does not correspond to a value in the primary key of the related table. Using our example of customers and orders, there could be a Customer and Order relationship where a Customer can have one or more Orders. If you were to try to delete only a Customer row without first deleting its Orders, then the Orders associated with that Customer would have a foreign key pointing to a record that no longer existed.

NOTE A properly configured relational schema will prevent this from happening by either rejecting the delete transaction outright or automatically deleting all the orders associated with the customer being deleted first. (This automation is called a **cascade delete**, and because it can lead to the deletion of millions of records without warning, it is usually not the preferred solution to resolving orphan keys.)

Isolation

A transaction is isolated if its effects are not visible to other transactions until it is complete. This is often referred to as *concurrency control*. A large database application may have hundreds or thousands of users making changes to it at the same time, so if transactions are not isolated, this could cause inconsistent data.

Imagine two users, John and Sally, accessing the database at once. John is updating data in the orders table. At the same time, Sally is reading data from the orders table, including records being edited by John. A DBMS has various levels of isolation it could apply. As a beginner, you need to know only two levels.

- **Serializable:** Sally will not receive her data until John's changes are committed. When John begins a transaction to change data, the data is **locked** until his transaction is complete.
- **Read Uncommitted:** Sally will get her data right away, including whatever changes John has made that haven't been committed yet. This is called a **dirty read** because it is possible that John's transaction could fail and be rolled back.

The default isolation in most DBMS systems is serializable because it does a better job of avoiding errors or corrupting data.

Durability

A transaction is durable if once it is **committed** (saved to the database), it will remain so, even in the event of catastrophic failures. Even if you kick the server's power cord out of the wall after a transaction, it will stay committed.

This means a transaction is not fully committed until it is written to permanent storage, such as a storage drive.

Databases and Log Files

In most ACID databases, a **transaction log** (sometimes referred to as a *journal* or *audit trail*) is a history of executed actions. The upshot of this is that even if there are crashes or hardware fails, the log file has a durable list of each change made to the database.

The log file is physically separate from the actual database data. This is important to ensure a database remains consistent. For example, when you insert a new row into a table, a few things happen.

1. The DBMS validates the incoming command.
2. A record is added to the log file specifying what changes are about to be made.
3. The DBMS attempts to make the changes to the actual data in the table or tables.
4. If successful, the log record is marked as committed.

If a failure occurs between steps 2 and 4, like a server reboot, the DBMS will scan the log file for uncommitted transactions when it comes back online. If it finds them, it will examine the actions performed and undo them, effectively restoring the database to its former, consistent state.

ENTITY INTEGRITY

One of the keystones of relational database design is **entity integrity**, which guarantees that each record in a table is unique within that table. All RDBMSs enforce entity integrity automatically, but the database creator has to appropriately define a primary key within each table for this to work. As data is added to a table, the RDBMS will check two properties to ensure that the new record is unique.

- That no other existing record in the table has the same primary key value as the new entry
- That there is a value entered for each field of the primary key

If a new record fails to meet both criteria, then the RDBMS will reject the record and prevent it from being added to the table.

Remember from our definitions for a relational database that a **record** is the collection of values for a single item in a table and that each record is independent of other records in a table. In this case, the term **unique** has its original definition of "one of a kind," so under the guidelines of referential integrity, the set of values in each row must be different from the set of values in all other rows of the table. This uniqueness serves two specific purposes.

- Reducing (but not necessarily eliminating) duplicate data
- Allowing the database to easily find specific records within a table

Ensuring Uniqueness

The relational design approach to meeting the requirements of entity integrity is to include one or more fields in each table whose sole purpose is to identify each individual record. In some cases, we can use an existing field to be the primary key. For example, we could use the entry date as a primary key in a table that tracks newspaper issues, on the grounds that each newspaper in the database issues only one paper per day. This is a called a **natural key** because it happens to be a piece of data we want to track anyway, so we don't need to create a separate field just to ensure uniqueness. Other examples of potential natural keys include a phone number or email address to identify people in a Contacts table. We typically want both of those values in such a dataset, and if each person has their own unique phone number or email address, either could work as a natural key.

A much more common approach is to use a **surrogate key**, a field (or collection of fields) that is created specifically to identify each record in a database, but which has no other purpose or meaning in the table. Because we are surrounded by databases, we are used to using surrogate keys for this purpose. For example, if you contact your bank, chances are good that they will want to use your account number to identify your account, rather than simply your name; your account number is unique, but your name probably isn't. In fact, we regularly use surrogate keys to identify things, such as Social Security numbers, bank account numbers, vehicle identification numbers, and product barcodes. Even when the key value includes some kind of meaning (such as where the person lived when they applied for a Social Security number or the product manufacturer code in a UPC), these values are only loosely connected to the object they identify, and the assignment itself is mostly arbitrary.

On their own, surrogate keys are completely meaningless, but they can serve the very useful purpose of uniquely identifying each object in a table. In a Contact table, a field named ContactID could be defined. Each person could then be assigned a different ContactID value as they are added to the table. While this doesn't prevent us from adding the same person to the contact list more than one time (and assigning a different ContactID value to each instance), it does allow the database to easily distinguish between John Johnson and his son John Johnson, Jr.

The term **candidate key** is used to refer to a field that is inherently unique to each record but that may not act as the primary key of the table. For example, in an employee table, it is highly likely that each employee's Social Security number will be included, and each employee has a unique value for that field. However, for security reasons, it will not be selected as the primary key, and the database designer will select either a different candidate key or a surrogate key to act as the primary key for that table.

Finding Records

The most important role of a primary key is to allow the database to find specific records in a database quickly and accurately. In a relational database, to reduce redundancy and improve efficiency, data is typically stored across many different tables, where each table focuses on one kind of data. In a product inventory database, for example, there will likely be separate tables for products, vendors, warehouse locations, and even categories. As a result, when all the details about a specific item must be retrieved, the database will have to search across tables to find the required information about that item.

For many databases, primary keys are **indexed** by default. Indexes are normally applied to columns (or groups of columns), and they are stored as a separate object from the rest of the table the index applies to. Each index acts as a pointer to records stored in tables in the database, and the RDMBS applies a default sort to the indexes, regardless of the order in which the records were added to the relevant table. These sorted primary keys include a connection (an index) back to the full records associated with each key. The fact that the indexes are both unique and sorted means that the database does not have to search through more records than necessary to find the required key values. Once the requested value is found in the index, the database knows it can stop looking for more instances of that value. Because the primary key is connected to the rest of the data in the associated record, the database can simply pull data from that record and safely ignore all other records in the table.

This is actually similar in concept to an index in a book. A book's index normally appears at the end of the book, where it is easy to locate. In addition, the terms in the index are sorted in alphabetic order, so the reader can easily find the term they are looking for. The index also tells the reader where to find the term in the book so that the reader can go straight to the correct location in the book.

This use of primary keys does have weaknesses, however. The biggest weakness is that the primary key index structure must be reindexed each time a new record is added or an existing record is deleted, which can slow down update processes within the database. In essence, this is similar to having to rewrite the index for a book when a chapter is deleted or added to the book. It is also not possible to change the value of a primary key, meaning that if a value is assigned incorrectly, you cannot change the value later to correct it. However, even with the weaknesses inherent to primary keys, they are an integral component in a relational database. While they can guarantee uniqueness only at the record level, they do allow the database to distinguish and find individual records in a table.

Other options for primary keys will be considered as we work through designing a database later in this book. For now, understand that the role of a primary key is to ensure that each record in a table is unique within that table.

BACKUP STRATEGIES

A lot of time and effort is put into the backup and recovery of database systems. In some businesses, losing access to the database can cost thousands of dollars per minute as orders can no longer be taken or customer data could be lost or compromised.

For this reason, it is important that the database administrator has backup and recovery options for both data and log files. Because logs contain all transaction information, they provide point-in-time restoration information. Full data backups tend to be very large and are done only periodically. Log backups tend to be much smaller.

As an example, nightly data backup might be performed nightly, while a log backup could occur every 10 minutes. If the data backup occurs at midnight and the server fails at 2:55 p.m., the last data backup would be restored, followed by restoring all the logged transactions until 2:50 p.m. We would lose changes only between 2:50 p.m. and 2:55 p.m. While this 5-minute loss is not great, it is better than the alternative if there were no log backups at all.

To further reduce losses, we could use multiple database servers and execute transactions on each. If one server fails, another can take its place. As you approach a true lossless solution, the cost of servers and software increases exponentially. An experienced database administrator has the job of matching budget to loss tolerance for a business.

SUMMARY

This lesson presented an overview of databases and relational database concepts, as well as information on ACID and the use of keys. This information is foundational for learning and working with Structured Query Language. Many of the terms presented in this lesson are used frequently when talking about data and databases, so it's good to learn what they mean early in the game. Most will also be revisited in subsequent chapters when they are directly applied to SQL. This includes the following:

- **Table/relation (academic)/entity (abstract):** A logical grouping of data in a relational database. A table defines valid facts and contains facts about one type of thing.
- **Row/record/tuple (academic):** A single, logical item in a table, comprised of one or more values or fields.
- **Column/field/attribute (academic):** Names a fact to be tracked in a table and restricts the size and type of the fact's data.

ACID was also covered in the lesson. When reliable data is essential, using an ACID-compliant database is a requirement. Only databases that are atomic, consistent, isolated, and durable are going to handle all the errors and failures that can occur while protecting the quality of the data.

Keys were also mentioned as they are a core part of organizing data in a way that can help you create uniqueness as well as help your records to be accessed quickly. As you begin to organize your data into a database that can be accessed with SQL starting in the next chapter, you'll see how primary and foreign keys are used.

Finally, no business should be without a backup and recovery plan that suits their budget and risk tolerance.

EXERCISES

The following exercises are provided to allow you to experiment with concepts presented in this lesson:

Exercise 1.1: Customers and Orders

Exercise 1.2: Libraries and the Books Within

Exercise 1.3: Your Scenario

> **NOTE** The exercises are for your benefit and help you apply what you learn in the lessons. Please note that these are to be done on your own, and thus solutions are not provided.

Exercise 1.1: Customers and Orders

In Figure 1.1, tables were shown for Customers and Orders. Create a list of at least three additional fields that could be added to each of the tables.

Exercise 1.2: Libraries and the Books Within

Assume that you will be creating a program that accesses a database containing information on libraries as well as information on the books each library contains. Do the following:

1. List the tables you could include.
2. Create the list of fields you might include within each row of each table you identified.
3. Create three rows of sample data for each of your tables.
4. Identify the fields in your tables (if any) that would be primary keys.
5. Identify the fields in your tables (if any) that would be foreign keys.

Exercise 1.3: Your Scenario

Come up with your own scenario for using a database. This can be a banking example that tracks accounts, a media database, a restaurant menu, or a store inventory system. Do the following for your scenario:

1. List the tables you could include.
2. Create the list of fields you might include within each row of each table you identified.
3. Create three rows of sample data for each of your tables.
4. Identify the fields in your tables (if any) that would be primary keys.
5. Identify the fields in your tables (if any) that would be foreign keys.