Appendix A. Exercise Solutions

[Chapter 1](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/ch01.html)

Exercise 1 Solution

The following list summarizes how the book provides (or doesn't) database goals:

* **CRUD**: This book doesn't let you easily CREATE information. You could write in new information but there isn't much room for that and that's not really its purpose. The book lets you READ information, although it's hard for you to find a particular piece of information (unless it is listed in the table of contents or the index). You can UPDATE information by crossing out the old information and entering the new. You can also highlight key ideas by underlining, by using a highlighter, and by putting bookmarks on key pages. Finally, you can DELETE data by crossing it out.
* **Retrieval**: The book's mission in life is to let you retrieve its data, although it can be hard to find specific pieces of information unless you have bookmarked them, or they are in the table of contents or the index.
* **Consistency**: I've tried hard to make the book's information consistent. If you start making changes, however, it will be extremely hard to ensure that you make related changes to other parts of the book.
* **Validity**: The book provides no data validation. If you write in new information, the book cannot validate your data. (If you write, "Normalization rocks!" the book cannot verify that it indeed rocks.)
* **Easy Error Correction**: Correcting one error is easy; simply cross out the incorrect data and write in the new data. Correcting systematic errors (for example, if I've methodically misspelled "the" as "thue" and the editors didn't catch it) would be difficult and time consuming.
* **Speed**: The book's structure will hopefully help you learn database design relatively efficiently but a lot relies on your reading ability.
* **Atomic Transactions**: The book doesn't really support transactions of any kind, much less atomic ones.
* **ACID**: Because it doesn't support transactions, the book doesn't provide ACID.
* **Persistence and Backups**: The book's information is non-volatile so you won't lose it if the book "crashes." If you lose the book or it is destroyed (my dog ate it), you can buy another one but you'll lose any updates you have written into it. You can buy a second copy and backup your notes into it but the chances of a tornado destroying your book are low and the consequences aren't all that earth-shattering, so I'm guessing you'll just take your chances.
* **Low Cost and Extensibility**: Let's face it, books are pretty expensive these days, although not as expensive as even a cheap computer. You can easily buy more copies of the book but that isn't really extending the amount of data. The closest thing you'll get to extensibility is buying a different database-related book or perhaps buying a notebook to improve your note taking.
* **Ease of Use**: This book is fairly easy to use. You've probably been using books for years and are familiar with its user interface.
* **Portability**: It's a fairly large book but you can reasonably carry it around. You can't read it remotely the way you can a computerized database, but you can carry it on a bus.
* **Security**: The book isn't password protected but it doesn't contain any top-secret material, so if it is lost or stolen you probably won't be as upset by the loss of its data as by the loss of the concert tickets that you were using as a bookmark. It'll also cost you a bit to buy a new copy if you can't borrow someone else's.
* **Sharing**: After you lose your copy, you could read over the shoulder of a friend and you could borrow someone else's book. Sharing isn't as easy as it would be for a computerized database, however, so you might just want to splurge and get a new copy.
* **Ability to Perform Complex Calculations**: Sorry, not in this edition.

Overall the book is a reasonably efficient read-only database with limited search and correction capabilities. As long as you don't need to make too many corrections, it's a pretty useful tool. The fact that instructional books have been around for a long time should indicate that they work pretty well.

Exercise 2 Solution

This book provides a table of contents to help you find information about general topics and an index to help you find more specific information if you know the name of the concept that you want to study.

Features that help you find information in less obvious ways include the introductory chapter that describes each chapter's concepts in more detail than the table of contents does, and references within the text.

Exercise 3 Solution

CRUD stands for the four fundamental database operations: Create (add new data), Read (retrieve data), Update (modify data), and Delete (remove data from the database).

Exercise 4 Solution

A chalkboard provides:

* **Create**: Use chalk to write on the board.
* **Read**: Look at the board.
* **Update**: Simply erase old data and write new data.
* **Delete**: Just erase the old data.

A chalkboard has the following advantages over a book:

* **CRUD**: It's easier to create, read, update, and delete data.
* **Retrieval**: Though a chalkboard doesn't provide an index, it usually contains much less data than a book so it's easier to find what you need.
* **Consistency**: Keeping the data consistent isn't trivial but again, because there's less data than in a book, you can find and correct any occurrences of a problem more easily.
* **Easy Error Correction**: Correcting one error is trivial; just erase and write in the new data. Correcting systematic errors is harder but a chalkboard contains a lot less data than a book so fixing all of the mistakes is easier.
* **Backups**: You can easily backup a chalkboard by taking a digital picture of it. (This is actually more important than it may seem in a research environment where chalkboard discussions can contain crucial data.)
* **Ease of Use**: A chalkboard is even easier to use than a book. Toddlers who can't read can still scribble on a chalkboard.
* **Security**: It's relatively hard to steal a chalkboard nailed to a wall.
* **Sharing**: Usually everyone in the room can see what's on a chalkboard at the same time. This is one of the main purposes of chalkboards.

A book has the following advantages over a chalkboard:

* **Persistence**: A chalkboard is less persistent. For example, someone brushing against the chalkboard may erase data. (I once had a professor who did that regularly and always ended the lecture with a stomach covered in chalk.)
* **Low Cost and Extensibility**: Typically books are cheaper than chalkboards, at least large ones.
* **Portability**: Books typically aren't nailed to a wall.

The following database properties are roughly equivalent for books and chalkboards:

* **Validity**: Neither provides features for validating new or modified data against other data in the database.
* **Speed**: Both are limited by your reading (and writing) speed.
* **Atomic Transactions**: Neither provides transactions.
* **ACID**: Neither provides transactions so neither provides ACID.
* **Ability to Perform Complex Calculations**: Neither can do this (unless you have some sort of fancy interactive computerized book or chalkboard).

In the final analysis, books contain a lot of information and are intended for use by one person, whereas chalkboards hold less information and are tools for group interaction. Which you use depends on which of these features you need.

Exercise 5 Solution

A recipe card file has the following advantages over a book:

* **CRUD**: It's easier to create, read, update, and delete data in a recipe file. Updating and deleting data is also more aesthetically pleasing. In a book, these changes require you to cross out old data and optionally write in new data in a place where it probably won't fit too well. In a recipe file, you can replace the card containing the old data with a completely new card.
* **Consistency**: Recipes tend to be self-contained so this usually isn't an issue.
* **Easy Error Correction**: Correcting one error in the recipe file is trivial; just replace the card that holds the error with one that is correct. Correcting systematic errors is harder but less likely to be a problem. (What are the odds that you'll mistakenly confuse metric and English units and mix up liters and tablespoons? Although NASA and Lockheed managed to mix metric and English to crash a $125 million Mars orbiter. See www.cnn.com/TECH/space/9909/30/mars.metric.)
* **Backups**: You could back up a recipe file fairly easily. In particular, it would be easy to make copies of any new or modified cards. I don't know if anyone (except perhaps Martha Stewart) does this.
* **Low Cost and Extensibility**: It's extremely cheap and easy to add a new card to a recipe file.
* **Security**: You could lose a recipe file but it will probably stay in your kitchen most of the time. Someone could break into your house and steal your recipes but you'd probably give copies to anyone who asked (except for your top-secret Death-by-Chocolate Brownies recipe).

A book has the following advantages over a recipe file:

* **Retrieval**: A recipe file's cards are sorted, essentially giving it an index, but a book also provides a table of contents. With this kind of recipe file, it would be hard to simultaneously sort cards alphabetically and group them by type (entrée, dessert, aperitif, midnight snack).
* **Persistence**: The structure of a recipe file is slightly less persistent than that of a book. If you drop your card file down the stairs, the cards will be all mixed up. (Although that may be a useful way to pick a random recipe if you can't decide what you want to eat.)

The following database properties are roughly equivalent for books and recipe files:

* **Validity**: Neither provides features for validating new or modified data against other data in the database.
* **Speed**: Both are limited by your reading (and writing) speed.
* **Atomic Transactions**: Neither provides transactions.
* **ACID**: Neither provides transactions so neither provides ACID.
* **Ease of Use**: Many people are less experienced with using a recipe file than a book but both are fairly simple. (Following the recipes will probably be harder than using the file, at least if you cook anything interesting.)
* **Portability**: Both books and recipe files are portable, although your recipe may never leave the kitchen.
* **Sharing**: Neither is easy to share.
* **Ability to Perform Complex Calculations**: Neither can do this. (Some computerized recipe books can adjust measurements for different number of servings but index cards cannot.)

Instructional books usually contain tutorial information and you are expected to read them in big chunks. A recipe file is intended for quick reference and you generally use specific recipes rather than reading a bunch of them. This is more like a dictionary and has many of the same features.

Exercise 6 Solution

*ACID* is an acronym describing four features that an effective transaction system should provide. ACID stands for Atomicity, Consistency, Isolation, and Durability.

* *Atomicity* means transactions are atomic. The operations in a transaction either all happen or none of them happen.
* *Consistency* means the transaction ensures that the database is in a consistent state before and after the transaction.
* *Isolation* means the transaction isolates the details of the transaction from everyone except the person making the transaction.
* *Durability* means that once a transaction is committed, it will not disappear later.

Exercise 7 Solution

If transaction 1 occurs first, Alice tries to transfer $150 to Bob and her balance drops below zero, which is prohibited.

If transaction 2 occurs first, Bob tries to transfer $150 to Cindy and his balance drops below zero, which is prohibited.

So transaction 3 must happen first: Cindy transfers $25 to Alice and $50 to Bob. Afterwards Alice has $125, Bob has $150, and Cindy has $25.

At this point, Alice and Bob have enough money to perform either transaction 1 or transaction 2.

If transaction 1 comes second, then Alice, Bob, and Cindy have $0, $275, and $25 respectively. (If he can, Bob should walk away at this point and quit while he's ahead.) Transaction 2 follows and the three end up with $0, $125, and $175.

If transaction 2 comes second, then Alice, Bob, and Cindy have $125, $0, and $175 respectively. Transaction 1 follows and the three end up with $0, $125, and $175.

So the allowed transaction orders are 3 - 1 - 2 and 3 - 2 - 1. Note that the final balances are the same in any case.

Exercise 8 Solution

If the data is centralized, it does not remain on your local computer. In particular, if your laptop is lost or stolen, you don't need to worry about your customers' credit card information because it is not on your laptop.

Be sure to use good security on the database so cyber-criminals can't break into it remotely.

[Chapter 2](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/ch02.html)

Exercise 1 Solution

Assuming the dogs are not inbred so they have the most genetic diversity, this is a tree so it could be stored in a hierarchical database. It would be a moderately small tree so it would fit easily in a small XML file.

You could also coerce the hierarchical data in a relational database if you wanted to be able to find dogs with certain characteristics such as dog show winners and flyball champions.

Exercise 2 Solution

This seems like a much bigger database than the one in Exercise 1 but it's still a tree. The tree has two main branches: one leading to descendants and one leading to ancestors. As long as you don't track other relationships such as uncles, cousins, and sisters-in-law, it's just another tree so you could still store it in a hierarchical database or an XML file. You could also use a relational database to allow more general queries.

Exercise 3 Solution

Application settings are easy to store in a flat file located in the user's directory hierarchy (for example, in My Documents or the user's Documents and Settings\UserName\Local Settings directory).

If you have a lot of settings and you want to access them by name as needed instead of reading the entire file all at once, you could store them in an INI file, again in a location specific to each user. You could also use the system registry's HKEY\_CURRENT\_USER hive.

I have written applications that stored this kind of information in a shared relational database. That made it centralized so it was easy for the system administrators to fix it when a user managed to make a window zero pixels wide or dragged a window completely off of the screen. It also meant that if you logged into the application from any computer you found your personal settings ready and waiting for you.

Exercise 4 Solution

This sounds like a very simple database whose major requirement is graphing so a spreadsheet can probably handle this. This does tie the application into a dead-end technology, however, and if the users decide that they want to store more complex data and perform sophisticated queries on it, you'll wish you'd chosen a relational database.

Exercise 5 Solution

A spreadsheet can also handle this requirement but there's the same risk that the users will later decide they need more features than a spreadsheet can handle.

Exercise 6 Solution

A spreadsheet will *still* work, with the same caveats. At this point, however, I would notice that the users are starting to add more and more features. I would want to explore the requirements more fully and make sure this is *really* their final request before committing to a spreadsheet. It would be better to move to a more complicated database model now than to have to rebuild everything from scratch in six months. (Or just as likely, have users complain about how the spreadsheet doesn't do all of the things they didn't tell you it was supposed to do.)

Exercise 7 Solution

This is a fairly simple tree so it will fit easily in a hierarchical database or XML file. It's such a small tree (relatively speaking) that it seems unlikely that you'll need to perform complex ad hoc queries.

Exercise 8 Solution

This needs to be some sort of relational database. They are great at handling large amounts of interconnected data and performing complex ad hoc queries.

Which flavor of relational database you should pick (regular, object-oriented, object-relational, object-relational mapping) depends largely on your development philosophy and environment.

Exercise 9 Solution

As in Exercise 8, this problem cries out for some kind of relational database. To make the boss happy, you could use an object-oriented database, object-relational database, or object-relational mapping. In several projects I've used an object-relational mapping approach planted on top of a relational database and it has always worked quite well.

Exercise 10 Solution

If the recipe book will be fairly small, you could just put each recipe on a separate page in a Microsoft Word document and use Word's search capability to find recipe names, part of meal, or main ingredient. (Fooled you, didn't I? That wasn't one of the main topics covered in this chapter! However, it would be a reasonable solution for such a simple application. Remember, the goal is to provide a useful solution with the minimum amount of work.)

Of the solutions that *are* described in this chapter, I would probably pick a relational database. It will provide better search capabilities than the simpler flat file, spreadsheet, or XML databases. Truly object-oriented databases are probably serious overkill for this project. (I would only pick one of them if I wanted practice with a particular new tool, for example, one that I knew was going to be used on a future project.)

Exercise 11 Solution

This one could require some serious sorting and searching so a relational database is your best bet. (You would use a separate table or two to define power decks.) Which flavor you should pick (regular, object-oriented, object-relational, object-relational mapping) depends largely on your development philosophy and environment.

Exercise 12 Solution

These collections could require some serious sorting, searching, and grouping so a relational database is your best bet. The statement "And anything else you might think of later." is a sure sign of a vague specification that will almost certainly require you to implement other queries later. A relational database's ability to perform ad hoc queries is just what you need.

Which flavor of relational database you should pick (regular, object-oriented, object-relational, object-relational mapping) depends largely on your development philosophy and environment.

To support those future unknown queries, you will need to be sure to include as much data as possible about every item in the database. If you don't record each DVD's Best Boy and Key Grip, you won't be able to search on them later.

Exercise 13 Solution

These databases will require some serious sorting, searching, and grouping so a relational database is in order. It will allow you to perform complex queries linking players and their teams.

Which flavor of relational database you should pick (regular, object-oriented, object-relational, object-relational mapping) depends largely on your development philosophy and environment.

Exercise 14 Solution

This situation lends itself naturally to a document-oriented database. I use a separate directory for each book with subdirectories for manuscript files, figure files, and planning files such as the schedule spreadsheet. It's simple and it works fairly well, although it is sometimes tricky to find some items. (For example, find all of the bitmap files containing figures that show image processing techniques. Or find all of the figures that contain pictures of people.)

Exercise 15 Solution

This data is so simple that it could conveniently be stored in just about any kind of database. If the application uses a database for some other purpose, you might consider adding this information to it because the database will be there anyway.

Otherwise you should use the simplest solution that makes sense. A plain old text file would work just fine.

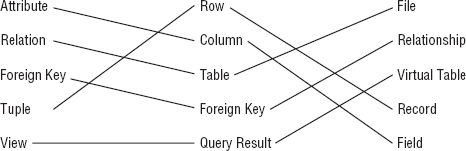
[Chapter 3](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/ch03.html)

Exercise 1 Solution

This constraint means that all salespeople must have a salary or work on commission but they cannot have both a salary and receive commissions.

Exercise 2 Solution

In [Figure A-1](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-1), lines connect the corresponding database terms.



**Figure A.1. Figure A-1**

Exercise 3 Solution

State/Abbr/Title is a superkey because no two rows in the table can have exactly the same values in those columns.

Exercise 4 Solution

Engraver/Year/Got is not a superkey because the table could hold two rows with the same values for those columns.

Exercise 5 Solution

The candidate keys are State, Abbrev, and Title. Each of these by itself guarantees uniqueness so it is a superkey. Each contains only one column so it is a minimal superkey and therefore a candidate key.

All of the other fields contain duplicates and any combination that doesn't have duplicates in the data shown (such as Engraver/Year) is just a coincidence (someone could engrave two coins in the same year). That means any superkey must include at least one of State, Abbrev, or Title to guarantee uniqueness so there can be no other candidate keys.

Exercise 6 Solution

The domains for the columns are:

* **State**: The names of the fifty U.S. states.
* **Abbrev**: The abbreviations of the fifty U.S. states.
* **Title**: Any text string that might describe a coin.
* **Engraver**: People's names.
* **Year**: A four-digit year. More precisely, 1999 through 2008.
* **Got**: "Yes" or "No."

Exercise 7 Solution

Room/FirstName/LastName and FirstName/LastName/Phone/CellPhone are the possible candidate keys.

CellPhone can uniquely identify a row if it is not null. If CellPhone is null, we know Phone is not null because all students must have either a room phone or a cell phone. But roommates share the same Phone value so we need FirstName and LastName to decide which is which. (Basically Phone/CellPhone gets you to the Room.)

Exercise 8 Solution

In this case, FirstName/LastName is not enough to distinguish between roommates. If their room has a phone, they might not have cell phones so there's no way to tell them apart in this table. In this case, the table has no candidate keys. That might be a good reason to add a unique column such as StudentId. (Or if the administration assigns rooms, just don't put two John Smiths in the same room. You don't have to tell them it's because of your poorly designed database!)

Exercise 9 Solution

The room numbers are even so you could use Room Is Even (don't worry about the syntax for checking that a value is even). You could also use some simple range checks such as (Room > = 100) AND (Room < 300) depending on what room numbers are actually allowed.

You might also notice that every Phone value has the same area code and exchange 202-237 so you could check for that.

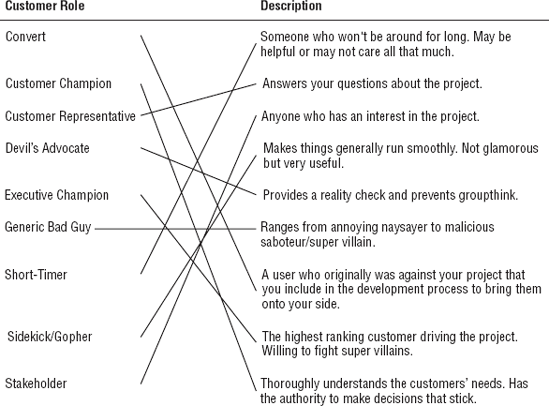
Exercise 10 Solution

Every student must have a Phone or CellPhone value so you could check that (Phone <> null) OR (CellPhone <> null).

[Chapter 4](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/ch04.html)

Exercise 1 Solution

In [Figure A-2](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-2), lines connect the customer roles with their corresponding descriptions.



**Figure A.2. Figure A-2**

Exercise 2 Solution

A use case can cover any part of the customers' operation including big or little pieces of the whole process. In fact, it's easier to test a big scenario if you break it into smaller pieces. The answer that doesn't describe a use case is:

c. Should cover the customer's entire operation from start to finish.

Exercise 3 Solution

Brainstorming sessions should include everyone interested so the correct answer is:

d. All of the above.

Although technically Customer Representatives and the Devil's Advocate are also Stakeholders.

Exercise 4 Solution

The correct answer is:

b. Ask the customer why he thinks that.

You never know if the customer knows more than he's admitting and he may have very good reasons for suggesting that kind of database. Even if he's wrong, the reasons he gives will tell you more about the situation and may lead to other important insights.

Exercise 5 Solution

Whenever you don't understand something about the customers' operation you should ask someone so the correct answer is:

a. Ask someone what that's all about.

The answer you get may be as arbitrary as "that's just the way Mark likes to do it" but in this fictitious scenario the customers use the first date stamp to record when the order was received and the second to indicate that the order entry operator looked at the back of the order to check for notes and comments.

If you didn't ask, you might have incorrectly placed two date fields in the Orders table. Once the process is online, however, you won't need the second date because there is no "other side" of the order to check. (Looking at the back of your computer monitor won't tell you much.) All of the notes and comments will be in a text box at the bottom of the online form.

Exercise 6 Solution

The following table summarizes the fields' data requirements:

| **Field** | **Required?** | **Domain** |
| --- | --- | --- |
| Address 1 | Yes | Valid street addresses or street names without numbers. |
| Address 2 | No | Apartment, suite, floor, etc. |
| City | Yes | Valid cities. |
| State | Yes | Valid states. |
| ZIP Code | No | Five digit or ZIP+4 codes as in 12345 and 12345-6789. |

The required fields are marked on the form with asterisks.

The form could use a foreign key validation for the City, checking against a table listing every city in the country. It would be a huge table and would probably contain errors so in many applications this might not be worth the effort. However, this application needs the city to look up the ZIP Code so if the City isn't legal the lookup will fail. (In fact, that may be the way to validate the data: see if you can look up the ZIP Code.)

The form could also verify that the ZIP Code is valid for the City, if the user enters both. Again the whole point is to look up a ZIP Code so it would be easy to check it against any value that the user entered.

Exercise 7 Solution

Backup policy is a data reliability issue more than a security issue so the correct answer is:

c. The frequency with which you need to perform backups.

Although the two issues are often closely related. For example, in many applications backups must be stored securely so sensitive data doesn't fall into the wrong hands.

Exercise 8 Solution

The correct answer is:

d. It depends (you need more information).

This is probably a priority 1 or 2 feature, depending on how serious Frank is and how soon he wants to add this feature. This doesn't sound too complicated (it would probably just require a few new fields in an inventory table or a new plant lookup table) so I would say if Frank is serious he should make this a priority 1 feature and add it to the database design. I would also make this data not required in case Frank doesn't have time to enter all of this information right away for every kind of plant.

Exercise 9 Solution

The answer to this one depends on the operating system you're using. I'm currently sitting at a computer running Windows XP so here's how my use case might read:

* **Goals**: Authorized users should be able to log in while unauthorized users should not.
* **Summary**: The user tries to enter a user name and password. If they are correct, the user is allowed access to the system.
* **Actors**:
  + The user—Tries to log in.
  + The operating system—Validates the user name and password and grants or denies access.
* **Pre-conditions**: No one is currently logged in to the system.
* **Post-conditions**: If the user enters a valid user name and password, the system is logged in and displays the user's desktop. If the user enters an invalid user name/password combination, the system remains logged out and the user cannot see the desktop or any of the data in the computer.
* **Normal Flow**: The user should try all of the possible combinations of blank, valid, and invalid user names and passwords and click OK. The following table lists the combinations and their desired results. The tester should fill in the blank column with "Pass" or "Fail" to indicate whether each test gave the desired result.

| **Username** | **Password** | **Desired Result** | **Pass/Fail** |
| --- | --- | --- | --- |
| Blank | Blank | No access |  |
| Blank | Valid | No access |  |
| Blank | Invalid | No access |  |
| Valid | Blank | No access |  |
| Valid | Valid | Access |  |
| Valid | Valid for different account | No access |  |
| Valid | Invalid | No access |  |
| Invalid | Blank | No access |  |
| Invalid | Valid | No access |  |
| Invalid | Invalid | No access |  |

* **Alternative Flow**: Instead of clicking OK, the user could click Cancel. The system should reset the screen, blanking the user name and password text boxes.
* **Notes**: In all cases that do not give the user access, the system should deny access in exactly the same way so the user cannot learn, for example, that he has guessed a valid user name but an invalid password. That would give a ne'er-do-well a valid user name to attack and that would be bad.

Note that this use case specifies the user's actions with enough detail that a relatively inexperienced user could follow it.

Exercise 10 Solution

When a heavy hitter such as a Vice President attacks, you need to call in your Executive Champion. Ideally he can point to your requirements document and show that you did, in fact, consider farbulistic granilation and that everyone agreed the allowance was sufficient. If you didn't consider this issue, you may need to put in some extra study to give your Executive Champion ammunition to fend off the attack.

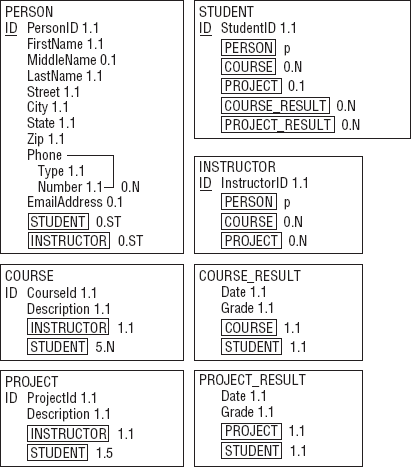
If your Executive Champion doesn't have enough clout to fight off the Super Villain, you could be in trouble.

One project I worked on really did have Super Villains and Executive Champions at that level in a pretty big company (many tens of thousands of employees). I won't bore you with the details but our Executive Champion and Customer Champion spent a huge amount of time fending off attacks for about two years before the project finished.

[Chapter 5](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/ch05.html)

Exercise 1 Solution

[Figure A-3](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-3) shows one possible solution.



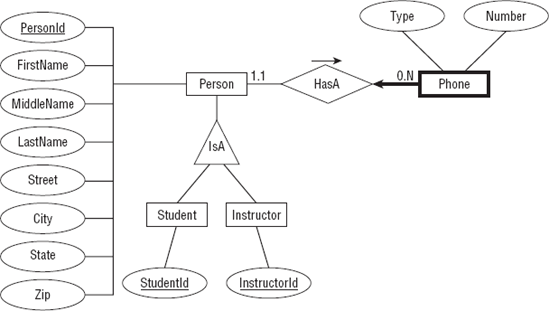
**Figure A.3. Figure A-3**

* In the STUDENT class, COURSE and PROJECT have cardinality 0.N and 0.1, respectively. This doesn't capture the fact that at least one of these two attributes must include at least one value.
* Similarly in the INSTRUCTOR class, does not capture the fact that at least one of the COURSE or PROJECT attributes must include at least one value.

Exercise 2 Solution

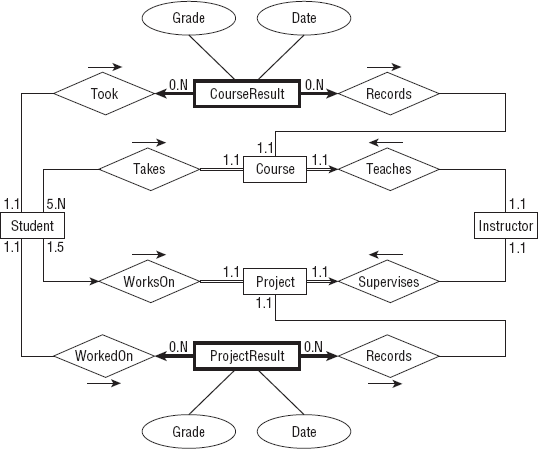
[Figure A-4](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-4) shows an inheritance diagram for the Person, Student, and Instructor entities. It also shows the relationship between the Person and Phone entities.

The Phone entity doesn't have a primary key because it doesn't make sense to search for just a Phoneentity by itself. Instead, you can find the Phone entities corresponding to a Person entity. That means Phone is a weak entity so it is surrounded by a thick rectangle and its identifying relationship is drawn with a thick arrow.



**Figure A.4. Figure A-4**

[Figure A-5](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-5) shows one possible ER diagram for the college course data.



**Figure A.5. Figure A-5**

The diagram's constraints are:

* It doesn't make sense to look for a particular CourseResult so it doesn't have a primary key. Instead you can look for CourseResults associated with a Student or with a Course. That means CourseResult is a weak entity so it is drawn with a thick rectangle and it is connected to its identifying relationships with thick arrows.
* Similarly ProjectResult is a weak entity.
* A Course must be involved in a relationship with a Student (or else the Course is canceled) so its line leading toward Student is double (a participation constraint).
* Similarly a Project must be involved in a relationship with a Student so its line leading toward Student is double (a participation constraint).
* A Course must be involved in a relationship with an Instructor (someone has to teach it) so its line leading toward Instructor is double (a participation constraint). A Course can have only one Instructor so the line is also an arrow (a key constraint).
* Similarly a Project must be involved in a relationship with exactly one Instructor so its line leading toward Instructor is a double arrow (participation and key constraint).
* A Student can work on at most one Project at a time so its line leading to Project is an arrow (key constraint).

Special notes:

* The Student entity's relationships with Course and Project do not indicate that a Studentmust be involved with at least one Course or a Project.
* Similarly the Instructor entity's relationships with Course and Project do not indicate that an Instructor must be involved with at least one Course or a Project.

Exercise 3 Solution

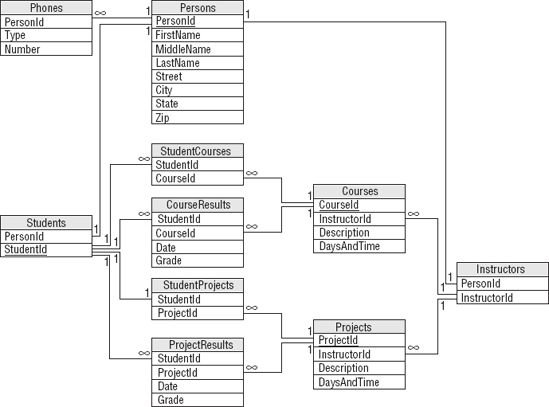
[Figure A-6](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-6) shows one possible solution.

Notice the way this model handles the fact that Student and Instructor inherit from Person. The Persons table holds the basic Person information and a PersonId. The Students and Instructors tables include PersonId foreign keys to link to the corresponding basic Person data.

Note also the different approach used for the Student/Course and Instructor/Course relationships. Because a course has exactly one instructor, the Instructors and Courses tables are connected with a simple one-to-many relationship. In contrast, a course has many students so the relationship uses an intermediate StudentCourses table to connect the two to build a many-to-many relationship. (The same reasoning applies to the Student/Project and Instructor/Project relationships.)

Finally, notice the difference between the Student/Course and Student/Project relationships. A student can be enrolled in any number of courses but at most one project so the first is a many-to-many relationship while the second is a one-to-one relationship.

Unfortunately this solution doesn't capture every aspect of the system either. In particular, it doesn't indicate that a Student must be enrolled in at least one Course or a Project. Similarly it doesn't show that an Instructor must teach at least one Course or supervise at least one Project. The model also doesn't include data type, required, and other domain data. All of this should be noted in separate documents.



**Figure A.6. Figure A-6**

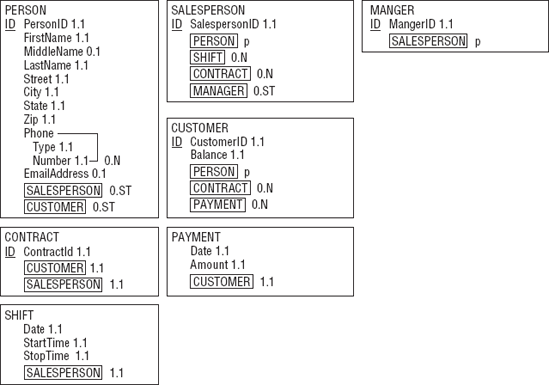
Exercise 4 Solution

[Figure A-7](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-7) shows one possible solution.

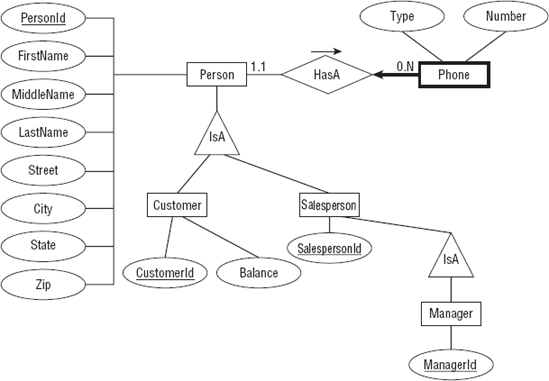
Special notes: The semantic object model actually does a pretty good job of capturing the Mike's Trikes data. About the only item that isn't described explicitly is the manager's role. In this model, you can deduce the manager at any given time by examining the manager's shift data. If Mike needed a more explicit record of who is managing during a salesperson's shift or when a contract was sold, the model would need to be modified.

Exercise 5 Solution

[Figure A-8](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-8) shows an inheritance diagram for the Person, Customer, Salesperson, and Manager entities. It also shows the relationship between the Person and Phone entities.



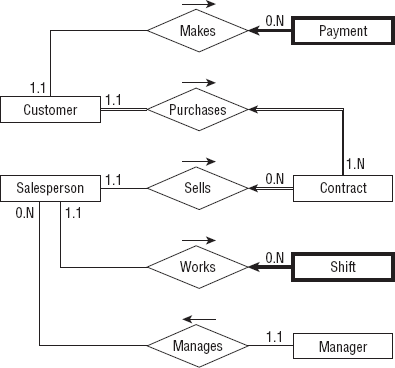
**Figure A.7. Figure A-7**



**Figure A.8. Figure A-8**

The Phone entity doesn't have a primary key because it doesn't make sense to search for just a Phoneentity by itself. Instead, you can find the Phone entities corresponding to a Person entity. That means Phone is a weak entity so it is surrounded by a thick rectangle and its identifying relationship is drawn with a thick arrow.

[Figure A-9](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-9) shows one possible ER diagram for Mike's Trikes.



**Figure A.9. Figure A-9**

The diagram's constraints are:

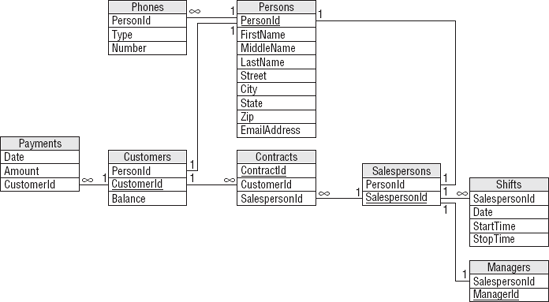
* Payment is a weak entity because you look up payments via the Customer who made them. Payment is drawn with a thick rectangle and a thick arrow pointing toward its identifying relationship.
* Shift is also is a weak entity because you look up shift data via the Salesperson who works them. Shift is drawn with a thick rectangle and a thick arrow pointing toward its identifying relationship.
* A Customer must be involved in at least one Contract (we don't make a Customer record until Customer Purchases Contract) so its line leading toward Contract is double (a participation constraint).
* A Contract must have exactly one Customer and exactly one Salesperson so the lines leading out of Contract toward those other entities are double (participation constraint) and arrows (key constraint).

Special notes:

* This diagram does not emphasize the fact that a Manager is also a Salesperson so a manager could play the role of the Salesperson in the diagram. You could add the Manager Works Shift relationship but that would complicate the diagram.

Exercise 6 Solution

[Figure A-10](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-10) shows one possible solution.



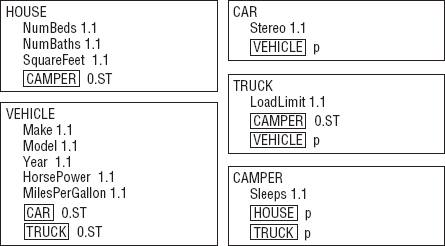
**Figure A.10. Figure A-10**

Notice how this model builds the inheritance hierarchy. The Customers and Salespersons tables use PersonId foreign key fields to link to their corresponding Persons records. The Managers table uses a SalespersonId foreign key field to link to Salespersons records.

As usual, the model doesn't capture all of the information available about the situation. In particular, it doesn't indicate that a Customers record must be associated with at least one Contracts record. You should write down this and other facts such as field data types and domain information in separate documents.

Exercise 7 Solution

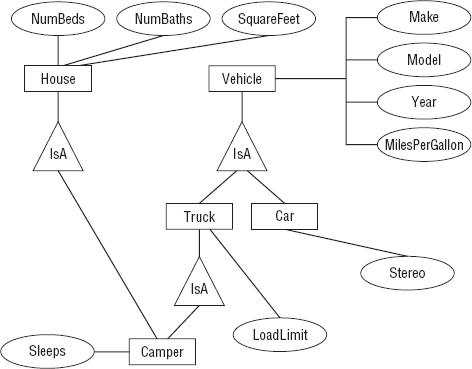
[Figure A-11](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-11) shows one possible solution.



**Figure A.11. Figure A-11**

Exercise 8 Solution

[Figure A-12](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-12) shows one possible solution.



**Figure A.12. Figure A-12**

[Chapter 6](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/ch06.html)

Exercise 1 Solution

The following chart describes the Phones table.

| **Field** | **Required** | **Data Type** | **Domain** | **Sanity Checks** |
| --- | --- | --- | --- | --- |
| PersonId | Yes | ID | Persons.PersonId |  |
| Type | Yes | String | List: Cell, Home, Fax |  |
| Number | Yes | String | Phone numbers |  |

The following chart describes the Persons table.

| **Field** | **Required** | **Data Type** | **Domain** | **Sanity Checks** |
| --- | --- | --- | --- | --- |
| PersonId | Yes | ID | Any ID |  |
| FirstName | Yes | String | Any string |  |
| MiddleName | No | String | Any string |  |
| LastName | Yes | String | Any string |  |
| Street | Yes | String | Any string |  |
| City | Yes | String | Any string |  |
| State | Yes | String | List: (states) |  |
| Zip | Yes | String | ZIP or ZIP+4 format | Verify ZIP or ZIP+4 format |
| EmailAddress | No | String | Valid email address | Contains one @ symbol |
| MedicalNotes | ? | String | Any string |  |
| IceQualified? | ? | Yes/No | Yes or No |  |
| RockQualified? | ? | Yes/No | Yes or No |  |
| JumpQualified? | ? | Yes/No | Yes or No |  |

The following chart describes the Guides table.

| **Field** | **Required** | **Data Type** | **Domain** | **Sanity Checks** |
| --- | --- | --- | --- | --- |
| PersonId | Yes | ID | Persons.PersonId |  |
| GuideId | Yes | ID | Any ID |  |
| IceInstructor? | Yes | Yes/No | Yes or No |  |
| RockInstructor? | Yes | Yes/No | Yes or No |  |
| JumpInstructor? | Yes | Yes/No | Yes or No |  |

The following chart describes the Explorers table.

| **Field** | **Required** | **Data Type** | **Domain** | **Sanity Checks** |
| --- | --- | --- | --- | --- |
| PersonId | Yes | ID | Persons.PersonId |  |
| ExplorerId | Yes | ID | Any ID |  |

The following chart describes the Organizers table.

| **Field** | **Required** | **Data Type** | **Domain** | **Sanity Checks** |
| --- | --- | --- | --- | --- |
| PersonId | Yes | ID | Persons.PersonId |  |
| OrganizerId | Yes | ID | Any ID |  |

The following chart describes the Adventures table.

| **Field** | **Required** | **Data Type** | **Domain** | **Sanity Checks** |
| --- | --- | --- | --- | --- |
| AdventureId | Yes | ID | Any ID |  |
| ExplorerId | Yes | ID | Explorers.ExplorerId |  |
| EmergencyContact | Yes | ID | Persons.PersonId |  |
| OrganizerId | Yes | ID | Organizers.OrganizerId |  |
| TrekId | Yes | ID | Treks.TrekId |  |
| DateSold | Yes | Date | Any date | Before the trek's start date. Between January 1, 2000 and December 31, 2050 (or some other very early and late dates). |
| IncludeAir? | Yes | Yes/No | Yes or No |  |
| IncludeEquipment? | Yes | Yes/No | Yes or No |  |
| TotalPrice | Yes | Currency | Monetary amount > $0 | Price > $250 (or some minimum sane value). |
| Notes | ? | Yes/No | Yes or No |  |

The following chart describes the Treks table.

| **Field** | **Required** | **Data Type** | **Domain** | **Sanity Checks** |
| --- | --- | --- | --- | --- |
| TrekId | Yes | ID | Any ID |  |
| GuideId | Yes | ID | Guides. GuideId |  |
| Description | Yes | String | Any string | Length > 100 (anything shorter couldn't say enough). |
| Locations | Yes | String | List of locations | Length > 5. |
| StartLocation | Yes | String | A location | Length > 5. |
| EndLocation | Yes | String | A location | Length > 5. |
| StartDate | Yes | Date | Any date | StartDate is on or before EndDate. Between January 1, 2000 and December 31, 2050 (or some other very early and late dates). |
| EndDate | Yes | Date | Any date | EndDate is on or after StartDate. Between January 1, 2000 and December 31, 2050 (or some other very early and late dates). |
| Price | Yes | Currency | Monetary amount > $0 | Price > $250 (or some minimum sane value). Price > some minimum price per day times the number of days (EndDate-StartDate). |
| MaxExplorers | Yes | Number | Number > 0 | Number > 0. Number < 20 (or some maximum sane amount). |
| IceRequired? | Yes | Yes/No | Yes or No |  |
| RockRequired? | Yes | Yes/No | Yes or No |  |
| JumpRequired? | Yes | Yes/No | Yes or No |  |

Exercise 2 Solution

The following list describes business rules that can be implemented in field or table checks for the Phones table:

* **Type**: Verify that the type is one of Home, Work, Cell, or Fax. Alternatively if you think this list might change in the future, you could put these values in a lookup table.
* **Number**: Verify that the value has a valid phone number format. In the United States, you would probably want to verify that it is a 10-digit number of the format ###-###-#### and you should allow for an extension.

The following list describes business rules that can be implemented in field or table checks for the Persons table:

* **FirstName/MiddleName/LastName**: Verify that this combination is unique. This will prevent you from adding the same person twice, perhaps as an explorer and as an emergency contact.

It would also be natural to try to validate the EmailAddress field in a field check. Unfortunately valid email address formats are quite complicated so this probably doesn't belong in the simpler field and table checks.

Similarly it might be nice to look up the explorer's City, State, and Zip values to make sure they are compatible. If you build a table listing all of the possible combinations, this wouldn't be a hard check but it would be an enormous table so it's probably not worth all of the extra effort. (Although for bonus points you could probably use a Web Service to perform this check over the Internet. If you don't know what a Web Service is, don't worry about it.)

You could also look up the State value in a list built into a field check. Though it's unlikely that the list of allowed states will change often, this list is so long that it's easier to manage in a separate lookup table rather than in a very long field check. (And who knows, Canada may eventually be officially recognized as "The Maple Leaf State.") (Just kidding! But this does bring up a whole series of questions about non-US explorers. This model ignores those issues completely.)

The Explorers, Organizers, and Guides tables should verify that their records are unique. That means checking uniqueness for the Explorers table's PersonId/ExplorerId fields, the Organizers table's PersonId/OrganizerId fields, and the Guides table's PersonId/GuideId fields.

The following list describes business rules that can be implemented in field or table checks for the Adventures table:

* **(Table)**: Verify that the trek has room for this explorer.
* **(Table)**: Verify that the explorer's IceQualified?, RockQualified?, and JumpQualified? values include those required for this trek.
* **ExplorerId/TrekId**: Verify that this combination is unique. An explorer should not buy the same trek twice. (We're assuming that the same trip on different dates gets a different record in the Treks table. Some people may very well want to go to the same places again.)
* **EmergencyContact**: Verify that the EmergencyContact is not going on the same trek listed for this Adventures record.
* **IncludeAir?/Notes**: If IncludeAir? is Yes, the Notes field should include flight information such as the explorer's starting airport and meal preferences. The database can probably not verify that the notes make sense (who knows if the low sodium meal is available on that flight?) but it can verify that the Notes entry has some minimum length if IncludeAir? is Yes.

The Adventures table would be a natural place to try to deal with the discounts for purchasing airline tickets or renting equipment. You would set TotalPrice equal to the trek's cost minus any discounts. (Note that this model doesn't have room to hold information about the equipment rented. The full model would need more order-related information along those lines.)

In any case, the discount schedule seems likely to change so it's better handled later, not in a simple field or table check.

The following list describes business rules that can be implemented in field or table checks for the Treks table:

* **(Table)**: Verify that the guide's IceQualified?, RockQualified?, JumpQualified?, IceInstructor?, RockInstructor?, and JumpInstructor? values include those required for this trek.

Exercise 3 Solution

The following list summarizes business rules that should be extracted from the database's structure:

* If you really want to validate email addresses, it would be better to do so outside of the field and table checks. You could put this code in a stored procedure, code library, or middle tier.
* If you use a lookup table to validate phone number types (Home, Work, Cell, or Fax), do so here.
* If you're going to perform a complex City/State/Zip lookup, this is where to do it. You might use a huge table or you might call a Web Service over the Internet.
* If you use a lookup table to validate State values, do so here.
* This is where you would calculate an adventure's TotalPrice. You would look up discount information stored in a separate table and perform the calculation. You could put this code in a stored procedure, code library, or middle-tier layer.
* The fact that one of the company's owners asked which calculation would give the customer the biggest discount if they both purchase airline tickets and rent equipment (adding the two discounts and take 15% off gives the biggest discount) further implies that they might someday change the way they perform this calculation. That gives you more reason to extract this rule from the database so it's easier to change later.
* If the adventure's IncludeAir? value is Yes, you could try to parse the Notes field to see if the flight and meal information is present. I've seen several systems that make these sorts of checks, mostly because their requirements changed after the database was built and they couldn't easily modify the database. If you really need this check, you should move the flight and meal information into separate fields so they are easier to find and examine.

Exercise 4 Solution

The PhoneTypes table would have only one field: Type. The records would initially include Home, Work, Cell, and Fax.

The States table would have only one field: State. The records would list all of the allowed State values: AL, AK, AS,..., WY.

The DiscountParameters table would have two fields: Type and Amount. Type would give the discount type (Air or Equipment) and Amount would be the discount amount (15% or 5%).

An additional Parameters table would have two fields: Name and Value. This table would hold parameters used in other calculations so they would be easier to update than they would be if they were embedded in check constraints. The following table describes the initial values in this table.

| **Name** | **Value** | **Purpose** |
| --- | --- | --- |
| MinimumDate | January 1, 2000 | Sanity check date for DateSold, StartDate, and EndDate. |
| Maximum Date | December 31, 2000 | Sanity check date for DateSold, StartDate, and EndDate. |
| Minimum TotalPrice | $250 | Sanity check price for an Adventure's TotalPrice. |
| Minimum TrekPrice | $250 | Sanity check price for a Trek's Price. |
| Minimum PricePerDay | $100 | Sanity check minimum price per day for a Trek's Price. |
| Maximum Explorers | 20 | Sanity check maximum number of explorers on a trek. |

[Chapter 7](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/ch07.html)

Exercise 1 Solution

1. The list isn't in 1NF because it violates these 1NF rules:
   1. Each column must have a unique name.

The two Email fields have the same name.

* 1. The order of the rows and columns doesn't matter.

The order of the Email columns represents the student's preferred email address.

* 1. Each column must have a single data type.

The MajorOrSchool field holds both majors and schools.

* 1. Each column must contain a single value.

The Name field contains the student's first and last names together.

Let's take these rules one at a time.

* 1. Each column must have a unique name.

The two Email fields have the same name. You could fix this problem by giving them different names. For example, you could name them Email1 and Email2. The numbers would indicate the student's preferred email address solving the problem with Rule 2. This is the approach taken by the Phone1 and Phone2 fields so it might work, right?

Not really. There's another equally important issue here. These two Email fields represent the same kind of data with only a minor difference: priority. Aside from the student's preference of which comes first, the two fields hold identical values. How do we know you won't want to add a third email address later? You've already got two, why not three or four? Simply renaming the fields solves the duplicate name issue but locks you in to exactly two email addresses. Not only would that prevent you from adding more email addresses, but in many cases the second field would be empty. It's also flirting with 1NF rule number 6: Columns cannot contain repeating groups.

A better solution to the multiple Email field problem would be to pull those fields into a new StudentEmails table.

While we're thinking about multiple fields holding the same kind of data, let's take a closer look at the Phone1, PhoneType1, Phone2, and PhoneType2 fields. Though they have different names, they also represent the same kind of information and you're probably even more likely to want a third phone number than you are to want a third email address. Though these fields technically don't violate 1NF (aside from Rule 6), it's probably worthwhile moving them into a new StudentPhones table.

* 1. The order of the rows and columns doesn't matter.

The order of the Email columns represents the student's preferred email address. The new StudentEmails table should have a Priority column to capture the student's preference. Similarly the new StudentPhones table should have a Priority column to indicate the student's preference.

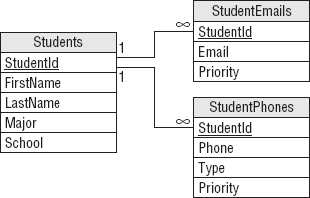
* 1. Each column must have a single data type.

The MajorOrSchool field holds both majors and schools. It should be split into Major and School fields. Note that a student has a school whether he has a major or not so the School field should always contain a value while the Major field may contain null.

* 1. Each column must contain a single value.

The Name field contains the student's first and last names together. Here you need to decide whether the name value is atomic. In other words, will you *ever* need to do something with just a first name or just a last name? Chances are good that you'll want to at least be able to search for last names (so you can easily look up students) so you should split this field into FirstName and LastName fields.

1. [Figure A-13](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-13) shows a relational diagram for this model.



**Figure A.13. Figure A-13**

Exercise 2 Solution

1. The list isn't in 1NF because it violates these 1NF rules:
   * 2. The order of the rows and columns doesn't matter.

The order of the rows represents the rows' priorities.

* + 5. Each column must contain a single value.

The Items column contains a comma-separated list of values.

1. The following table shows one way to convert the list into 1NF.

| **Location** | **Item** | **Priority** |
| --- | --- | --- |
| Grocery store | milk | 1 |
| Grocery store | eggs | 1 |
| Grocery store | bananas | 1 |
| Office supply store | paper | 2 |
| Office supply store | pencils | 2 |
| Office supply store | divining rod | 2 |
| Post Office | stamps | 3 |
| Computer store | flash drive | 4 |
| Computer store | 8" floppy disks | 4 |

The primary key for this table is the combination Location/Item.

Exercise 3 Solution

1. The list isn't in 2NF because it violates the 2NF rule:
   * 2. All of the non-key fields depend on all of the key fields. The Priority field depends on Location but not Item. That's why its values are repeated so many times in the table.
2. The solution is to pull the non-key field (Priority) out into a new table and use the key field that it depends on (Location) as the link to the original data. [Figure A-14](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-14) shows the new relational design.

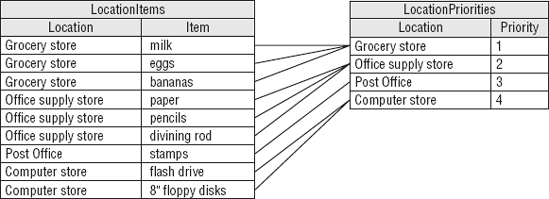
Figure A-14

**Figure A.14. Figure A-14**

[Figure A-15](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-15) shows the new tables holding the original data.

Exercise 4 Solution

1. The list isn't in 3NF because it violates the 3NF rule:
   * 2. It contains no transitive dependencies.



**Figure A.15. Figure A-15**

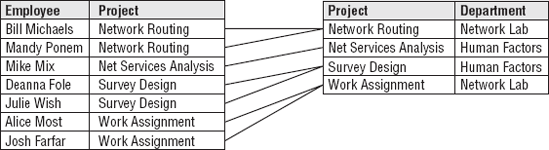
In this table, the Department field depends on the Project. Because those fields are not key fields, this is a transitive dependency.

1. The solution is to pull the dependent field (Department) out into a new table and use the field that it depends on (Project) as the link to the original data. [Figure A-16](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-16) shows the new relational design.

Figure A-16

**Figure A.16. Figure A-16**

[Figure A-17](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-17) shows the new tables holding the original data.



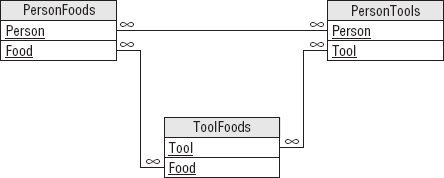
**Figure A.17. Figure A-17**

Exercise 5 Solution

1. The table isn't in 5NF because it violates the 5NF rule:
   * 2. It contains no related multi-values dependencies.

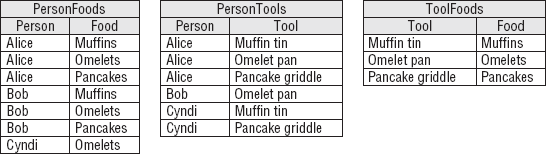
In this table, Person determines Food (the type the person can make), Person determines Tools (those in the person's kitchen), and Tool partially determines Food (you can't make muffins without a muffin tin). This makes a related multi-value dependency.

[Figure A-18](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-18) shows an ER diagram for this model.



**Figure A.18. Figure A-18**

1. The solution is to break the single table into three new tables that record the three different relationships: Person/Food, Person/Tool, and Tool/Food. [Figure A-19](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-19) shows the new relational model.



**Figure A.19. Figure A-19**

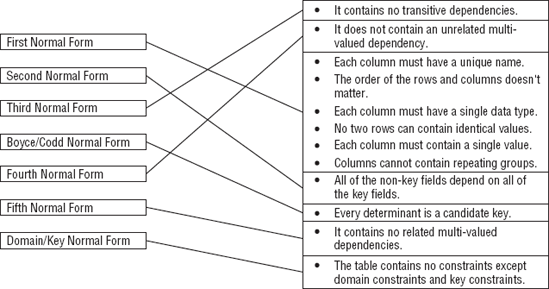
Exercise 6 Solution

[Figure A-20](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-20) shows the matching between normal forms and their rules.

[Chapter 8](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/ch08.html)

Exercise 1 Solution

The following ShipClasses table contains the allowed combinations of Ship and Class.



**Figure A.20. Figure A-20**

| **Ship** | **Class** |
| --- | --- |
| Luxury Liner | 1st Class |
| Luxury Liner | 2nd Class |
| Luxury Liner | 3rd Class |
| Luxury Liner | 4th Class |
| Luxury Liner | 5th Class |
| Schooner | 1st Class |
| Schooner | 2nd Class |
| Tuna Boat | 1st Class |

Because the validation involves two fields, this must be a two-field foreign key constraint. In the Trips table, the combination of fields Ship/Class will be a foreign key referencing the ShipClasses table's Ship/Class fields.

Exercise 2 Solution

The Students table holds information about students so it is an object table. Similarly the Departments table holds information about the school's departments and the Classes table holds information about classes so they are also object tables.

The StudentClasses table links the Students and Classes tables so it is a link table. Similarly the DepartmentClasses table links the Departments and Classes tables so it is also a link table.

Exercise 3 Solution

This table is trying to hold information about three different concepts: the first player, the second player, and the match they will play.

To fix it, create a Players table with fields PlayerId, Name, and Rank. Put all of the player information in this table for all of the Player1 and Player2 entries. This is an object table holding information about players.

Then create a Matches table that has fields PlayerId1, PlayerId2, and MatchTime. This is a link table that links the Players table to itself. It also holds extra information about the link: the times of the matches.

Exercise 4 Solution

1. Average minutes late for an airline at a particular airport.

This will require finding and averaging up to a few hundred values so it should be possible to calculate as needed.

1. Average minutes late for all airlines at a particular airport.

This will require finding and averaging several hundred values. It may still be possible to perform this calculation as needed.

1. Average minutes late for an airline across the country.

This could require a *lot* of calculations. If this is a common query (for example, if lots of people are asking for this information all over the country hundreds of times per day), it might be better to store and update the information as planes take off and land rather than calculating it as needed.

1. Average minutes late for all airlines across the entire country.

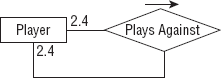
This will require a huge number of calculations. This could take quite a while even if the database isn't heavily used so it might be best to store this value rather than calculating it as needed.

Of course, as long as you're going to store some of these values, you might want to just store them all so you can treat them uniformly.

[Chapter 9](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/ch09.html)

Exercise 1 Solution

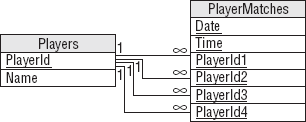
[Figure A-21](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-21) shows an ER diagram to represent Parcheesi matches.



**Figure A.21. Figure A-21**

Exercise 2 Solution

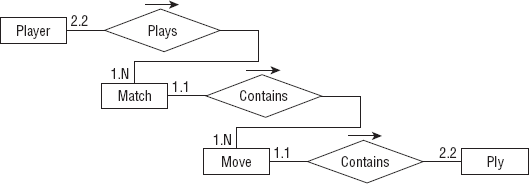
[Figure A-22](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-22) shows a relational model for recording information about Parcheesi matches. PlayerId1 finished first, PlayerId2 finished second, PlayerId3 finished third, and PlayerId4 finished fourth.



**Figure A.22. Figure A-22**

Exercise 3 Solution

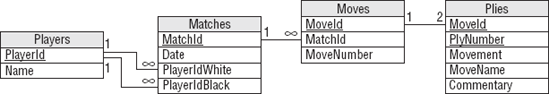
[Figure A-23](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-23) shows an ER diagram that represents the relationships between Match, Move, and Ply.



**Figure A.23. Figure A-23**

Exercise 4 Solution

[Figure A-24](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-24) shows a relational model for recording chess Match, Move, and Ply data.

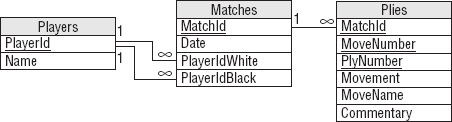


**Figure A.24. Figure A-24**

You can model the one-to-two relationship between Moves and Plies by making the domain of the PlyNumber field include the values 1 and 2. You can implement that as a field-level check constraint on PlyNumber. Note that the fact that MoveId/PlyNumber is the Plies table's primary key ensures that each move cannot contain two plies with the same PlyNumber.

Exercise 5 Solution

[Figure A-25](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-25) shows the chess model without the Moves table.



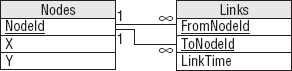
**Figure A.25. Figure A-25**

The new diagram doesn't explicitly show that there should be exactly two plies per move. It has converted the old one-to-two relationship into a new one-to-many relationship.

The database still needs to verify that there are only two plies per move. You can still use a field-level check constraint to verify that the PlyNumber is either 1 or 2. The fact that MatchId/MoveNumber/PlyNumber is the Plies table's primary key ensures that any move in a given match cannot contain two plies with the same PlyNumber.

Exercise 6 Solution

The network solution described earlier in [Chapter 9](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/ch09.html) and shown in [Figure A-26](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-26) uses two tables. The Nodes table holds node IDs and coordinates. The Links table holds link times and the IDs of the nodes that the link connects.



**Figure A.26. Figure A-26**

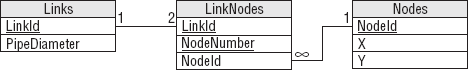
This exercise is slightly different because it is an undirected network. In other words, each link has the same "value" no matter which direction you cross it. The solution shown in [Figure A-26](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-26) isn't perfect because the FromNodeId and ToNodeId fields imply a direction for the link. To use this design you would either need to recognize that a Links record connecting node1 to node2 also represents a link connecting node2 with node1. Or you could insert two records for each link with the order of the node IDs switched, but that would double the number of records and all of that duplication screams out, "I'm not normalized!"

In normalization terms, FromNodeId and ToNodeId store the same kind of data. For a directed network, the two fields are not exactly the same thing so there's some benefit to using two fields with different names to store their data and differentiate them.

Normalization purists would say that the link's node data should be moved into a new table with an extra field to tell you which was the "from" node and which was the "to" node. For a directed network, the extra layer of indirection seems like a lot of work for little benefit. In addition to making you follow extra links to find the data, you would also need to perform some new validations to ensure that every link corresponded to exactly two nodes.

However, this more normalized design works somewhat better for an undirected network because moving the link's nodes into a new table removes the implication that one is the "from" node and one is the "to" node.

You still need a way to ensure that each link has two nodes, however. One way to do that is to give the new table a NodeNumber field to indicate which node this is, make the domain of NodeNumber be the numbers 1 and 2, and make LinkId/NodeNumber the primary key. That ensures that any link can have only two nodes. This design is shown in [Figure A-27](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-27).



**Figure A.27. Figure A-27**

This is the same as the normalized design for a directed network. The only difference is that in the undirected network you treat the NodeNumber field as a simple index to ensure that a link has two nodes whereas in a directed network you use that field to tell which node is "from" and which is "to."

Exercise 7 Solution

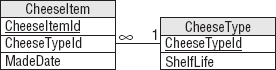
This is fairly straightforward temporal data. [Figure A-28](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-28) shows a model to hold cheese item data. A CheeseItem record would probably hold other information such as the quantity of cheese purchased, the lot number, and so forth.



**Figure A.28. Figure A-28**

Exercise 8 Solution

[Figure A-29](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-29) shows the new model to hold cheese item data. Instead of a SellByDate, this version stores the date the cheese was made and a link leading to the shelf life.



**Figure A.29. Figure A-29**

In the model, the CheeseItem table is the same size as the model for Exercise 7 and there's a new table, so you could ask if this is an improvement. In terms of looking up expiration data for a particular cheese item, however, this model isn't an improvement. It takes more space and requires an extra lookup plus a calculation (MadeDate + ShelfLife) to find the cheese item's sell-by date.

However, this model provides more consistency because it ensures that each item of a particular kind of cheese uses the same shelf life.

[Chapter 10](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/ch10.html)

Exercise 1 Solution

This table has a lot of problems. Specific problems include:

* The Name field includes two logical fields, FirstName and LastName, so the table isn't even in First Normal Form.
* Your client plans to look up the state from the Zip value. Why doesn't he also look up the city? The table should be changed to either also look up the city or have separate City, State, and Zip fields (the second option is a lot easier).
* The two phone number fields are not differentiated. In other words, how do you know which number is a home phone, cell phone, or work phone? Which is the daytime number and which is the evening number? These fields should be moved into a Phones table with an additional field indicating the type of the phone number.
* Two phone numbers is also an arbitrary limit. Some day a customer will probably want to leave more than two numbers. When you create the Phones table, you should not restrict a customer to two entries.
* The Address field has a bad name because Address implies that the field contains an entire address when in fact it only contains the street information. This field's name should be something like Street or StreetAddress.
* The Stuff field has a terrible name because "Stuff" could mean just about anything! This field's name should be changed to Interests.
* The freshly renamed Interests field lists more than one value. (The fact that the name is plural is a hint.) This field's data should be moved into a new CustomerInterests table. You should also make an Interests lookup table to list the allowed values so CustomerInterests can use it as a foreign key constraint.
* Planning for future changes, you might also suggest adding an Email field.

Your client's assumption that you can just build Orders and other tables implies the plan isn't very well thought out. This project definitely needs a lot more planning and a complete database design before you start slapping tables together. This kind of homegrown project also rarely includes documentation of any kind so you'll need to do a lot of documentation work early in the project. (Though this type of project often provides many hours of lucrative consulting later for debugging, it's the frustrating kind of consulting.)

Exercise 2 Solution

Because this client is opening a new store, you should wonder if they will grow even more in the next few years. Blu-ray is also a brand new technology and, if it becomes as popular as current growth indicates, demand for rentals could skyrocket.

This database will need extra testing at very high loads to verify that the database design can meet ever-increasing performance demands.

In contrast, a well-established party rental store probably won't experience explosive growth in the near future because it's been around for a while and it isn't selling new technology. You still need to thoroughly test their application but your load testing doesn't need to run at loads as far beyond the current level.

Exercise 3 Solution

This table is hyper-normalized. Though you can break a street address into name, number, prefix, and so forth, there are very few applications where that is necessary. If you will only ever need to use the address information to send mail to someone, you can combine all of this information in a single Street field. You can even include the apartment or suite number.

Similarly you can probably combine the Zip and PlusFour fields into a single Zip field. If you're only going to use the ZIP Code to write addresses, there's no need to use separate fields.

The Floor and Neighborhood information is also probably not useful. (Although if your business is renting apartments, you might want to be able to search for ground floor apartments or apartments within a certain neighborhood. In that case, these fields might make sense.)

Here's the new list of fields:

* CustomerId
* Street
* City
* State
* Zip

Exercise 4 Solution

In this model, the Phones table is fairly unconstrained because it allows a person to have any number of any type of phone number. All of the fields are required. Some other validations that you could build into the database include:

| **Field** | **Constraint** | **Implementation** |
| --- | --- | --- |
| PersonId | Exists | Foreign key match to Persons.PersonId. |
| Type | Enumerated value | Foreign key match to new PhoneTypes table. |
| Number | Format | Let the database verify that the value has format ###-###-####. |

In the Persons table, every field except MiddleName should be required. The table can implement the following constraints:

| **Field** | **Constraint** | **Implementation** |
| --- | --- | --- |
| State | Enumerated value | Foreign key match to new States table. |
| Zip | Format | Let the database verify that the value has format #### or ####-####. |

All of the fields in the Courses and Projects tables should be required, although you may want to allow a blank InstructorId and DaysAndTime so you can create a course before you're ready to schedule it. This table should also have a foreign key constraint requiring that the InstructorId exist in the Instructors table.

The Students and Instructors tables should require all fields. They should also have a foreign key constraint requiring that their PersonId fields have values that exist in the Persons table.

StudentCourses and StudentProjects are linking tables used to implement many-to-many relationships. Their fields should be required and foreign key constraints should verify that their values exist in the corresponding tables.

CourseResults and ProjectResults are also linking tables that implement many-to-many relationships. They should require all fields and foreign key constraints should verify that their ID values exist in the corresponding tables.

CourseResults and ProjectResults should also use constraints to verify that the Grade fields contain acceptable values. If Grade is numeric, a check constraint should verify that it is between 0 and 100 (or whatever scale the school uses). If the Grade value includes A+, A, A-, B+, and so forth, the tables should use foreign key constraints to verify that the Grade exists in a new PossibleGrades table.

Finally, you could check that the Date fields in the CourseResults and ProjectResults tables come after the corresponding student's enrollment date.

[Chapter 11](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/ch11.html)

Exercise 1 Solution

The following table summarizes the Course entity's fields.

| **Field** | **Req'd?** | **Data Type** | **Domain** |
| --- | --- | --- | --- |
| Title | Yes | String | Any string |
| Description | Yes | String | Any string. |
| MaximumParticipants | Yes | Integer | > 0 |
| Price | Yes | Currency | > 0 |
| AnimalType | Yes | String | One of Cat, Dog, Bird, Bat, and so on. |
| Dates | Yes | String | List of dates. |
| Time | Yes | Time | Between 8am and 11pm. |
| Location | Yes | String | One of Room 1, Room 2, yard, arena, and so on. |
| Trainer | No | Reference | The Employee teaching the course. |
| Students | No | Reference | Customers table. |

Because the Dates and Time fields are required, we cannot create a course until it is scheduled.

A more complex validation for new records should verify that there are no other courses scheduled for the same location with overlapping dates and times.

Exercise 2 Solution

The following table summarizes the Employee entity's fields.

| **Field** | **Req'd?** | **Data Type** | **Domain** |
| --- | --- | --- | --- |
| FirstName | Yes | String | Any first name. |
| LastName | Yes | String | Any last name. |
| Street | Yes | String | Any street name and number. Not validated. |
| City | Yes | String | Any city name. Not validated? |
| State | Yes | String | Foreign key to States table. |
| Zip | Yes | String | Valid ZIP Code. Not validated? |
| Email | No | String | Valid email address. If provided, send the customer a monthly email newsletter. |
| HomePhone | No | String | Valid 10-digit phone number. |
| CellPhone | No | String | Valid 10-digit phone number. |
| SocialSecurityNumber | Yes | String | Valid Social Security number. |
| Specialties | No | String | Zero or more of: Dog, Cat, Horse, Bird, Fish, Snail, and so on. |

Exercise 3 Solution

Alicia and The Pampered Pet employees think of work shift assignments as coming in one week batches. Alicia posts schedules one week at a time.

However, the database may not actually need to create records representing weeks of assignments. Instead it can track individual work assignments that represent an employee working certain hours on a given day. The interactive interface and any work assignment reports will gather the assignments for a particular week and display the results in the familiar week-at-a-time format.

That means the Shift entity can be relatively simple:

| **Field** | **Req'd?** | **Data Type** | **Domain** |
| --- | --- | --- | --- |
| Employee | Yes | Reference | Refers to the assignment's employee. |
| Date | Yes | Date | Valid dates. For new records, verify that the date is on or after today. |
| StartTime | Yes | Time | > = 6am. |
| StopTime | Yes | Time | > = StartTime + 1 hour, and < = 11pm. |

Exercise 4 Solution

The following table summarizes the Customer entity's fields.

| **Field** | **Req'd?** | **Data Type** | **Domain** |
| --- | --- | --- | --- |
| FirstName | Yes | String | Any first name. |
| LastName | Yes | String | Any last name. |
| Street | See notes | String | Any street name and number. Not validated. |
| City | See notes | String | Any city name. Not validated? |
| State | See notes | String | Foreign key to States table. |
| Zip | See notes | String | Valid ZIP Code. Not validated? |
| Email | See notes | String | Valid email address. If provided, send the customer a monthly email newsletter. |
| HomePhone | See notes | String | Valid 10-digit phone number. |
| CellPhone | No | String | Valid 10-digit phone number. |
| Pets | No | String | Pet names, ages, and types. |

The system only creates customer records in one of the following circumstances:

* The customer enrolls in a course. In that case, we require either a home or cell phone number so we can contact the customer in case there's a change in schedule or some other unexpected event occurs (for example, Sveta contracts Capgras syndrome and won't work with Charlie anymore).
* The customer wants to receive postal mailings about sales and courses. In that case, the address information is required.
* The customer wants to receive email about sales and courses. In that case, the email address is required.
* We are shipping items to the customer. In that case, the address information and at least one phone number is required.

Exercise 5 Solution

Like the Shift entity, TimeEntry is simpler than it might appear. Users typically think of timekeeping as a weekly chore so they tend to think of a week's worth of time entries. However, individually each time entry is quite simple. The interactive timekeeping interface and any related reports (including printing payroll checks) will gather the assignments for a particular week and display the results appropriately.

The following table summarizes the TimeEntry entity's fields.

| **Field** | **Req'd?** | **Data Type** | **Domain** |
| --- | --- | --- | --- |
| Employee | Yes | Reference | The employee who worked (or at least pretended to work). |
| Date | Yes | Date | < = now. |
| StartTime | Yes | Time | < = now. |
| StopTime | Yes | Time | < = now, and > StartTime. |
| PaidDate | No | Date | < = now. |

The PaidDate field records the date on which the employee's check was printed covering this time entry.

A more complex check for new records should verify that no existing record for this employee has an overlapping date and times.

Exercise 6 Solution

The Vendor entity gives the name of a company that provides Pampered Pet products. (Peter Piper picked a peck of Pampered Pet products.) It includes information about a contact person at the company.

The following table summarizes the Vendor entity's fields.

| **Field** | **Req'd?** | **Data Type** | **Domain** |
| --- | --- | --- | --- |
| CompanyName | Yes | String | Any company name. |
| ContactFirstName | Yes | String | Any first name. |
| ContactLastName | Yes | String | Any last name. |
| Street | Yes | String | Any street name and number. Not validated. |
| City | Yes | String | Any city name. Not validated? |
| State | Yes | String | Foreign key to States table. |
| Zip | Yes | String | Valid ZIP Code. Not validated? |
| ContactEmail | No | String | Valid email address. |
| ContactPhone | Yes | String | Valid 10-digit phone number. |
| Notes | No | String | Miscellaneous instructions and notes. |

[Chapter 12](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/ch12.html)

Exercise 1 Solution

Food items could be treated like any other inventory item, although their expiration dates would probably be much shorter. Some items might not even be counted in inventory if they expire quickly. For example, the database will need an entry for coffee so you can add one to an order but there's no point trying to update the QuantityInStock every time someone makes a new pot.

Exercise 2 Solution

An easy solution would be to add a new Certifications attribute to the EMPLOYEE class listing the courses that the employee can teach. This would be a foreign key field referring to COURSE classes. In the ER model, the Employee entity would have a new relationship with the Course entity. This would be a moderately hard change but probably doable.

Alternatively you could create a new Instructor subclass that inherits from Employee. This would require creating a new class/entity so it would be harder.

Exercise 3 Solution

Add a new StoreId attribute to the Order entity. That part wouldn't be too hard. At a minimum, you would also need to add a Store entity to look up allowed store IDs. That would be a little harder. If you also want to store real information about each store, such as an Address (which would require a link to the Addresses table), the change would be a lot harder.

Exercise 4 Solution

You could add a link between the Course entity and the Address entity. This wouldn't be too hard but it does require a new relationship so it would be harder than adding a new attribute to the Course entity.

Exercise 5 Solution

It would be easy to store these as Course entities with a Price of $0. The Pampered Pet could advertise them just like any other course. Probably no one would care if people attended without creating Customer entities.

Exercise 6 Solution

Adding more addresses to an order would make the Order/Address relationship many-to-many. You would need to add an intermediate table to represent the Order/Address pairs and replace the existing one-to-many relationship with two new one-to-many relationships. This would be a fairly difficult change.

Exercise 7 Solution

The easy solution would be to add a Phone attribute to the Order entity. However, [Figure 12-9](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/ch12.html#figure_12-9) shows that the design already has a Phone entity associated with the Person entity. Rather than creating a new attribute, it would be slightly more complicated but more flexible to reuse the Phone entity.

Before doing any of this, however, it would be worth asking the customers whether they will ever need to allow multiple phone numbers for an order. After all, they're adding one and there's nothing to stop them from adding another, particularly because the Person entity already allows any number of phone numbers.

Unfortunately adding multiple phone numbers to the Order entity would create a many-to-many relationship (one order can have many phone numbers and one phone number might be used to place any number of orders, probably by the same customer). To implement this, you would need to make an OrderPhone entity and two new one-to-many relationships. That would be a much harder change than simply adding a new Phone attribute to the Order entity.

Exercise 8 Solution

The obvious solution is to add a new Department attribute to the InventoryItem entity. However, that would create a functional dependency in that entity's attributes. InventoryItem already has a ShelfLocation field that tells where the item is when it is on display in the store. That location is in some department so adding a new Department attribute would partially duplicate that data and that could lead to inconsistent data. For example, an item could be listed as shelved in the Fish department but its Department field could be set to Reptile.

A better solution would be to make a new Departments entity that maps ShelfLocation values to departments. This requires adding a new table and a new relationship between InventoryItem and Departments so it would be fairly difficult.

Exercise 9 Solution

This would require a couple of changes. First, you would need to add effective date attributes to the Address entity. You would also need to change the user interface significantly to let the user decide which of a customer's addresses to use for any given operation. If the program simply uses the address that was in effect when an order was placed, that might be manageable.

Overall, however this change seems like a lot of trouble and the need is so unclear that I would ask the customers why they wanted to do this and try to talk them out of it if they don't have a good reason.

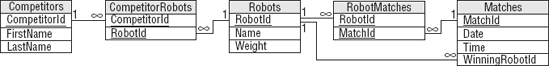
Exercise 10 Solution

The discount applied to an order would need to be recorded so the simplest solution would be to add a new Discount attribute to the Order entity.

More complicated solutions could track types of discounts to ensure consistency. Then, for example, the employee entering an order would enter a coupon or discount code rather than the actual discount percentage so entering in an incorrect discount would be less common. This solution would require creating a new Discounts entity and a relationship between it and the Order entity so it would be a more complicated solution.

Exercise 11 Solution

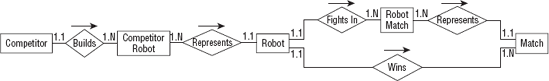
[Figure A-30](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-30) shows one possible solution. It uses the CompetitorRobot entity to implement the Competitor/Robot relationship and it uses the RobotMatch entity to implement the Robot/Match relationship.



**Figure A.30. Figure A-30**

Exercise 12 Solution

[Figure A-31](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-31) shows one possible relational design.



**Figure A.31. Figure A-31**

[Chapter 13](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/ch13.html)

Exercise 1 Solution

The following list describes the primary keys and required fields for each table:

* **Competitors**: —The CompetitorId field is the primary key. Other required fields are FirstName, LastName, Street, City, State, Zip.
* **CompetitorRobots**: —This is an intermediate table. Its primary key includes both of its fields. Both of its fields are also foreign key constraints to other tables so they are completely constrained.
* **Robots**: —The RobotId field is the primary key. Name and Class must be required in order to compete. The Weight, MaxSpeed, and Chassis fields could also be required.
* **RobotMatches**: —This is an intermediate table. Its primary key includes both of its fields. Both of its fields are also foreign key constraints to other tables so they are completely constrained.
* **Matches**: —The MatchId field is the primary key. Date, Time, and Location are also required. WinningRobotId cannot be required because the Matches record will probably be created before the match occurs and at that time the winner isn't known (unless it's a fixed fight).
* **RobotWeapons**: —This table lists the weapons that are built into each robot (chainsaw, axe, grapple, laser cannon). Both of its fields are part of the primary key so both are required. The RobotId field is a foreign key constraint referring to the Robots table so it is completely constrained.

Exercise 2 Solution

The following list describes sanity checks for each table:

* **Competitors**: —Zip should have a valid ZIP Code format similar to either 12345 or 12345-6789.
* **CompetitorRobots**: —None.
* **Robots**: —If present, Weight > 0 and Weight < 10,000 lbs. If present, Speed > = 0 and Speed < 30 mph.
* **RobotMatches**: —None.
* **Matches**: —When created, Date > = today. Time > 8am and Time < 11pm.
* **RobotWeapons**: —None.

Exercise 3 Solution

The following list describes lookup tables for each table's fields:

* **Competitors**: —You could build a full City/State/Zip lookup table but it would be big and hard to maintain. You could use the trick described in this chapter of using a table to validate common City/State/Zip values but allow values not in the table, but competitors in Robot Wars come from all over the country so there's no good list of the most likely City/State/Zip combinations.
* **CompetitorRobots**: —Both of this table's fields are used in foreign key constraints already.
* **Robots**: —Chassis should be one of 4 Wheel, 6 Wheel, Tank Tread, Hovercraft, and so forth. Class should be one of Light, Medium, Heavy, Under $1000, and so forth. The allowed values should be moved into new Chasses and Classes lookup tables.
* **RobotMatches**: —Both of this table's fields are used in foreign key constraints already.
* **Matches**: —Location should be one of Arena 1, Arena 2, Pond, and so forth. These values should be added to a Locations lookup table.
* **RobotWeapons**: —WeaponType should be one of Chainsaw, Axe, Rail Gun, Plasma Cannon, and so forth. Those values should be added to a WeaponTypes table.

Exercise 4 Solution

The three somewhat more complicated business rules that I thought of that really should be implemented in some manner are:

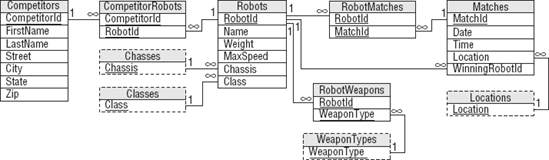
* Two matches should not be scheduled for the same place at the same time. This can be implemented as a uniqueness constraint on the Matches table's combined Date/Time/Location values. (This assumes the matches fit in nice time slots so we don't need to worry about them overlapping.)
* A robot should not be scheduled for two matches at the same time.
* Because competitors must control their robots during a match, none of a competitor's robots should not be scheduled for two matches at the same time. (If two robots share multiple co-owners, the team could split up and be in two matches at once but that would make the database just plain ugly. If that sort of change is required, you'll be glad you provided this check in a stored procedure, a middle tier, or some other place that's reasonably easy to change.)

Some other possible rules that I thought of include:

* A competitor can have no more than one robot in each match.
* A competitor can have no more than one robot in each class.
* A robot can have no more than two weapons.
* Weight, speed, chassis, and weapons could be part of what determines class. For example, classes could include Heavy, Light & Fast, Wheeled, or Single Weapon. Those definitions would be complicated and would probably change regularly.
* How the matches are assigned could be part of a set of business rules. For example, it could be single elimination, double elimination (if a robot can be repaired), winners and losers brackets, or a giant brawl.

Exercise 5 Solution

[Figure A-32](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-32) shows the new relational model with the lookup tables added.



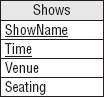
**Figure A.32. Figure A-32**

[Chapter 14](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/ch14.html)

Exercise 1 Solution

This table isn't in 1NF because it contains two columns that hold multiple values. The Show column holds the names of all shows at a particular venue and the Times column holds all of the times for shows at a location.

[Figure A-33](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-33) shows a relational design that stores this data in 1NF.



**Figure A.33. Figure A-33**

The following table shows the data in this new format.

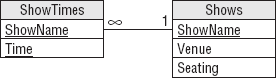
| **ShowName** | **Time** | **Venue** | **Seating** |
| --- | --- | --- | --- |
| Sherm's Shark Show | 11:15 | Sherman's Lagoon | 375 |
| Sherm's Shark Show | 3:00 | Sherman's Lagoon | 375 |
| Meet the Rays | 1:15 | Sherman's Lagoon | 375 |
| Meet the Rays | 6:00 | Sherman's Lagoon | 375 |
| Deb's Daring Dolphins | 11:00 | Peet Amphitheater | 300 |
| Deb's Daring Dolphins | 12:00 | Peet Amphitheater | 300 |
| Deb's Daring Dolphins | 6:30 | Peet Amphitheater | 300 |
| The Walter Walrus Comedy Hour | 2:00 | Peet Amphitheater | 300 |
| The Walter Walrus Comedy Hour | 5:27 | Peet Amphitheater | 300 |
| Flamingo Follies | 2:00 | Ngorongoro Wash | 413 |
| Wonderful Waterfowl | 3:00 | Ngorongoro Wash | 413 |

This table contains so much redundant information that there must be something wrong with it.

Exercise 2 Solution

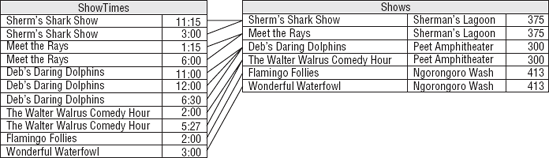
The solution to Exercise 1 isn't in 2NF because some non-key fields depend on only some of the primary key fields. A particular show only occurs in one location (it would be hard to move the dolphins to different amphitheaters for different shows) so the Venue and Seating fields depend only on Show and not on Time.

The solution is to move the Venue and Seating data into a new table connected to the original table by the ShowName. Because the original table now only holds show time information, I'm going to rename it ShowTimes and call the new table Shows. [Figure A-34](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-34) shows the result.



**Figure A.34. Figure A-34**

[Figure A-35](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-35) shows the new tables holding their data.



**Figure A.35. Figure A-35**

Exercise 3 Solution

The solution to Exercise 2 isn't in 3NF because the Shows table contains a transitive dependency: the Seating field is determined by the Venue field. In the original table, the dependency isn't obvious because the same Venue and Seating values are not repeated. In [Figure A-35](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-35) the problem is shown by the repeated Venue/Seating pairs in the Shows table.

The solution is to move the seating information into a new table to match venues with their capacities. The new table should use the Venue field to link back to the Shows table. Because this table describes the venues, I'll call it Venues. (Clever, huh?) [Figure A-36](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-36) shows the new design.

Figure A-36

**Figure A.36. Figure A-36**

[Figure A-37](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-37) shows the data in the new tables.

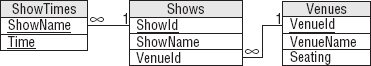


**Figure A.37. Figure A-37**

Exercise 4 Solution

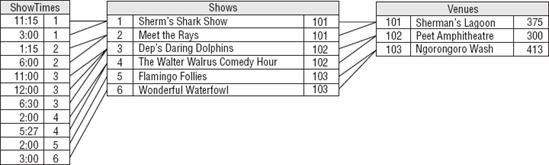
Changing show names, time, or venue names is difficult for the design shown in [Figure A-36](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-36) because those fields are used as primary keys. To increase the database's flexibility, all you need to do is make artificial keys (ID numbers) for the tables. Because the ShowName was only in the ShowTimes table to provide a link to the Shows table, it is no longer needed in ShowTimes. Similarly the Venue field in the Shows table was only there to link to the Venus table so Venue is no longer needed in the Shows table.

[Figure A-38](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-38) shows the more flexible design.



**Figure A.38. Figure A-38**

[Figure A-39](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html#figure_a-39) shows the data in the new tables. The ShowId values are between 1 and 6 and the VenueId values are between 101 and 103 so it's easy to see which are which.



**Figure A.39. Figure A-39**

Notice that the tables contain no repeated data other than their ID values so you can easily change a show's name or time, or a venue's name.

[Chapter 15](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/ch15.html)

These exercises ask you to do something and there's no reasonable way to put answers for them here. If you visit the book's Web site at wrox.com, you can download the Access database I created for these exercises and then you can compare them to your own.

[Chapter 16](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/ch16.html)

These exercises ask you to do something and there's no reasonable way to put answers for them here. If you visit the book's Web site, you can download the MySQL Workbench solutions and the database creation scripts I built for these exercises, and then you can compare them to your own.

[Chapter 17](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/ch17.html)

Exercise 1 Solution

The following code creates the Venues, Shows, and ShowTimes tables. Note that you must create the tables in this order because you cannot create a foreign key constraint that refers to a table that doesn't yet exist.

CREATE TABLE Venues(

VenueId INT NOT NULL,

VenueName VARCHAR(45) NOT NULL,

Seating INT NOT NULL,

PRIMARY KEY (VenueId)

);

CREATE TABLE Shows(

ShowId INT NOT NULL,

ShowName VARCHAR(45) NOT NULL,

VenueId INT NOT NULL,

PRIMARY KEY (ShowId),

INDEX fk\_Shows\_Venues (VenueId),

CONSTRAINT fk\_Shows\_Venues

FOREIGN KEY (VenueId)

REFERENCES Venues (VenueId)

ON DELETE NO ACTION

ON UPDATE NO ACTION

);

CREATE TABLE ShowTimes(

ShowId INT NOT NULL,

Time TIME NOT NULL,

PRIMARY KEY (ShowId, Time),

INDEX fk\_ShowTimes\_Shows (ShowId),

CONSTRAINT fk\_ShowTimes\_Shows

FOREIGN KEY (ShowId)

REFERENCES Shows (ShowId)

ON DELETE NO ACTION

ON UPDATE NO ACTION

);

Exercise 2 Solution

The following code inserts the values for the aquarium show schedule. Note that the statements must insert data in tables used as foreign key constraints before inserting the values that refer to them. For example, the statement that creates the Sherman's Lagoon Venues record must come before the Sherm's Shark Show record that refers to it.

Also note that some of the text contains an apostrophe so that text is delimited by double quotes instead of apostrophes. For example, the text "Sherman's Lagoon" contains an apostrophe. Alternatively you could double-up the apostrophes to indicate characters that are part of the text value as in 'Sherman' 's Lagoon' (here ' ' are two apostrophes, not a double quote).

INSERT INTO Venues VALUES (101, "Sherman's Lagoon", 375);

INSERT INTO Venues VALUES (102, "Peet Amphitheater", 300);

INSERT INTO Venues VALUES (103, "Ngorongoro Wash", 413);

INSERT INTO Shows VALUES (1, "Sherm's Shark Show", 101);

INSERT INTO Shows VALUES (2, "Meet the Rays", 101);

INSERT INTO Shows VALUES (3, "Deb's Daring Dolphins", 102);

INSERT INTO Shows VALUES (4, "The Walter Walrus Comedy Hour", 102);

INSERT INTO Shows VALUES (5, "Flamingo Follies", 103);

INSERT INTO Shows VALUES (6, "Wonderful Waterfowl", 103);

INSERT INTO ShowTimes VALUES (1, "11:15");

INSERT INTO ShowTimes VALUES (1, "15:00");

INSERT INTO ShowTimes VALUES (2, "13:15");

INSERT INTO ShowTimes VALUES (2, "18:00");

INSERT INTO ShowTimes VALUES (3, "11:00");

INSERT INTO ShowTimes VALUES (3, "12:00");

INSERT INTO ShowTimes VALUES (3, "18:30");

INSERT INTO ShowTimes VALUES (4, "14:00");

INSERT INTO ShowTimes VALUES (4, "17:27");

INSERT INTO ShowTimes VALUES (5, "14:00");

INSERT INTO ShowTimes VALUES (6, "15:00");

Exercise 3 Solution

The following statement updates the Flamingo Follies time:

UPDATE Shows, ShowTimes SET Time = "14:15"

WHERE Shows.ShowId = ShowTimes.ShowId

AND Time= "14:00"

AND ShowName = "Flamingo Follies";

The following statement updates the Sherm's Shark Show time:

UPDATE Shows, ShowTimes SET Time = "15:15"

WHERE Shows.ShowId = ShowTimes.ShowId

AND Time= "15:00"

AND ShowName = "Sherm's Shark Show";

Exercise 4 Solution

The following code produces the desired result in MySQL:

SELECT

ShowName AS "Show",

LPAD(DATE\_FORMAT(Time, "%l:%i %p"), 8, " ") AS Time,

VenueName AS Location

FROM Shows, ShowTimes, Venues

WHERE Shows.ShowId = ShowTimes.ShowId

AND Shows.VenueId = Venues.VenueId

ORDER BY "Show", TIME(Time);

Exercise 5 Solution

The following code produces the desired result in MySQL:

SELECT

LPAD(DATE\_FORMAT(Time, "%l:%i %p"), 8, " ") AS Time,

ShowName AS "Show",

VenueName AS Location

FROM Shows, ShowTimes, Venues

WHERE Shows.ShowId = ShowTimes.ShowId

AND Shows.VenueId = Venues.VenueId

ORDER BY TIME(Time), "Show";

[Chapter 18](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/ch18.html)

Exercise 1 Solution

One order in which you could build these tables is: MpaaRatings, Genres, Movies, Persons, MovieProducers, MovieDirectors, RoleTypes, MovieActors.

Exercise 2 Solution

The following code shows one possible SQL script for creating the movie database:

CREATE DATABASE MovieDb;

USE MovieDb;

CREATE TABLE MpaaRatings (

MpaaRaiting VARCHAR(5) NOT NULL,

Description VARCHAR(40) NOT NULL,

PRIMARY KEY (MpaaRaiting)

);

CREATE TABLE Genres (

Genre VARCHAR(10) NOT NULL,

Description VARCHAR(40) NOT NULL,

PRIMARY KEY (Genre)

);

CREATE TABLE Movies (

MovieId INT NOT NULL AUTO\_INCREMENT,

Title VARCHAR(40) NOT NULL,

Year INT NOT NULL,

MpaaRating VARCHAR(5) NOT NULL,

Review TEXT NULL,

NumStars INT NULL,

Minutes INT NOT NULL,

Description TEXT NULL,

Genre VARCHAR(10) NULL,

TrailerUrl VARCHAR(255) NULL,

PRIMARY KEY (MovieId),

INDEX FK\_Movies\_Ratings (MpaaRating ASC),

INDEX FK\_Movies\_Genres (Genre ASC),

CONSTRAINT FK\_Movies\_Ratings

FOREIGN KEY (MpaaRating)

REFERENCES MovieDb.MpaaRatings (MpaaRaiting)

ON DELETE NO ACTION

ON UPDATE NO ACTION,

CONSTRAINT FK\_Movies\_Genres

FOREIGN KEY (Genre)

REFERENCES MovieDb.Genres (Genre)

ON DELETE NO ACTION

ON UPDATE NO ACTION

);

CREATE TABLE Persons (

PersonId INT NOT NULL AUTO\_INCREMENT,

FirstName VARCHAR(40) NOT NULL,

LastName VARCHAR(40) NOT NULL,

PRIMARY KEY (PersonId)

);

CREATE TABLE MovieProducers (

MovieId INT NOT NULL,

PersonId INT NOT NULL,

PRIMARY KEY (MovieId, PersonId),

INDEX FK\_Producers\_Persons (PersonId ASC),

INDEX FK\_Producers\_Movies (MovieId ASC),

CONSTRAINT FK\_Producers\_Persons

FOREIGN KEY (PersonId)

REFERENCES MovieDb.Persons (PersonId)

ON DELETE NO ACTION

ON UPDATE NO ACTION,

CONSTRAINT FK\_Producers\_Movies

FOREIGN KEY (MovieId)

REFERENCES MovieDb.Movies (MovieId)

ON DELETE NO ACTION

ON UPDATE NO ACTION

);

CREATE TABLE MovieDirectors (

MovieId INT NOT NULL,

PersonId INT NOT NULL,

PRIMARY KEY (MovieId, PersonId),

INDEX FK\_Directors\_Persons (PersonId ASC),

INDEX FK\_Directors\_Movies (MovieId ASC),

CONSTRAINT FK\_Directors\_Persons

FOREIGN KEY (PersonId)

REFERENCES MovieDb.Persons (PersonId)

ON DELETE NO ACTION

ON UPDATE NO ACTION,

CONSTRAINT FK\_Directors\_Movies

FOREIGN KEY (MovieId)

REFERENCES MovieDb.Movies (MovieId)

ON DELETE NO ACTION

ON UPDATE NO ACTION

);

CREATE TABLE RoleTypes (

RoleType VARCHAR(40) NOT NULL,

PRIMARY KEY (RoleType)

);

CREATE TABLE MovieActors (

MovieId INT NOT NULL,

PersonId INT NOT NULL,

CharacterName VARCHAR(40) NOT NULL,

RoleType VARCHAR(40) NULL,

Review TEXT NULL,

NumStars INT NULL,

PRIMARY KEY (MovieId, PersonId, CharacterName),

INDEX FK\_Actors\_Persons (PersonId ASC),

INDEX FK\_Actors\_RoleTypes (RoleType ASC),

INDEX FK\_Actors\_Movies (MovieId ASC),

CONSTRAINT FK\_Actors\_Persons

FOREIGN KEY (PersonId)

REFERENCES MovieDb.Persons (PersonId)

ON DELETE NO ACTION

ON UPDATE NO ACTION,

CONSTRAINT FK\_Actors\_RoleTypes

FOREIGN KEY (RoleType)

REFERENCES MovieDb.RoleTypes (RoleType)

ON DELETE NO ACTION

ON UPDATE NO ACTION,

CONSTRAINT FK\_Actors\_Movies

FOREIGN KEY (MovieId)

REFERENCES MovieDb.Movies (MovieId)

ON DELETE NO ACTION

ON UPDATE NO ACTION

);

DROP DATABASE MovieDb;

[Chapter 19](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/ch19.html)

Exercise 1 Solution

The following table shows a backup schedule. In this case, you have time for a full backup every night so you may as well use it.

| **Night** | **Off-Peak Start** | **Off-Peak End** | **Off-Peak Hours** | **Backup Type** |
| --- | --- | --- | --- | --- |
| Monday | 11:00pm | 4:00am | 5 | Full |
| Tuesday | 11:00pm | 4:00am | 5 | Full |
| Wednesday | 11:00pm | 4:00am | 5 | Full |
| Thursday | 11:00pm | 4:00am | 5 | Full |
| Friday | 11:00pm | 6:00am | 7 | Full |
| Saturday | 9:00pm | 6:00am | 9 | Full |
| Sunday | 9:00pm | 4:00am | 7 | Full |

Exercise 2 Solution

The following table shows a new backup schedule. Now you can make a full backup only on Saturday night and on other nights you only have time for an incremental backup of changes since the previous night's backup.

| **Night** | **Off-Peak Start** | **Off-Peak End** | **Off-Peak Hours** | **Backup Type** |
| --- | --- | --- | --- | --- |
| Monday | 12:00am | 3:00am | 3 | Incremental from Sunday |
| Tuesday | 12:00am | 3:00am | 3 | Incremental from Monday |
| Wednesday | 12:00am | 3:00am | 3 | Incremental from Tuesday |
| Thursday | 12:00am | 3:00am | 3 | Incremental from Wednesday |
| Friday | 12:00am | 5:00am | 5 | Incremental from Thursday |
| Saturday | 10:00pm | 5:00am | 7 | Full |
| Sunday | 10:00pm | 3:00am | 5 | Incremental from Saturday |

This backup schedule is pretty full so you should probably start thinking about other strategies to use if your database continues to grow. For example, you might need to perform some backups during peak hours (naturally during the "off-peak" peak hours) or you could partition the database so areas handling different time zones are stored separately so you can back them up separately.

[Chapter 20](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/ch20.html)

Exercise 1 Solution

An order entry clerk doesn't need to read or update any existing order records so you don't need to set privileges for individual fields in the Orders or OrderItems tables. (Although you may want the user interface program to read previous orders so it can copy their values.)

The clerk will need to read existing Customers records for existing customers and create Customers records for new customers. Many applications also allow the clerk to update customer data when creating a new order so the clerk needs Update access to the Customers table. The clerk should not change the CustomerId field, however, because that would disconnect the customer from previous orders. (In general, you should not update primary key values because that causes this kind of problem.)

The clerk needs Read access to the InventoryItems table to select the items that the customer wants to buy. (If there isn't enough inventory, assume the clerk creates the order anyway and sets the order's status to Back Order.)

The clerk also needs Read access to the OrderStatuses table to pick an initial status.

The following table lists the privileges that an order entry clerk needs for each table.

| **Table or Field** | **Privileges** |
| --- | --- |
| Customers | C |
| CustomerId | R |
| FirstName | RU |
| LastName | RU |
| Street | RU |
| City | RU |
| State | RU |
| Zip | RU |
| Phone | RU |
| CreditCardType | RU |
| CreditCardNumber | RU |
| Orders | C |
| OrderId | − |
| CustomerId | − |
| OrderDate | − |
| ShippedDate | − |
| OrderStatus | − |
| OrderItems | C |
| OrderId | − |
| SequenceNumber | − |
| ItemId | − |
| Quantity | − |
| InventoryItems | − |
| ItemId | R |
| Description | R |
| Price | R |
| QuantityInStock | R |
| OrderStatuses | − |
| OrderStatus | R |

The following SQL statements create an order entry clerk with appropriate privileges:

CREATE USER EntryClerk IDENTIFIED BY 'secret';

- - Revoke all privileges for the user.

REVOKE ALL PRIVILEGES, GRANT OPTION FROM EntryClerk;

- - Grant needed privileges.

GRANT INSERT, SELECT ON ShippingDb.Customers TO EntryClerk;

GRANT UPDATE (FirstName, LastName, Street, City, State, Zip, Phone,

CreditCardType, CreditCardNumber)

ON ShippingDb.Customers TO EntryClerk;

GRANT INSERT ON ShippingDb.Orders TO EntryClerk;

GRANT INSERT ON ShippingDb.OrderItems TO EntryClerk;

GRANT SELECT ON ShippingDb.InventoryItems TO EntryClerk;

GRANT SELECT ON ShippingDb.OrderStatuses TO EntryClerk;

Exercise 2 Solution

A customer service clerk must be able to read everything to give information about an existing order. This clerk doesn't need to create records but needs to be able to update and delete Orders and OrderItems records for orders that have not yet shipped.

Though the clerk can update Customers data, the CustomerId should never change because that would disconnect it from previous orders.

Note that it doesn't make sense for the clerk to update Orders data. Changing OrderId would disconnect the items from the order, changing CustomerId would disconnect the order from the customer, changing OrderDate would be revising history (popular with politicians but not a good business practice), and changing ShippedDate and OrderStatus is the shipping clerk's job.

This clerk should also not be able to change an OrderItems record's OrderId value because it would disconnect the item from the order.

Whether the clerk can delete Customers records is a business rule. In this case, assume the clerk cannot delete customers so you don't need to worry about old orders without corresponding Customers records.

Finally, whether the clerk can update OrderItems records or should just delete old records and create new ones is another business rule. In this case, it will probably be easier for the user interface application to delete the old records and create new ones so the clerk needs Create, Read, and Delete privileges for the OrderItems table.

The following table lists the privileges that a customer service clerk needs for each table.

| **Table or Field** | **Privileges** |
| --- | --- |
| Customers | − |
| CustomerId | R |
| FirstName | RU |
| LastName | RU |
| Street | RU |
| City | RU |
| State | RU |
| Zip | RU |
| Phone | RU |
| CreditCardType | RU |
| CreditCardNumber | RU |
| Orders | D |
| OrderId | R |
| CustomerId | R |
| OrderDate | R |
| ShippedDate | R |
| OrderStatus | R |
| OrderItems | CD |
| OrderId | R |
| SequenceNumber | R |
| ItemId | R |
| Quantity | R |
| InventoryItems | − |
| ItemId | R |
| Description | R |
| Price | R |
| QuantityInStock | R |
| OrderStatuses | − |
| OrderStatus | R |

The following SQL statements create a customer service clerk with appropriate privileges:

CREATE USER ServiceClerk IDENTIFIED BY 'secret';

- - Revoke all privileges for the user.

REVOKE ALL PRIVILEGES, GRANT OPTION FROM ServiceClerk;

- - Grant needed privileges.

GRANT SELECT ON ShippingDb.Customers TO ServiceClerk;

GRANT UPDATE (FirstName, LastName, Street, City, State, Zip, Phone,

CreditCardType, CreditCardNumber)

ON ShippingDb.Customers TO ServiceClerk;

GRANT SELECT, DELETE ON ShippingDb.Orders TO ServiceClerk;

GRANT INSERT, SELECT, DELETE ON ShippingDb.OrderItems TO ServiceClerk;

GRANT SELECT ON ShippingDb.InventoryItems TO ServiceClerk;

GRANT SELECT ON ShippingDb.OrderStatuses TO ServiceClerk;

Exercise 3 Solution

The inventory manager's main task is to order new inventory and maintain the InventoryItems table. That requires Create, Read, Update, and Delete privileges on that table.

To change an order's status from Back Ordered to Ordered, the inventory manager must look in the Orders table to find orders in the Back Ordered status, look up the items for that order, and see if there is now enough inventory to fulfill the order. That means the manager must be able to look at the Orders table's OrderId and OrderStatus fields, and update the OrderStatus field. The manager must also be able to look at the OrderItems table's OrderId, ItemId, and Quantity fields.

The following table lists the privileges that an inventory manager needs for each table.

| **Table or Field** | **Privileges** |
| --- | --- |
| Customers | − |
| CustomerId | − |
| FirstName | − |
| LastName | − |
| Street | − |
| City | − |
| State | − |
| Zip | − |
| Phone | − |
| CreditCardType | − |
| CreditCardNumber | − |
| Orders | − |
| OrderId | R |
| CustomerId | − |
| OrderDate | − |
| ShippedDate | − |
| OrderStatus | RU |
| OrderItems | − |
| OrderId | R |
| SequenceNumber | − |
| ItemId | R |
| Quantity | R |
| InventoryItems | CD |
| ItemId | R |
| Description | RU |
| Price | RU |
| QuantityInStock | RU |
| OrderStatuses | − |
| OrderStatus | R |

The following SQL statements create a customer service clerk with appropriate privileges:

CREATE USER ServiceClerk IDENTIFIED BY 'secret';

- - Revoke all privileges for the user.

REVOKE ALL PRIVILEGES, GRANT OPTION FROM ServiceClerk;

- - Grant needed privileges.

GRANT SELECT ON ShippingDb.Customers TO ServiceClerk;

GRANT UPDATE (FirstName, LastName, Street, City, State, Zip, Phone,

CreditCardType, CreditCardNumber)

ON ShippingDb.Customers TO ServiceClerk;

GRANT SELECT, DELETE ON ShippingDb.Orders TO ServiceClerk;

GRANT INSERT, SELECT, DELETE ON ShippingDb.OrderItems TO ServiceClerk;

GRANT SELECT ON ShippingDb.InventoryItems TO ServiceClerk;

GRANT SELECT ON ShippingDb.OrderStatuses TO ServiceClerk;

* [**Copy**](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html)
* [**Add Highlight**](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html)
* [**Add Note**](https://www.safaribooksonline.com/library/view/Beginning+Database+Design+Solutions/9780470385494/apa.html)