Chapter 2
Relational Data

Relational database systems store data in repositories called *databases* or *schemas*. The latter name is due to the fact that (as we will see) data repositories have schemas, like datasets. In this book, we use ‘database’ for the data repository and reserve ‘schema’ for the description of the structure.

After starting a database server and connecting to it (see Appendix A for instructions), it is necessary to create a database, which is done with the SQL command

```
CREATE DATABASE name
```

or, alternatively

```
CREATE SCHEMA name
```

The name is just a label, but it is customary to give databases descriptive names that bring to mind the data they contain.

It is possible to create several databases in the same server. In that case, it is required that each database be given a different name. Whenever a user connects to a database server, she must specify which database to work on by giving the name: in Postgres, the command is

```
connect database-name
```

and in Mysql,

```
use database-name
```

After that, all commands that are explained in this chapter and the next one will work inside that database (it is also possible to work with several databases by switching from one database to another).

Once a database is created, the next step is to create *tables*. This can be quite a complex step, and it is explained in detail in this chapter.

A. Badia, *SQL for Data Science*, Data-Centric Systems and Applications,
https://doi.org/10.1007/978-3-030-57592-2_2

2.1 Database Tables

Relational databases store data in units called *tables*. In fact, a database can be considered a collection of tables. In almost all cases, such tables are linked to each other, in that they contain related or connected data. How this connection can be expressed is discussed later in Sect. 2.2.

SQL implements the concept of table in a straightforward manner. To create a table, the SQL language has a command, called (not surprisingly) `CREATE TABLE`. A very simple example of this command is

```
CREATE TABLE Employees (
  name char(64),
  age int,
  date-of-birth date,
  salary float)
```

In order to explain the meaning of this command, we need to give a description of a table's four components: the *name*, the *schema*, the *extension*, and the *primary key*.

Each table must have a name. Since a database may contain several tables, this name must be unique inside a database (tables in different databases can have the same name). As in the case of databases, this name is simply a label to be able to refer to the table, but it is customary to give the table a descriptive name that refers to whatever the data in the table denotes. Typically, tables have names like 'Employees,' 'country-demographics,' or 'New-York-flights.' This is the first component of the `CREATE TABLE` command.

The name of the table is followed by a description, in parentheses, of the table schema; this is a list of *attributes*. Each attribute refers to a column in the table (some books use 'attribute' and 'column' interchangeably). An attribute has a *name* and a *data type*. The name must be unique among all the attributes in the table. As in the case of tables, it is customary to give attributes meaningful names that evoke the kind of information or domain that they represent.

2.1.1 Data Types

A data type refers to the way in which a computer represents data values. All relational database systems have a repertoire of data types to choose from when defining a table. Even though each database system uses slightly different types with slightly different names, they are all very similar and easy to understand. Each system has the following basic types:

- *strings*: strings are the name given in Computer Science to sequences of characters. They are used to represent values from a discrete domain; usually, this is the data type for any nominal/categorical attribute and may also be used for ordinal types (see Sect. 1.3). Most systems will take any sequence made up of any combination of letters and number that does not start with a number (some systems may even admit labels that start with numbers). Thus, `rain_amount` or `DB101` are legal strings, but in some systems `4thterm` or `2011` may not be. Later on, when entering values (see Sect. 2.4) we will see that some systems require strings to be surrounded by quotes (single or double). The single quote is the standard SQL string delimiter.[1] Doing this allows the system to accept additional characters like whitespaces; this way the string 'Jim Jones' (or "Jim Jones") can be accepted as a single value.

Strings are also needed to give names to database elements, like tables and attributes on them; these names are called *identifiers*. Both MySQL and Postgres allow identifiers with and without quotes; however, when not using quotes, certain restrictions apply: an identifier cannot use hyphens or start with a digit or an underscore or use a reserved SQL word (like CREATE, ATTRIBUTE, etc.). When using quotes, these restrictions are all lifted.

The *size* of a string is the number of characters that make it up. Strings can be of several types, depending on their sizes and on how sizes are handled. With respect to the former, some strings are appropriate for short strings, like many categorical variables use; others are useful for large strings, like short snippets of text. With respect to size handling, some strings have *variable* size (called `character varying(n)` in SQL, with `n` a positive integer), others have *fixed* size (called `character(n)`). When a string value is shorter than the maximum size `n`, variable size strings adopt the size of the actual value, while fixed size strings are padded with whitespace (up to the maximum `n` characters). When the string value is longer than `n`, the value is truncated in either case. As an example, if we declare an attribute `CountryA` as `character varying(5)` and `CountryB` as `character(5)`, the value "USA" would be stored as such in `CountryA`, and as "USA " (note the two added whitespaces) in `CountryB`, while the value "Australia" would be truncated to "Austr" in both cases. When using a string data type, it is customary to examine existing data values for the attribute and choose a size that would allow even large values to be stored (with a few extra characters just in case). Many systems easily support strings of up to 256 characters, which is fine for most categorical variables (but may be short for certain uses. See Sect. 2.3.3 for handling long strings).

- *numbers*: used to represent numerical values, be they interval, ratio, or absolute (see Sect. 1.3). Most systems support several types of numbers: *integers* (called `int` or `integer` in most systems) for whole numbers and *float* or *decimal* for reals (rationals are expressed as reals). It is important to note, for those who are going to make heavy use of numerical calculations, that all numbers in a computer are expressed using a certain level of precision: for integers, this means that some numbers may be too large in absolute value; for real numbers, it means that some numbers may be too large or too small. In many systems, it is possible to specify a certain level of precision for real numbers, by using the notation

[1] In MySQL, double quotes work as a string delimiter by default.

`decimal(precision, scale)`: here, precision refers to the maximum (total) number of digits in the number, while scale refers to the maximum number of decimal places. For instance, the number 1234.567 has a precision of 7 and a scale of 3. Each system has a different limit on how precise numbers can be. In Postgres, the `double precision` data type offers 15 digit precision, and `float8` offers 17 significant decimal digits, while the type `bigint` can store integer values between -2^{63} and $+2^{63}$. MySQL offers similar types, with the same names and ranges. In both systems, the use of floating-point values may result in unexpected behavior when doing arithmetic with very large or very small values; we mention this in several relevant places in Chap. 3.

- temporal information types: usually, *date*, *time*, *timestamp*, and `interval` types are supported. A date is a combination of year, month, and day of the month that denotes a certain calendar day. A time is composed of hour-minute-second parts (some systems allow times with further precision, going down to tenths and hundreds of second). A timestamp is a combination of a date and a time, with all the components of both. Finally, an interval is a pair of timestamps intended to denote the beginning and end of a time period.

In addition to these, pretty much all systems have other types, like *Booleans* (to represent value True and False). In particular, many systems offer special types to store complex information. We will discuss types to store XML, JSON, and text data later in Sect. 2.3.[2]

It is easy to see that the same value could be represented by several data types. This leads to an interesting issue: we can pretty much declare anything as a string. For instance, one could declare an attribute `age` to be of string type in order to enter values like “almost 35” or “sixty-two” or “24.”[3] However, as we will see in Sect. 3.1.2, the database provides several ways to manipulate each data type, represented by *functions*. Such functions are tailored to the expected values of the type. For instance, you cannot add two strings—but you can add two numbers. What this means is that, if this `age` attribute is declared as a string, we could not do simple arithmetic on it, even though it represents a numerical value. This would deprive us of the opportunity of even elementary analysis, like calculating the mean (average) of all the ages in our table. In order to do so, we should declare `age` as an integer; in general, we should always declare an attribute with the type that is most faithful to the values it contains. As noted in Sect. 1.3, one must take into account the semantics of the attribute: just because something looks like a number, it does not mean that it is a number (recall the example of the zip code; such value is better represented by a string).

[2]Even more advanced types exist that are able to store collections of objects. We will not be discussing them in this book, since they are usually not needed for data analytics and they add quite a bit of complexity to data modeling.

[3]Note that this is the string “24,” not the number 24; these two are completely different.

Example: CREATE TABLE Statement

To create a table for the ny-flights dataset, we can use a command like the following:

```
CREATE TABLE NY-FLIGHTS(
flightid int,
year int,
month int,
day int,
dep_time int,
sched_dep_time int,
dep_delay int,
arr_time int,
sched_arr_time int,
arr_delay int,
carrier char(2),
flight char(4),
tailnum char(6),
origin char(3),
dest char(3),
air_time int,
distance int,
hour int,
minute int,
time_hour timestamp);
```

In choosing the data types, we have observed some sample values to determine an appropriate choice. Several aspects of this example are worth pointing out:

- Information about the date of the flight is expressed by 3 attributes, `year`, `month`, and `day`. Why not express it as a single attribute of type date? Many times, the *granularity level* at which to express the data is decided for us (as in this case). As we will see in Sect. 3.1.2, we can divide a string into parts fitting into a pattern, and we can also put together several strings into a single one. In these cases, it is easy to extract parts from the original value and to *concatenate* (put together) strings, so we can create an attribute of type date that contains the same information. If accessing the parts of a value is going to be done repeatedly, it may be a good idea to split the data in the first place. If breaking the value into parts is not an easy task (for instance, in real life dealing with dates is usually much more complicated), it may also be a good idea to break the attribute into parts, so that accessing the right data does not become a nightmare. Conversely, if the original data comes with the value as one big string, we may have to take it in as a single value and break up this value into parts with some database tools later. The right choice depends mainly on two circumstances: the raw data and how we are going to use the data.
- Attributes like departure or arrival time are given an integer data type. This is because that is how it is expressed in the raw data, with values like "517," instead of a time with hours and minutes. Again, if we are going to do arithmetic on those, it would be a good idea to have them as time values (note that as numbers, the difference between 517 and 455 is 65, but as times the difference is 22 min).

We cannot declare the attribute as time because the system cannot transform the raw values to time values, but we can convert between the integer value and a time value (the details of this transformation are explained in Sect. 3.3.1.3).

- attributes `carrier`, `flight`, `tailnum`, `origin`, and `dst` are given a fixed length. This is because most of these are codes (for instance, `origin` and `dst` are expressed as airport codes, which are 3 letters). If this were not the case, it would be better to go with a varying size attribute.
- the values of attribute `flight` look like numbers (‘1545,’ ‘1714’) but are really codes, hence the choice of a string for data type. Again, ask yourself *how am I going to use this data?* (i.e. will I ever add two flight numbers?).
- the values of attribute `time_hour` are real timestamps (i.e. a date and a time) and therefore need to be declared as such so that the database can make sense of these values.

Exercise 2.1 Create a database and execute the CREATE TABLE statement for dataset `ny-flights` in both Postgres and MySQL.

2.1.2 *Inserting Data*

Finally, a part of the table that is not seen in the `CREATE TABLE` statement is the extension of the table, which contains the data in *rows* (AKA *tuples*). A row represents a record, object, or event; it gives a sequence of data values, one value for each attribute in the schema. In SQL, we first create the table; once, this is done, we can add rows to it, hence putting data into our database. This is done through the `INSERT` statement; the syntax is

```
INSERT INTO table-name VALUES(...)
```

The parentheses must enclose a list of data values that respects the schema of the table. Note that, since each value must match some attribute in the schema, we must pair values with attributes. This is done in one of the two ways:

- simply enumerating the values will pair them up with attributes following the order used in the `CREATE TABLE` statement (the default order). That is, if we created table T declaring (in this order) an attribute A of type integer, an attribute B of type string, and an attribute C of type date, the system will expect rows of the form (integer, string, date).
- specifying a customer order in the INSERT statement by following the table name with a list, in parentheses, of the attributes in the table, in whichever order we want. The values will then be expected to follow this order. For instance, reusing our previous example, we could use the statement

  ```
  INSERT INTO T(B,C,A) VALUES(b,c,a)
  ```

where value `b` would be expected to be of type string (to correspond to attribute B), value `c` would be expected to be of type date (to correspond to attribute C), value `a` would be expected to be of type integer (to correspond to attribute A).

Example: Inserting Data

The statement

```
INSERT INTO NY-FLIGHTS VALUES(1, 2013, 1, 1, 517, 515, 2, 830,
819, 11, "UA", 1545, "N14228", "EWR", "IAH", 227, 1400, 5, 15,
2013-01-01 05:00:00);
```

is legal once the table `NY-FLIGHTS` has been created (we are using the 'default' order here). Observe that string values use double quotes.

Exercise 2.2 Insert two rows into the table `ny-flights` of the previous exercise. NOTE: some systems allow you to insert more than one row with a single INSERT statement; find out how to do that in Postgres and/or MySQL.

Note that the only data that can be *inserted* into the table is data that respects the schema, that is, rows that have exactly one value, and of the right type, for each attribute in the schema. It is sometimes the case that data does not *exactly* follow the schema, a problem that we examine in more detail later. SQL offers some flexibility for the cases where we have *incomplete* data (some values are missing): a special marker, called the NULL marker, can be used to signal that a value is missing.

Example: Inserting Incomplete Data

Assume that a row of `NY-FLIGHTS` is missing the departure time, the scheduled departure time, the tail number, and the distance. Then we can use the following statement to insert the row:

```
INSERT INTO NY-FLIGHTS VALUES(2, 2013, 1, 1, null, null, 4,
850, 830, 20, "UA", 1714, null, "LGA", "IAH", 227, null, 5,
29, 2013-01-01 05:00:00);
```

It is also possible to destroy a table by using the SQL command
`DROP TABLE tablename`
This will get rid of the table *and* of any data in the table.
A few final observations about tables:

- for simplicity, most tables in this book have very simple schemas, and only a small portion of their data is shown. Real-life tables may have from a few to a few hundred attributes in the schema and may have hundreds to thousands or millions of rows in the extension. Since it may be necessary to remember what attributes exist in a table, and what their exact name is, all systems provide a method to retrieve information about a table, including its schema. In MySQL, there is a command

```
SHOW COLUMNS [FROM table_name] [FROM database_name]
```

that will give schema information about the (optional) `table_name` argument. The same information can be retrieved in Postgres at the command line using

```
\dt *.*
```

for all tables in all databases, or

```
\dt database_name.table_name
```

for a particular table in a particular database. It is also possible to retrieve this information directly, since all relational databases store metadata (data about the databases) in tables! In MySQL, this is done with the statement

```
SELECT COLUMN_NAME, DATA_TYPE
FROM INFORMATION_SCHEMA.COLUMNS
WHERE table_name = 'table_name'
    [AND table_schema = 'database_name'];
```

A similar query will also work in Postgres. Queries (SELECT statements) are explained in detail in the next chapter.

- As a rule, the schema of a table changes very rarely; the rows of data may change rapidly. As we will see, it is very easy to change the schema of a table before it contains data, but it becomes problematic once data is on it. This is one reason that schema changes are infrequent. However, they are possible, as discussed in Sect. 3.5.
- Even though database tables can be depicted as described in the previous chapter, it is important to note that the order of the columns or the rows is completely irrelevant. In other words, a table with the same number, type, and names of columns in a different order would be considered to have the same schema. And two tables over the same schema with the same rows, even in different order, would be considered to have the same data.

2.1.3 Keys

Among all attributes that make up the schema of a table, it is common that some of them together are enough to identify rows, that is, to tell a row apart from all the others.[4] Such attribute sets are called *keys* in databases (in other contexts, they

[4]Technically, this is *guaranteed* to be the case, since all attributes together (the whole schema) could be a last-instance key. But in most cases, a small subset of attributes works.

are called *identifiers*, and the rest of the attributes, *measurements*). When creating tables, we identify and declare keys for each table.

Example: Keys in Tables

In the table `ny-flights`, a key is a combination of attributes that identifies a flight (i.e. distinguish a certain flight from all others). There is an obvious key (the first attribute, `flightid`). There are also several other keys: a combination of departure date (`year, month`, and `day`), `carrier` and `flight` number will also identify each flight uniquely (unless an airline has the same flight that departs several times a day under the same flight number!). Note that the tail number (which identifies the airplane) may also be used instead of flight number if each flight is assigned a certain plane, so that departure date, `carrier`, and `tailnum` may also be a key (there is no rule forbidding two keys from sharing attributes!). There may be other keys, like a combination of departure date, `dep_time`, `origin`, and `dest`. Whether any of these is a true key depends on the exact semantics of the attributes.

It is important to understand two things about keys:

- In SQL, a table can have *repeated rows* (that is, the same row can be added to a table several times). Obviously, repeated rows have the same key. Thus, when we say that a key can tell a row from any other, we mean "from any *distinct* row." A key value is always associated with one or more copies of a given row. What should never happen is that a key is present in two (or more) *different* rows. In that case, what we have is not really a key.
- a key is a group of attributes that is *as small as possible*—that is, all the attributes in the key are needed for the key to do its job of telling rows apart. Without this stipulation, we would have a large number of (useless) keys. Consider, for instance, a table with information about people, where one attribute is the *social security number* of the person. Since this is (at least in the USA[5]) a unique code per person, we can think of this attribute as a single-attribute key. But then, technically, the pair of *social security number* and *name* (or any other attribute) could also work as a key, since given a *social security number* and a *name*, we can tell any two rows apart. The problem, of course, is that *social security number* is doing all the work; *name* is just along for the ride. In fact, without the requirement of minimality, any set of attributes that included *social security number* would be a key. So when we talk about a key we will always mean a *minimal* set of attributes.

Finding keys is very easy sometimes and far from trivial in others. We may have situations in which a table has more than one key. In such a scenario, we choose one key to be the *primary key* of the table; all other keys are called *unique* attributes

[5]Other countries also give their citizens some sort of identification number.

in SQL. Sometimes it is unclear what constitutes a key for a table; it has become customary in many scenarios to *create* artificial keys, called *identifiers* (*id* for short). Usually, an id is simply an integer that is assigned to each row by giving a 1 to the first row inserted in the table, a 2 to the second row, etc. This guarantees that each row has a different value; hence, an id works perfectly as a key (we will see an example of how this is done in SQL later on). There are two things worth mentioning about ids: first, they are meaningless. They technically satisfy all the requirements of a key, but (unlike other attributes) do not refer to anything in the real world. Therefore, they are useless for most data analyses. Second, if there is a key in a table, and we decide to create an id, the key is still there, and if it goes undetected (so it is not even declared as UNIQUE) this may lead to some problems later on. The moral of the story is that we should not rush to create ids; rather, we should examine the existing attributes carefully to identify any possible keys.

Example: Ids and Multiple Primary Keys

In the case of `ny-flights`, the attribute `flightid` is an example of id, and we have multiple (possible) keys. As another example, suppose we have a dataset about cars, where some of the attributes are: *VIN*, or Vehicle Identification Number, a unique number given to each car on the road in the USA; *State*, the state where a vehicle is registered; *license-number*, the license (plate) number for a card; *make*, or manufacturer; *model*; and *year*.

Vin	State	License-number	Make	Model	Year
WBAA3185446N384	KY	444ABC	BMW	325i	1989

We can tell that:

- *VIN* is, by itself, a key. This is due to the fact that it is designed to work as such; a different VIN is assigned to each car.
- *State* and *License-number* are, together, another key. This is because each state gives each car licensed in that state a unique plate. Thus, two cars from different states could end up with the same place, but no two cars from the same state can have the same license number.

In this case, we have two keys. We choose one as primary and declare the other one as unique. It is customary (but not required) to choose as primary the simplest (smallest) key, so we would choose *VIN* as primary and *State, license-number* as unique.

Note that whether a group of attributes is a key depends entirely on the semantics of the attributes in the schema. If a group of attributes is a key, it should work as a key for any possible extension of the table, not just from the data that we have right now. Hence, it is not possible to determine whether a group of attributes is a key by inspecting the data. It is, however, perfectly possible to prove that a group of attributes is *not* a key by finding a counter-example in the data.

As we will see shortly, primary keys fulfill an important mission: they allow us to refer to specific tuples in a table. This will become necessary when organizing complex datasets that require more than one table (see Sect. 2.2) and when making changes to the data (see Sect. 2.4.2).

Example: Declaring Keys

We can declare keys (and uniques) when creating a table; for instance, in Postgres, we could have used

```
CREATE TABLE ny-flights(
  flightid int PRIMARY KEY,
  ...);
```

in our original CREATE TABLE statement—or, alternatively, we could add a line after the last attribute:

```
CREATE TABLE ny-flights(
  flightid int,
  ...
  time_hour timestamp,
  PRIMARY KEY (flightid));
```

instead of our original example. We can also modify an existing table to add information about keys. For instance, in Postgres we could create table `ny-flights` as before and then issue the SQL command:

`ALTER TABLE ny-flights ADD PRIMARY KEY(flightid);`

Technically, the declaration of a primary key is what is called a *constraint* and it can also be written as such, by using the keyword `CONSTRAINT` and giving a name:

```
CREATE TABLE ny-flights(
  flightid int CONSTRAINT nyf-pk PRIMARY KEY,
  ...);
```

or as

```
CREATE TABLE ny-flights(
  flightid int,
  ...
  time_hour timestamp,
  CONSTRAINT nyf-pk PRIMARY KEY (flightid));
```

This gives a name (`nyf-pk`) to the primary key declaration. The syntax is the same in MySQL and in all relational systems, as it is part of the SQL standard.

Exercise 2.3 Alter the `ny-flights` table in Postgres and MySQL to add a primary key to it.

Example: Creating Ids

As stated earlier, most systems have some way of creating id attributes. In Postgres, an attribute is given type `SERIAL` and declared a primary key, and the system automatically creates values for this attribute:

```
CREATE TABLE books (
  book-id   SERIAL PRIMARY KEY,
  title   VARCHAR(100) NOT NULL,
  author  VARCHAR(100),
  publisher VARCHAR(100),
  num-pages INT);
```

In this case, when adding data into the table it is not necessary to provide a `book-id`; the system will automatically generate a new value for each row. That is, insertions into `books` will provide rows with only 4 values: one for `title`, one for `author`, one for `publisher`, and one for `num-pages`.

In MySQL, the same example is written as follows:

```
CREATE TABLE books (
  book-id   INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  title   VARCHAR(100) NOT NULL,
  author  VARCHAR(100),
  publisher VARCHAR(100),
  num-pages INT);
```

Note that an id primary key can also be added to a table after creation; if we had not declared `book-id` as above, we could add

```
ALTER TABLE books
ADD book-id INT AUTO_INCREMENT PRIMARY KEY;
```

in MySQL, and similarly in Postgres:

```
ALTER TABLE books
ADD book-id SERIAL PRIMARY KEY;
```

Other keys beside the primary one can be declared exactly like the primary key, but using the keyword `UNIQUE` instead of `PRIMARY KEY`.

Example: Unique Keys

For the table `CAR` described previously, we can use the declaration

```
CREATE TABLE CAR (
 Vin VARCHAR(36) PRIMARY KEY,
 State CHAR(2),
 License-number CHAR(6),
 Year INTEGER
 UNIQUE (State, License-number)
)
```

The last line could also be

```
CONSTRAINT uniq-car UNIQUE (State, License-number)
```

Likewise, ALTER TABLE can also be used to add a unique constraint after the fact:

```
ALTER TABLE car ADD UNIQUE (State, License-number)
```

A primary key cannot have NULL markers in any of its attributes; if we declare a set of attributes as the primary key of a table, any attempt to enter a row with null markers on any part of the key will be automatically rejected by the system. This is because the system will check that a primary key (or another key declared as UNIQUE) is indeed a (primary) key; if a (primary) key contains nulls, the system cannot ascertain whether it is a correct key or not. If we declare a (primary) key and misuse it (by entering the same value of the so-called key in two distinct tuples), the system will reject the insertion because it violates the condition of being a key.

2.1.4 Organizing Data into Tables

A large proportion of data used for data analytics is presented in a tabular format. In fact, many algorithms for Data Mining and Machine Learning assume that data is in a table. This is why it is important to understand what a proper table is, and how to deal with data that is not in the right format.

We start with the basic observation that the same data can be structured in several ways. Not all of them are conducive to analysis.

Example: Wide (Stacked) and Narrow (Unstacked) Data

We have some data about people, presented in two different ways.[6]

Person	**Age**	**Weight**
Bob	32	128
Alice	24	86
Steve	64	95

Person	**Variable**	**Value**
Bob	Age	32
Bob	Weight	128
Alice	Age	24
Alice	Weight	86
Steve	Age	64
Steve	Weight	95

[6]This example is a simplification of the one used in Wikipedia at `en.wikipedia.org/wiki/Wide_and_narrow_data`.

The first table is called a *wide* or *unstacked* table, while the second table is sometimes called a *narrow* or *stacked* table. For the purpose of analysis, narrow/stacked tables are a very inconvenient structure; we want our tables always in the wide format. Note that in the wide table the key is `Person`, suggesting that each row describes a person, while in the narrow table the key is (`Person, Variable`), a strange combination that is a hint that something is not quite right.

We will see later that there are ways to transform data from narrow (stacked) to wide (unstacked) and vice versa. Data in narrow format is, unfortunately, a common occurrence.

There are other ways in which data may be presented in an undesirable format.

Example: Untidy Data

A very common situation is to have data as follows:

Product	S	M	L
Shirt	12.4	23.1	33.3
Pants	3.3	5.3	11.0

This is undesirable, since the sizes S (small), M (medium), and L (large) can be considered the values of a categorical attribute but are used in the schema. This makes some analysis rather difficult. The following arrangement works better:

Long table

Product	Size	Price
Shirt	S	12.4
Shirt	M	23.1
Shirt	L	33.3
Pants	S	3.3
Pants	M	5.3
Pants	L	11.0

Again, the first table is an example of narrow/stacked data, while the second is an example of wide/unstacked data. A proper table data (called *tidy data* in [19]) is wide/unstacked; to achieve this, it follows certain conventions:

- all values are in the data cells, never in the schema. For instance, S (small), M (medium), and L (large) are values of attribute `size`, so they should be in data cells, not part of the schema.
- all attributes are in the schema, never in the data. For instance, `Age` and `Weight` are attributes of a person, so they should be in the schema, not in the data.
- each row corresponds to a data record (observation, in statistics) and each column to an attribute (variable, in statistics). Intuitively, in a table about people we want

each row to describe a person; in a table about experiment runs, each row should be a run. Note that, in the last example above, each row describes a combination of product and size; this is due to the fact that each product is sold in several sizes. We will see how to deal with this situation shortly.

Having untidy data is a common problem; both R and Python provide tools to deal with this situation.[7] However, both R and Python are a bit more liberal than databases about the data they admit. The following provides an example of what pandas accept that a database does not.

Example: Tidy Data in Other Frameworks

The following tables provide an example of how the panda framework would unstack/pivot a data frame (the name of the structure in pandas that holds tabular data). The dataset on the left is a stacked data frame, the one on the right is an unstacked data frame.

		Spring	**Fall**
Emma	History	82	91
	Physics	81	79
Gabi	History	80	88
	Physics	83	89

	Spring		**Fall**	
	History	Physics	History	Physics
Emma	82	81	91	79
Gabi	80	83	88	89

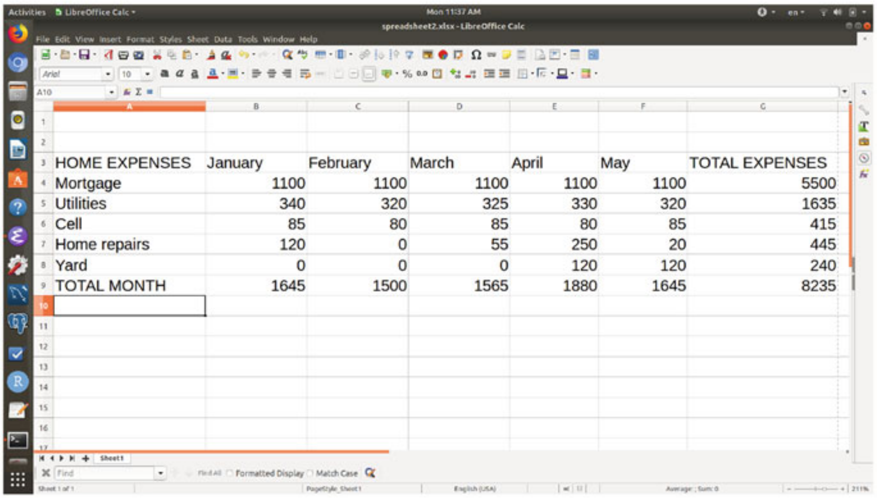
However, we would not call the data frame on the left a proper table in SQL (and it would not be tidy according to [19]). The proper table would look as follows:

Student	**Subject**	**Semester**	**Score**
Emma	History	Spring	82
Emma	Physics	Spring	81
Emma	History	Fall	91
Emma	Physics	Fall	79
Gabi	History	Spring	80
Gabi	Physics	Spring	83
Gabi	History	Fall	88
Gabi	Physics	Fall	89

Recall that in a table the order of rows (or attributes) is irrelevant. On the data frame, the 'location' of a score is crucial; in the table, it is the values in the tuple that matter. The tuple itself can be anywhere in the table.

The problem of ill-structured data is very common in spreadsheets. It is compounded there by the fact that data and analysis are often mixed up. The following is an example of the problems found when data comes from spreadsheets.

[7] R has a `tidyr` package; Python provides several functions on the `Panda` library.



	B	C	D	E	F	G
HOME EXPENSES	January	February	March	April	May	TOTAL EXPENSES
Mortgage	1100	1100	1100	1100	1100	5500
Utilities	340	320	325	330	320	1635
Cell	85	80	85	80	85	415
Home repairs	120	0	55	250	20	445
Yard	0	0	0	120	120	240
TOTAL MONTH	1645	1500	1565	1880	1645	8235

Fig. 2.1 Non-tabular data in a spreadsheet

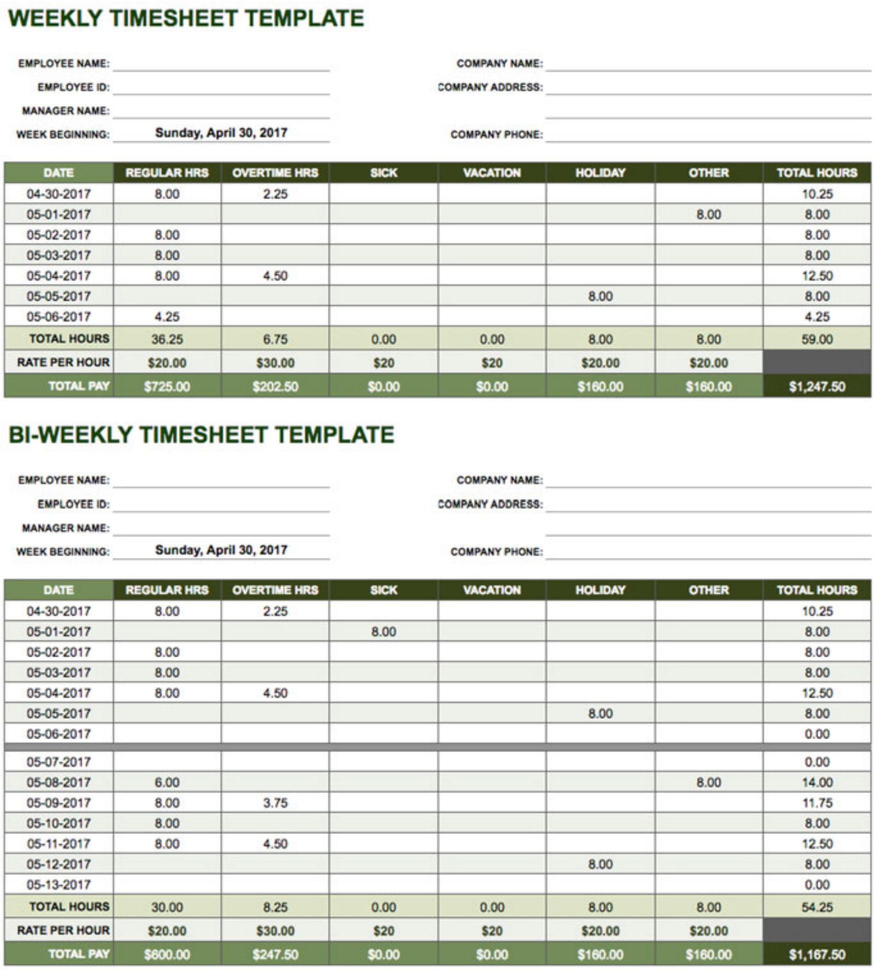
Example: Spreadsheets Problems

Another typical arrangement of data that is not tabular can be seen in Fig. 2.1. As before, the problem is that the data has been put on both down the columns and across the rows (the date when an expense was incurred is on the top row). In a tabular (tidy) format, this data would have schema *(Expense, Month, amount)*, with data like

Expense	Month	Amount
Mortgage	January	1100
Utilities	January	340
...		
Mortgage	February	1100
...		

There is an additional issue here. The bottom row and the rightmost column are what is usually called *margin sums*; they are data derived by calculation from the original data. When bringing the information in the spreadsheet into a table, there is no sense in copying these derived data; these results can be easily calculated in SQL, and they are different in nature from the raw data in the table.

Sometimes the problems with spreadsheets go deeper than this. The root of the problem is that spreadsheets do not enforce any rules on how data is organized. As a result, a user can do whatever seems convenient for a given situation. For instance, if we look at the spreadsheet in Fig. 2.2, which is a generic template from Google docs, we will see that part of it looks like the table in the previous example, but in addition there is also other data (the attributes at the top: ‘Employee Name,’ ‘Company Name,’ . . .) that is different from the data in the table. Another common situation arises when a spreadsheet has data spread across multiple pages or tabs. Suppose, for instance, that we have a dataset about properties in a spreadsheet, which each property described in a separate page. There is no guarantee that the same or even similar data is present on each sheet or that it is in the same format. And even though the data is in separate pages, it is all part of the same datasets. In many cases, all the data in the spreadsheet can be put into a database, sometimes



WEEKLY TIMESHEET TEMPLATE

EMPLOYEE NAME:		COMPANY NAME:	
EMPLOYEE ID:		COMPANY ADDRESS:	
MANAGER NAME:			
WEEK BEGINNING:	Sunday, April 30, 2017	COMPANY PHONE:	

DATE	REGULAR HRS	OVERTIME HRS	SICK	VACATION	HOLIDAY	OTHER	TOTAL HOURS
04-30-2017	8.00	2.25					10.25
05-01-2017						8.00	8.00
05-02-2017	8.00						8.00
05-03-2017	8.00						8.00
05-04-2017	8.00	4.50					12.50
05-05-2017					8.00		8.00
05-06-2017	4.25						4.25
TOTAL HOURS	36.25	6.75	0.00	0.00	8.00	8.00	59.00
RATE PER HOUR	$20.00	$30.00	$20	$20	$20.00	$20.00	
TOTAL PAY	$725.00	$202.50	$0.00	$0.00	$160.00	$160.00	$1,247.50

BI-WEEKLY TIMESHEET TEMPLATE

EMPLOYEE NAME:		COMPANY NAME:	
EMPLOYEE ID:		COMPANY ADDRESS:	
MANAGER NAME:			
WEEK BEGINNING:	Sunday, April 30, 2017	COMPANY PHONE:	

DATE	REGULAR HRS	OVERTIME HRS	SICK	VACATION	HOLIDAY	OTHER	TOTAL HOURS
04-30-2017	8.00	2.25					10.25
05-01-2017			8.00				8.00
05-02-2017	8.00						8.00
05-03-2017	8.00						8.00
05-04-2017	8.00	4.50					12.50
05-05-2017					8.00		8.00
05-06-2017							0.00
05-07-2017							0.00
05-08-2017	6.00					8.00	14.00
05-09-2017	8.00	3.75					11.75
05-10-2017	8.00						8.00
05-11-2017	8.00	4.50					12.50
05-12-2017					8.00		8.00
05-13-2017							0.00
TOTAL HOURS	30.00	8.25	0.00	0.00	8.00	8.00	54.25
RATE PER HOUR	$20.00	$30.00	$20	$20	$20.00	$20.00	
TOTAL PAY	$600.00	$247.50	$0.00	$0.00	$160.00	$160.00	$1,167.50

Fig. 2.2 Spreadsheet example

in a single table, sometimes in multiple tables (we discuss this example again in Sect. 2.2).

We can, with some discipline, use spreadsheets to represent tabular data, but we can also do all kinds of weird stuff to our data—one of the several reasons for not using spreadsheets for serious data analysis.[8]

Sometimes data presents multiple problems at once. In fact, massaging the data into the right format for analysis is usually a (time-consuming and) necessary step before any analysis is done (we discuss this in Sect. 3.4.1).

Example: Very Untidy Data

The United Nations has several datasets publicly available at `data.un.org`. One such dataset shows international migrants and refugees; when downloaded as a csv file, it looks like this:

Region/Country/Area		Year	Series	Value
1	Total	2005	International migrant stock: Both sexes (number)	190531600
1	Total	2005	International migrant stock: Both sexes (% total population)	2.9124
1	Total	2005	International migrant stock: Male (% total population)	2.9517
1	Total	2005	International migrant stock: Female (% total population)	2.8733
1	Total	2010	International migrant stock: Both sexes (number)	220019266
1	Total	2010	International migrant stock: Both sexes (% total population)	3.162
1	Total	2010	International migrant stock: Male (% total population)	3.2381
1	Total	2010	International migrant stock: Female (% total population)	3.0856

This dataset has several problems: it clearly is narrow/stacked; if we go down looking at the second (unnamed) attribute, we find values `Total` (shown above), `Africa`, `Northern Africa`, `Algeria`,...Clearly, the data is shown in several levels of granularity. This really corresponds to what we call later *hierarchical data* (see Sect. 2.3.1) and should not be mixed up in the same table. Besides pivoting, this data needs also to be separated into different tables.

2.2 Database Schemas

Up until now, we have assumed that data fits in a ‘tabular’ format. Basically, this means that

- each record (event/experiment/object) in our domain has a well-defined set of attributes (features/variables/measurements), which are the same for each record (that is, the record set is homogeneous). For instance, in the table `Books` each

[8]There are other good reasons, but we will not discuss them in this book. After all, we have only so many pages.

row represents a book; in the table `ny-flights` each row represents a flight, and so on.
- each such attribute has exactly one value per record. For instance, in a table about People, each person has exactly one age, hence one value for attribute `Age`.

But sometimes data does not follow these rules. We now examine the most common reason for data not fitting well in a tabular format, and what can be done about it (in this section, we still assume structured data. Semistructured and unstructured data is dealt with in Sect. 2.3).

There are three common scenarios for data not to be 'tabular':

- heterogeneous data, that is, data where records may have values for different attributes;
- data with multi-valued attributes, that is, attributes that have more than one value for a record; and
- complex data involving different (but related) events or entities.

We study each one next.

2.2.1 Heterogeneous Data

In the first scenario, we may have records in our dataset that are not completely homogeneous: while sharing many common attributes, some entities may be somewhat different. This may be due to some attributes being present only under certain circumstances and not applying to each record (these are sometimes called *optional* attributes). For instance, an attribute `spouse-name` only makes sense when the record is about a married person, but would not have a value for single people. In this case, we have two options:

1. create a single table and add all attributes present in any record to the schema. Then, on each row, we describe one record; when the record does not have a value for an attribute, we use a NULL marker. This option keeps the database simple, but at analysis time all the NULLs need to be accounted for (we discuss this in Sect. 3.3).
2. identify sets of records that share the same attributes and create a table for each. This leads to *fragmentation* (the same dataset is represented by several tables, not just one) but eliminates the problem with NULLs. Clearly, this option is only advisable if all data can be divided into a few groups of homogeneous attributes; fortunately, this is not an uncommon situation in real-life datasets.

Example: Optional Attributes

In the previous chapter, we used the Chicago employee dataset as an example of tabular data. The schema of the table had attributes *Name, Job Title, Department,* *Full or Part Time, Salary or Hourly, Typical Hours, Annual Salary, and Hourly Rate*. We noted that there are missing values in each record. This is pretty much guaranteed by the fact that the table includes both salaried and hourly employee (as recorded in attribute *Salary or Hourly*): salaried employees will have an *Annual Salary* but no *Hourly Rate*, while hourly employees will have an *Hourly Rate* but not *Annual Salary*. The problem with this table is that it is going to have a large number of NULL markers. However, if we are going to run an analysis that involves all the employees, we may want to keep them in a single table:

Employee							
Name	**Job Title**	**Department**	**F/P**	**S/H**	**TH**	**AS**	**HR**
Jones	Pool Motor Truck	Aviation	P	Hourly	10	Null	$32.81
Smith	Aldermanic Aide	City Council	F	Salary	Null	$12,840	Null

(again, we have abbreviated the attribute names). If we want to avoid any missing data, then we should split this table into two, one for salaried employees and one for hourly employees, and use only the relevant attributes on each (note that we can make do without attributes `Salary/Hourly` since each table is only for one type of employee).

Hourly Employee					
Name	**Job Title**	**Department**	**Full/Part Time**	**Typical Hours**	**Hourly Rate**
Jones	Pool Motor Truck	Aviation	P	10	$32.81

Salaried Employee				
Name	**Job Title**	**Department**	**Full/Part Time**	**Annual Salary**
Smith	Aldermanic Aide	City Council	F	$12,840

We can go back and forth between the two designs; SQL allows combining the two tables together into a single table design (this is explained in Sect. 5.4) as well as breaking down the single, original table into two separate ones (as shown in Sect. 3.1).

2.2.2 *Multi-valued Attributes*

In the second scenario, we have that some records are characterized by attributes that have several values for a given record (these are called *multi-valued attributes* in database textbooks). An example of this was seen in a previous example, where a record came in several sizes, each one with a price. This forces us to describe the record using several rows, since we can only put one value for the attribute on each row. This is fine and will not present a problem for analysis.

Example: Transactional Data

In business applications, it is common that we think in terms of ‘transactions,’ each one of each involves a set of products (usually, buying or selling transactions, with products being the products/services being bought or sold). To describe this data, we usually assign an identifier to each transaction and associate all involved products with the transaction they belong to. The following example describes a couple of sales in a store:

Products	
Transaction-id	**Product**
1	Bread
1	Beer
1	Diapers
2	Eggs
2	Milk

In this table, there are two transactions; the first one involves three products; the second one, two products. Because of this, the first transaction is shown in 3 rows; the second one spans 2 rows. Note that, even though `Transaction-id` is meant to identify the transaction, the key of this table is (`Transaction-id, Product`), since the key needs to be unique for each row.

This scenario is quite common; for instance, any situation where we take repeated measures over time leads to this situation (we discuss the influence of time in databases at the end of this section).

In many cases we have mixed situations, in which a record has both regular (single-valued) and repeated (multi-valued) attributes. Staying within the single table format in a mixed situation creates some problems.

Example: Transactional Data, Revisited

Assume that, for each transaction, we have some information that is unique to the transactions, like the date and time and the store. In that case, the strategy above would involve repeating the information concerning the single-valued attributes.

Transactions-and-Products				
Transaction-id	**Product**	**Date**	**Time**	**Store**
1	Bread	10/10/2019	10:55pm	Market-street
1	Beer	10/10/2019	10:55pm	Market-street
1	Diapers	10/10/2019	10:55pm	Market-street
2	Eggs	10/11/2019	8:25pm	Main-street
2	Milk	10/11/2019	8:25pm	Main-street

Repeating data in this manner may be fine for analysis purposes (it is common to have categorical or ordinal attributes repeated like this, as we will see), but it can present some serious trouble for other purposes. In particular, if we are storing the data and it is possible that we modify it later (by deleting or modifying existing data or adding more), we are facing what is called *anomalies*, situations where the database may be left in an inconsistent state by changes (inconsistency here means that data in some tuples contradicts the data in other tuples). A typical example is the *update anomaly*: suppose that the date of transaction 1 is wrong, and I want to change it. As we will see in Sect. 2.4.2, there is an `UPDATE TABLE` command that allows to change existing rows in a table. However, here I have to make sure that I update the 3 rows spanned by the description of transaction 1; if I change one or two of the rows only, I would leave a transaction associated with two dates. Under the assumption that each transaction happens in one and only one date, this means that the data in the database is inconsistent. But once the change has been made, it is impossible to decide which date is the correct one and which date is the incorrect one (unless we have access to the original record of sale). Hence, it is better to avoid the anomalies in the first place. The method to do so is called *normalization*, and it consists of spreading the data into several tables by separating the multi-valued attributes from the single-valued ones.

Example: Database Design

We are going to store the same information as in the previous example using normalization. First, we separate the table into two tables, one with the transaction information (single-valued attributes) and another one with the transaction products (multi-valued attribute).

Transactions

Transaction-id	Date	Time	Store
1	10/10/2019	10:55pm	Market-street
2	10/11/2019	8:25pm	Main-street

Products

Transaction-id	Product
1	Bread
1	Beer
1	Diapers
2	Eggs
2	Milk

Note that the first table has only two rows: a single row per transaction is enough. The second table is now back to our original example: the id of the transaction is repeated, but nothing else. In fact, if we count the number of cells in these two tables, we see that they have 8 + 10 = 18 cells total. The original table in the previous example has 25 cells. And yet, these two tables have exactly the same information as the original one. The extra cells were created because of redundancy.

The key of the first table is `Transaction-id`, as it stores each transaction information in one row. This is possible because only single-valued attributes are used in the schema of the table. Conversely, the second table stores the multi-valued attribute `Product` and (as stated above) has a different key. However, it keeps a copy of `Transaction-id` so it can connect records to the transactions as described in the previous table. Copies of primary keys are used in databases to keep connections in data and are called *foreign keys*. This technique is crucial in database design, as we will see shortly.

As in the previous case, we can use SQL to transform a single table into several and to combine several tables back into a single one (this is discussed in detail in Sect. 3.1.1). The decision whether to normalize tables or not is, therefore, entirely a pragmatic one. The decision should be guided partly by what we intend to do with the data (are we at risk of having anomalies? That is, is the data going to be modified or just analyzed?) and partly by the structure of the data itself: as we will see shortly in another example, sometimes data has a complex structure and it is not a good idea to put it all in one table.

2.2.3 Complex Data

The strategy of normalization is also used in this third scenario: situation in which we must deal with complex data. This is the case where we have data about several related records (events and/or entities). For instance, we may have data about the customers of a business and the orders they have placed; we can think of customers and orders as different records, but they are related—since orders are placed by customers. As a different example, we may have a group of patients who participate in a drug study; during this study, each patient has several vital signs (blood pressure, etc.) measured daily for a period of time. The patient is a record; the daily measurements, an event—but clearly, the measurements and the patients are related. We may even have information about the study (who is in charge of it, when it started, etc.); the study is related to the patients and the measurements.

In general, we may have an arbitrary number of events/entities and connections between/among them. However, in most scenarios the connections among event/entities are *binary* (they involve two events/entities) and, for the purposes of database design, they can be classified in one of the three types:

- A collection of record I_1 has a *one-to-one* relationship with a collection of records I_2 if a record in I_1 is related to only one record in I_2 and vice versa (a record in I_2 is related to only one record in I_1). For instance, assume we have a demographic database with data about people, but also data about addresses (the exact longitude/latitude, state, etc.) and that each person is associated with one address and each address with one person.
- A collection of record I_1 has a *one-to-many* relationship with a collection of records I_2 if a record in I_1 is related to only one record in I_2 but a record in I_2

may be related to several records in I_1. For instance, in our example above of customers and orders, we can assume that some customers have placed several orders, but each order is associated with one and only one customer. Similarly, in the case of the clinical trial, each participant has several measurements taken, but each measurement is related to only one participant.

- A collection of record I_1 has a *many-to-many* relationship with a collection of records I_2 if a record in I_1 may be related to several records in I_2 and vice versa (a record in I_2 may be related to several records in I_1). For instance, assume we have a dataset of different chemical compound suppliers and another one of chemical laboratories. Each laboratory buys from several suppliers; each supplier sells to several laboratories: there is a many-to-many relationship between supply companies and laboratories.

In general, we want each event/entity to have its own table; this will allow us to tailor the schema of the table to those attributes that describe the event/entity in the most useful or meaningful way. For instance, a table for `Patients` may have attributes like `Name`, `date-of-birth`, `insurance-company`, etc., while a table for `Studies` may have attributes like `date-started`, `sponsor`, and so on. If an event/entity has single-valued and multi-valued attributes, then normalization (as shown in our previous examples) is traditionally used—although, as we have seen, whether to normalize a table or not is a design decision that must take into account the uses of the data.

Besides that, we have to capture the relationships between/among events/entities. The way such relationships are expressed in databases is by using the primary key of an event/entity as its surrogate: primary keys are copied to another table to represent the event/entity and express the relationships. In the previous example we saw that `TransactionID` is the primary key of the table describing transactions, but is also part of the schema (and part of the key) of the table that relates transactions with their records. Recall that a primary key copied to another table is called a *foreign key* in SQL (hence, in the second table above `TransactionID` is a foreign key). All relationships are expressed through foreign keys in relational databases; therefore, a foreign key must be declared to the database, just like a primary key.

Example: One-to-One Relationships

Assume, as before, that we have information about people (heads of households, really) and addresses, with each person having at most one address and each address belonging to one and only one person. Then we can express this simply combining all information in a single table:

Name	Age	...	Street-number	Street-name	...
"Jim Jones"	39	...	1500	Main-Street	...
"Fred Smith"	45	...	1248	Market-Street	...

There are other options for representing this information: we could have two separate tables (`People` and `Addresses`) and copy the primary key of one of them in the other (i.e. as a foreign key). It does not matter which key is copied; either one will do. This is done in SQL as follows:

```
CREATE TABLE PEOPLE (
Name VARCHAR(64) PRIMARY KEY,
....)

CREATE TABLE ADDRESS (
street-number INT,
street-name VARCHAR(128),
city VARCHAR(64),
...
Name VARCHAR(64) FOREIGN KEY REFERENCES PEOPLE
PRIMARY KEY (street-number, street-name, city))
```

We have taken the (pragmatic) decision of using the primary key of table `People` as a foreign key in table `Address` because it is simpler than the primary key of `Address`. However, even a primary key with several attributes can be used as a foreign key; the whole key needs to be copied. For instance, we could have used the primary key of `Address` as a foreign key in `People` as follows:

```
CREATA TABLE PEOPLE (
Name VARCHAR(64),
....
street-number INT,
street-name VARCHAR(128),
city VARCHAR(64),
...
PRIMARY KEY (Name),
FOREIGN KEY (street-number, street-name, city)
      REFERENCES ADDRESS)
```

For data analysis purposes, the single table option is usually the best in this case. For the next cases, the choice is not so clear-cut.

Example: One-to-Many Relationships

In the example used above, we have customers and orders, and each order corresponds to one customer, but a customer may place several orders. First, we would create a table `Customer`, with primary key `CustomerID` and table `Order`, with primary key `OrderID`, to hold information about customers and orders. Then we would add an attribute `CustomerID` to the schema of `Order`; on each row (for each order) we would add the id of the customer to identify who placed the order. That is, tables would be created as follows:

```
CREATE TABLE CUSTOMER(
CustomerID INT PRIMARY KEY,
...);

CREATE TABLE ORDER(
OrderID   INT PRIMARY KEY,
CustID INT FOREIGN KEY REFERENCES Customer,
...);
```

In this table, we may have data like the following:

Customer			
CustomerID	**Name**	**Address**	...
1	Wile E. Coyote	999 Desert Rd.	...
2	Elmer Fudd	123 Main St.	...

Order				
OrderID	**CustID**	**Date**	**TotalAmount**	...
300	1	9/12/2010	20,000	...
301	1	1/2/2011	15,000	...
302	2	2/5/2011	500	...
303	2	6/5/2012	800	...

If put together in a single table, the customer information would have to be repeated (we do not repeat the foreign key, as it is redundant in this table):

Customer-Order						
CustomerID	**Name**	**Address**	**OrderID**	**Date**	**TotalAmount**	...
1	Wile E. Coyote	999 Desert Rd.	300	9/12/2010	20,000	...
1	Wile E. Coyote	999 Desert Rd.	301	1/2/2011	15,000	...
2	Elmer Fudd	123 Main St.	302	2/5/2011	500	...
2	Elmer Fudd	123 Main St.	303	6/5/2012	800	...

Note how each customer information is repeated as many times as orders the client has.

A foreign key is declared, just like a primary key, by telling the database which attribute(s) form the foreign key and which table (hence which primary key) this foreign key is a copy of. Note that the type and number of attributes in the foreign key must match the primary key it copies: if the primary key is a single attribute of type integer (as in this example), the foreign key must be a single attribute of type integer. If the primary key were made of two attributes, one of type string and one of type date, any foreign key should also have two attributes, one of type string and one of type date. The reason for this is that any values in the foreign key must be values already appearing in the referenced primary key. For instance, in the example above, any `CustID` entered as part of a tuple of table `Order` must already appear in the primary key of table `Customer`. The database will enforce this rule by rejecting any data that does not obey it. This implies that orders for a customer cannot be entered in the database until the customer herself has been entered in the database (with an entry in table `Customer`). The rationale for this is that any order must come from a real customer, and as far as the database is concerned, the only real customers are those that are in the database. This makes sure that each order is connected to a real customer and there are no *dangling* references. In this sense, the database helps us keep our data internally consistent.

To help keep consistency, a database offers several (optional) commands when declaring foreign keys:

```
ON [DELETE|UPDATE] [CASCADE|SET NULL|SET DEFAULT|RESTRICT]
```

This instructs the database on what to do if the primary key being referenced is deleted or updated: we can also delete or update the foreign key that references it ('CASCADE'), set the foreign key to a NULL (foreign keys, unlike primary keys, can have null markers), set to a default value, or forbid the deletion or update of the primary key ('RESTRICT'). Suppose, for instance, that an existing customer has placed several orders. Then, there will exist one row in the `Customer` table with the information about this customer (and a `CustomerID`), and several rows in the `Orders` table, one for each order, and all of them using the customer ID to identify this customer. However, suppose that we made some mistake when entering the ID of the customer and we want to correct that (using UPDATE TABLE). The problem is that customer ID is used in `Orders` table as a value of a foreign key and would, after the change, not point to any existing customer. Clearly, the thing to do here is to propagate the change:

```
ON UPDATE CASCADE
```

Conversely, say the customer leaves for whatever reason and cancels her orders. When we delete the customer from the database (that is, from the `Customer` table), the customer ID is gone and, again, that `CustID` used in `Orders` does not refer to an existing customer anymore. In this situation, it may make sense to delete the order too, so we would use

```
ON DELETE CASCADE
```

Under other circumstances, another behavior may make more sense. We need to examine the semantics of the datasets and decide accordingly.

As this example shows, to represent a one-to-many relationship we can copy the key of the table representing the 'one' side to the table representing the 'many' side. However, many-to-many relationships need a different approach.

Many-to-Many Relationships

Suppose, as before, two datasets, one of chemical compound suppliers and one of laboratories, and a relationship between them where a lab buys some amount of a compound at some date. Then our database would look as follows:

```
CREATE TABLE SUPPLIER (
SupName character(100) PRIMARY KEY,
....);

CREATE TABLE LABORATORY (
LabName character(200) PRIMARY KEY,
....);

CREATE TABLE BUYS (
SupName character(100) FOREIGN KEY REFERENCES SUPPLIER,
LabName character(200) FOREIGN KEY REFERENCES LABORATORY,
CompoundId character(10),
Amount int,
Date date,
...
PRIMARY KEY (SupName, LabName));
```

As this example shows, in the case of a many-to-many relationship we create a separate table where we copy, as foreign keys, the primary keys of the tables representing the records involved in the relationships. These two foreign keys, together, are the primary key of this new table.

There are more complex cases, requiring further analysis. It is impossible to cover all of them and give hard and fast rules; database design is part art and part science. However, what has been shown in this section covers the most frequent situations and should help in all but a few cases.

Example: Very Complex Data

Assume that, as earlier, we have information about customers and their orders, connected by a one-to-many relationship. Further assume that we also have *point-of-contact (POC)* information for each customer, and that several customers have several POCs, although a POC relates only to one client. Thus, in addition to the tables shown in an earlier example, we also have

POC			
CustID	**POC-name**	**Phone**	**Email**
1	Road Runner	888-8888	rruner@gmail.com
1	ACME, Inc.	999-9999	manager@acme.com
2	Bugs Bunny	777-7777	bugsbunny@warnerbros.com
2	Duffy Duck	666-6666	duffyduck@warnerbros.com

Note that here Cust-id is also a foreign key to Customer. What would happen if we insist on having all information about our customers in a single table? We would have something like the following:

Customer-Order-POC								
CID	**Name**	**Address**	**OID**	**Date**	**TotalAmt**	**POC-name**	**Phone**	...
1	Wile E. Coyote	999 Desert Rd.	300	9/12/2010	20,000	Road Runner	888-8888	...
1	Wile E. Coyote	999 Desert Rd.	300	9/12/2010	20,000	ACME, Inc.	999-9999	...
1	Wile E. Coyote	999 Desert Rd.	301	1/2/2011	15,000	Road Runner	888-8888	...
1	Wile E. Coyote	999 Desert Rd.	301	1/2/2011	15,000	ACME, Inc.	999-9999	...
2	Elmer Fudd	123 Main St.	302	2/5/2011	500	Bugs Bunny	777-7777	...
2	Elmer Fudd	123 Main St.	302	2/5/2011	500	Bugs Bunny	777-7777	...
2	Elmer Fudd	123 Main St.	303	6/5/2012	800	Duffy Duck	666-6666	...
2	Elmer Fudd	123 Main St.	303	6/5/2012	800	Duffy Duck	666-6666	...

Note that in this table:

- Customers are repeated many times: a customer with n orders and m POCs will appear $n \times m$ times.
- on each customer, all combinations of Order and POC are shown. That is because Orders and POCs are *orthogonal*, that is, which orders a customer places do not depend on a particular POC, and who is a POC for a particular customer does not depend on that customer's (current) orders.

As a result, all data (that about customers, about orders, and about POCs) is repeated in the above table. The result can be pretty devastating for analysis that only wants to look at some of the data; an example of the problems that come up in this scenario is given in Sect. 5.1.

We close this section with one more consideration that is likely to appear in many real-life projects: time.

Example: Time in Databases

Often, databases record events that happen over time, hence temporal information is very important. Temporal information can make the schema of a database more complex. For example, in one of the examples above we described a clinical trial where participants had several vital signs checked daily. Note that a person has only one blood pressure *at a given moment*. Hence, when considering the person alone, the attribute is single-valued; however, when considering the person and the time of the measurement, it becomes a multi-valued attribute. Hence, we need to adjust our database accordingly. Assuming that all vital signs are measured together (at the same time), we could have tables

```
CREATE TABLE PARTICIPANT (
PId INT PRIMARY KEY,
....);

CREATE TABLE MEASUREMENTS (
PId  int FOREIGN KEY REFERENCES PARTICIPANT,
Date date,
Time time,
blood_pressure_sys int,
blood_pressure_dia int,
respiratory_rate int,
heart_rate int,
temperature float,
PRIMARY KEY (PId, Date, Time));
```

Note that we need to know the patient and the date and time to determine which vital signs we are talking about (if they were taken only once a day, patient and date would suffice).

We can now describe a database as a collection of tables where all tables are related—that is, each table has one (or several) foreign keys referencing other tables in the database or is referenced by other tables in the database (or both). The *schema of a database* is simply the collection of schemas of the tables in the database, including the foreign keys that link some tables to others. It is clear now that one of the reasons to give each table a primary key is so that it can be referenced by other tables in the database if need be.

Note that most approaches to data analysis require only relatively simple, tabular or near-tabular datasets that can be handled in a single table. However, it is important to be aware of the fact that databases can handle more complex cases whenever, for any of the reasons mentioned, data cannot fit in a single table. It is also important to know that data on a single table can always be normalized by splitting it into several, and that data in several tables can be combined to create a single table—both of those transformations are easy in SQL, as we will see.

2.3 Other Types of Data

Whether data fits into one table or several, it is still what we called *structured data* in Sect. 1.2. Relational databases were designed to deal with this type of data. However, over the years they have evolved to deal with semistructured and unstructured data as well. Here we discuss how to store such data in the database.

2.3.1 XML and JSON Data

As discussed in Sect. 1.2, XML and JSON are used with *hierarchical* or *tree structured* data. One way to see this type of data is as a chain of one-to-many relationships. What distinguishes this from the troublesome example we saw earlier is that the relationships are 'chained' together, with entities that are on the 'one' side of a relationship being on the 'many' side of another—hence forming a tree. Imagine, for instance, a university that is divided into various schools; each school is divided into several departments, and each department into several sections. Then we can think of the relation university–schools as one-to-many: the university is associated with several schools, but each school is associated with one university (in this case, there is only one university, but the pattern repeats at each level: a school is associated with several departments, but each department is associated with only one school. The same holds for departments and sections).

Hierarchical data can be stored inside the database in one of the two ways: by *flattening* it or by using the new XML or JSON data types added to the SQL standard.

In this first case, we can put hierarchies in tables in one of the two ways: normalized and unnormalized. Normalized data breaks the hierarchy into levels by representing each one-to-many relation in the hierarchy in its own table. Unnormalized data puts all the data in a single table by repeating data in one level as many times as needed to fill lower levels.

Example: Hierarchical Data

The hierarchy in Fig. 2.3 can be expressed in XML as

```
<superfamily name="Hominoidea>
  <family name="Hominidea">
     <subfamily name="Homininae">
      <tribe name="Homonini">
        <genera name="Hominini">
          <species name="Human"/>
       </genera>
       <genera name="Pan">
          <species name="Bonobo">
          <species name="Chimpanzee">
       </genera>
     </subfamily>
     <subfamily name="Ponginae">
        <genera name="Pongo">
          <species name="Orangutan"/>
        </genera>
     </subfamily>
    </family>
    <family name="Hylobatidae">
       <genera name="Hylobates">
          <species name="Orangutan"/>
       </genera>
    </family>
```

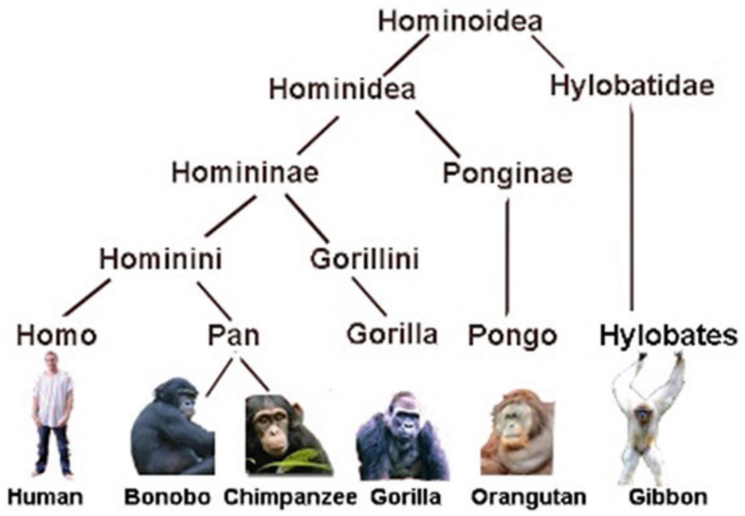


Fig. 2.3 An example of hierarchical data: biological classification

```
</superfamily>
```

In a flattened table, this becomes

Superfamily	Family	Subfamily	Tribe	Genera	Species
Hominoidea	Hominidea	Homininae	Hominini	Homo	Human
Hominoidea	Hominidea	Homininae	Hominini	Pan	Bobobo
Hominoidea	Hominidea	Homininae	Hominini	Pan	Chimpanzee
Hominoidea	Hominidea	Homininae	Gorillini	Gorilla	Gorilla
Hominoidea	Hominidea	Ponginae	Null	Pongo	Orangutan
Hominoidea	Hylobatidae	Null	Null	Hylobates	Gibbon

The schema has one attribute per hierarchy level, and the extension has one row per leaf value: values in higher levels are then repeated as necessary to fill in all rows (another way to see this is that each row represents the path from a leaf to the root). This causes redundancy, so we can split this table into a collection of tables:

Superfamily	Family
Hominoidea	Hominidea
Hominoidea	Hylobatidae

Family	Subfamily
Hominidea	Homininae
Hominidea	Ponginae

Subfamily	Tribe
Homininae	Hominini
Homininae	Gorillini

Tribe	Genera
Hominini	Homo
Hominini	Pan

Genera	Species
Homo	Human
Pan	Bobono
Pan	Chimpanzee
Pongo	Orangutan
Hylobates	Gibbon

Each table here represents the links connecting a pair of levels. This avoids repeating higher levels. Note, however, that if there are ‘holes’ in the hierarchy (as is the case here), reconstructing the original table from these smaller ones is, in general, *not* possible. For that, it is necessary to ‘fill in’ the holes with some artificial value, one different value per hole.

It is important to point out something about this example. We have seen before a situation with two one-to-many relationships, the example of customers, orders, and POC. There, there was a one-to-many relationship between Customer and Order (with Customer being the ‘many’ side) and another one-to-many relationship between Customer and POC (with Customer again being the ‘many’ side). However, orders and POCs were independent of each other. We called this case problematic, a situation that should be avoided.[9] However, in scenarios like the one above (with hierarchical data), we also have several one-to-many relationships, but no such problems. The crucial difference is in the “orientation” of the relationships. Suppose that instead of Customers, Orders, and POCs we have Customers, Orders, and Regions, where a customer is associated with one (and only one) region, but a region may have several customers associated with it. We again have two one-to-many relationships, one between Regions and Customers, and another one between Customers and Orders. However, now Customers play different roles in each relationship: it is the ‘one’ side with Regions and the ‘many’ side with Orders. This can be handled just like the examples shown in this section.

Another way to store the data above, especially if it comes with other (simple) data, is to create a table that has an XML or JSON attribute. In modern relational systems, a column can be declared as an XML or JSON type. This is a data type offered by the database system, so that such columns can contain data expressed in XML or JSON directly, without any need to flatten it.

Example: Hierarchical Data in XML/JSON

Using an XML type, the example above could be stored as follows:

```
CREATE TABLE HIERARCHY(
 ID INT PRIMARY KEY,
 Name character(50),
 Data XML);

INSERT INTO HIERARCHY VALUES(1, "Biological",
 "<superfamily name="Hominoidea>
    <family name="Hominidea">
       <subfamily name="Homininae">
        <tribe name="Homonini">
          <genera name="Hominini">
           <species name="Human"/>
```

[9] This is technically known as a *4th Normal Form* problem.

```
            </genera>
            ...
    </superfamily>");
```

Note that the XML value has been entered as a string, in quotes. A similar idea would work with JSON.

We will see later (Sect. 4.4) that it is possible to change data in XML/JSON to a flattened table using database functions, so that a large XML/JSON dataset can be loaded into the database directly as such and then transformed into a tabular format for analysis.[10] As stated earlier, most data mining and machine learning tools assume tabular data, so when presented with data in XML/JSON, one possibility is to load it into the database as such, using a table with an XML/JSON column, and then flattening this into a regular table, which can then be fed to the right tool.

2.3.2 Graph Data

Representing graph data in a relational database is very easy (doing interesting things with it is not so easy, as we will see in Sect. 4.6). There are, in fact, several options for dealing with graph data, but the most popular is to arrange the graph into two tables:

- A *nodes* table, where each node/vertex is represented by a row. The attributes that all nodes have in common constitute the schema of this table. If no primary key exists, some kind of identifier is generated and added to this table. For many real-life datasets, this works well since in many graphs nodes tend to represent the same type of entity and hence they are pretty homogeneous. In graphs where nodes are of several types, different tables may have to be used (one table per type). For instance, a graph that linked People and Books through a set of 'read' edges could use two separate tables, `People` and `Book`, to store the nodes.
- An *edges* table, where the links between nodes are stored. Each row in this table represents an edge, and the schema has, at the least, two attributes, one to represent each node involved in the edge (additional attributes can be used for graphs where edges are labeled or additional information is present). When the edges are directed, one attribute represents the *source* (or *origin*) nodes of each edge and another attribute the *destination* nodes, and edges are always interpreted as going from source to destination. When the edges are not directed, no distinction is made between node attributes. If it is necessary to make clear that the graph is undirected, we can either repeat each edge twice (once as (a,b),

[10]The opposite (data in a flattened table transformed into XML/JSON format) is also possible, but we do not cover it in this textbook.

another as (b,a)) or we can, in our queries, read the rows in either order (see Sect. 4.6 for examples).

Using generic SQL, the tables look like this:

```
CREATE TABLE nodes (
 id INTEGER PRIMARY KEY,
 name character(16) NOT NULL,
 feature1 datatype1,
 feature2 datatype2,
 ...);

CREATE TABLE edges (
 a INTEGER NOT NULL REFERENCES nodes(id)
           ON UPDATE CASCADE ON DELETE CASCADE,
 b INTEGER NOT NULL REFERENCES nodes(id)
           ON UPDATE CASCADE ON DELETE CASCADE,
 label character(256),
 PRIMARY KEY (a, b));
```

Example: Graph Data as Tables

The example of (fake) Tweeter users can be expressed in tables as follows:

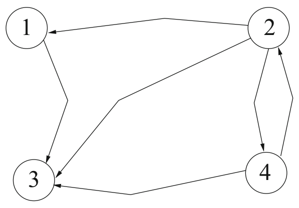
```
INSERT INTO NODES VALUES
   (1, ‘‘Shaggy’’, ...), (2, ‘‘Fred’’, ...),
   (3, ‘‘Daphne’’,...), (4, ‘‘Velma’’, ...);

INSERT INTO EDGES VALUES
   (1,2, ‘‘follows’’), (1,3,’’follows’’),
   (1,4,’’follows’’), (3,4,’’re-tweets’’), (2,3,’’likes’’);
```

Another approach is to store graphs as matrices. For readers unfamiliar with the concept, one can think of a matrix as a representation of a set of values in a rectangle, determined by rows and columns. For instance,

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

is a matrix with 2 rows and 3 columns representing 6 data values, a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, and a_{23}. Each value a_{ij} sits in row i, column j.

The *adjacency matrix* of a graph with n vertices is an $n \times n$ matrix where each row/column represents a vertex in the graph (vertices in the graph are numbered $1, \ldots, n$ to facilitate this representation), and entry (i, j) in the matrix (the entry in row i, column j) represents information about the edge between nodes i and j, if such an edge exists. A *Boolean* adjacency matrix simply has a 1 to indicate that the edge exists and a 0 to indicate that it does not exist. In the case of matrices



$$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

Fig. 2.4 Simple graph and its Boolean adjacency matrix representation

with weights in the edges, we get a *edge weight* matrix by setting entry (i, j) to the weight of the edge between i and j if such an edge exists, 0 otherwise. As an example, we show in Fig. 2.4 a very simple graph and its Boolean adjacency matrix. The graph has 4 nodes (nodes are numbered to make the representation clear) so the matrix is 4×4 (4 rows and 4 columns); by convention, rows are numbered top to bottom (so top row is row number 1) and columns are numbered left to right (so rightmost column is column 1).

The matrix M can then be represented via a table with schema *(row,column, value)*, where *row* and *column* are the integers giving the row and column of the matrix and *value* giving the appropriate entry. That is, if $M(i, j) = v$, we add row (i, j, v) to the table (it is a convenient convention not to add a row whenever $M(i, j) = 0$. This helps save lots of space, especially for *sparse* graphs, those with a low number of edges). Note that this is essentially the same as the EDGE relation described above; however, matrix algorithms treat this data differently. As we will see in Sect. 4.6, SQL can do matrix multiplication, sum, difference, multiplication by scalar, and transposition but cannot do more complex stuff (like finding linear independence (rank), determinants, eigenvector, and eigenvalues). Still, quite a few algorithms require nothing more complex than matrix multiplication, so this representation may be useful in some cases.

The matrix representation of a graph has some interesting properties:

- if there are no self-loops in the graph (no edges between a node and itself), the *diagonal* of the matrix (the collection of all the entries (i, i), that is, in row i and column i, for $i = 1, \ldots, n$) is all zeroes.
- if the graph is undirected, the adjacency matrix is *symmetric*, that is, entry (i, j) is the same as entry (j, i). This is not the case for a directed graph, where we store an edge only in one 'direction.'
- the most interesting property (for our present purposes) of adjacency matrices is that by multiplying the matrix by itself we get information about paths in the graph.[11] Let M be an adjacency matrix for graph G; then

[11]For readers not familiar with matrix multiplication, we explain the operation (and SQL code for it) in Sect. 4.6.

- $M^2 = M \times M$ represents paths of length 2 in G, that is, pairs of edges where the first edge goes from a node i to a node j, and the second edge goes from node j to a node k—this is a path of length 2 between i and k. If M is Boolean, then $M^2(i, j) = 1$ if and only if there is a path from node i to node j of length 2. If M is weighted, then $M^2(i, j)$ is the weight of the path of length 2 between i and j (if one exists; 0, otherwise).
- The above is true for any lengths, not just 2. In general, M^l (the product of M with itself l times) contains information about paths of length l. In particular, M^n is the adjacency matrix of the transitive closure of M (recall that n is the number of nodes in G, so there cannot be paths longer than n in G if we do not admit loops). Thus, we can use matrix multiplication to find out the existence of paths (using a Boolean matrix) or their length (using a weight matrix), even shortest paths.

While it is not convenient or possible to always use SQL for graph algorithms, there are some basic graph analysis tasks that can be carried out in the database. If nothing else, the database can be used to store and maintain very large graphs, which can later be downloaded (perhaps in small chunks) for analysis with graph tools, which tend to not do well with graphs that are larger than what a computer can handle in memory.

2.3.3 Text

Text refers to data expressed as fragments of natural language (English, Chinese, etc.). The unit of text is usually called a *document* and it may refer to something very short (a tweet, an email body) or long (a whole book). As stated earlier, usually text comes as a collection of documents, called a *corpus*; in some cases, documents in a corpus come with some structured data. For instance, a collection of emails may have attributes like *sender*, *receiver*, *date*, *subject*, and *body*, where *subject* is a short string and *body* is a string of arbitrary size, which can be considered text.

Text can be stored as an attribute of a table by taking advantage of a data type available in most systems: the *text* type. Essentially, this is a very long string, but in most systems it comes with functions that facilitate manipulation and analysis of strings (text analysis is described in Sect. 2.3.3). In Postgres, *text* is a variable, unlimited length string. In MySQL, there are types TINYTEXT (up to 256 characters), TEXT (up to 64,000 characters, approximately), MEDIUMTEXT (up to 16 million characters, approximately), and LONGTEXT (up to 4 billion characters, approximately). Even if the data is simply a corpus with no attached structured data of any type, it is possible to create a table with a single attribute in the schema, of type text, and to store each document in a row. This will enable basic analysis of the text like *keyword search* or *sentiment analysis* (see Sect. 2.3.3 for details).

Example: Text Data

In the case of the Hate Crime dataset from ProPublica, we had schema *Article-Date, Article-Title, Organization, City, State, URL, Keywords, and Summary*. As we saw, the last two attributes could be quite long and probably need to be analyzed in terms of their contents, so it makes sense to make them of type text. In Postgres:

```
CREATE TABLE HATE-CRIME(
article-date Date,
article-title char(256),
Organization char(128),
City char(64),
State char(2),
URL char(32),
Keywords Text,
Summary Text);
```

An insertion into this table looks as follows:[12]

```
INSERT INTO HATE-CRIME VALUES(
"3/24/17",
"Kentucky Becomes Second State to Add Police to Hate Crimes
Law",
"Reason",
"Washington",
"District of Columbia",
"http://reason.com/blog/2017/03/24/kentucky-becomes-second-
stateto-add-pol",
"add black blue ciaramella crime delatoba donald gay hate
 law laws lives louisiana matter police trump add black
 blue ciaramella crime delatoba donald gay hate law laws
 lives louisiana matter police trump",
"Technically, this is supposed to mean that if somebody
 intentionally targets a person for a crime because they are
 police officers, he or she may face enhanced sentences for
 a conviction. That is how hate crime laws are used in cases
 when a criminal targets");
```

Exercise 2.4 In the example of email collection, the data can be stored in a table with schema (email-id, sender, receiver, date, subject, body). Create a table in Postgres or MySQL for this dataset using text type for the last two attributes and try to insert some Enron data (or made-up data) into this table.

[12]We show each value in a separate line for clarity; it is not necessary to enter values this way. Putting the values in quotes and separating them by commas is what really matters.

2.4 Getting Data In and Out of the Database

Once a table or tables have been created in a database, it is time to bring in the data. We can also, once we have the data, make changes to it, delete (some subset of) it, or add more whenever additional data is available. The following subsections describe these activities.

2.4.1 Importing and Loading Data

There are several ways to do put data in the tables of a database, depending on the circumstances, but the two most common ones are to insert tuples or to load in bulk. The first one (which we have already seen) uses the `INSERT` SQL statement. This statement allows us to add one row (or a few rows) to an existing table, and it can be used whenever we need to add data in small quantities and we want to have total control on how data is entered in the database. However, this statement is too tedious and error-prone when we already have a non-small dataset that needs to be added to the database.

If data is already in a file, it is possible to *load* the data into the database in one swoop. All database systems have some command which takes a file name and a table name and brings in the data from the file into the table—as far as the data in the file is compatible with the schema of the table. This command assumes that the data in the file can be broken down into lines, each line corresponding to a row/tuple for the table.

The *load* command is system dependent, although the basic outline is the same for most systems. Generally, one specifies the location of the file in the computer, the table to load into, and provides a description of how the lines in the file are to be broken down into the values that make up a row/tuple by indicating how values are separated from each other and a few other characteristics. For instance, the typical `csv` files use the comma as a separator; other typical separators are the semicolon character or the tab. The table must already exist before this command is used, and the data in the file must fit into the schema of this table. The loader will read the file line by line and split each line according to the given instructions. It will expect that there are as many values on each line as there are on the schema of the table and of the right type. When this is the case, the loader will create a tuple/row for each line in the file and will insert it in the table.

However, datasets often come with *dirty data* and this creates a problem when trying to load the data in the database. Some of the most common problems are:

- missing values. Lack of values can be manifested in a file in two ways: by an *empty field* (that is, two consecutive appearances of the delimiter) or by some marker (markers like '\N' or 'NA,' for 'Not Available,' are particularly common). Dealing with empty fields is relatively straightforward. Some systems will automatically set the corresponding attribute in the table to a *default* value:

the empty string, for string types; zero, for numeric types; and the date or time 'zero' for dates and times. Others will create a NULL marker in the database. However, dealing with markers tends to be messy. The first problem is that different datasets may use different conventions to mark missing values; the dataset may need to be explored *before* it can be loaded in the database to identify such markers.[13] The second problem is that null markers may confuse the loader about the type of data it is reading. In numerical attributes, the system expects strings that can be transformed into numbers (essentially, strings made up of digits and optionally a hyphen (-) or a period (.). Even numbers with commas can create problems). When a string like 'NA' is found, the system is unable to transform it into a number or recognize it as a missing value marker. The same problem happens with temporal information, where the system expects a string in a certain format that it can parse and recognize as a date or time. Even in string based values, the system may likewise confuse a string like 'NA' with a valid value, not a missing value marker. The best way to avoid errors is system dependent: in some systems, it is better to delete unrecognized markers and leave empty fields; in others, it may be necessary to create a special value of the right type (for instance, using -1 for a numeric field that contains only positive values).

- strings are represented differently in different datasets. In some cases, strings are stored by surrounding them with single quotes ('), sometimes with double quotes ("); sometimes they are stored without any quotes. This can cause confusion in the loader, especially when strings include characters other than letters or numbers. For instance, the string "Spring, Summer, Fall, Winter ... and Spring" is the title of a movie, but it includes 3 commas and 3 dots and may confuse a loader when stored in a CSV file—the commas on it may be confused with separators.[14] Again, solutions depend on the system: some systems are smart enough to leave everything in quotes alone (in which case, making sure that all strings are surrounded by quotes is the way to go); others may require that those extra commas go away.
- dates and times are typically recognized by most database systems if they follow a certain format (we discuss such formats in detail in Sect. 3.3.1.3). When the data in the file does not conform to the format, the system tends to read it as a string.

Different systems use different tricks to help deal with these problems. In MySQL, one would write

```
LOAD DATA INFILE filename
INTO TABLE table
 [FIELDS [TERMINATED BY string]
         [ENCLOSED BY char]
         [ESCAPED BY char]]
```

[13]Command line tools are the appropriate tools for this task [9].

[14]This example is from a dataset in the `Imdb` website.

in order to load data from file `filename` into table `table`. The optional TERMINATED BY clause allows the user to specify the character that separates one field from the next; the default is tab, but it can be changed to comma or another character with this clause. The optional ENCLOSED BY clause allows the user to indicate how values of type string are enclosed; usually, this is a quote or a double quote, although other values can be used. Finally, the optional ESCAPED BY is used to indicate how to handle special characters: these are usually indicated by a backslash ('\') followed by a character. For instance, '\n' denotes the newline (linefeed) character, a control character used to separate lines. In addition, a statement `IGNORE n LINES`, where *n* is a positive integer, can be used if we need to not load the first *n* lines of the file; in particular, `IGNORE 1 LINES` is used when the data file contains a *header*, which we do not want to load (it could be confused with data).

Exercise 2.5 Load data from the file ny-flights.csv into the table `ny-flights` in MySQL. Caution: this can be a long process, depending on the computer. As an alternative (and to be able to fix errors easily), start with a small sample by choosing the first 1000 lines or so.

In MySQL, it is possible to perform some data transformations on the LOAD command by using *user variables*. User variables are labels that start with '@'; they can be used in assignment operations, as the following example shows.

Example: LOAD and Data Transformation in MySQL

The following example assumes a file `T` with schema `(column1,column2)`; it uses the first input column directly for the value of `T.column1` and divides the value of the second column in the file by 100 before using for `T.column2`:

```
LOAD DATA INFILE 'file.txt'
  INTO TABLE T
  (column1, @var1)
  SET column2 = @var1/100;
```

This approach can be used to supply values not derived from the input file. The following statement sets `T.column3` to the current date:

```
LOAD DATA INFILE 'file.txt'
  INTO TABLE T
  (column1, column2)
  SET column3 = CURRENT_DATE;
```

You can also discard an input value by assigning it to a user variable and not assigning the variable to a table column:

```
LOAD DATA INFILE 'file.txt'
  INTO TABLE T
  (column1, @dummy, column2, @dummy, column3);
```

This statement reads and uses the first, third, and fifth columns of the file and reads but ignores the second and fourth columns.

When the LOAD DATA statement finishes, it returns a message indicating how many records (lines) were loaded, how many were skipped (because of some problem), and how many warnings the LOAD process generated. This can be used to check whether there were problems and, if so, how many.

In Postgres, the equivalent command is called `COPY`. COPY FROM copies data from a file to a table (appending the data to whatever is in the table already). The format is

```
COPY table_name [ ( column\_name [, ...] ) ]
 FROM  'filename'
 [ [ WITH ] ( option [, ...] ) ]
```

The optional WITH option is used to give different hints as to how to handle the data. The most important option is FORMAT, which determined the data format to be read or written: TEXT (default), CSV, or BINARY. Another useful option is NULL AS, which is used to specify which characters or strings should be interpreted as NULLS; this is very useful when the dataset uses its own convention to mark missing values (the default in CSV is an empty value with no quotes, that is, two commas together or a comma at the end of the line). Finally, QUOTE AS can be used to tell the system how strings are quoted (single or double quotes; the default in CSV mode is double).

When the TEXT format is used, COPY FROM will raise an error if any line of the input file contains more or fewer columns than are expected. Hence, it is important to make sure that the right delimiter and the right options are used, so that each line can be parsed correctly.

Option HEADER specifies that the file contains a header line with the names of each column in the file. This option is allowed only when using CSV format.

Example: Loading Data in Postgres

To copy data from a csv file containing a header, we would use a command like the following:

```
COPY ny-flights FROM '~/DATA-MNGMNT/DBS/ny-flights.csv'
 DELIMITER ',' WITH FORMAT CSV HEADER;
```

Note that the DELIMITER specification is redundant in this case.

Exercise 2.6 Load data from the file ny-flights.csv into the table `ny-flights` in Postgres.

2.4.2 Updating Data

Once data is in the table, we may need to modify some of it. Modifications are accomplished with two SQL commands: DELETE and UPDATE.

The DELETE command is used to delete existing data. The format of this command is

DELETE FROM table-name WHERE condition

Here condition is an expression that is evaluated in each row in table-name and results in a True or False value. In its simplest form, it involves an attribute name from the schema of the table, a comparison operator (like '=,' '≤,' <) and a constant. For instance, in table Chicago-employee, a condition could be: name = 'Jones'. In each row of the table, the system will look up the value of attribute name and see whether this value is 'Jones' (in which case the condition evaluates to True) or something different (in which case the condition evaluates to False). More complex conditions can be built from simple ones (conditions are explained in depth in Sect. 3.1). The DELETE statement works as follows: on each row of the table, the condition is checked. If it returns True, then the row is deleted. If the condition returns False, the row is left unchanged on the table. Clearly, the condition controls the effect of this statement, and it must be written carefully. In particular,

- if the condition is not true in any row of the table, the command has no effect, as the table is left unchanged.
- if the condition is true in every row of the table, all the data in the table is erased (this will also happen if no condition is given, see below). The table is now empty.

It is customary, when using this command, to use a primary key or a foreign key in the condition, in order to control exactly where the command is applied.

Example: Table Deletion

Suppose that in table ny-flights we are told that the flight from EWR to IAH on January 1st, 2013 was canceled. We could use this information directly in SQL to update the data:

```
DELETE FROM NY-FLIGHTS
WHERE year = 2013 and month = 1 and
      day = 1 and origin = 'EWR' and dest = 'IAH';
```

But this will delete any tuple that fulfills the conditions—that is, if there are several flights on that day from EWR to IAH, all of them will be deleted. A better idea is to find out the flight id of the canceled flight uses a condition like id = ... to make sure we are singling out for deletion only the right flight. In general, using a key in the condition (primary or unique) is the safe way to proceed.

Exercise 2.7 Delete all flights that go from JFK to ATL in the afternoon (departure time between noon and 5 pm).

The DELETE command can be issued *without* a WHERE clause, in which case it will delete all the data in the table (in other words, in the absence of a condition, the command applies to every row). Note the difference with the DROP TABLE command: the DELETE will get rid of the data, but leave the table (empty) in place;

the DROP TABLE will get rid of table and data. After a `DELETE`, we can continue using the table (for instance, putting some data on it again); after the `DROP TABLE`, the table itself is gone so if needed, it would have to be recreated (with a `CREATE TABLE` statement).

The `UPDATE` command is used to change existing data. By 'updating' a row we mean to change the value of one or more of the attributes for that row (this is called 'modifying' the row in some textbooks). The format of this command is `UPDATE table-name SET attribute = value WHERE condition`

Here, `condition` is exactly the same as in the DELETE statement; changes are made only to tuples on which the condition is true. In addition, the SET clause tells the system exactly what changes to make: `attribute` is the name of an attribute in the schema of the table, `value` is a value of the right data type. It is possible to make several changes at once by giving a sequence of `attribute = value` expressions separated by commas. Any attributes not mentioned are left unchanged.

Example: Table Update

Suppose that in table `ny-flights` we are told that the flight with id 1 did have flight number 8501, not 1545 as it appears in the data. We can change the data with

```
UPDATE NY-FLIGHTS SET flight = "8501" WHERE id = 1;
```

As in the case of deletion, we can use any condition, but it is best to use a condition involving a primary key to make sure we only change the row (or rows) that we want to change.

It is important to make clear that the old data is gone forever. If we want to keep old data and just register that there has been a change, we can create *versions* of data. For this, a temporal attribute is added to the table to reflect when data is valid; instead of updating existing values, new rows are inserted with the new value, and old ones are left in place—the temporal attribute is used to tell which row reflects the *current* situation and which ones are *historical* data. This may be important in applications that want to examine changes over time, and it is the foundation of *time series* data.

Note that one could change a row by first deleting it and then inserting it with the new values. However, doing an update is almost always preferred, as it is much more efficient.

Exercise 2.8 Change the destination of all flights that fly into JFK to LGA.

2.4.3 Exporting Data

Sometimes we are interested in taking data *from* the database and putting it in a file, so we can use it with other tools. This is called *exporting* or *dumping* the data. We also want, from time to time, to make a copy of data in the database in case there is a serious problem, to avoid data loss.

The process for saving the data into a file for *backup* purposes is also, like the load, system dependent, although the idea is basically the same for most databases. In MySQL, there are two basic procedures for exporting data. The first one is to use the reverse of the LOAD statement:

```
SELECT columns INTO OUTFILE filename
 [FIELDS [TERMINATED BY string]
        [ENCLOSED BY char]
        [ESCAPED BY char]]
FROM table-name;
```

This command will take the data in table `table-name` and put it in the file `filename`. If the file already exists, this statement gives an error; the user should specify a new file name for this command. As we will see later, this command can be used to extract only certain parts of a table or database into a file. This is useful when we want to carry out focused analysis (using R or other tools) on part(s) of a large database.

The other way to get data out of a MySQL database is to execute a `mysqldump` command. This command generates an SQL script, that is, a file with SQL commands; it is customary to give such files a `.sql` extension. The script contains step-by-step instructions (in SQL) to recreate a table or a database. For a single database, the command is written (in the command line):

```
mysqldump --databases database-name > mydump.sql
```

or as

```
mysqldump database-name > mydump.sql
```

Both commands instruct the system to save the script as a data file called `mydump.sql`. The difference between the two preceding commands is that without `-databases`, the dump output contains no CREATE DATABASE or USE statements.

To dump only specific tables from a database, name them on the command line following the database name:

```
mysqldump database-name table-name > mydump.sql
```

To reload a dump file written by mysqldump that consists of SQL statements,

```
mysql < mydump.sql
```

Alternatively, from within MySQL, use a source command:

```
mysql> source dump.sql
```

Both commands execute all the statements inside the SQL script `mydump.sql`, recreating whatever data was on the database or file we dumped.

In Postgres, one can also reverse the COPY FROM command using COPY TO:

```
COPY { table_name [ ( column_name [, ...] ) ] | ( query ) }
    TO 'filename'
    [ [ WITH ] ( option [, ...] ) ]
```

Besides getting data from an existing table (using `table_name`), one can pick certain data from the database using the 'query' option. A *query* is the name for a process to pick certain data in the database; it is a fundamental tool in data analysis, and it is explained in the next chapter.

In the options, DELIMITER specifies the character that separates columns within each row (line) of the file (the default is a tab character in text format, a comma in CSV format); HEADER specifies that the schema of the table is written to the first line of the file. The filename must be specified as an absolute path. As in the case of MySQL, this command can be used to extract only certain parts of a table or database.

The command equivalent of `mysqldump` in Postgres is called `pg_dump`. To dump a single table into a file, the syntax is

```
$ pg_dump -t table-name mydb > filename.sql
```

As before, this creates a SQL script. To get the data back from the file into the table, use the command

```
$ psql -d new-table-name -f filename.sql
```

A whole database can also be dumped into a file:

```
$ pg_dump database-name > filename.sql
```

As before, this SQL script can be used to recreate the database from scratch with:

```
$ psql -d new-database-name -f filename.sql
```

The command `pg_dump` can also be used to archive a database, by using the `-F switch`. For instance,

```
$ pg_dump -Fd database-name -f dumpdir
```

will create a directory-like archive of the given database. To restore this archive, the command `pg_restore` is used:

```
$ pg_restore -d new-database-name dumpdir
```

Exercise 2.9 In either Postgres or MySQL, export the table `ny-flights` into an SQL script. Drop the table and restore it using the saved script.