CHAPTER 2

Relational databases

2.1 A brief history of SQL and relational databases

Databases have served a central role in civilization for millennia and, at its core, anything which stores information persistently is a form of database. Cave paintings, stone tablets, and papyrus are all valid examples of a database, but they each have their own critical shortcomings. The history of databases is essentially motivated by a quest to build systems, which can simultaneously meet the demands of accurately and durably storing information over long periods of while also allowing users to effectively search for information from among the stored data. The advent of computers unleashed new horizons for database technology, allowing information stored on disk to be rapidly created and accessed; however, simply having a computerized system does not automatically make data powerfully interactive. In fact, it was a parallel and powerful achievement in engineering to create a system of representing information, which would allow for much more than just targeted retrieval but also for the creation of an expressive language interrogating stored information.

Although the terms “SQL” and “relational database” are often used interchangeably, they are quite distinct entities. SQL or the “structured query language” is an implementation of the underlying relational model, which is a mathematical and logical format for how data should be represented such that they can be effectively queried. The relational model thus preceded its implementation in SQL and was the product of computer scientist and mathematician Edgar F. Codd, which he first described in 1969.[1] This model was deeply influential and the formal implementation of the relational model was led by Donald Chamberlain and Raymond Boyce at IBM who, after learning about it, set about the creation of the Structured English Query Language, at that time called SEQUEL, in the early 1970s.[2] Later in the decade, a company by the name of Relational Software Inc. (now Oracle Corporation) introduced the first commercially available implementation of SQL, which they dubbed “Oracle V2.” The applications to business intelligence were clear and, by 1986, SQL had become an official and organized language implemented almost ubiquitously in data products. There are several mission critical and extremely popular modern relational database products such as PostgreSQL, MySQL, Oracle, Microsoft SQL Server, a testament to the lasting value of SQL as a way to interrogate stored information. Almost invariably, hospital data warehouses rely upon some form of relational database, typically one of the aforementioned products. SQL even persists in modern Big Data technologies such as Spark, which we discuss later in Chapter 20.

An Introduction to Healthcare Informatics. https://doi.org/10.1016/B978-0-12-814915-7.00002-8
© 2020 Elsevier Inc. All rights reserved.

2.2 Overview of the relational model

The core of the relational model is, unsurprisingly, the "relation," which is a conceptual schematic for representing information that consists of "tuples" of data and "headers" that describe the elements of those tuples. In order to unpack this, let's start with tuples: a tuple is data structure that resembles a list with the constraint that each element must be unique (and as a convention, tuples are often represented with parentheses at either end). For example, a tuple consisting of four elements describing a patient might look like this:

```
('Peter', 'McCaffrey' 30, 'Male')
```

The tuple itself is a "record" and each element in the tuple is an attribute of that record providing information about something that record described. In this example, the patient tuple is a collection of four pieces of information and because each of these pieces of information resides within the same tuple, they all apply to the same thing, in this case the same patient. The concept of a "Relation" extends this notion in some important ways. Imagine for a moment that we have the four following tuples:

```
('Peter', 'McCaffrey', 30, 'Male)
('Hydrocodone', 5, 'mg', 'BID', 'PO')
('John', 'Smith', 25, 'Male')
('Paul', 'Jones', 'MD')
```

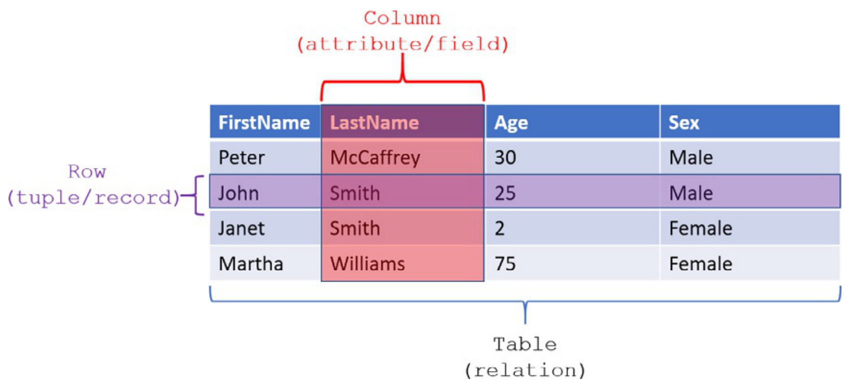
We can understand that the information *within* each of these tuples is connected in that it describes the same thing, but how do we organize these tuples to understand how they relate to one another and how do we know exactly what sort of attributes these tuples are intended to describe? The Relation solves this problem by linking a tuple of values to a tuple of attributes known as a header. We can thus turn the aforementioned tuples into relations by providing these attributes (in bold):

```
('FirstName', 'LastName',  'Age', 'Sex')
('Peter',     'McCaffrey', 30,    'Male')

('MedicationName', 'Dose', 'Units', 'Frequency', 'Route')
('Hydrocodone',     5,     'mg',    'BID',       'PO')

('FirstName', 'LastName',  'Age', 'Sex')
('John',      'Smith',     25,    'Male')

('FirstName', 'LastName',  'IdNumber', 'State')
('Paul',      'Jones',     210214,     'MD')
```

Now two powerful things have happened. First, we can determine, for an individual tuple, what it is intended to describe; we know that 30 corresponds to Peter's age and not something else and we know that 210214 corresponds to Paul's ID Number, not his age. Second, we know that the tuples describing Peter McCaffrey and John Smith are both talking about the same attributes. The power of Relations is that we can know this sort of depth about how records describe the world and we can aggregate records that describe the same kinds of things. A Relation in the strict sense is therefore a collection of one or more tuples, which share the same collection of attributes. Quite naturally, this lends itself to a familiar understanding of a Relation as a table with the attributes describing its columns and the record tuples describing its rows.

Perhaps more importantly is that this strict organization of data under a header and the resulting assignment of unambiguous meaning to each element in a record allows for meaningful logical statements to be applied to data contained within a relation. In other words, if we lacked this strict association between data and a header, we would not be able to formulate a question such as "return all patient records where the first name is Peter" because, lacking a clear way to determine whether the element "Peter" in a given record refers to a patient's first name or something else, we would not be able to tell for which records the query statement was true or false. In relational theory, this is referred to as describing a predicate over a set of predicate variables where a "predicate" is simply a function that can be evaluated as true or false (i.e., our query) and "predicate variables" are the elements of data in the record which would determine whether a predicate evaluates to true or false. While this might seem like a simple concept in this example, this logically consistent way to constrain how data are stored and retrieved is what allows the relational model—and its implementation in SQL—to reliably construct and execute rich analytical queries over billions of rows of data.

Putting all of this together, Fig. 2.1 depicts the connection between tuples, headers, and relations. Databases often contain anywhere from tens to potentially tens of thousands (in the case of some healthcare data warehouses) of relations. Typically, a single relation describes a single type of entity such as patients or encounters and each row or tuple within that relation describes an instance of that entity such as a particular patient or a particular encounter.



FirstName	LastName	Age	Sex
Peter	McCaffrey	30	Male
John	Smith	25	Male
Janet	Smith	2	Female
Martha	Williams	75	Female

Fig. 2.1
"Patients" table as a relation with a standard tuple of attributes and multiple tuples describing specific patients as rows.

As we will discuss in the next chapter covering the mechanics of SQL, any individual table may be specified as the target of a search query and through a process known as "joining" tables, multiple tables can be elegantly combined in the process of resolving a query to retrieve data.

One last but very important point to make with regard to the relational model is that it was designed from the very beginning to support a "declarative" formulation for queries and this declarative nature is an important strength of SQL as a language. In many conventional programming languages, the author writing statements in the language is concerned not just with the inputs and outputs of her program but also with the intricate internal details of how those inputs are handled and how outputs are generated. In other words, the user of the language takes significant responsibility for the execution of her code. The relational model was designed to fulfill the goal of supporting a purely declarative query language, which stood apart from the implementation details of the queries it describes. The mathematical underpinnings of relational theory could ensure that, as long as data were represented and queried in accordance with their premises, there would be logical integrity across many different approaches to execution of that query. As you will see later when we discuss SQL, composing SQL queries really amounts to describing the query output one desires without much focus on exactly how those queries are run or optimized. Many relational database products contain complex and sometimes proprietary internal "query planner," which handle many of a query's implementation details on the user's behalf. This flexibility is one of the reasons why SQL has retained its wide popularity and grown across so many platforms and products. It's worth mentioning that the existence of a query planner does not obviate the need to write queries thoughtfully as certain questions—especially those containing complex joins—can lead to inefficient query execution. Getting around this is a task both for query writing but also for database design itself.

2.3 Differences between the relational model and SQL

Although the relational model is the underpinning of SQL, the Query Language itself—and the practices of those who create and maintain databases–often does not completely adhere to the model's restrictions. That being said, it is far more important for you as an informaticist to be familiar with SQL as a language and as an interface to retrieving data from relational databases than it is for you to be a master of the relational model. Nonetheless, understanding a few of the most important differences between the model and the language will deepen your perspective as a user.

First and foremost is the fact that, in relational theory, every value in every tuple must be such that a predicate function can determine whether it is true or false. This means no NULL values as NULL values are logically ambiguous. In reality, NULL values are rampant in databases but, strictly speaking, they violate the logical integrity of relations. This is more than an esoteric aside as there is a long-standing discussion in the field of database theory as to what to do with NULL values stemming from the fact that NULL values are often coerced to false—that is, they are assumed to be equivalent to FALSE—even though they do not have such a logical meaning. In fact, Codd himself proposed an alternative, 3-value logic system that accounts for intermediacy between true and false although this is not widely implemented.[3]

Second, there is the fact that relations are formally defined as collections of unique tuples. That is, tables that truly capture relations shouldn't have any duplicate rows and yet this is very common in practice. The motivation in relational theory was that a predicate function should evaluate uniquely for each set of variables and, thus, having duplicate rows means that there are identical collections of predicate variables for which the predicate function (again, the query) redundantly evaluates the same. Having two identical rows each describing Peter McCaffrey in the patient's table is ambiguous because, without additional information, it is unclear *why* there are two identical rows. Could this mean two visits? We have no idea unless the relation contains some other attribute like a VisitID. Strictly identical tuples are like NULLs in the sense that they mask some real information. Relational database products don't impose this restriction though, and this is a very good example of the subtle differences between the academic and mathematical concepts of relational theory and the practical decisions made when designing a product for people to use.

There are also several additional departures, which exist to make SQL more convenient to write. As such, it is most accurate to describe modern SQL databases as "pseudo-relational," although most just call them "relational." Another important semantic detail is that, while SQL is a query language almost ubiquitously used for accessing and manipulating data within a relational database, many products like Microsoft SQL Server or MySQL are referred to as "Relational Database Management Systems" (RDBMS) because they themselves are a collection of features and user interfaces for the purpose of interacting with data stored (mostly) according to the relational model.

2.4 Primary and foreign keys

Having covered the relational model and the representation of data as tuples of information according to various attributes, it's appropriate to discuss another fundamental concept in relational databases: keys. Keys accomplish two critically important tasks. They ensure that rows are unique within tables and they provide associations between rows in different tables, allowing logic to be applied across more than one type of thing. In the next chapter we will discuss the SQL language itself and, in that discussion, we will cover JOINs as a way to query using data from multiple tables. This section serves as a good foundation for that discussion as JOINing would not be possible without the associations provided by keys. Before going further, however, let's introduce an additional Encounters relation in our database:

```
('EncounterID', 'Clinic', 'Date')
(1,             'Uptown', 10/6/2018)
```

```
(2,           'Uptown',   10/6/2018)
(2,           'Downtown', 12/10/2018)
(1,           'Downtown', 3/10/2019)
```

This relation contains four records and three attributes: the ID of the encounter itself, the clinic site where the encounter occurred, and the date of the encounter. Importantly, the Encounter ID is unique for each record, providing a way to individually identify encounters even if all of the other information is identical between rows. In this example, we have two encounters on 10/6/2018 at the Uptown clinic, but we can ensure these are unique rows by giving them each a unique Encounter ID. We could to the same with our Patients relation as well, granting each Patient a unique ID:

```
('PatientID', 'FirstName', 'LastName',  'Age', 'Sex')
(1,           'Peter',     'McCaffrey', 30,    'Male')
(2,           'John',      'Smith',     25,    'Male')
```

In both cases, we have provided each relation with a "primary key," which is an attribute or a collection of attributes sufficient to uniquely identify each row. It may be the case that some tables have preexisting attributes adequate for this purpose, as with our Patients relation where we could have used just the FirstName, LastName, or Age in order to uniquely identify each row. This is risky, though, because we don't know what future data our table may end up containing and, however unlikely, it is certainly possible that we could have another 30-year-old male patient named Peter McCaffrey. Therefore, it is common practice to create an attribute designated for this purpose that uniquely identifies each row, just like a row number as we have done here. For most relational database systems, users declare an attribute to be the primary key when they initially create a relation and this declaration enforces that all values in that attribute are unique and defined for each row (that is they are not NULL anywhere). Moreover, most relational database systems optimize searching based upon the primary key, which, if you have a table with millions or even billions of rows, can become quite important.

Now let's imagine that we wish to link encounters to the Patients who had those encounters. Thanks to the Primary Keys Encounter ID and Patient ID we have the foundations to do this because we can talk about specific encounters and specific patients. The missing link, however, comes when we need to add information to the records of one relation that associates it to the record of another. One easy way to do this is to add a PatientID attribute to the Encounters relation:

```
('EncounterID', 'Clinic',   'Date',      'PatientID')
(1,             'Uptown',   10/6/2018,    1)
(2,             'Uptown',   10/6/2018,    2)
(2,             'Downtown', 12/10/2018,   1)
(1,             'Downtown', 3/10/2019,    2)
```

Now we have the ability to determine which specific Patient was present for a particular encounter. In this capacity, the PatientID attribute serves as a "Foreign Key" within the Encounters relation. It does not have to be unique for each row in that relation—that's the job of the primary key—but it must correspond to a primary key in another table for which its values are unique. Thus, foreign keys serve the complementary purpose of referencing a primary key for another, associated relation. Almost invariably, relational databases contain multiple relations that describe things for which we would want to use their data together to describe a query. In our database, we have a Patients relation and an Encounters relation and we would often want to access tuples in one relation based upon information in another relation. Therefore, since encounters belong to patients and we want to maintain organization and association between encounters in the Encounters relation and the patients to whom those Encounters refer, then we need a Foreign Key to link these relations. Having this foreign key in place lets us unambiguously resolve a single patient to whom a single encounter refers using their linked Patient IDs—this is critical for relational databases to function properly.

This concept is technically termed "referential integrity" and is one of the core requirements of a functional relational database. Strictly speaking, referential integrity means that each foreign key in a relation must refer to a real primary key in another relation; otherwise, references made by that foreign key would be invalid. If there was a PatientID referenced in the Encounters relation to which encounters were assigned but there was no corresponding Patient in the Patients relation who had that ID, then we would have broken referential integrity.

As mentioned previously, Foreign Keys need not be unique and most often they are not. The Encounters relation, for example, would likely have many encounters for each patient as patients typically have multiple appointments and so there would be several tuples for which the PatientID attribute was identical. This is totally permissible for foreign keys but inadmissible for Primary Keys as each time a PatientID appears in the Encounters relation, it denotes one of potentially countless encounters for a given patient but each time a PatientID appears in the Patients relation, it must denote one unique and individual person. Likewise, the Encounter ID of the Encounters relation can correspond to a Foreign Key in other relations such as an Orders relation, which would often list several orders for a single encounter. Relational databases typically have many relations with complex linkages to one another. Fortunately, these databases will typically have documentation often including a schematic representation of all tables and their linkages to one another via keys, known as an "entity relationship diagram." If available, it is wise to request and gain access to such a representation to aid in your ability to navigate and explore these databases. Fig. 2.2 depicts an entity relationship diagram for our example database.

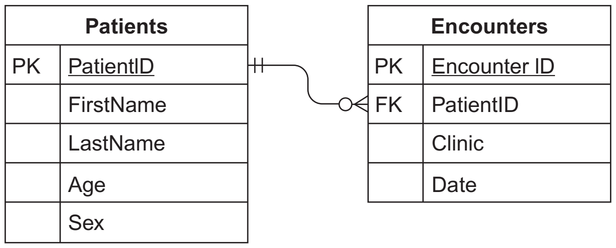


Fig. 2.2
Entity relationship diagram showing Patients and Encounters relations and their keys. Primary Keys are labeled "PK" and underlined. Foreign Keys are labeled "FK."

2.5 ACID and transactions with data

Now that we've covered the relational model and keys, which maintain the associations between relations, we need to cover another essential concept: state. As mentioned at the beginning of the chapter, databases are only valuable insofar as they preserve data with integrity. The "state" of database—that is, the particular information constituting the tuples of the relations in a database—must be preserved and consistent over time or else the database becomes a corrupted record. Relational databases aren't just records from which users read data but also records to which users write data and many mainstream relational databases universally support multiple users performing read and write operations simultaneously. This greatly enhances the value of these databases but also greatly increases the risk that the data they contain becomes inconsistent or incorrect.

Fortunately, this has been thought about extensively in the development of relational database technology and has culminated in the development of a collection of properties that form the acronym ACID and which comprise the core constraints that must exist around interactions with data in order to ensure that these interactions don't compromise the integrity of the database. Typically, these interactions are called "transactions," which begin with data in a given initial state, include one or more operations performed upon that data, and conclude with data in another state. We will cover each property briefly and then walk through some explanatory scenarios that should illustrate value of these properties. The ACID properties are as follows:

"Atomicity" means that the collections of mutations performed on the database in adding or updating information is an all-or-nothing process and that the state of the database will either remain as it was before the user performed any alterations or will be wholly converted to the state after the user performed all of their alterations. In either case, a transaction will not conclude with the state anywhere in between these two poles. To see why this is critically important, imagine a billing database that tracks claims and reimbursements. To completely record a reimbursement requires two important steps: a claim has to be declared to be reimbursed and removed from accounts receivable and money has to be added to the institution's bank account. If only one of these operations occurred, the database would be erroneous either regarding a bill as paid but not reflecting the payment in the hospital's account or in double dipping by recording the payment in the account but still marking the claim as unpaid and needing to be resubmitted to insurance. Therefore, in order for a claim to be fulfilled, it must be a single atomic mutation to both update the status of the claim and the amount of money in the account.

The remaining three ACID components are simpler to explain and can be covered more briefly. "Consistency" means that mutations made to a database must only occur in allowable ways. That is, these mutations must adhere to the constraints placed upon attributes and relations such as data types and uniqueness of primary keys must be enforced. For example, we should not be able to write a number in the FirstName attribute of the Patient's relation. "Isolation" means that the state of one transaction does not depend upon the state of another simultaneous transaction. If two users are both trying to update the value of an account and process a claim, they will each be operating within their own independent transaction. Recall that the intermediate steps of a transaction can create invalid states in the database, such as having a claim declared resolved before the payment is reflected in the account balance. Atomicity ensures that either both steps or no steps of this transaction will occur but Isolation ensures that simultaneous transactions do not see or work off of the intermediately inconsistent representations of data contained within each other's transactions. Lastly, "Durability" simply means that once a given state has been recorded for the database, data are preserved in that state unless operated upon by another transaction. After all, a database that could randomly lose or revert parts of its state would be hopelessly invalid. Many relational database products allow for multiple strategies through which to set up backups, fail overs, and redundancy in order to help improve durability.

Given all of these requirements, one might ask how database technology actually keeps transactions in order especially when working with multiple users. In reality, systems often record a log of all transactions, known as a "write-ahead log." The purpose of this log is to record each transaction and each SQL statement contained within each transaction before initiating any of those transactions against the actual database. While this delay is imperceptibly brief in practice, this serves as a sort of buffer to ensure that the state of the database at any point in time is not only valid at that moment but can also be reverted back to any previous point or recreated from its origin since every mutating operation performed on the database has been recorded in order in the log sequence.

2.6 Normalization

Before concluding, there is one last and critical aspect of relational databases that merits discussion. Heretofore, we have discussed relations, keys, and transactions through which data can be created and modified. One question a scrupulous reader may ask is why databases contain multiple tables at all and why everything isn't just one big table. In practice, you will likely find that operational systems have complex relationships between several relations each of which contains a few attributes, while data warehouses and analytical databases more often contain one or a few very large relations with many attributes. This is not by chance. Instead, there is a very specific design principle known as "normalization" that accounts for this schematic difference.

When Edgar Codd conceived of the relational model, he also put forth the concept of normalization in order to reduce the redundancy of specific records and improve the overall integrity of relational databases. Under this model, normalization proceeds in three successive forms from the first normal form through to the third normal form. In the first form, the domain of each attribute can contain only atomic values and that the values for that attribute are singular for each record. In other words, attributes cannot correspond to lists of other values. The second normal form extends this definition to include the requirements that every nonprime attribute of the relation—that is, any attribute which isn't the primary key and thus does not have to contain unique values—is dependent upon the primary key. In other words, the attribute must be determined by the primary key and, therefore, describe what the primary key identifies. If we have a patients table that contains attributes, which aren't dependent upon the PatientID and therefore, don't describe a particular patient, then normalization would remove them from the table. This is a large reason why many relational databases contain separate relations with only the attributes necessary to describe properties of that relation. Our patients relation should not contain attributes regarding the appointment date as any given patient can have many such dates. Instead, appointment date should be an attribute of an encounter relation. Lastly, third normal form stipulates that all attributes of a relation should be dependent upon the primary key and not upon each other. For example, in our patients table, we may include the date of birth or the age but to include both is to include columns, which are "transitively dependent" in that the value of one can be determined from the value of the other even without knowledge of the primary key.

These normal forms are difficult to grasp by description so we will walk through an example. First, let's add to our patients relation a phone number attribute and consider that a patient may easily have two or more phone numbers (note that the square brackets indicate a list of values):

```
('PatientID', 'First Name', 'Last Name', 'Age', 'Sex',  'PhoneNumber')
(1,           'Peter',      'McCaffrey', 30,    'Male', [1234567890, 4562223333])
```

Because Peter has two phone numbers, the "value" for the PhoneNumber attribute in that tuple is a list containing two items. In order to be in first normal form, we need to ensure that each attribute contains single values. To accomplish this, we may create an additional relation that associates phone numbers to PatientIDs like so:

```
('PhoneNumberID', 'PatientID', 'PhoneNumber')
(1,               1,           1234567890)
(2,               1,           4562223333)
```

Notice how, for this relation, the PatientID is a foreign key that refers to the associated primary key in the patients relation. Moreover, these two tuples contain the same PatientID since they belong to the same patient. Also note that we have a primary key for this relation, which we call PhoneNumberID and this does have to be unique for all tuples in this relation. As we see here, it is quite common that normalization results in data being broken out into multiple smaller relations. This is an important difference between normalized and denormalized databases where the former, often used for production applications, contains many tables with more complex relationships and the latter, often used for analytical data warehouses, contains fewer tables.

Second normal form requires that the primary key for each relation be contained within a single attribute. This is important because it ensures that all attributes in a relation serve to describe a single instance of the things the relation describes. Fortunately, we fulfill this requirement because we explicitly included PatientID, PhoneNumberID, and EncounterID attributes. Sometimes, however, this is not the case. Imagine that our patients relation looked like this:

```
('PatientID', 'FirstName', 'LastName', 'Age', 'ClinicID', 'VisitLocation')
(1,           'Peter',     'McCaffrey', 30,    1,          'Downtown')
(1,           'Peter',     'McCaffrey', 30,    2,          'Uptown')
(2,           'John',      'Smith',     25,    1,          'Downtown')
```

Here, the VisitLocation is dependent upon the ClinicID and uniquely identifying a tuple here actually requires a "composite key" created by combining both the PatientID and the ClinicID attributes. Following second normal form, we could break this apart into two relations for which the attributes depend only upon the single-attribute primary key of their respective relation.

```
('PatientID', 'FirstName', 'LastName', 'Age')
(1,           'Peter',     'McCaffrey', 30)

('ClinicID', 'Location')
(1,          'Downtown')
```

Also, since a patient may visit many clinics and have many ClinicIDs associated with him, we associate the clinic and patient tuples through the encounter relation:

```
'EncounterID', 'PatientID', 'ClinicID', 'Date')
(1,            1,           1,          10/6/2018)
```

Lastly, as the third normal form requires that all attributes be "nontransitively dependent" so we would avoid relations that might have records like this:

```
('EncounterID','PatientID','Date',     'Discharge', 'LengthOfStay')
(1,            1,         10/6/2018, 10/8/2018,   2)
```

To one resembling our conventional encounters table since LengthOfStay is entailed by the dates of the encounter.

One might reasonably ask what the value is in normalizing databases at all since it is seems to make things more scattered. The answer ultimately boils down yet again to preserving database integrity, especially in highly trafficked operational systems. When data are denormalized, it may be easier to browse and query for some purposes, but the tradeoff is that individual pieces of data are replicated across tuples. Consider the relation we discussed in second normal form:

```
('PatientID', 'FirstName', 'LastName', 'Age', 'ClinicID', 'VisitLocation')
(1,           'Peter',     'McCaffrey', 30,    1,         'Downtown')
(1,           'Peter',     'McCaffrey', 30,    2,         'Uptown')
(2,           'John',      'Smith',     25,    1,         'Downtown')
(2,           'John',      'Smith',     25,    1,         'Downtown')
```

We have first names, last names, ages, locations, and IDs all repeated in multiple locations. In an analytical warehouse where we intend almost exclusively to read this information, this is acceptable but in an operational application, this introduces risk. Imagine we needed to update a value such as the last name for patient ID 1, changing from "McCaffrey" to "Wilson." Managing that transaction to update the same value in several locations amid high-volume read and write operations traffic bears the risk of failure wherein the last name is updated in some places but not others and this violates database integrity. Thus, normalization, while it complicates database schema by expanding the number of relations, protects database integrity by limiting redundancy and database mutation.

2.7 Conclusion

We've discussed the relational model behind many popular databases that support SQL as well as key concepts around keys, transactions, and normalization. In the next Chapter, we will discuss SQL as a language and how to construct and interpret SQL queries. Although many try to learn SQL without insight into the relational model that underlies it, being equipped with a conceptual grasp of relations, tuples, attributes, and keys will significantly enhance your ability not just to understand but to master SQL as a language and to use if effectively. Furthermore, as we proceed throughout subsequent chapters, these concepts will add important context and bearing for understanding "nonrelational" systems.

References

1. Baxendale P, Codd EF. Information Retrieval A Relational Model of Data for Large Shared Data Banks. *Commun ACM*. 1970;13.
2. Chamberlin DD, Boyce RF. Sequel: A Structured English Query Language B. Proceedings of the ACM SIGFIDET Workshop on Data Description, Access, and Control. https://doi.org/10.1145/800296.811515.
3. Codd EF. The Relational Model for Database Management: Version 2. Addison Wesley; 1990.

This page intentionally left blank