Chapter 3
Data Cleaning and Pre-processing

In this chapter, we introduce the basic tool that SQL uses to extract information from a database, that is, the `SELECT` statement. We then show how to use it to carry out some basic tasks to be done during the Exploratory Data Analysis (EDA), data cleaning, and pre-processing stage of the data life cycle (see Sect. 1.1). After introducing the basic blocks of `SELECT` in the next section, we discuss EDA in Sect. 3.2, data cleaning in Sect. 3.3, data pre-processing in Sect. 3.4, and the implementation of *workflows* in Sect. 3.5.

3.1 The Basic SQL Query

SQL offers a statement, called the `SELECT` statement because of its initial keyword, in order to ask questions (in database parlance, *queries*) of the data. This statement allows us to extract information from the database, but it is also used when manipulating data. It is the most useful, used, and complex of all SQL statements. In this chapter, we show its basic structure; a discussion of additional features is left for Chap. 5.

In its basic form, the `SELECT` statement is written as

```
SELECT result-list
FROM data-sources
WHERE condition;
```

where

- `result-list` is a list of attribute names, or functions applied to attribute names (see Sect. 3.1.2). This list determines what gets retrieved from the database; hence, this list represents what information we are interested in getting as an answer to our question.

A. Badia, *SQL for Data Science*, Data-Centric Systems and Applications,
https://doi.org/10.1007/978-3-030-57592-2_3

- `data-sources` denotes where the data examined for this query comes from. There are basically two types of data sources in SQL:
  - Tables from the database: the FROM keyword is followed by one or more table names, separated by commas. The data in the extension of the table(s) is used for processing the query.
  - Another query: a whole query, i.e. an expression:
    ```
    (SELECT .... FROM ... WHERE ...) AS new-name
    ```
    can be used inside the FROM clause. This query is followed by the expression `AS new-name` because it has be given a name. The rationale for this is that, as we will see, all queries in SQL return a table as an answer. Hence, queries (including subqueries) can be seen as denoting a table (the answer to the query). When there is a query A (called a *subquery*) in the FROM clause of another query B (called the *outer* or *main* query), the system first evaluates A and uses the result of this evaluation (which is a table) as one of the data sources for evaluating B. The reason that subqueries use `AS table-name` to give their result a name is that this name is then used in the main query to refer to the result. Since B acts upon the results of A, using subqueries is a very useful tactic for computing complex results by breaking the process down into steps: the data is prepared with query A, and the final result is obtained with query B. We will see many examples where this approach is used.

  It is possible to mix, in the same FROM clause, table names and subqueries. This is because in the end, both table names and subqueries denote table extensions, i.e. collections of data. When more than one table is used, the extensions of all the tables are combined into a single table/extension. Exactly how this is done depends on the context and is discussed in depth in Sect. 3.1.1.
- `condition` is an expression that is evaluated on each row of the data extension, and it returns True or False. This is similar to the conditions used in the DELETE and UPDATE statements (see previous chapter). A simple condition compares the value of some attribute with a constant. More complex conditions can be formed by taking the conjunctions or disjunction of two conditions, or the negation of a condition.

Each query is evaluated as follows: the collection of rows denoted by the data sources in FROM are combined into a single table (if there are subqueries, these are processed first to obtain a result/table for each subquery); then the condition in WHERE is evaluated on each row. Rows where the condition is not True are disregarded, and those where the condition is True are kept (the WHERE clause may be absent from a query; in this case, all rows of the data source(s) are used). Using these filtered rows, the values of the attributes in `result-list` are picked up. Note that, for this evaluation to work, it must be the case that every attribute mentioned in the `condition` or in the `result-list` is present in the schema of

the tables in FROM. If this is not the case, the system will not attempt to evaluate the query; it will return an error message instead of a result.[1]

Example: Simple Query

Looking at the data in table ny-flights, we may want to answer several questions. For instance, we may ask: are there any flights into JFK on November 10?

```
SELECT id
FROM NY-FLIGHTS
WHERE year = 2013 and month = 11 and day = 10
      and dest = "JFK";
```

In this example, the data source is the table ny-flights; the condition is a conjunction of 3 simple conditions, and only the attribute id is retrieved. This is an *existential* query (one asking if data fulfilling some constraints does exist), so retrieving the primary key is enough. The system evaluates this query as follows: the extension of table ny-flights (all the rows on it) is examined; on each row, the condition of the WHERE clause is evaluated (all 3 simple conditions are applied, and if all 3 return True, the whole condition returns True).[2] The collection of rows where the condition returned True is then processed; on each row, the system picks the desired attributes, as specified in the SELECT (in this case, id).

Example: Another Simple Query

Using again table ny-flight, we ask where those flights coming into JFK on December 10 are coming from, but only for trips longer than 1,000 miles. We can reuse our previous query as follows:

```
SELECT origin
FROM (SELECT id, distance, origin
      FROM ny-flights
      WHERE year = 2013 and month = 11 and
            day = 10 and dest = "JFK") AS T
WHERE distance > 1000;
```

In this example, the evaluation proceeds as follows: the subquery in the FROM clause is done first, starting with table NY-FLIGHTS and applying the WHERE conditions to it (so only flights from 11/10/2013 and destination 'JFK' are used) resulting in a (temporary) table named T (as indicated by the AS T)[3] with 3 columns

[1] This is typically the case when the user makes a typo or forgets the exact name of an attribute.

[2] Complex conditions in SQL are evaluated using the traditional rules of Boolean logic: for A and B to be true, both A and B must be true; for A or B to be true, it is enough that one of A or B is true (but it is okay if both are); for not A to be true, A must be false.

[3] In some systems, including MySQL and Postgres, giving an *alias* (a temporary name) to any table created by a subquery in FROM is required; in some, it is optional.

(attributes `id, distance, origin`); using this table, the main query is run: the condition `distance > 1000` is evaluated, and for all surviving rows, the `origin` attribute is picked. Note that in the subquery we selected attributes `distance` and `origin` because they were necessary in the following step. Note also that in this case, we could have written the query in one step, as follows:

```
SELECT origin
FROM ny-flights
WHERE year = 2013 and month = 11 and
      day = 10 and dest = "JFK" distance > 1000;
```

It will often be the case that there is more than one way to write an SQL query. Using subqueries is convenient in complex cases, as it breaks down a problem into simpler sub-steps. We will see other examples where using a subquery is a good idea.

Conditions depend on the type of the attribute(s) involved. There are specific conditions for numbers, strings, and dates. We introduce a few basic ones here, and more as we go along. For number, arithmetic conditions (comparisons using $<$, $\leq$, $=$, $>$, $\geq$) are common. For strings, it is possible to compare two strings with equality, but it is also possible to compare a string with a *string pattern*, an expression that requires the occurrence (or non-occurrence) of certain characters in the string in a certain order. The SQL standard requires the predicate LIKE, which takes a string and a simple pattern and compares them. Suppose, for instance, that we are unsure about an airport name, then a predicate like

`dest LIKE 'SD_'`

stipulates that the value of `dest` must start with 'SD' but has an additional character after that (the '_' stands for any character). The '*' symbol can also be used in patterns for LIKE; it stands for 'no character, or any one character, or any string of characters'). Most systems allow for much more complex patterns, including what is called a *regular expression*. We do not cover regular expressions in this book, but we discuss string functions in some depth in Sect. 3.3.1.2.

Another available predicate (both for numbers and strings) is IN. It compares an attribute to a list of constants; if the value of the attribute equals any of the constants, the IN predicate is satisfied.

Example: IN Predicate

We want to retrieve all flights that go into New York City. However, going to New York can be accomplished by flying into any of its two airports (JFK and LGA) and even by flying into neighbor New Jersey (Newark, EWR). Any one of them will do, so we write

```
SELECT *
FROM ny-flights
WHERE dest IN ("JFK", "LGA", "EWR");
```

Note that IN is equivalent to a disjunction: `attribute IN (`$value_1$`,...,`$value_n$`)` is semantically equivalent to

`attribute =` $value_1$ `or ...or attribute =` $value_n$[4]

so we could have written

```
SELECT *
FROM ny-flights
WHERE dest = "JFK" or dest = "LGA" or dest = "EWR";
```

Exercise 3.1 Select all the flights from `ny-flights` that fly on weekends (Saturdays or Sundays).

Finally, a predicate called BETWEEN is also offered. Like IN, it is really a shortcut. A predicate of the form `attribute BETWEEN` $value_1$ `and` $value_2$ is short for `attribute` $\geq$ $value_1$ `and attribute` $\leq$ $value_2$. This predicate requires an order in the attribute domain (i.e. the attribute must be an ordinal or a number); that is why it is commonly used with dates and numbers.

Example: BETWEEN Predicate

Suppose we are interested in the origin of all flights that landed between 4 and 5 am. The query

```
SELECT origin
FROM ny-flights
WHERE arr_time BETWEEN 400 and 500;
```

will retrieve all information about such flights.

Exercise 3.2 Select all the flights from `ny-flights` that fly longer than 500 but less than 1000 miles.

It is possible to refer, in the SELECT clause (and other clauses that we introduce shortly), to the attributes of a table by a number. This number refers to the position of the attribute in the table, as given when the table was created. That is, if the command:

```
CREATE TABLE foo(
A integer,
B varchar(48),
C float);
```

[4]This equivalence, like many others, only holds if there are no nulls involved. See the discussion of null markers in Sect. 3.3.2.

is used, attribute `A` can be referred to as 1, attribute `B` can be referred to as 2, and attribute `C` can be referred to as 3. The query

```
SELECT  1, 3
FROM foo;
```

is equivalent to

```
SELECT  A, C
FROM foo;
```

However, this practice is not recommended, as it can make queries difficult to read. The attributes retrieved in the SELECT clause become the schema of the result table, and it is a good idea to make sure we are retrieving exactly what we want. These attributes can be given a new name in the result, using the AS keyword:

```
SELECT dest AS destination, flight  AS flight-number
FROM ny-flights;
```

will produce a result with schema `(destination, flight-number)`. This is usually done when saving the result to be used later or shown to others; we will see how to save the results of a query in Sect. 3.5. A query can also be used with the command to download data from the database into a file, so this mechanism can also be used to give our data meaningful names for data sharing.

What if we want *all* the attributes of a table? A real-life table may have dozens of attributes; listing them all makes writing queries long, tedious, and error-prone. SQL provides a shortcut: the '*' ('star' or 'asterisk') can be used in a `SELECT` clause to mean "all attributes."

Example: Getting All Attributes

The query

```
SELECT *
FROM ny-flights
WHERE dest = "JFK" and arr_time < 800;
```

will return all information (all attributes in the schema) about flights coming from "JFK" and arriving before 8 am.

One final thing to note about the results is that sometimes we may get *duplicates*, that is, two or more rows that contain the exact same values. This may happen because the table that we used in the FROM clause contained duplicates or because duplicates are created while processing the query.[5] When we want to get rid of duplicates, SQL provides the keyword DISTINCT that can be added to SELECT to signal to the system that we want duplicates eliminated.

[5]Recall that we said that the same tuple may be inserted multiple times in a table. But even if all rows in a table are different from each other, all this means is that two arbitrary rows are

Example: Eliminating Duplicates

The query

```
SELECT carrier, origin
FROM ny-flights;
```

will retrieve the airlines and the places that they fly from in our New York flight dataset. Note that the rows of the original table are likely all different, but there also very likely share values for some attributes. In this example, each row in the original table is a different flight; however, when focusing on carrier and city, we are going to see duplicates if the same carrier has more than one flight starting at a given city. To eliminate these duplicates, we could use

```
SELECT DISTINCT carrier, origin
FROM ny-flights;
```

The typical use for DISTINCT is to identify the different values that make up an attribute, especially a categorical/nominal one. When applied to a numerical attribute, DISTINCT will produce all unique values, and this will make a difference when applying arithmetic or other mathematical functions, as we will see.

Exercise 3.3 Make a list of all the carriers mentioned in the `ny-flights` dataset. That is, show all the unique contents of the `carrier` attribute.

3.1.1 Joins

As stated in the previous section, a FROM clause may mention more than one data source. If that is the case, the system combines all data sources into a single table. This is accomplished through one of two ways: *Cartesian products (henceforth, CPs)* and *joins*. Both play an important role in Data Analysis.

When the tables are simply listed in the `FROM` clause, the system will automatically produce the *Cartesian product* of all tables named and/or denoted by subqueries. The Cartesian product of two tables T and R yields a single table with a schema, which is the concatenation of the schemas of T and R (i.e. all attributes in T followed by all attributes in R); tuples for this table are created by combining tuples in T and tuples in R (all tuples in T are combined with all tuples in R).

different from each other *in at least one attribute*. Two rows may coincide in some attributes and be different in others. If our query retrieves only those attributes where the rows coincide, we will obtain duplicates.

Example: Cartesian Product

Explaining the behavior of CPs is best done through an (artificial) example. Assume tables `T(A,B,C)` and `R(D,E)`, with these extensions:

T		
A	**B**	**C**
a_1	b_1	c_1
a_2	b_2	c_2
a_3	b_3	c_3

R	
D	**E**
d_1	e_1
d_2	e_2

In SQL, the CP is written as:

```
SELECT *
FROM T,S;
```

This produces a table with schema `(A,B,C,D,E)` and the following tuples:

CP of **T, R**				
A	**B**	**C**	**D**	**E**
a_1	b_1	c_1	d_1	e_1
a_1	b_1	c_1	d_2	e_2
a_2	b_2	c_2	d_1	e_1
a_2	b_2	c_2	d_2	e_2
a_3	b_3	c_3	d_1	e_1
a_3	b_3	c_3	d_2	e_2

Note that **T** has 3 tuples and `R` has 2 tuples, and their CP has $3 \times 2 = 6$ tuples. Each tuple in the CP is made up of a tuple from `T` (giving values to attributes `A`, `B`, and `C`) concatenated from a tuple from `S` (giving values to attributes `D` and `E`).

CPs can be used to combine more than two tables by combining them pair-wise: given 3 tables `T`, `R`, and `S`, we can take the CP of `T` and `R` and then the CP of the resulting table and `S`, to produce a single table. Because CPs are associative, we can do this in any order, and the result is still the same.

Example: A Common Cartesian Product

Sometimes when analyzing data, we need to compute a single value. In SQL, this will be done with a query. As stated earlier, all SQL queries return a table; what happens in this cases is that we have a table with a single attribute and a single row. For instance, a query may compute a single value v called `F`:

F
v

It is a common pattern to combine such a result with a data table. The Cartesian product of the above table R and table T from the previous example yields this result:

A	B	C	F
a_1	b_1	c_1	v
a_2	b_2	c_2	v
a_3	b_3	c_3	v

The net effect is to add the single value v to each row of T; now the value v can be compared to values in every row in the data. Note that the result has $3 \times 1 = 3$ rows. This is a common pattern that will be used over and over in what follows (usually, R is some statistic on the dataset itself).

In general, the size of the CP of T and R is the product of the size of T and the size of R, so this can get very large if T and R are even medium-sized tables: for instance, if T and R have 1,000 tuples each (a very small size for real-life datasets), their CP has 1 million rows (a more respectable size). CPs are only needed in certain occasions, as we will see; therefore, when writing an SQL query we always need to check, if we are using a CP, whether it is really needed. A typical use of CP is illustrated in the previous example; because one of the tables has size 1, there is no 'blow up' effect on the data size. However, beyond this pattern, CPs should be used with caution.

The other way of combining tables (and by far the more common) is to specify a *join* between them. To join two tables, it is necessary to give a condition involving attributes of both tables. This is written in SQL in one of two ways:

```
FROM Table1, Table2
WHERE attribute1 = attribute2

FROM Table1 JOIN Table2 on (attribute1 = attribute2)
```

Here, `attribute1` comes from (the schema of) `Table1`, and `attribute2` comes from (the schema of) `Table2`. What this condition does is to constrain which rows of `Table1` should be paired with rows of `Table2`. Among all tuples in the CP of the tables, only those that fulfill the condition are kept.[6] This allow us to combine data from two tables but to keep all the combinations that 'make sense.'

In order for the comparison to be possible, `attribute1` and `attribute2` should have the same data type. Also, for the result to be meaningful, both attributes should have related values. If you recall the discussion of Sect. 2.2, when information in two tables is related, we use foreign keys to indicate this fact. 99% of joins are on a primary key/foreign key connection—that is, `attribute1` is a primary key and `attribute2` is foreign key that refers to it, or vice versa. The reason is that,

[6] Technically, a join is a CP followed by a selection on the result of the CP.

when designing a database, we used the foreign key to connect related data across tables (see Sect. 2.2); therefore, it makes sense to put data back together using such connections. It is possible to use arbitrary conditions when writing a join; however, joining tables using something else than a primary key/foreign key connection rarely makes sense.

Example: Join

Recall the example where we had transaction data in a table and we split the table into two in order to avoid redundancy. We can now see how in SQL we can go from a single table to several, and vice versa. Joining the smaller, normalized tables `Transactions` and `Products` produces the table `Transaction-and-Products` with all the data combined:

```
SELECT *
FROM Transaction, Items
WHERE Transaction.transaction-id = Items.transaction-id;
```

Note that `transaction-id` is a primary key in `Transaction` and a foreign key in `Items`; hence, this is an example of a primary key/foreign key join.

Likewise, if we are given the full table `Transaction-and-Products` and we decide to split it into two normalized tables, this can be done by *projecting* (database parlance for picking only certain attributes from a table):

```
SELECT transaction-id, product
FROM Transactions-and-products;

SELECT distinct transaction-id, date, time, store
FROM Transactions-and-Products;
```

Note the use of `DISTINCT` on the case of `Transactions`; this is to combat the redundancy introduced by the design of `Transactions-and-products`.

Exercise 3.4 It is easy (and recommended) to check that the queries in the previous example indeed produce the expected results with the data in the example of the previous chapter.

The above example hints at an important technical detail: the schema of a CP or a join is created by concatenating the schemas of the relations being combined. We stated earlier that, in a given schema, each attribute name must be unique. However, when two schemas are combined, there is no guarantee that this will be the case. This is indeed what happens in the example above, where both table `Transaction` and `Product` have an attribute named `transaction-id`. The problem is that mentioning attribute `transaction-id` in the query now would create an ambiguity: there are two attributes with that name. To break this ambiguity, SQL allows the `dot notation`, `table-name.attribute-name`, where each attribute is preceded with

the name of the table it belongs to, both names separated by a dot ('.'). This is exactly what we used in the example above and can always be used as far as each table has a different name—as they should inside a database.

Exercise 3.5 The `world` database is a sample database available from MySQL[7] and is made up of 3 tables:

- `City(id, Name, CountryCode, District, Population)`, which describe cities around the world;
- `Country(Code, Name, Continent, Region, SurfaceArea, IndepYear, Population, LifeExpectancy, GNP, GNPOld, LocalName, GovernmentForm, HeadofState, Capital, Code2)`, which describes countries;
- `CountryLanguage(CountryCode, Language, IsOfficial, Percentage)`, which lists the languages spoken in each country, together with the percentage of the population that speaks the language.

Join the tables `City` and `Country` to show a result where each city population is shown next to the country's population, for the country where the city is. Hint: `CountryCode` in `City` is a foreign key.

Exercise 3.6 Join the tables `CountryLanguage` and `Country` to show a result where each country name is shown next to the country's languages. Hint: here again attribute `CountryCode` is a foreign key.

The dot notation is necessary because sometimes name clashes are unavoidable. As we will see later, there are some occasions where we want to combine a table with *itself* : we want to take the CP or the join of a table `T` with `T`. What does this mean, and how is it accomplished? This simply means we take two *copies* of the data in `T` and take the CP or join of these two copies. But now *all* attributes are ambiguous! This problem is resolved by renaming the tables themselves, as the following example shows.

Example: Self-Join and Self-Product

The following query produces the CP of table `T` with itself:

```
SELECT T1.A, T1.B, T1.C, T2.A, T2.B, T2.C
FROM T AS T1, T AS T2;
```

[7] https://dev.mysql.com/doc/index-other.html.

resulting in table:

CP of **T** (as T1) and T (as T2)					
T1.A	**T1.B**	**T1.C**	**T2.A**	**T2.B**	**T2.C**
a_1	b_1	c_1	a_1	b_1	c_1
a_1	b_1	c_1	a_2	b_2	c_2
a_1	b_1	c_1	a_3	b_3	c_3
a_2	b_2	c_2	a_1	b_1	c_1
a_2	b_2	c_2	a_2	b_2	c_2
a_2	b_2	c_2	a_3	b_3	c_3
a_3	b_3	c_3	a_1	b_1	c_1
a_3	b_3	c_3	a_2	b_2	c_2
a_3	b_3	c_3	a_3	b_3	c_3

Note that the result has 9 rows (3×3) and that each row is combined with a copy of itself. The following query takes the join of table T with itself using a condition that requires equality on attribute A:

```
SELECT *
FROM T AS T1, T AS T2
WHERE T1.A = T2.A;
```

and results in table:

Join of **T** (as T1) and T (as T2)					
T1.A	**T1.B**	**T1.C**	**T2.A**	**T2.B**	**T2.C**
a_1	b_1	c_1	a_1	b_1	c_1
a_2	b_2	c_2	a_2	b_2	c_2
a_3	b_3	c_3	a_3	b_3	c_3

The result is a subset of the CP.[8]

As in the case of CP, it is possible to join 3 or more tables using a sequence of pair-wise join (in any order, since joins are also associative). In particular, a table can be joined with itself an arbitrary number of times by renaming the table as many times as needed. We will see a use for this later.

Exercise 3.7 Write a query that joins table T with itself 2 times.

[8] Since, as stated above, a join is a CP followed by a selection, the join is always a subset of the CP.

3.1.2 Functions

Functions are one of the most useful features of any database system. Unfortunately, the SQL standard specifies only a few, basic functions; most systems offer quite a few functions beyond these basic ones. The good side of this is that a rich functionality is available from most databases; the bad side is that the functions offered, even when they do the same thing, tend to vary in name and other details from system to system (and different systems may offer different functions). But most databases tend to cover the same basic operations in a similar way and therefore what is presented here is doable in pretty much any SQL database.

Functions in SQL can be used in the SELECT and in the WHERE clause. They are always applied to an attribute. There are two types of functions in SQL:

- (Standard) functions. These are functions that are applied to the value of some attribute in a row and yield a (single) result. Since most attributes are of type string, numerical, or date, the typical system has string functions (that is, functions that take a string value, and perhaps some additional parameters, and manipulate the value in certain ways), numerical functions (including typical arithmetic functions), and date functions (that is, functions that take a date value, and perhaps some additional parameters, and manipulate the date in certain ways). We introduce these functions in an as-needed basis throughout this chapter and the next one.

  We will use several mathematical functions in what follows. The most common ones are:

  - `POW(m,n)`, which calculates the exponentiation function m^n. It is important to note that in many systems there is a function `SQRT(m)` to take the square root $\sqrt{m}$, but for higher roots it is common to use `POW(m, 1.0/n)` (since $\sqrt[n]{m} = m^{\frac{1}{n}}$).
  - `LOG(m, b)`, which computes its inverse, the logarithm function (using base b);[9]
  - `MOD(m,n)`, which returns the remainder of dividing m by n (and is often used to compute modular arithmetic);
  - `CEIL(m)`, which returns the *ceiling* of m (that is, the least integer that is greater than or equal to m).

  These functions are available in Postgres, MySQL, and most other systems. There are also functions that deal with string and dates; a few are mentioned in the next example, and they are discussed in much more detail in Sects. 3.3.1.2 and 3.3.1.3.

[9]Confusingly, MySQL uses `LOG(base, number)` instead. Also, in Postgres a special notation exists for base e (`ln()`) and when `log` is used with a single argument, the default is base 10, while it is base e in MySQL.

- Aggregate functions. These are functions that take in a *collection* of values and produce a single result. *Aggregate functions can only be used in the SELECT clause*. The reason is that any function used in the WHERE clause as part of a condition applies to a single value (since it is evaluated in a single row), while aggregate functions, as stated, apply to collections. The SQL standard only requires the functions `AVG` (average or mean), `SUM` (sum of values), `COUNT` (count number of values), `MIN` (minimum), and `MAX` (maximum). However, most systems offer a large number of additional functions, including some statistical functions that we will put to a good use in the next sections (for instance, most systems provide *standard deviation*, usually called `std` or `stddev`).

Some examples will show the difference in meaning and use.

Example: Standard Functions

In the query

```
SELECT to_date(concat_ws("-", year, month, date))
            as flight-date,
       distance/1000.0 as distance-thousands,
       carrier || tailnum as plane-identifier,
       trim(both from flight) as flight-number,
       lower(dest) as destination
FROM ny-flight;
```

there is no `WHERE` clause, which means that all rows in table `ny-flight` qualify for the answer. On each row, the system will pick the value of attributes `year, month, date, distance, carrier, tailnum, flight`, and `dest` and on each row will use these values as follows:

- The function `concat_ws` takes several strings as arguments: the first one denotes a string that acts as a 'separator,' that is, it is used in between all other arguments as they are *concatenated* or composed into a single string. Thus, the function call above will put together the values of attributes `year, month`, and `date` in one string, but with a hyphen (the first argument) in between them.
- The function `to_date` takes the result of the previous function, which is a single string, and transforms it into a value that the system recognizes as a date. Note how this function takes as input the output of another function; *composition* of functions is fine as far as the output type of a function is the input type of another one. The result of this is called `flight-date`.
- The '/' is used to denote division, so this is an arithmetic function. The value of attribute `distance` is expressed in thousands after being divided by 1000. Note that the number is written as '1000.0'; the reason for this is that the system will interpret such a number (with a decimal point) as a real, not an integer, and '/' will act as real division (between two integers, '/' acts as integer division and provides always an integer result). A common trick when doing arithmetic is to divide or multiply an integer value by 1.0, thereby making sure that all other

operations (division, etc.) produce real numbers as result and no information is lost. The result of this division is called `distance-thousands`.

- The function'||' is another way of expressing concatenation in Postgres. Note that the values of attributes `carrier` and `tailnum` are put together as they are, without separators, so values "UA" and "N14228" end up as "UAN14228." The result is called `flight-identifier`.
- The function `trim` gets rid of whitespaces in a string. Recall that strings are enclosed in (double) quotes, and anything in between the quotes is part of the string. For instance, the string " UA " has 2 empty spaces before the non-whitespace characters "UA" and 2 more after it; they are all part of the string. In some cases, this causes confusion, as strings that intuitively seem the same are considered different by the system, due to the whitespaces. It is common to attack this problem by eliminating all whitespaces from a string; most system allow eliminating only whitespaces `before`, `after`, or `both` (as above) the non-white characters in the string. The result of the trimming is called `flight-number`.
- The function `lower` transforms all alphabetic characters in a string to their lower-case counterparts. Non-alphabetic characters and letters already in lowercase are left unchanged. Note that, to the computer, "Jones" and "JONES" are different strings. Hence, functions like `lower` are often used to make sure all values are expressed in the same way (this task, called *normalization*, is discussed in more detail in Sect. 3.4). The result is called `destination`.

The above is just a small sample showing some string, numeric, and date functions, using Postgres notation. In MySQL, concatenation of strings is also expressed using `CONCAT_WS` (if a separator is required) or `CONCAT` (if no separator is required) and conversion of strings to dates with `STR_TO_DATE`; trimming is written as `LTRIM` (for leading spaces) and `RTRIM` (for trailing spaces); converting all characters to lowercase is accomplished with `LOWER`. As we can see, the differences are minor.

Exercise 3.8 In the `world` database, calculate the number of years that a country has been independent (hint: there is an `IndepYear` attribute giving the year of independence, and all systems have functions that will tell you the current date and year).

Example: Aggregate Functions

Using table `Chicago-employees`, we want to calculate the average pay for salaried workers.

```
SELECT avg(salary)
FROM chicago-employees
WHERE salaried = 'T';
```

the system will first filter out rows according to the condition `Salaried = 'T'`; then, it will take the values of attribute `salary` in all the rows left and will apply aggregate function `avg` to them. The result is a single number.[10]

It is obvious now why aggregate functions cannot be used in the WHERE clause; it does not make sense to ask for the average, sum, or count of a single value.

Exercise 3.9 Count how many data points (flights) there are in the `ny-flights` table.

Example: Number of Unique Values

DISTINCT can be combined with aggregates like COUNT to find out the number of unique values in an attribute. The query

```
SELECT count(DISTINCT carrier)
FROM nyc-flights;
```

will tell us the number of airline companies mentioned in our New York flight database. Note the syntax: the DISTINCT keyword goes inside the parenthesis.

One of the characteristics of SQL that drives data analysts mad is that it takes a query to calculate an aggregated value. If you want to *use* it, you will have to write an *aggregated subquery*. Suppose, for instance, that you have the table *Chicago employees*, and you want to find out, among the salaried employees, which ones make more than average. One query is needed to compute the average salary, and another one to use this result and compute the answer.

Example: Aggregated Subquery

The query

```
SELECT name
FROM chicago-employees, (SELECT avg(salary) AS stat
                         FROM chicago-employees
                         WHERE salaried = 'T') AS Temp
WHERE salary > stat;
```

is evaluated as follows: the subquery in `FROM` is run, producing a result (which we can think of as a table named `Temp` with a single row and a single attribute). Following the idea of CP, this attribute is added to every row of table `table-name`,

[10]Technically, all SQL queries return a table as a result; therefore, for this kind of queries (sometimes called *aggregated queries*) where the system is guaranteed to return a single value, the result is technically considered as a table with a single attribute in the schema and a single row in the extension.

that is, it forms an additional attribute (with name `stat`). It is this 'extended' table that is evaluated in the `WHERE` clause; that is why the condition there refers to attribute `stat` as if it were an attribute of table `Chicago-employees`.

Subqueries that return a single result are very common since, as we stated above, SQL requires a query to compute an aggregate. Therefore, if the aggregate is to be used for further computation, a pattern like the one above is commonly used to express this.

Exercise 3.10 In the `world` database, give the name of countries with a GNP greater than the average GNP (for all countries).

Another annoying characteristic of aggregates is that calculating more than one aggregate at the same time can be sometimes difficult. For situations where we want to do several related calculations at once, SQL provides the CASE statement. The CASE is a *conditional* statement: its syntax is

`CASE WHEN` *`condition`* `THEN expression1 ELSE expression2 END`

and it means: evaluate the *condition*; if it is true, do `expression1`; if it is false, do `expression2`. It is often used in combination with aggregates, as the next example shows.

Example: Aggregates with CASE

Assume a table `Temperatures(day,month,temp)` that gives a temperature reading on a certain date, indicated by the month (1–12) and the day within the month. If we want the highest and average temperatures for January (1), this is easy to do:

```
SELECT max(temp), min(temp)
FROM Temperatures
WHERE month = 1;
```

Note that the WHERE filters the temperatures for January for both aggregates. However, if we want to have the highest temperatures for January (1) and also the highest for February (2), we have an issue. Which temperatures are used to compute a result is determined by the conditions of the WHERE clause of which there is only one. One solution is to write two queries, one for January and one for February. The first one is

```
SELECT max(temp)
FROM Temperatures
WHERE month = 1;
```

The second query is identical, except that the WHERE condition is `month = 2`. Experienced SQL programmers would write this as a single query, as follows:

```
SELECT max(CASE WHEN month = 1 THEN temp ELSE 0 END)
         as JanMax
       max(CASE WHEN month = 2 THEN temp ELSE 0 END)
```

```
        as FebMax
FROM Temperatures;
```

This can be read as follows: since there is no WHERE clause, all rows in table `Temperatures` are fed to the aggregates. However, on each row the system runs the CASE statement *before computing the aggregate*. In the first statement, we compare the value of the `month` attribute to 1; if it is, we pass the value of the attribute `temp` on that row to the `max` aggregate; otherwise was pass 0. This means that the max will only consider the temperatures of the rows where the month is 1. The same happens for the second `max` aggregate, but for `month = 2`.

This trick is used to do in one query what would otherwise require two queries. When the table extension is large, this can save not only some typing but also some time.

Exercise 3.11 For all flights originating at LGA, calculate the average duration ("air_time") when the flight is less than 500 miles and the average duration when the flight is longer than 500 miles.

There is a subtlety involving the computation of averages (means): we cannot write

`avg(CASE WHEN month = 1 temp ELSE 0 END)`

to compute the average since this would have taken the average of all January temperatures *and* a bunch of zeros. Instead, we should use

`avg(CASE WHEN month = 1 temp ELSE NULL END)`

since null markers are ignored by aggregate functions.

Example: Percentages

A very common use of the CASE pattern is to compute *percentages*. Assume that, using table `Chicago-employees`, we want to know what percentage of all money paid to hourly workers goes to employees in aldermanic duties. In SQL, we would compute (a) the sum of all money paid to hourly workers; (b) the sum of all money paid to hourly workers who are in aldermanic duties; the final result is the latter divided by the former. To do all in a single query, we can use

```
SELECT sum(CASE WHEN type = 'aldermanic-duties' wages ELSE 0)
       / (sum(wages) * 1.0)
FROM chicago-employees
WHERE hourly-wages = 'T';
```

Note that the condition in the WHERE clause affects both aggregates, since the WHERE clause is executed before the SELECT clause. Thus, both sums only take into account employees with wages, not salaried ones. However, among those, the CASE makes the first sum to only add the wages if the occupation type of the employee is aldermanic duty, while the second sum simply adds all the wages. The

division then gives us the percentage we are after (we are using again the trick of multiplying by 1.0 to make sure that '/' is real division).

Example: Probabilities

Another very common use of this pattern is to approximate *probabilities*. Taking the data in a table as a sample of some underlying population, we can calculate probabilities for certain events by counting, out of all possible cases, which ones are of the form that we are interested in. Assume, for instance, that we want to know how many workers are employed to cover 'aldermanic duties'; we can write

```
SELECT sum(CASE WHEN type = 'aldermanic-duties' THEN 1
                ELSE 0)/(count(*) * 1.0)
FROM chicago-employees;
```

In this example, the first `sum` adds one for each row where the type is 'aldermanic-duties' and ignores others (effectively counting how many times the type is 'aldermanic-duties'), while the `count` counts all the rows. The division then provides a percentage or estimated probability.

The pattern from these examples can be generalized: the probability that event A happens, among all the population, is given by

```
SELECT sum(CASE WHEN A THEN 1 ELSE 0)/(count(*) * 1.0)
FROM table;
```

The conditional probability that event A happens, given that event B has happened, is given by

```
SELECT sum(CASE WHEN A THEN 1 ELSE 0)/(count(*) * 1.0)
FROM table
WHERE B;
```

Exercise 3.12 In the `ny-flights` table, all flights originate in one of 3 airports: 'JFK' (Kennedy), 'LGA' (La Guardia), and 'EWR' (Newark in New Jersey). Count how many flights originate at 'JFK.' Then show how many flights originate at 'JFK' as a percentage of all flights.

3.1.3 Grouping

We have seen how to calculate aggregates over the whole table or subsets of the whole table. However, it is very common in data analysis to be interested in some statistic (be it raw counts, or means, or something else) for different categories within the same dataset (especially using categorical attributes). Because this is a

very common pattern across scientific and business fields,[11] SQL has a set of tools specifically designed for this purpose. The GROUP BY clause tells the system to partition the table into groups: then, any aggregates or actions in the SELECT clause are carried out within each group. The table returned as final result contains *one row per group*. This row contains whatever aggregates or information we have requested in our SELECT for each group.

Example: Split-Apply-Combine in SQL

The following gives the number of flights out of JFK for each airline.

```
SELECT carrier, count(*)
FROM ny-flights
WHERE origin = 'JFK'
GROUP BY carrier;
```

Here, the system scans the table `ny-flights` and picks the rows where the `origin` attribute has value 'JFK'; this set of rows is fed to the GROUP BY. In this case, `GROUP BY carrier` will break the set into groups of rows; each group will be composed of all the rows that share the same value for attribute `carrier`—note that the number of groups, and the number of rows on each group, depends on the particular dataset. Once this partition is done, the system will apply a `count(*)` aggregate *to each group*. Finally, the system will return as answer a table with two attributes, the first one the carrier name, and the second one the number of rows in the group representing that carrier name. In other words, it will tell us how many flights there are in the dataset for each airline, but counting only the flights that originate at JFK. Importantly, for this to work we assume that each flight is represented by one row in the table. Under different conditions, the query would have to be written differently.

Note that when the GROUP BY is combined with a WHERE clause, the WHERE is always done first and it filters the rows that are available to the GROUP BY to do its partitioning.

Exercise 3.13 As stated earlier, all flights in the `ny-flights` table originate in one of 3 airports: 'JFK' (Kennedy), 'LGA' (La Guardia), and 'EWR' (Newark in New Jersey). Count how many flights originate at each one of these airports.

Exercise 3.14 Using the `world` database, count the number of languages spoken in each country. Identify the country by its code (no join needed). Repeat identifying each country by name (join needed).

One of the most typical uses of this clause is the creation of *histograms*: a histogram gives us the frequency associated with a value, i.e. the number of times a value appears in a dataset.

[11] It is sometimes given the name *split-apply-combine* [18].

Example: Histograms

Suppose we want to know the associated frequency to each destination airport. Then we would write

```
SELECT dest, count(*) as frequency
FROM ny-flights
GROUP BY dest;
```

This would return a table with all destination airports and, for each one, the number of times (raw frequency) from which it appears in the dataset (under the assumption that one flight = one row, the frequency represents the number of flights into that airport).

The GROUP BY clause takes a list of attribute names; in fact, expressions that return an attribute name, like a CASE, can be used. When more than one attribute is mentioned, then all the values involved are used to determine the groups that data is broken into, as the next example shows.

Example: Complex Grouping

Suppose we are interested in the number of different ways to get from one airport to another. The query

```
SELECT origin, dest, count(*) as joint-frequency
FROM ny-flights
GROUP BY origin, dest;
```

breaks the table into groups using attributes `origin` and `dest`: that is, to be in the same group, two rows must have the same value in both attributes. Hence, each group represents all the flights that take us from `origin` to `dest`.

As before, we can convert raw counts to percentages, but we need to be careful. The frequency should be divided by the total number of data points, i.e. the number of rows in the whole table. However, aggregates computed withing grouping apply to each group. Therefore, we need to compute this total number apart (in a subquery). When we attempt to use the total, we may run into trouble with SQL. The reason is that using a GROUP BY restricts what can be extracted out of the data in the SELECT clause. In any SQL query with a GROUP BY, the SELECT can only mention attributes that appear in the GROUP BY or aggregations. This is because the result of a query is a table consisting of one row per group, so only expressions that guarantee single values per group are allowed. This restriction can get on the way of analysis sometimes; as we will see in Sect. 5.3, SQL provides a more flexible way of partitioning data that lifts this restriction.

Example: Incorrect Grouping

In table ny-flights, I want to see the number of flights per destination airport and also the airlines that provide those flights, so I write

```
SELECT dest, carrier, count(*)
FROM ny-flights
GROUP BY dest;
```

This query is **not** legal SQL and will generate an error.[12] The reason is that when the system attempts to generate one tuple per group, it finds out that there is only one value of `dest` (since this is the attribute used to create the group, all tuples in the group have the same value for it) and only one value of `count(*)` (since it is an aggregate) but there are multiple values of `carrier`, and room for only one. One way to get this information is to split the query into two and get the information separately; if necessary, both answers can be combined into one:

```
SELECT T1.dest, T1.carrier, T2.num
FROM ny-flights as T1, (SELECT dest, count(*) as num
                        FROM ny-flights
                        GROUP BY dest) as T2
WHERE T1.dest = T2.dest;
```

This table will show one row for each carrier–destination combination and repeat the number of flights for that destination in all tuples with the same destination.

This limitation of SQL syntax can sometimes be aggravating, as we see next.

Example: Grouping and Percentages

If we wanted to convert the table where we counted the numbers of flights per destination into percentages, we need to divide by the total number of flights. It would seem that the way to do this is as follows:

```
SELECT dest, count(*) / (total * 1.0)  as probability
FROM ny-flights,
     (SELECT count(*) as total FROM ny-flights) AS T
GROUP BY dest;
```

This query uses a *subquery* in the FROM clause to count the number of rows in the whole table. One way to see what is happening is to imagine that this subquery is evaluated before anything else. It produces, as a result, a table with a single row and a single column. The system then takes the Cartesian product of this table and table `ny-flights`. Since we have only one row in the table, the end result is to attach, to each row in `ny-flights` a new value named `total`, which contains the result of

[12] Early versions of MySQL would return a non-sensical answer instead of giving an error.

the subquery (note that this single value is repeated in each row!). However, most systems will actually throw an error with this query. This is because the attribute `total` will be seen as a non-aggregated, non-grouping attribute of the table being grouped (which is the CP of `ny-flights` and `T`). There are two work-arounds for this. The first is to observe that the value of `total` is actually the same in all rows; hence, we can write

```
SELECT dest, count(*) / (min(total) * 1.0)  as probability
FROM ny-flights,
     (SELECT count(*) as total FROM ny-flights) AS T
GROUP BY dest;
```

By applying the aggregate `min`, we have converted attribute `total` into an aggregated result. Since the value of `total` is the same in all rows, applying `min` to it does not change its value. The second approach is to compute the value as follows:

```
SELECT dest, sum(1.0 / total) as probability
FROM ny-flights,
     (SELECT count(*) as total FROM ny-flights) AS T
GROUP BY dest;
```

Here we are adding the individual 'weight' of each fact, with the 'weight' indicating how much a single fact counts ($\frac{1}{n}$ in a dataset with n facts). Note that, in all cases, we use the trick of multiplying by 1.0 to make sure that floating number division, not integer division, is used.

Exercise 3.15 Turn the joint histogram above into a joint probability distribution of attributes 'origin' and 'destination' by converting the raw counts in the example above to percentages/probabilities, using the same approach as the previous example.

We can combine GROUP BY with the use of CASE in the definition of the groups, so that we can make up groups more generally and not just by value.

Example: GROUP BY and CASE

The following query counts how many flights are small, medium, and large distance, according to some prefixed cut points:

```
SELECT  CASE WHEN distance < 500 THEN 'Short'
             WHEN distance < 100 THEN 'Medium'
              ELSE 'Long' END as flightDuration,
              count(*)
FROM ny-flights
GROUP BY flightDuration;
```

Of course, CASE can also be used in the aggregates of a group-apply-combine query.

Example: GROUP BY, Aggregates, and CASE

The following query will count the number of short, medium, and long flights per destination, according to the same cut points as before:

```
SELECT dest
    sum(CASE WHEN distance < 500 THEN 1 ELSE 0 END)
         as shortFlights,
    sum(CASE WHEN distance BETWEEN 500 and 1000
              THEN 1 ELSE 0 END) as  mediumFlights,
    sum(CASE WHEN distance > 1000 THEN 1 ELSE 0 END)
         as LargeFlights
FROM ny-flights
GROUP BY dest;
```

Exercise 3.16 In `ny-flights`, count the number of flights with a departure delay of less than 5 min, a departure delay between 5 and 10 min, and a departure delay of more than 10 min.

An additional tool for this type of analysis is the HAVING clause. This clause can only be used with GROUP BY, never on its own. The reason is that HAVING puts a condition on partitions; that is, it examines partitions created by GROUP BY and it lets them through or filters them out.

Example: HAVING Clause

Suppose that, as before, we want to know the associated frequency to each destination airport but are only interested when the frequency is higher than some threshold, say 10. Then we would write

```
SELECT dest, count(*)
FROM ny-flights
GROUP BY dest
HAVING count(*) > 10;
```

This query would be executed as the previous example, with the table being partitioned by destination, and the aggregate applied to each partition. However, the system will only return a row for those partitions where the aggregate `count(*)` generates a value greater than 10.

Queries with grouping are very useful for examining overall tendencies of the data, especially distribution of values; HAVING clauses are useful in narrowing down such an examination.

Exercise 3.17 In `ny-flights`, count the number of flights with an arrival delay of more than 15 min for each airline but show only airlines where that number is at least 5 flights.

Exercise 3.18 In the `world` database, show the number of cities with more than 100,000 inhabitants for each country but show only countries with at least 5 such cities.

Exercise 3.19 In the `world` database, show the number of languages per country but show only countries where more than 2 languages are spoken.

Exercise 3.20 In the `world` database, show (for each country) the number of languages spoken by at least 10% of the population of that country but show only countries with more than 2 such languages.

3.1.4 Order

So far, all the queries we have seen return tables as answers. One of the characteristics of the table is that row order is not important, that is, an answer is a *set* of rows: the only thing that matters is which rows made to the answer. However, sometimes we want more than that. In particular, sometimes we want results that are *ordered*. For these cases, SQL has an additional clause, `ORDER BY`, which allows us to impose an order on the result of a query. The `ORDER BY`, if used, is added after any `WHERE` or `GROUP BY` clauses.

Example: ORDER BY Clause

Assume (once again) that we want to know, in the `ny-flights` dataset, about the number of flights per destination. However, while we plan to examine all destinations, we would like to be able to see the most frequent destinations first. We can write

```
SELECT dest, count(*)
FROM ny-flights
GROUP BY dest
ORDER BY count(*) desc;
```

The keyword `desc` stands for *descending order*; `asc` can be used for ascending order (but it is the default, so that is usually not necessary).

Exercise 3.21 In the `world` database, sort the countries by number of languages spoken (most languages first).

Ordering can be used in conjunction with another clause, `LIMIT`. This clause takes a number as its argument: `LIMIT n` means that the system will return at most `n` rows as answer. Note that using `LIMIT n` by itself will simply truncate an answer; most of the time, `LIMIT` is combined with `ORDER BY` to answer *top k* queries: these are questions when we want the most important k elements in the answer, not all

of them, with 'importance' decided by some aggregate or measure defined in the query.

Example: Top k Query

The query

```
SELECT dest, count(*)
FROM ny-flights
GROUP BY dest
ORDER BY count(*) desc;
LIMIT 10;
```

will retrieve the top 10 destinations by number of flights. The query

```
SELECT dest, count(*)
FROM ny-flights
GROUP BY dest
ORDER BY count(*) desc;
LIMIT 10 OFFSET 20;
```

will pick destinations 21 to 30 (10 destinations, starting after 20) in the order created by the count.

Top k queries are especially useful for distributions where we can expect a *long tail*, that is, a long list of marginally useful results. This is typical in scenarios where can expect a *power law* distribution of results: for instance, an analysis of computers in a network may show a few computers that handle most of the traffic (the servers), while a bunch of computers have only limited traffic (individual PCs). Another typical example is wealth distribution: there are, in most countries, a few billionaires, a small number of millionaires, followed by a very large number of people with limited income. When we want to focus on the 'top group' and avoid the long tail, a top k query is our friend. Note that it is trivial to get "bottom k" results by using ascending instead of descending order.

However, `LIMIT` can be used by itself, in which case it can be seen as a rudimentary form of *sampling*), since the rows to be retrieved are picked by the system. `LIMIT` can also be combined with `OFFSET` to pick not just the top k, but any k answers: `OFFSET n` makes LIMIT to start counting at row $n + 1$ instead of at row 1.

Example: Top k Query

The query

```
SELECT dest, count(*)
FROM ny-flights
GROUP BY dest
ORDER BY count(*) desc;
LIMIT 10 OFFSET 20;
```

will retrieve destinations 21 to 30, with the order coming (as in the previous example) by number of flights.

Exercise 3.22 In the `world` database, show the 10 most populous cities.

Exercise 3.23 In the `ny-flights`, show the flight origin, destination, and airline with the top 10 flights by delay (i.e. largest arrival delay).

Ordering can be also made explicit with *ranking*, whereby each row gets an additional attribute that gives its order within the result. We discuss how to produce and use rankings in Sect. 5.3.[13]

3.1.5 Complex Queries

We have seen above several examples where a subquery in the FROM clause is used to help obtain a result. Using such subqueries is not uncommon; sometimes, the SQL syntax forces us to break down a computation into steps, each step requiring a query. A typical example, as already observed, is the use of aggregates, which need to be computed first.

Using subqueries can be helpful when writing complex queries, as they allow us to break down a problem into sub-problems. Because of this, SQL has several ways in which subqueries can be used. Besides subqueries in FROM, another useful construct is the WITH clause. This clause precedes a query and introduces a temporary table that can then be used in the query. Its syntax is

```
WITH table-name AS (SELECT ... FROM ... WHERE ...)
SELECT ...
FROM table-name, ...
WHERE ...
```

The first query (in parenthesis) defines a table that is given a name (and optionally, a schema); this name can then be used in following query. This is equivalent to defining a subquery in the FROM clause:

```
SELECT ...
FROM (SELECT ... FROM ... WHERE ...) as table-name, ...
WHERE ...
```

but is preferred by some programmers as it makes more clear that we are creating a temporary table for further computation.

[13] Rankings can be computed without the advanced methods of Sect. 5.3, but doing so is quite costly and it should not be done except with small datasets.

Example: Subqueries in FROM, Revisited

In a previous example, we ask where those flights coming into JFK on December 10 are coming from, but only for trips longer than 1,000 miles:

```
SELECT origin
FROM (SELECT id, distance, origin
      FROM ny-flights
      WHERE year = 2013 and month = 11 and
            day = 10 and dest = "JFK") AS T
WHERE distance > 1000;
```

This could also have been written as

```
WITH T AS
 (SELECT id, distance, origin
  FROM ny-flights
  WHERE year = 2013 and month = 11 and
        day = 10 and dest = "JFK")
SELECT origin
FROM T
WHERE distance > 1000;
```

Another reason to use WITH is that it can be combined with subqueries in FROM clause for complex cases where several queries are required.

Example: Complex Queries

Suppose a company has a table `ASSIGNMENTS(pname, essn, hours)`, which states which employees (denoted by `essn`) are working in which projects (denoted by `pname`) for how many `hours` a week. The boss wants to know which project(s) require the largest number of person-hours. In order to compute this in SQL, we need to take the following steps:

1. Compute the number of person-hours per project.
2. Find out the largest number of person-hours.
3. Find the project(s) with that largest number.

This requires 3 queries, which we can put together as follows:

```
WITH PERSON-HOURS(pname,total) AS
 (SELECT pname, sum(hours)
  FROM ASSIGNMENTS
  GROUP BY pname)
SELECT pname
FROM PERSON-HOURS,
     (SELECT max(total) as maxt FROM PERSON-HOURS) AS T
WHERE total = maxt;
```

The WITH clause is evaluated first, creating a temporary table `PERSON-HOURS`, which is then used in evaluating the subquery in the FROM clause of the second query and the rest of the second query.

Breaking down an answer into steps, writing a query for each step, and then combining the queries to obtain a final result is a good tactic when using SQL. In the following, we will use WITH and subqueries in FROM liberally to solve problems.

Exercise 3.24 Let the *linguistic diversity* of a country be defined as the number of languages spoken in the country divided by the country's population expressed in millions. Find the country(ies) with the largest linguistic diversity in the database.

3.2 Exploratory Data Analysis (EDA)

The first thing to do with a dataset is to find out its meaning and main characteristics. The meaning is whatever events or entities the tables refer to, and whatever attributes or characteristics the attributes denote—in other words, what information the data represents. The goals of Exploratory Data Analysis (EDA)[14] on a dataset are: to determine what type of dataset it is (structured, semistructured, or unstructured); if it is not unstructured, what is schema is—that is, what attributes compose the data (and, in the case of semistructured, how they are combined—what the tree structure is) and to determine, for each attribute, the type of domain it represents (categorical/nominal, ordinal, or numerical) and what the data values in the domain are like: what a typical value is, what the range of values is, and so on.

In the rest of this chapter, we will assume for now that we are dealing with structured data and that the schema is known. Hence, the main task will be to find out about each domain. In such an scenario, EDA will proceed as follows:

1. Examine each attribute in isolation. This is called *univariate* analysis in the statistical literature, the name indicating we are dealing with *one* variable at a time. The first task is to determine the type of domain. Typically, categorical attributes are represented by string data types, numerical attributes by number data types, and temporal information by times and/or dates (ordinal domains can be represented by strings or numbers, depending on the context). However, this is not always the case, and EDA should be used to find any mismatch between the domain type and the data type and correct it. Once we know the type of domain, we explore the values on it as follows:
   - For categorical attributes, histograms and its variations are the main tool.

[14] This step is sometimes called *data profiling* or *data exploration*.

- For numerical values, we want to find the measures of central tendency (mean, median, mode, and a few more) and dispersion (standard deviation, quartiles). We may also try to fit a known distribution to a domain to see if the domain can be described succinctly by a formula.

At this state, we should also try to identify potential issues on the domain of each attribute. Some of the main problems to look for are *missing values*, *outliers*, and errors (data in the wrong type, etc.). Discovering whether such problems are present in our dataset will help with the data cleaning step (see Sect. 3.3).

2. Examine the relationships (or lack thereof) among several attributes. This is called *multivariate* analysis in the statistical literature. It is common that some attributes in a dataset are related to other attributes; since each attribute describes an aspect or characteristics of an object or event, there is the possibility that two or more attributes exhibit some common traits or are connected in some way. The techniques for examining potential connections depend on the types of attributes involved; some basic tools that we will see later are (classified by variable type):

   - For **categorical–categorical** analysis (both variables involved are categorical): contingency tables, chi-square test.
   - For **categorical–numerical** analysis (one variable is categorical, the other one is numerical): logistic regression, ANOVA.
   - For **numerical–numerical** analysis (both variables are numerical): covariance, correlation, PMI, linear regression.
   - For **ordinal–ordinal** analysis (both variables are ordinal): Spearman's rank, Kendall's rank.

In principle, any set of schema attributes could be related to another set of schema attributes. Unfortunately, a schema with n attributes has 2^n sets of attributes, so the number of potential connections between these would be of the order of 2^{2^n}, a number that is too large even for small values of n. Therefore, EDA typically focuses on *pairs* of attributes A and B, trying to decide if there is any connection between the two of them. More complex connections can be also explored, but this usually happens during the data analysis phase itself (see next chapter).

The main tools of EDA are *visualization* and *descriptive statistics*. Visualization is very useful because people are very good at discerning patterns when these are presented in a graphical manner. For a single attribute, a *bar chart* of a *histogram* can give a quick overview of value distributions. *Boxplots* can also be used for exploration. The boxplot of a symmetric distribution looks very different from the one of an asymmetric distribution. For two attributes, a *scatterplot* or a *density plot* is useful. Unfortunately, visualization is a weak point of databases. Most database systems offer nothing and depend on third-party, external tools for visualization. Tools like R are much better at this (see Sect. 6.1 for an overview of R). We will not discuss visualization any further in this book. Fortunately, simple descriptive statistics can be computed in databases; in the rest of this section, we describe how to carry out EDA in SQL.

To summarize, EDA can be described as a set of tools that help us build a description of the dataset. The goal of EDA is to gain an intuition about what the data in the dataset is like, to summarize it and to try to identify any issues with it. This is purely a descriptive task, so assumptions made are minimal at this stage. EDA is very important because the information that we obtain at this stage will guide further work in the next stages, from data cleaning (Sect. 3.3) and data preprocessing (Sect. 3.4) to data analysis (Chap. 4).

3.2.1 Univariate Analysis

As stated, the first task is to determine whether the domain of the attribute is categorical, numerical, or ordinal. It should be the case that categorical attributes are expressed with some kind of string data type, and numerical attributes with some kind of number data type, but this does not need to be the case. For instance, products in a catalog can be divided into 5 categories, which are called (regrettably) '1,' '2,' '3,' '4,' and '5.' This is not a numerical attribute, not even an ordinal one (unless there is an underlying reason to use those numbers, for instance reflecting increased price range). It is also very common to have temporal information like dates entered as strings, instead of as a date data type. This should be avoided, since dates have their own functions that can be fruitfully used during analysis. Establishing the (intended) meaning of each attribute is essential here; this is what allows us to compare what we find to what could be expected.

We first consider numerical attributes. In this case, we are interested in the measures of the range (minimum, maximum), central tendency (different means —arithmetic, geometric and harmonic, median, mode), and dispersion (standard deviation, variance, skewness, kurtosis) [4, 5]. It is easy to calculate several of these values at once; for a given attribute `Attr` in table `Data`, we can use the following query to extract some basic information:[15]

```
SELECT count(Attr) as number-values,
       count(distinct Attr) as cardinality,
       min(Attr) as minimum,  max(Attr) as maximum,
       max(Attr) - min(Attr) as range,
       avg(Attr) as mean,
       stddev(Attr) as standard-deviation
FROM Data;
```

We examine each of these in more detail next.[16] However, it is important to point out right away that aggregate functions *skip nulls*, that is, if there are null markers in

[15] For readers familiar with R, note the similarity with the `summary` command.

[16] From this point on, we show SQL *templates*, that is, we use generic attributes and tables to show how queries should be written. For instance, in the example above `Attr` should be substituted by a particular attribute name, and `Data` by a particular table name.

the attribute `Attr` being analyzed, those null markers will be ignored (except when using `count(*)`, which simply counts the number of tuples, regardless of what the tuples contain). We deal with null markers in Sect. 3.3.2.

The first aggregate simply counts how many data points (i.e. number of rows) we have; the second counts how many `unique` values are present in the attribute. The difference between the minimum and the maximum value gives us the *range*. The (arithmetic) *mean* of an attribute (also called *average* in popular parlance, and *expectation* or *expected value* in statistics) in a dataset is simply the sum of the values divided by the number of values. In all SQL systems, as we have seen, there is an aggregate function, `avg()`, that calculates this:

```
SELECT Avg(Attr) as mean
FROM Data;
```

Note that, by definition, this is the same as

```
SELECT Sum(Attr) / (Count(Attr) * 1.0) as mean
FROM Data;
```

We are using once again the trick of multiplying by 1.0 to make sure the system uses floating point division, not integer division.

For readers who see this concept for the first time, it will become important later to note the connection with frequencies and probabilities: when we add all the values in an attribute, if value a appears n times, we will add it n times; hence, we could achieve the same by adding each distinct value once after multiplying it by its frequency:

```
WITH Histogram(Value, Frequency) AS
(SELECT Attr, count(*)
 FROM Data
 GROUP BY Attr)
SELECT (1.0 * Sum(Value * Frequency)) / Sum(Frequency) as mean
FROM Histogram;
```

And, since we end up dividing by the total number of values, we could substitute the frequency by its normalized value, the probability:

```
WITH NHistogram(Value, Prob) AS
 (SELECT Attr, sum(1.0/total)
  FROM Data,
       (Select count(*) AS total FROM Data) AS Temp
  GROUP BY Attr)
SELECT Sum(Value * Prob) as mean
FROM NHistogram;
```

The mean is known to be affected by *outliers*, extreme values that may or may not be correct. Assume, for instance, that in our demographic dataset we have a `height` attribute, giving the height of each person in the dataset in feet and inches. What happens if we find a very low (say, 4 feet 10 inches) or very high (say, 7 feet and 10 inches) value? They could simply reflect that we have a very short or very

tall person, or they could be the result of a mistake in measurement. What is clear is that this value will have a strong influence on the value of the mean. For this reason, some people prefer to use the *trimmed mean*, which is calculated after disregarding extreme value (usually, the maximum and minimum). This can be generalized, if one wants, to *k% trimmed mean*, where the highest/lowest k% of the values is removed [8]. A simple trimmed mean is quite easy in SQL:

```
SELECT avg(A)
FROM Data, (SELECT max(A) as Amax FROM Data) AS T1,
           (SELECT min(A) as Amin FROM Data) AS T2
WHERE A < Amax and A > Amin;
```

Note that, since the `WHERE` part is run first, the maximum and minimum values are eliminated from the `avg(A)` computation. However, a k% trimmed mean is much trickier; we will see how to compute this in Sect. 5.3.

Sometimes we require other kinds of mean. The *geometric mean* is the *product* of the n values divided by the n-th root of the values. This mean has the advantage of not being as sensitive to outliers as the arithmetic mean (in particular, large values do not disturb it by much, although small values have a significant effect). SQL does not have an aggregate function for multiplication; the standard work-around for this is to use logarithms, as follows: since $\log(ab) = \log a + \log b$, we can calculate $ab = \exp(loga + \log b)$; this generalizes to more than two values. Thus,

```
SELECT exp(sum(log(Attr))
FROM Data;
```

will give us the product of the values of attribute `Attr`. For the geometric mean, we need to compute the n-th root, which we can achieve with

```
SELECT pow(exp(sum(log(Attr))), 1.0 /total)
FROM Data, (SELECT count(Attr) as total FROM Data) AS T;
```

However, the same can be achieved by applying the exponentiation to the arithmetic mean of the sum of the logs. In SQL,

```
SELECT exp(sum(log(Attr)) / count(Attr))
FROM Data;
```

Since the sum divided by the count is the average, this can be simplified to

```
SELECT exp(avg(log(Attr))
FROM Data;
```

However, one needs to be careful for this due to issues of *numerical stability*: real numbers (such the result of calculating logarithms) are represented in the database (in the computer, really) with a certain degree of precision; numbers that are very small (or very large) may get rounded. The logarithm of a small number will tend to

be a very small number,[17] and as a consequence the result obtained from the query above may, in some cases, be inexact.

The geometric mean is useful when dealing with growth (or decay) rates. The typical example is calculating interest rates: suppose a bank proposes, for a savings account, to give different (increasing) interest rates depending on how long the money is left in the bank. For the first year, it will give a 1% interest rate, for the second, 1.5%, for the third, 2%, for the fourth, 2.5%, and for the fifth, 3%. The (geometric) mean of the interests is simply

$$gm = (1.01 \times 1.015 \times 1.02 \times 1.025 \times 1.03)^{\frac{1}{5}} = 1.0199.$$

Note that to calculate how much an amount, say $1000, grows, we normally would calculate

$$(1000 \times s1.01.015 \times 1.02 \times 1.025 \times 1.03) = 1103.94,$$

which is the same as $1000 \times gm^5$.

Another mean is the *harmonic mean*, which uses the sum of the reciprocals:

```
SELECT Count(Attr) / Sum(1/Attr)
FROM Data;
```

This mean is used for measures based on ratios (that is, any measure that depends on some unit) or for when different amounts contribute with a different weight to the mean. The typical example used to illustrate this context is calculating speeds: assume a car travels at 60 miles/hour a certain distance and then comes back, traveling at 30 miles/hour. The average speed is the harmonic mean, 40 miles/hour. The reason is that the car traveled the same distance on both speeds, but since the return speed was a third of the original one, the car took 3 times as long going back. If the car had traveled the same time at both speeds, its average would be the traditional (arithmetic) mean, 45 miles/hour.

Exercise 3.25 Assume a table called `Trips` with an attribute `speed` and write a query to compute the arithmetic, geometric, and harmonic mean of this attribute.[18]

[17] This depends on the base, of course; the above uses e as the standard base.

[18] Any exercise that starts with *Assume some data ...* will describe a (highly simplified) scenario. These exercises can be carried out in either MySQL or Postgres (or any other relational system) as follows: (1) create the table described in the scenario (in this scenario, a table called `Trip` with a numeric attribute called `speed`); (2) insert some made-up data in this table (usually, 4–6 rows are enough); (3) write the query for the exercise. This will make it possible to ensure that: (a) the query is syntactically correct (otherwise the system will give an error message instead of an answer); and (b) the query is semantically correct (the answer can be checked by hand against the data).

The *mode* is the most frequently occurring value. In SQL, it must be calculated in steps:

```
WITH Histogram as
(SELECT Value as val, count(*) as freq
FROM Data
GROUP BY Attr)
SELECT val
FROM Histogram, (SELECT max(freq) as top FROM Histogram) AS T
WHERE freq = top;
```

Note that there can be several modes in a dataset. When we know that there is only one mode, the following is more efficient:

```
SELECT Value, count(*) as freq
FROM Data
GROUP BY Value
ORDER BY freq DESC
LIMIT 1;
```

However, this is correct only when we know for sure the number of modes—if there are k modes, we can use `LIMIT k`. Otherwise, this approach is incorrect.

As we will see, in some systems it is possible to call an aggregate function to compute the mode directly using windows (see Sect. 5.3).

Exercise 3.26 In `ny-flights`, compute the most visited destination (the destination with more flights arriving to it).

Finally, the *median* is the value that would appear in the middle position if all values were ordered from smaller to larger. That is, the median has as many values larger than it as values smaller than it. While the concept is very simple, it is actually a bit difficult to calculate in SQL. First, it requires values to be sorted; second, it requires that the middle position be found. Note that, for an odd number of values, the middle position is clearly defined (for instance, for 21 values, position 11 has 10 values before and 10 values after it), but for an even number of values, there is no real 'middle': this is usually solved by averaging the 'before' and 'after' values (that is, for 20 values, we would take the average of the values in positions 10 and 11). As a consequence of this, finding the median in SQL is a bit elaborate; the most efficient way is to sort the values with ORDER BY and use a combination of LIMIT and OFFSET to find what we want:

```
SELECT avg(Attr)
FROM (SELECT Attr
      FROM Data, (SELECT count(*) as size FROM Data)
      ORDER BY value
      LIMIT 2 - MOD(size, 2)
      OFFSET CEIL(size / 2.0)) AS T;
```

The idea here is that when `size` is odd, `MOD(size, 2)` is 1 and `CEIL(size / 2.0)` will be the 'middle' so we will pick a single value, the one in the middle

position; but when `size` is even, `MOD(size, 2)` is 0 and `CEIL(size / 2.0)` will be the 'middle' minus 1, so we will pick two values, corresponding to the 'before' and 'after' the middle position. Note that this approach does not ignore NULLs, as most aggregates do; to discard them, one should add a WHERE clause using the `is not null` predicate. A simpler way to calculate the median will be shown in Sect. 5.3.

As for the measures of dispersion, the simplest is the *range*, or difference between the maximum (largest) and minimum (smallest) value. We have seen that this is trivial to calculate in SQL. The *standard deviation* is probably the most common dispersion measure. It is also easy to calculate, since it is a built-in function in most database systems:

```
SELECT stddev(Attr) FROM Data;
```

is how Postgres expresses it. In MySQL, the function is call `std`.[19]

It is instructive, though, to look at how to calculate this value. The typical formula for standard deviation for some collection of values x_i is

$$\sqrt{\frac{\Sigma_{i=1}^{n}(x_i - \mu)^2}{(n-1)}}$$

with μ the mean. In SQL, this yields

```
SELECT sqrt((sum(power(Attr - mean, 2)) / (count(*) -1))
FROM Data, (SELECT avg(Attr) as mean FROM Data) as T;
```

The following formula is equivalent to it and is easier to compute (it requires only one pass over the data):

$$\sqrt{\frac{(\Sigma_{i=1}^{n}x_i^2)}{(n-1)} - \left(\frac{(\Sigma_{i=1}^{n}x_i)}{(n-1)}\right)^2}$$

In SQL:

```
SELECT sqrt( sum(pow(Attr, 2)) / (count(*) - 1) -
             pow(sum(Attr) / (count(*)-1), 2)) as std-dev
FROM Data;
```

The value of this example is to show that, in many cases, when a function is not available in the system, it is still possible to produce a result by implementing its definition—something we exploit often in this chapter and the next.

[19]The function `stddev()` in Postgres is an alias for `stddev_samp` (standard deviation over the sample), which divides (as we do here) over $n - 1$ when there are n data values. There is also a `stddev_pop` function that divides over n. Similar functions exist in MySQL.

In some cases we will need the *variance*, which is simply the square of the deviation. Variance is also available as a built-in in most systems; in both Postgres and MySQL, one can use:[20]

```
SELECT variance(Attr) FROM Data;
```

Exercise 3.27 Pretend the variance is not available in your system; compute it from scratch using the approach for standard deviation.

Both standard deviation and variance are also affected by outliers, so sometimes the MAD (Median Absolute Deviation) is calculated too. The MAD of a set of values is the median of the absolute value of the difference between each value and the median:

$$MAD(x_i) = \text{median}(|x_i - \text{median}_i(x_i)|)$$

It can be seen as a deviation measure that uses the median instead of the mean, and the absolute value instead of the square root. Using the median as calculated above, this formula can be transformed into an SQL query.

Exercise 3.28 Give the SQL to calculate the MAD of attribute `Value` in table `Data`. Hint: reuse the definition of median above; use a WITH clause.

Exercise 3.29 When data can be grouped, it is common to examine the *between-classes variance*: if attribute `A` in table `T` can be divided into groups $g_1, \ldots, g_n$, let n_i be the size of interval g_i and avg_i the mean of the values in interval g_i. Also, let avg be the mean of all `A` values and m the total number of `A` values. Then

$$\frac{1}{m}\Sigma_{i=1}^{n} n_i(avg_i - avg)^2$$

is the between-class variance. Transform this formula into an SQL query. Hint: use subqueries in `WITH` and/or `FROM` to compute the intermediate results needed (the means within each interval, the number, and mean of all values).

Exercise 3.30 If we add all the between-class variances for a variable, we get less than the total variance, because we disregard the variance *within* the class. However, with access to the raw data we can actually make up for this and compute the variance as the sum of the between-class variance and the *within-class* variance:

$$\frac{1}{m}\Sigma_{i=1}^{n} n_i(avg_i - avg)^2 + \frac{1}{m}\Sigma_{i=1}^{n} n_i var_i^2$$

[20] As in the case of the standard deviation, there are functions `var_samp` and `var_pop` in both systems.

where var_i is the variance in interval g_i. Transform this formula into an SQL query. Hint: the first part is the same as the previous exercise; reuse all your work in a subquery. The second part can also be calculated with a separate subquery.

Sometimes a couple of additional statistics can prove useful: the *third moment* about the mean or *skewness*, and the fourth moment about the mean or *kurtosis* can be calculated in SQL following the approach for the standard deviation. What do they mean? Skewness measures the symmetry of a distribution. A distribution is *symmetrical* if the mean is equal to the median. In that case, the mean is in the 'middle' of the distribution; when drawn, the distribution looks symmetrical around the mean. A symmetrical distribution has zero skew; but when a distribution is not symmetrical, it either has positive skew (or right skew, or right tail) or negative skew (or left skew, or right tail). The kurtosis describes the *tail* of a distribution, that is, values that are not close to the mean. In this sense, kurtosis is useful in that it indicates the propensity of a distribution to produce extreme values (i.e. outliers).[21] To compute skewness, we can use

```
SELECT sum(pow(Value - mean, 3)) / count(*)  /
       pow(sum(pow(Value - mean, 2)) / (count(*) - 1), 1 / 3)
FROM Data, (SELECT avg(Attr) as mean FROM Data) AS T;
```

Recall that the formula `pow(x, 1 / 3)` is used to express $\sqrt[3]{x}$ as $x^{\frac{1}{3}}$. However, in many systems this formula is not numerically stable, so results should be checked for accuracy.

To calculate kurtosis, we can use

```
SELECT sum(pow(Value - mean, 4)) / count(*)  /
       pow(sum(pow(Value - mean, 2)) / (count(*) - 1), 1 / 3)
FROM Data, (SELECT avg(Attr) as mean FROM Data) AS T;
```

As before, one must be aware that this formula may not be numerically stable.

We turn our attention now to categorical attributes. The most important tool in this case is the *histogram*. A histogram, in its simplest form, is a list of values in the domain together with their *frequency* (number of times the value appears). We have already seen how to calculate histograms earlier, when computing the mode. The general schema to build a histogram of categorical attribute `A` is

```
SELECT A, count(*)
FROM R
GROUP BY A;
```

[21] Because the kurtosis of a univariate, normal distribution is always 3, the kurtosis of a dataset can be computed and compared to this value: if it is greater than 3, it indicates that the dataset may have outliers. However, this is just a heuristic.

For instance, suppose that we have a demographic dataset with an attribute `zip code`; we can count how many residents live on each zip code with

```
SELECT zip-code, count(*) as population
FROM Dataset
GROUP BY zip-code;
```

Histograms are really one example of a more general technique, *binning* (also called *bucketing*). In binning, the values of a variable are divided into disjoint intervals (called *bins* or *buckets*), and all the values that fall within a given interval are replaced by some representative value. This makes the technique applicable to continuous (numerical) variables too. For categorical values, it is customary that each value is its own interval, but this does not necessarily have to be the case. Binning also generalizes histograms in that the representative value for an interval is not limited to the frequency; it can be another statistic too.

We can do general binning in SQL. For this, the values must be distributed into intervals, so the first task is to determine the number of intervals. There are basically two approaches to this. The first one is called an *equi-depth* histogram, and it sets the boundaries of the bins so that each has an equal number of data points. This is usually done by defining *quantiles*, cut points that split a domain into intervals with the property that each interval has as many data points as any other (plus or minus one, if the number of data points does not divide the number of intervals). The most commonly used quantiles are

- the *percentiles*, which divide the domain into 100 intervals, so that each one has 1% of the data;
- the *quartiles*, which divide the domain into 4 intervals, corresponding to 25, 50, 75, and 100% of the data;
- the *deciles*, which divide the domain into 10 intervals, corresponding to 10, 20,..., and 100% of the data.

The median is sometimes called a 2-quantile, since it splits the data into two intervals, both with equal number of points too.

Example: Binning with Quartiles

Assume a table `Heights(age,size)` where the age is an integer, and we want to create a histogram for the `age` attribute, and we want our histogram to be based on quartiles, the first bin representing the first 25% of all ages, the second one representing from 25 to 50%, the third one from 50 to 75%, and the fourth one above 75%. First, we need to find out how many elements there are and divide them into 4 groups. Second, we want to divide all the elements into those 4 groups; this is done after sorting them.

```
WITH OrderedData AS
(SELECT *, number
 FROM Heights,
```

```
      (SELECT count(*)/4 as number FROM Heights) AS T
 ORDER BY age)
SELECT 'bottom25' as quartile, age, size
FROM OrderedData
LIMIT number
UNION
SELECT '25to50' as quartile, age, size
FROM OrderedData
LIMIT number OFFSET number
UNION
SELECT '50to75' as quartile, age, size
FROM OrderedData
LIMIT number OFFSET number*2
UNION
SELECT 'top25' as quartile, age, size
FROM OrderedData
LIMIT number OFFSET number*3;
```

The table that results from answering this query can in turn be used to calculate other values, like $IQS(X)$, the inter-quantile range of the sample (the number of values in the middle 50% of the data).

Clearly, this approach is too cumbersome to be used with more bins (imagine the example with percentiles!). Modern versions of SQL have tools to make this task much simpler; they are discussed in Sect. 5.3.

The second approach is the *equi-space* histogram, which makes all bins the same *width* (i.e. all intervals of the same size; this approach is sometimes called *fixed-width*). In categorical attributes, this means the same number of values on each interval but, unless there is a special reason to group certain categorical values together, this approach is normally only used for ordinal and numerical domains. Note that the width of the bins determines, for a given numerical domain D, the number of bins; if h is the chosen width, then the number of bins is $\lceil \frac{max(D)-min(D)}{h} \rceil$ bins.

Example: Fixed Width Histograms

Assume table `Heights(age,size)` as before. We again want to create a histogram for the `age` attribute, and we want our intervals to have a fixed width, say 4.

```
SELECT age, size, ceil(((age-minage)+1)/4) as bin
FROM Heights, (SELECT min(age) as minage FROM Heights) as T
ORDER BY bin;
```

This maps each value to a bin starting at 1; the minimal age gets mapped to 1 by (age-minage)+1; the second smallest to 2, and so on. Note that the ceiling of the division acts as a modulus function, since this time we are using integer division. With the result of this query, we can group by bin and manipulate the data as required.

In general, if we have a column with attribute `Values` on it and want an equi-space histogram of `Values` with width n,

```
SELECT Values, ceil(((Values - minval)+1)/n) as bin
FROM Data, (SELECT min(Values) as minval FROM Data) as T
ORDER BY bin;
```

will do the trick.

Exercise 3.31 To avoid outlier problems, one can try getting rid of extreme values. Repeat the equi-spaced (fixed-width) histogram of `Heights` after removing the largest and smallest age. Hint: this removal should be one prior to computing anything for constructing the histogram.

One has to be careful here as divisions into bins that are too 'wide' (large intervals) result in a few, coarse bins and may hide important characteristics of the data, while a divisions into bins that are too 'narrow' or 'thin' results in a large number of bins, which may make the data look quite irregular if it does not fit well into any known distribution. There is no general rule to choose the number of bins, but there are several rules of thumb. The simplest one is pick, for n data points, $\sqrt{n}$ intervals for the histogram. There is also *Sturges' rule*: the number of bins for n data points should be (assuming all bins have equal width) $\lceil(\log_2 n + 1)\rceil$, but this works best for normal distributions of moderate size. Another approximation is to use the sample's standard deviation, s, and calculate $\frac{3.5s}{\frac{1}{n^3}}$. Instead of $3.5s$, another possible value is $2IQS(X)$, where $IQS(X)$ is the inter-quantile range of the sample (which we just saw how to calculate). All these methods are very sensitive to *outliers*, which force the bins to become too wide. One may want to ignore outliers in these calculations, by removing extreme values prior to breaking up the data into bins.

Sometimes the bins are determined by the semantics of the attribute being analyzed. As we have already seen, if cut points can be established with domain knowledge, we can bin based on them.

Example: General Binning

Assume a dataset with a `price` attribute; we are interested in determining how many 'cheap,' 'medium,' and 'expensive' products we have, having determined beforehand what those are.

```
SELECT type, count(*)
FROM Data
GROUP BY (CASE WHEN price > 100 THEN 'expensive'
          WHEN price <= 100 and price > 50 THEN 'medium'
          ELSE 'cheap' END) as type;
```

Exercise 3.32 In `ny-flights` dataset, count the number of flights that are 'very late' (arrival delay >20 min), 'late' (arrival delay between 10 and 20 min),

'somewhat late' (arrival delay <10 min), 'on time' (no arrival delay), and 'early' (negative arrival delay).

Exercise 3.33 Using the previous exercise as a temporary result (hint: use a `WITH` clause or a `FROM` subquery), count the number of flights of each type for each airline.

Exercise 3.34 In the `world` database, count the number of countries that call themselves a 'Republic' and the ones that do not (attribute `GovernmentForm`).

Note that for each bin, only a count is kept, so other information about the distribution (like mean) may be hard to recover. One can keep additional information if needed or desired (for instance, the mean of each bin). From the histogram, we can identify generic properties of the data distribution, like symmetry and skewness, as well as whether the distribution is unimodal, bimodal, or multimodal. This can help us narrow down the choice of a theoretical distribution that fits the data. Alternatively, we can compare the histogram generated from the data with the histogram that a theoretical distribution would generate, as described in Sect. 3.2.3. Histograms have their limitations. In particular, distributions with heavy tails are usually not well accounted for with histograms.

A couple of variations of binning can be useful. Sometimes we may want to use percentages instead of raw counts; this is sometimes called a *normalized* binning or normalized histogram.

Exercise 3.35 Using the example shown, compute a normalized, equi-space histogram for the fictitious `Heights` table.

Another idea, especially useful with ordinal attributes, is to have *cumulative totals* on each bin: if the bins can be put in a certain order, each bin counts its own values plus all the values of the bins preceding it. This idea can be applied to any type of histogram.

Example: Cumulative Histogram

Assume a simple histogram of heights `HHeight` that we want to use to calculate cumulative counts.

```
WITH HHeight(value, freq) AS
 (SELECT height AS value, count(*) as freq FROM Height)
SELECT D2.value, sum(D1.freq) as cumulative
FROM HHeight D1, HHeight D2
WHERE D1.value <= D2.value
GROUP BY D2.value;
```

The query joins the `HHeight` table with itself, but the condition `D1.value <= D2.value` matches each tuple in copy `D2` of the table with all those tuples in copy `D1` that have smaller values; when grouping by `D2.value`, all the frequencies of those smaller values are added up.

The same idea can be used to compute cumulative percentages, or other types of binning.

Example: Cumulative Binning

Assume we have split the table `Heights` into bins numbered 1,2,...,10 by calculating deciles; the result is in table `Deciles(decil,frequency)`. We now want a cumulative histogram.

```
SELECT D1.decil, count(*)
FROM Deciles D1, Deciles D2
WHERE D2.decil <= D1.decil
GROUP BY D1.decil;
```

This query takes a join of table `Deciles` with itself (hence the renaming), but on the D2 side it puts all values of `decil` that are less than or equal to ('precede') the current decil we are considering for grouping. That is, when `D1.decil` is 3, the join qualifies deciles 1, 2, 3 for `D2.decil`; this is what is counted by `count(*)`.

Exercise 3.36 Compute a cumulative histogram of percentages over table `HHeight`.

As we will see, using window functions (Sect. 5.3) makes some of these tasks, like calculating quantiles and cumulative results, much easier.

We close this section by presenting a powerful idea that is the basis of some sophisticated analysis. As we have seen, it is possible to associate with any type of attribute (categorical, ordinal, or numerical) a probability distribution by counting the frequency of each value and normalizing all such counts with the total number of data points. Once this is done, we can calculate the *entropy* of an attribute A, which is traditionally defined as

$$H(A) = \Sigma_{a \in A} P(a) log P(a)$$

that is, we add the product of each probability with the logarithm (usually, in base 2, although other bases can be used) of the probability. In SQL,

```
SELECT sum(Pa * log(Pa))
FROM (SELECT A, sum(1.0 /total) as Pa
      FROM Data, (SELECT count(*) as total FROM Data) AS T
      GROUP BY A);
```

The idea of entropy is to represent the 'information content' of attribute A, in the following sense: the 'information content' of value $a \in A$ is considered inversely proportional to its probability of happening; since common or normal events are expected to happen, their occurrence is not very informative. In contrast, uncommon, rare events are unexpected, and therefore their occurrence is very informative. The entropy of A is the 'average' of the information content of all

values of A. This simple definition is the basis of very sophisticated forms of analysis; we will see a couple of the most elementary ones in the next section.

Exercise 3.37 Calculate the entropy of attribute `GNP` in the `Country` table of the `World` database.

3.2.2 Multivariate Analysis

Here we look at possible connections between attributes. We focus on the case of two single attributes; investigating sets of attributes is much more complex (some of it is done under full data analysis).

Let A and B be two attributes from our data table. There are two ways of looking at relationships between A and B: in one, we can assume that one of them is *caused* or influenced by the other. In this case, we say there is an *independent* attribute (in statistics, a *predictor*) and a *dependent* attribute (in statistics, an *outcome* or *criterion*). In the other case, we can assume that they do not depend on each other, although they may still be connected (for instance, there can be a third attribute Z that causes both A and B, therefore linking their values).

In either case, analysis can be further subdivided depending on the type of attributes we are dealing with. We can distinguish 3 cases: both attributes are categorical; both are numerical; and the mixed case (one attribute categorical, one numerical).[22]

Most of the time, we may not know for sure whether two attributes are independent or not. Hence, we start analysis with some simple tests to try to determine whether there is some connection between the attributes. The simplest test, and one that can be used in all types of attributes, uses their probabilities (recall that we saw early on how to calculate the probabilities of categorical attributes using grouping and counting; the same approach can be used with ordinal attributes). Given attributes A and B, we compute the probability of A, $P(A)$, and the probability of B, $P(B)$, as usual; we also compute the *joint probability* of A and B, $P(A, B)$:

```
SELECT A, B, sum(1.0/total) as JointProb
FROM Data, (SELECT count(*) as total FROM Data)
GROUP BY A, B;
```

We can create, for a given table `Data` with attributes A and B, another table `Probabilities` that has $P(A)$, $P(B)$, and $P(A, B)$ as new attributes. Once this is one, we use the following test: if $P(A, B) = P(A)P(B)$, the attributes are *independent*. This can be done very easily: all we have to do is check whether the following query returns zero:[23]

[22] Ordinal attributes can many times be grouped with numerical for the purposes of this subsection.

[23] Recall that, due to numerical instability, we may see a very small, but non-zero, result.

```
WITH ProbA AS
 (SELECT A, sum(1.0/total) as PrA
  FROM Data, (SELECT count(*) as total FROM Data) as T
  GROUP BY A),
     ProbB AS
 (SELECT B, sum(1.0/total) as PrB
  FROM Data, (SELECT count(*) as total FROM Data) as T
  GROUP BY B),
     ProbAB AS
 (SELECT A, B, sum(1.0/total) as PrAB
  FROM Data, (SELECT count(*) as total FROM Data) as T
  GROUP BY A, B)
SELECT sum(PrAB - (PrA * PrB))
FROM (SELECT A, B, PrA, PrB, PrAB
      FROM ProbA, ProbB, ProbAB
      WHERE ProbA.A = ProbAB.A and ProbB.B = ProbAB.B)
      AS Probabilities;
```

Note that we must compute the single and joint probabilities apart, since they require different groupings (different ways to look at the data).[24] Note also that we compute probabilities for all values of A and B, but when joining with the joint probability, only combinations that occur in the data are kept.

Probabilities are also used in other important measures, like *Pointwise Mutual Information (PMI)*, defined as follows:

$$PMI(A, B) = \log \frac{P(A, B)}{P(A)P(B)}.$$

Obviously, it is also the case that when this measure is zero, attributes are independent. In SQL:

```
SELECT log(sum (PrAB / (PrA * PrB)))
FROM Probabilities;
```

The *mutual information* of A and B (in symbols, $I(A, B)$) in turn exploits PMI to give a measure of how dependent A is on B and vice versa:

$$I(A, B) = E_{P(A,B)} PMI(A, B),$$

where E is the expectation (mean) calculated over $P(A, B)$.

Exercise 3.38 Assuming a table `Probabilities` as above, with $P(A)$, $P(B)$, and $P(A, B)$, write an SQL query to compute $I(A, B)$.

The closer the mutual information is to zero, the more independent the attributes are. Because this value is not normalized, it is hard to interpret on its own; mutual information is often used to compare pairs of attributes for feature selection or other tasks.

[24] In Statistics, the single probabilities are called, in this context, *marginal* probabilities.

For numerical attributes, the *covariance* is the simplest measure of a possible connection between attributes. Covariance is given by

$$C(A, B) = E((A - E(A))(B - E(B))),$$

where A, B are the attributes and E the expectation (mean). In a sample, this resolves to:

$$C(A, B) = \frac{\Sigma_{i=1}^{N}(a_i - \bar{A})(b_i - \bar{B})}{N-1}(1) = \frac{\Sigma_{i=1}^{N} a_i b_i}{N} - \frac{(\Sigma_{i=1}^{N} a_i)(\Sigma_{i=1}^{N} b_i)}{N(N-1)}(2),$$

where $\bar{A}$ denotes the mean (average) of A and $\bar{B}$ the mean (average) of B.
The covariance can also be expressed via the formula:

$$Cov(A, B) = E(A, B) - E(A)E(B),$$

which is easier to compute if necessary. Note the similarity to the test of independence using the joint probability distribution; just like the mean of an attribute is related to its probability distribution, the covariance is related to the join probability. This is why it is another test for independence.

In PostgreSQL, there is an aggregate function, `covar_samp(X,Y)`, to compute the covariance of two attributes in a table. There is no aggregate for covariance in MySQL as of this writing (2019), but it is easy to simulate from the definition above. The simplest way to write the covariance of attributes A and B is to use the last definition:

```
SELECT (avg(A*B) - (avg(A)* avg(B)))
FROM Data;
```

Exercise 3.39 Write an SQL query to compute the covariance using the original formula (1).

Exercise 3.40 Write an SQL query to compute the covariance using the original formula (2).

Exercise 3.41 Compute covariance not from original data but from table `Probabilities` with $P(A)$, $P(B)$, and $P(A, B)$.

The Pearson correlation coefficient is simply a normalized covariance:

$$\rho = \frac{Cov(A, B)}{std(A)std(B)},$$

where $std(A)$ is the standard deviation of A, and similarly for $std(B)$. This value is always between -1 and $+1$, with larger absolute values reflecting stronger correlation, and the sign indicating positive (both A and B move in the same direction) or negative (A and B move in opposite directions) correlation. If two

attributes are independent, the value should be zero. However, if this value is zero, it does not mean that the attributes are independent (there could be a non-linear relationship), because correlation has an important drawback: it only detects *linear* associations between the variables. As a simple (and typical) example, let $A = (-2, -1, 0, 1, 2)$ and $B = (4, 1, 0, 1, 4)$. Then $B = A^2$, but their correlation is zero.[25]

In PostgreSQL, the aggregate `corr(A,B)` computes the correlation of attributes A and B. There is no aggregate for correlation in MySQL; fortunately, this is another concept that can be expressed in SQL—in several ways, actually. The simplest way to expression correlation in SQL is

```
SELECT (avg(A*B) - (avg(A)* avg(B))) / (std(A) * std(B))
FROM Data;
```

when average and standard deviation are available; or, if an aggregate for covariance exists:

```
SELECT Covar(A,B) / (std(A) * std(B))
FROM Data;
```

Exercise 3.42 A formula typically used for correlation is

$$r_{AB} = \frac{n\Sigma(a_i b_i) - (\Sigma a_i \Sigma b_i)}{\sqrt{n\Sigma a_i^2 - (\Sigma a_i)^2}\sqrt{n\Sigma b_i^2 - (\Sigma b_i)^2}}.$$

Write the SQL query that implements that formula.

Just like covariance was connected to probability, so is Pearson correlation, so it can be computed from probabilities too.

Exercise 3.43 Compute Pearson correlation not from original data but from table `Probabilities` with $P(A)$, $P(B)$, and $P(A, B)$.

For ordinal attributes, it is common to use *rank correlation*, a measure of the relationship between the rankings on each variable. The idea here is to compare the rank of the attributes (their position in the order), instead of the attribute values. We assume that our table `Data` contains, besides attributes A and B, attributes $Arank$ and $Brank$ giving their ranks in their respective orders (i.e. they both are $1, 2, \ldots, n$ for a dataset with n rows). There are several rank correlation measures; the most popular ones are Kendall's τ and Spearman's ρ. The idea behind Kendall's is as follows: given two pairs of (X, Y) values (x_1, y_1) and (x_2, y_2), we say they are *concordant* if the ranks of both elements agree: either the rank of x_1 is higher than that of x_2 *and* the rank of y_1 is higher than that of y_2 or the rank of x_1 is lower than

[25] (Pointwise) mutual information could be used here as is not limited to linear associations, so it is a more general measure of dependence, but it is also harder to interpret.

that of x_2 *and* the rank of y_1 is also lower than that of y_2. Otherwise, we say the pairs are *discordant*. Kendall's rank correlation is computed as

$$\tau = \frac{2(\text{number of concordant pairs})(\text{number of discordant pairs})}{n(n-1)},$$

where n is the number of data pairs. To calculate this in SQL, we need to compare the values of $Arank$ and $Brank$:

```
SELECT
 sum(CASE WHEN ((D1.Arank < D2.Arank AND D1.Brank < D2.Brank)
           OR (D1.Arank > D2.Arank AND D1.Brank > D2.Brank))
          THEN 1 ELSE 0 END) as concordant -
 sum(CASE WHEN ((D1.Arank < D2.Arank AND D1.Brank > D2.Brank)
           OR (D1.Arank > D2.Arank AND D1.Brank < D2.Brank))
          THEN 0 ELSE 1 END) as discordant
      / (count(*) * (count(*) - 1)
FROM Data D1, Data D2;
```

Note that we are using the Cartesian product of the dataset with itself, in order to consider all possible pairs of data points. Note also that when either of the ranks coincide, the pair is neither concordant nor discordant, so we must check for both conditions explicitly.

The value of Kendall's correlation is always between -1 and $+1$: if the agreement between rankings is perfect, we get $+1$; if the disagreement is perfect, we get -1; if the ranks are independent, we get 0.

Exercise 3.44 The above formula for Kendall is expensive due to the Cartesian product. A more direct way to calculate this correlation is

$$\frac{2}{n(n-1)} \Sigma_i (sign(Arank_i - Brank_i),$$

where $Arank_i$ is the rank of the ith element in A, $Brank_i$ is the rank of the ith element in B, and the function `sign` simply tells us whether its argument is positive, zero, or negative and is available in many SQL systems (including PostgreSQL and MySQL). Using this, express this definition in SQL.

To calculate Spearman's Rho, we can use the definition directly:

$$\rho = 1 - \frac{6\Sigma_i d_i^2}{n^3 - n},$$

where $d_i = Arank_i - Brank_i$ is the difference in rank between the ith pair of elements, as above.

```
SELECT 1 - (6 * sum(pow(Arank - Brank, 2)))  /
           (pow(count(*), 3) - count(*))
FROM Data;
```

When one attribute is categorical and another continuous, there are two ways to consider influence: in one, the continuous variable influences the categorical one. The typical approach here is to see if we can use the continuous variable to predict the continuous one using *classification*, which is explained in the next chapter. On the other direction, the typical approach is to see if the categorical attribute has an influence on the continuous value by analyzing the differences between the means of the values generated by each category. In full generality, this is the ANOVA (Analysis of Variance) approach from statistics. When talking about a single categorical variable and a single continuous one, we can use a simplified version, usually called *one-way ANOVA*. We explain this technique through an example:[26] assume we have a table `Growth(fertilizer, height)` where we keep the results of an experiment: several plants where grown for a period of time using different kinds of fertilizer. The first attribute gives the class of fertilizer used, and the second one gives the growth of a plant using that type of fertilizer. We take the following steps:

1. Calculate mean of continuous value for each type:

```
CREATE TABLE group-means AS
SELECT fertilizer, avg(height) as group-mean
FROM Growth
GROUP BY fertilizer;
```

2. Calculate overall mean of the group means.

```
SELECT avg(group-mean) as overall-mean
FROM group-means;
```

3. Calculate the sum of square differences between group mean and overall mean; divide over number of groups minus 1 (degrees of freedom). Note that this is the *between-groups* variability.

```
SELECT sum(pow(group-mean - overall-mean, 2)) * size /
       (num-groups - 1) AS between-groups
FROM (SELECT count(*) AS size FROM Growth),
     (SELECT count(DISTINCT fertilizer) as num-groups
      FROM Growth),
     group-means;
```

4. Calculate the sum of square differences between the group mean and the values on each group and divide by number of groups times number of data points minus 1. Note that this is the *within-groups* variability.

```
SELECT sum(pow(height - group-mean, 2)) /
       (num-groups * (size - 1)) AS within-groups
FROM Growth, group-means
WHERE Growth.fertilizer = group-means.fertilizer;
```

[26] The example is taken from Wikipedia but changed to reflect the structure of the data as a tidy table.

5. The F-ratio is the within-groups variability divided by the between-groups variability. This ratio can be compared to the *F-distribution*, for which tables exist, to determine if the value obtained is significant (this is sometimes called an *F-test*).

To put it all together in a single query, we use the typical strategy of pre-computing needed results with the `WITH` clause and subqueries in the `FROM` clause:

```
WITH group-means AS
     SELECT fertilizer, avg(height) as group-mean
     FROM Growth
     GROUP BY fertilizer
SELECT between-groups / within-groups
FROM (SELECT sum(pow(group-mean - overall-mean, 2)) *
             (size * (num-groups - 1)) AS between-groups
      FROM (SELECT count(*) AS size FROM Growth),
           (SELECT count(DISTINCT fertilizer) as num-groups
            FROM Growth),
           group-means) AS temp1,
      (SELECT sum(pow(height - group-mean, 2)) /
              (num-groups * (size - 1)) AS within-groups
       FROM Growth, group-means
       WHERE Growth.fertilizer = group-means.fertilizer)
           AS temp2;
```

When both attributes are categorical, we use the basic histogram technique to compare counts, in what is called a *contingency table* or *cross-tabulation*. Based on the counts, we can compute the *chi-square (χ^2) test* of independence. In its simplest form, the idea is to compare the number of co-occurrences one observes between their values with the number of co-occurrences one would expect if the attributes were independent of each other. This is a basic technique that can also be applied to ordinal and numerical attributes by counting their frequencies.

The idea is simple: if attribute A has n possible values and attribute B has m possible values, let c_{ij} be the count of data with value i of A and value j of B, $c_{i\_}$ the count of data with value i of A, $c_{\_j}$ the count of data with value j of B (these last two are sometimes called *margin sums*), and c the total count of values; then we call $E(i, j) = \frac{c_{i\_} c_{\_j}}{c}$ the *expected* value of i, j and compare this with c_{ij}, the *observed* value (since it comes from the data):

$$T(X, Y) = \Sigma_i \Sigma_j \frac{(c_{ij} - E(i, j))^2}{E(ij)}.$$

This is the χ^2 test of independence. The data is usually presented as a matrix (spreadsheet) with the margin sums, with format:

	Y_1	Y_m	Row total
X_1	c_{11}	c_{1m}	$c_{1\_} = \Sigma_j c_{1j}$
$\vdots$	$\cdots$	$\cdots$	$\vdots$
X_n	c_{n1}	c_{nm}	$c_{n\_} = \Sigma_j c_{nj}$
Column total	$c_{\_1} = \Sigma_i c_{i1}$	$c_{\_m} = \Sigma_i c_{im}$	n

where $c_{i\_} = \Sigma_j c_{ij}$, $c_{\_j} = \Sigma_i c_{ij}$ (this two-way table of A-B counts is an example of bivariate histogram).

As we saw in Chap. 2, in databases the data should be in a tidy table; this means an attribute describing the rows, another one describing the columns, and another one giving the values for each combination—that is, we would have a table with schema `Data(X,Y,Attr)`. Getting the counts from this is very easy; for instance, the different $c_{i\_}$ are given by

```
SELECT X, count(*)
FROM Data
GROUP BY X;
```

and likewise for the B counts $c_{\_j}$ and the joint counts c_{ij}. The result of the tabulation is compared to the chi-square distribution with $(m-1)\times(n-1)$ degrees of freedom; if the result if greater than the chi-square, then the attributes are not independent. If the result is smaller than the chi-square, then the attributes are independent.

Example: Chi-Square Test

Assume a table that gives, for the subscribers to a magazine, the city where they reside and their status—whether they are currently active subscribers or have stopped their subscription.[27] The question is whether the city where they live affects the rate of subscription.

CUSTOMERS		
City	Status	Count
Gotham	Active	1462
Gotham	Stopped	794
Metropolis	Active	749
Metropolis	Stopped	385
Smallville	Active	527
Smallville	Stopped	139

[27] This example comes from a website that, unfortunately, seems to be invisible to searches right now.

To answer the question, we proceed as follows:

1. Compute the margins: total by status, by city, and total overall.
2. Compute the overall stop (active) rate: total stop (active) divided by total.
3. Compute the expected values (per city): total number per city times overall rate (for active and passive).
4. Compute the deviation: the difference between observed and expected value (for active and passive).
5. Compute the χ^2: deviation squared divided by expected value.

In SQL, this can be written in a single query, using WITH and subqueries in the FROM clause to organize the margin sum calculations:

```
WITH
 (SELECT Cust.city, Cust.status, Cust.Count,
         perCity * (perStatus / total) as expected
  FROM Cust,
       (SELECT city, sum(Count) as perCity
        FROM Cust
        GROUP BY city) as Cities,
       (SELECT status, sum(Count) as perStatus
        FROM Cust
        GROUP BY status) as Statuses,
       (SELECT sum(Count) as total FROM Cust)
  WHERE Cust.city = Cities.city
        and Cust.status = Statuses.status)
SELECT sum(pow(Count-expected,2)/expected);
```

Exercise 3.45 Assume a car company is testing 3 new car models, called A, B, and C. It tests the car for 4 days, giving a score between 1 and 10 each day. They end up with a matrix (spreadsheet):

	A	B	C
Day 1	8	9	7
Day 2	7.5	8.5	7
Day 3	6	7	8
Day 4	7	6	5

The question is: are there significant differences between the models? Do the test results depend on the day and not the model? Put the data in a relational format and carry out a χ^2 test.

Exercise 3.46 Assume the simplest contingency table, one with two binary attributes A and B: A can only take values x_1 and x_2, and B can only take values y_1 and y_2. Let us give names to the counts as follows:

		Y		
		y_1	y_2	
X	x_1	a	b	a+b
	x_2	c	d	c+d
		a+c	b+d	n

where $n = a + b + c + d$. Then we can write the chi-square test as follows:

$$\chi^2 = \frac{n(ad - bc)^2}{(a + b)(c + d)(a + c)(b + d)}.$$

Create a tidy table (one with schema `(X, Y, count)`) to represent the table above and write an SQL query to implement this formula.

Exercise 3.47 Using this contingency table again, we can define the *odds ratio* of A as $\frac{ad}{bc}$. Write an SQL query over your tidy table to compute this result.

Exercise 3.48 Again using this contingency table, we can define the *relative risk* of x_1 as $\frac{a/(a+c)}{b/(b+d)}$. Write an SQL query over your tidy table to compute this result.

3.2.3 Distribution Fitting

Sometimes, it is suspected that some of the (numerical) data in our dataset has been generated by a process that follows some standard probability distribution (at least approximately, since real data always has some noise). Whether this is the case, it can be checked by generating artificial values with a formula for the distribution and comparing what we obtain with the data values. When the difference is not significant, we consider that indeed the data has been generated by a process with an underlying distribution. This process is called *distribution fitting*. When used to check whether some hypothesis a researcher has actually holds in the data, this is part of the process of *(statistical) hypothesis testing*.

To guess a distribution, one can start by using histograms and calculating some basic measures of centrality and distribution, as indicated earlier. There are more sophisticated techniques, like Kernel Density Estimation, which we will not cover here. However one comes up with the initial guess, one should check that indeed the chosen distribution is a good fit. Several checks are available for this; chi-square can be used.

The general approach is this:

1. Set the data in the form of a distribution, that is, create a table with schema `(value, probability)`. This can be done as seen previously, by estimating probabilities from percentages, and percentages from raw counts of data.
2. Find out the parameters of this empirical distribution, in particular, mean and standard deviation.
3. Add an attribute for the estimate to the data table. We end up with a table with schema `(value, probability, estimate)`. This new third column has nulls on all rows for now.
4. Choose a distribution. Using the formula for this distribution, and the mean and standard deviation of the sample, generates an estimate for each value and leaves it on the `estimate` attribute.
5. Compare the estimate obtained (`estimate`) with the empirical `probability`.

Example: Fitting a Normal Distribution

A phone company records the lengths of telephone calls.[28] The data is converted into a table DATA with columns *number of minutes, observed number of calls with those minutes*. This can be done for each number of minutes (1,2,3,...) or as a histogram with time intervals, after choosing a width (0 to 2 min, 2 to 4 min, etc. for a width of 2 min). We calculate estimates for μ (mean) and σ^2 (variance) from the data in table DATA. We then apply the density function of the normal distribution:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

to the *number of minutes* column (so *number of minutes* is x) to get the column *estimated number of minutes*. In SQL:

1. Create table DATA and populate it with data. If the data is in *raw* format `(callid, number of minutes)`, a simple group by and count query will produce the table DATA with probabilities instead:

```
CREATE TABLE DATA AS
SELECT num-minutes, sum(1.0/ total) as prob
FROM Raw-data, (SELECT count(*) as total FROM Raw-data)
GROUP BY num-minutes;
```

2. Create a table with columns for the data, the observed frequencies, and the expected frequencies and fill in first two columns from the data and the last one with nulls.

```
CREATE TABLE NORMAL(data,observed-freq,expected-freq) AS
SELECT num-minutes, freq, null
FROM DATA;
```

[28] This example is from [11].

Note that the previous two steps can be combined into one; we separate them to explain the process in a step-by-step manner, but they can be easily combined (see Exercise below).

3. Calculate mean and standard deviation in the data as usual. Then populate the column for expected frequencies by applying the formula of the normal distribution to the data column and the mean and standard deviation. In PostgreSQL,

```
UPDATE NORMAL
SET expected-freq = (1 / sqrt(2*pi()*stddev)) *
    exp(- (pow(observed-freq - avg, 2) / (2*stdev)));
```

4. Compare observed frequencies and expected frequencies and decide if the difference is small enough. A very simple way to do this is to add the absolute differences and express them in terms of standard deviations (see Exercises). A more sophisticated way to attack the problem is to run a chi-square test.

Exercise 3.49 Combine the first two steps above, creating the table NORMAL in one single query.

Exercise 3.50 Write an SQL query to compare observed and expected frequencies as suggested above (sum of absolute differences over standard deviation).

To fit a different distribution, we simply use the appropriate formula. For instance, we may want to fit a Poisson distribution to data. This distribution rises very rapidly, and once the peak is reached it drops very gradually. This is typical of discrete events, where the non-occurrence makes no sense. This is also typical of arrival and departure events, and it is why this distribution is so frequently used in queue theory. The density function for Poisson is

$$\frac{e^{-\mu}\mu^x}{x!},$$

where μ as usual is the mean of the distribution, $x!$ is the *factorial* of x, and x is the expected number of times the event can happen in a given time period (so the above can be calculated for $x = 1, 2, \ldots$). For instance, if x is the number of orders per day in an e-commerce site, this is the percentage of days we can expect x orders. Thus, we can build a table `Poisson(Xval, Expected)`, where for $Xval = 1, 2, \ldots$ we can calculate the expected value with the formula above, multiplying the result (which is a percentage) by a total value of x. For example, for 100 days, we multiply the percentage by 100, which will tell us how many days, out of 100, we can expect x orders. Again, this is an expected value. We can compare this with the observed value from the data, using χ^2 again. Note that the Poisson values can also be accumulated, to determine the days where orders will be x or less.

Exercise 3.51 A website collects the number of orders they get each day for a certain time period (n days). This is then converted to a table DATA with schema

number of orders per day, observed number of days with that number of orders. From the original data or from DATA, we can calculate the mean, which is used for μ. Then the table DATA is expanded by using the density function and computing, for each number of orders per day, the Poisson expected value: that is, x is the number of orders per day, and we calculate the expected percentage of days that would have that value (column *expected percentage*). This expected percentage is then multiplied by the total number of days to get the expected number of days with the value (column *expected number of days*). Implement this in SQL with some made-up values.

Fitting distributions can be used to find and remove outliers, as we will see in Sect. 3.3.3.

3.3 Data Cleaning

Data cleaning is the set of activities and techniques that identify and fix problems with the data in a dataset. In general, we want to identify and get rid of 'bad' or 'wrong' data. However, it may be very hard to identify such data, depending on the domain and data type:

- For (finite) enumerated types, a simple membership test is needed. For instance, months of the year can be expressed as integers (in which case only values between 1 and 12 are allowed) or as strings (in which case only value "January," ..., "December" are allowed). For this types, distinguishing 'good' from 'bad' data is usually not hard, unless some non-standard encoding is used (for instance, using 'A,' 'B,' ... for months of year).
- For pattern-based domains (like telephone numbers or IP addresses), a test can rule out values that do not fit the pattern. However, when a value fails to follow the expected pattern, it may be a case of bad formatting; such values must be rewritten, not ignored, or deleted (see next section for more on this). Telling badly formatted from plain wrong values can become quite hard.
- In open-ended domains (as most measurements are), what is a 'bad' or 'wrong' value is not clear cut. An in-depth analysis of the existing values and domain information have to be combined to infer the range of possible and/or likely values, and even this knowledge may not be enough to tell 'good' from 'bad' values in many cases.

In general, the problems attacked at this stage may be quite diverse and have different causes; hence, data cleaning is a complex and messy activity. Different authors differ in what should be considered at this stage; however, there are some basic issues that most everyone agrees should be addressed at this point, including

- *Proper data*. For starters, we need to make sure that values are of the right kind and are in a proper format. As already mentioned in Sect. 2.4.1, when we load data into the database we must make sure that values are read correctly. In spite

of our efforts, we may end up with numbers that are read as strings (because of commas in the values, or other problems), or dates that are not recognized as such and also read as plain strings. Even if the data is read correctly, the format may be not appropriate for analysis. Hence, making sure data is properly represented is usually a first step before any other tasks.

- *Missing values*. In the context of tabular data, this means records that are missing some attribute values, not missing records. The issues of whether the data records we have in a dataset are all those that we should have and whether the dataset is a representative sample of the underlying population are very important but different and treated in detail in Statistics. Missing values refers exclusively to records that are present in the dataset but are not complete—in other words, missing *attribute* values. Detection may be tricky when the absence of a value is marked in an ad-hoc manner in a dataset, but the real problem here is what to do with the incomplete records. The general strategies are to ignore (delete) the affected data (either the attribute or the record), or to predict (also called *impute*) the absent values from other values of the same attribute or from value of other attributes. The exact approach, as we will see, depends heavily on the context.
- *Outliers*. Outliers are data values that have characteristics that are very different from the characteristics of most other data values in the same attribute. Detecting outliers is extremely tricky because this is a vague, context-dependent concept: it is often unclear whether an outlier is the result of an error or problem, or a legitimate but extreme value. For instance, consider a person dataset with attribute `height`: when measured in feet, usually the value is in the 4.5 to 6.5 range; anyone below or above is considered very short or very tall. But certainly there are people in the world who are very short (and very tall). And, of course, we would expect different heights if we are measuring a random group of people or basketball players. In a sense, an outlier is a value that is not *normal*, but what constitutes normal is difficult to pin down [15]. Thus, the challenge with outliers is finding (defining) them. Once located, they can be treated like missing values.
- *Duplicate data*. It is assumed that each data record in a dataset refers to a *different* entity, event, or observation in the real world. However, sometimes we may have a dataset where two different records are actually about the same entity (or event, or observation), hence being duplicates of each other. In many situations, this is considered undesirable as it may bias the data. Thus, detecting duplicates and getting rid of them can be considered a way to improve the quality of the data. Unfortunately, this is another very difficult problem, since most of the time all we have to work with is the dataset itself. The simpler case where two records have exactly the same values for all attributes is easy to detect, but in many cases duplicate records may contain attributes with similar, but not exactly equal, values due to a variety of causes: rounding errors, limits in precision in measurement, etc. This situation make duplicate detection much harder. The special case of *data integration* (also called *data fusion*), where two or more datasets must be combined to create a single dataset, usually brings the problem of duplicate detection in its most difficult form [6, 7].

There are other activities, like discretization, that are sometimes covered under data cleaning. In this textbook, some of those activities are discussed in the next section under the heading of *data pre-processing*. Here, we focus on the four topics introduced above. However, in real life one may have to combine all activities in a feedback loop; for instance, one may have to standardize values before analyzing them looking for outliers or duplicates.

3.3.1 Attribute Transformation

In general, *attribute transformations* are operations that transform the values of an attribute to a certain format or range. These are needed to make sure that data values are understandable to database functions, so that data can be manipulated in meaningful ways. Many times it may be necessary to transform data before any other analysis, even EDA, can begin.

For categorical attributes, a typical transformation is *normalization standardization*, a process whereby each value is given a unique representation so that equality comparisons will work correctly. For instance, a comparison between strings may differentiate between uppercase and lowercase characters, something that may create artificial distinctions (i.e. if values like 'Germany' and 'germany' are present in the dataset). For numerical attributes, typical transformations include *scaling* and *normalization*. Because these transformations are highly dependent on the data type, they are discussed separately next.

3.3.1.1 Working with Numbers

Two common operations on numerical values are *scaling* and *normalization*. Scaling makes sure that all values are within a certain range. *Linear scaling* transforms all values to a number between 0 and 1; the usual approach is to identify maximum and minimum values in the domain (or in the attribute, as present in the database[29]) and transform value v to $\frac{v - min}{max - min}$. This is easily implemented:

```
UPDATE Dataset
SET Attr = (Attr - (SELECT avg(Attr) FROM Dataset)) /
            ((SELECT max(Attr) FROM Dataset) -
             (SELECT min(Attr) FROM Dataset));
```

Unfortunately, this will not work in all SQL databases (for instance, it works in Postgres, but it does not in MySQL), due to the following: we are changing values in table `Dataset`, but we also want some statistics (mean, minimum, maximum) obtained from it. The intended meaning, of course, is that such statistics must be

[29] What statisticians call the sample.

obtained from the table *before* any changes are applied to the table—and that is indeed what Postgres does. However, some systems will not understand this order of evaluation and will consider that we are asking to examine and modify table `Dataset` at the same time. To work around this problem, sometimes the query must be rewritten to isolate the statistics computation and make clear that it should happen before any changes:

```
UPDATE Dataset
SET Attr = (Attr - (SELECT mean
                    FROM (SELECT  avg(Attr) as mean
                          FROM Dataset) as D1)) /
                   (SELECT maxa - mina
                    FROM (SELECT max(Attr) as maxa,
                                 min(Attr) as mina
                          FROM Dataset) as D2 ) ;
```

Another scaling that provides values between 0 and 1 but is not linear in nature is the *logistic* scaling where value v is mapped to

$$\frac{1}{1+e^{-v}}.$$

The logistic function is used when the presence of outliers (very large or very small values) is suspected, as it can accommodate them at the top or the bottom of the range. This function gives the typical 'sigmoid' graph, softly approaching the minimum and maximum (0 and 1) without ever reaching it.

Exercise 3.52 Assume a generic table `Dataset` with attribute `Attr`. Implement logistic scaling of `Attr` in SQL.

Normalization consists of making sure that all values are expressed using a known quantity as the unit. The typical example is the *z-score*, where values are expressed in terms of standard deviations from the mean:

$$Z(v) = \frac{x-\mu}{sdev},$$

where μ is the mean of the domain of x and $sdev$ the standard deviation. This again is easy to express in SQL:

```
UPDATE Data
SET Attr = (Attr - (SELECT avg(Attr) FROM Data)) /
            (SELECT std(Attr) FROM Data);
```

Many other transformations are possible.

Exercise 3.53 The same caveat as in the previous transformation applies: in some systems, like MySQL, this query needs to be written with subqueries to force an evaluation order. Do this following the model provided in the scaling case.

When we have an attribute with a skewed distribution (see Sect. 3.2.3) needs to be corrected, the variable is typically transformed by a function that has a disproportionate effect on the tails of the distribution. The most often used transformations in this case change value v using the log transform ($log(v)$ in base 2, e or 10, commonly), the multiplicative inverse ($\frac{1}{v}$), the square root ($sqrt(v)$), or power ($power(x, n)$ for some n).

3.3.1.2 Working with Strings

In general, cleaning categorical attributes means making sure that one and only one name (string) represents each category. One does not want small, irrelevant differences (like letters being uppercase or lowercase) to interfere with tasks like grouping or searching for values.

Because strings can be used to represent many diverse values, there are no hard and fast rules about how to deal with strings. Ideally, one would like to make sure that the strings being used are standardized, that is, they use a set of standard names so that no confusion can occur. This is easy in the case of closed (enumerated) domains; for others, the best approach is to come up with a set of *conventions* that are followed through the analysis.

Example: String Normalization

Assume a student table that contains, among other attributes, the name of the department where the student is majoring. Unfortunately, and because of manual entry, this attribute contains many different ways of spelling the same value:

Student-id	...	Department
1	...	Dept of Economics
2	...	Econ
3	...	Department of Econ

All these names should be unified to a single, canonical one. In this simple example, the following SQL command will do the trick:

```
UPDATE Student
SET Department = "Economics"
WHERE position('Econ' IN Department) > 0;
```

As we will see, `position` is a string function that tells us at what position the string denoted by the first argument appears in the second; asking for a position greater than zero simply means that the string 'Econ' has been found in the value of attribute `Department` in a given tuple. In this case, this happens to capture all variations of the name. In more complex, real-life cases, several commands may be needed to capture all cases.

The above example is typical of how string functions are used to put string values in an appropriate format. The most common tasks are to 'trim' the strings (getting rid of whitespaces and other non-essential characters), to modify the string to some 'standard' form (like making sure the whole string is lowercase), or to extract part of a string for further analysis.

There are many string functions, and unfortunately different systems may express them in slightly different ways. Instead of trying to go over each possibility, we organize the functions by what they do and give some examples of each type, focusing on commonly used functions that are present in both Postgres and MySQL:

- Functions that *clean* the string: they change it by getting rid of certain characters or transforming existing characters. Among them are TRIM(), LOWER(), and UPPER(). The inverse of this (adding characters to a string) is called *padding*, expressed with LPAD(), RPAD(), and others.
- Functions that *find* elements (characters or substrings) within a given string. The most popular ones are POSITION() and STRPOS().
- Functions that *extract* elements of a string or *split* a string into parts. This includes functions SUBSTR() and SPLIT(). The inverse of this (putting together several strings into a single one) is usually called *concatenation*, expressed by CONCAT().

Other functions typically available include several forms of *replacement*, where parts of a string are removed and other characters substituted for them, like SUBSTR().

Finally, most functions include the useful function LENGTH(), which returns the number of characters in a string.

We now provide examples for some of these functions. A typical cleaning function, `TRIM(position characters FROM string)`, removes any character in `characters` (the default is whitespace) from string `string` starting at `position` 'leading' (leftmost), 'trailing' (rightmost), or 'both' (the default). Postgres also has functions `LTRIM()` (equivalent to `TRIM('leading',...)`) and `RTRIM()` (equivalent to `TRIM('trailing',...)`). As an example of use, in the dataset `ny-rolling-sales`, several attributes (like Neighborhood) have a fixed length, meaning they take the same space regardless of actual length. This may result in padding (extra blanks added to the value); to get rid of it, we write

```
SELECT TRIM(both ' ' FROM Neighborhood)
FROM ny-rolling-sales;
```

to get the values without padding (note that whitespaces in the middle are not affected).

LOWER() is used to force every character in a string to become lowercase (if the character is already in lowercase, it is left untouched). A similar effect is achieved with UPPER().

```
SELECT UPPER(address)
FROM ny-rolling-sales;
```

These functions are usually applied to categorical attributes in combination with UPDATE so that values are described in a uniform manner; this way, searches do not miss values and grouping works correctly.

Among finding functions, `POSITION(substring IN string)` (equivalently, `STRPOS(string, substring)`) returns a numerical value denoting the character position where the substring first appears in the string (character positions are numbered starting at 1 on the left; if the substring is not found, 0 is returned). This function is useful because the value it returns can be used by other functions, as we will see.

```
SELECT POSITION('East' IN address)
FROM ny-rolling-sales;
```

Extracting functions include `LEFT(string, n)`, which gets n characters from the beginning (left) of the string (function `RIGHT(string, n)` does the same from the end of the string). This function is useful when all string values follow a certain pattern and we need to extract a part of the string based on those patterns.

Example: Combining String Functions

Old datasets from the Imdb website[30] contain a field where both movie title and year are combined together as the value of a (single) attribute, as in "Amarcord (1973)"; we can extract the year with

```
SELECT RIGHT(title, 6)
FROM imdb;
```

since the year (together with the parenthesis) constitutes the last 6 characters of each title. If we wanted to extract the title, we cannot rely on a fixed position, but we know we should go all the way to the opening parenthesis, '(,' so we could write

```
SELECT LEFT(title, STRPOS(title, '('))
FROM imdb;
```

The number of characters to extract is calculated by finding the position where the '(' appears (in our example, 9). Note that, when evaluating functions, just like in Math, innermost functions are evaluated first.

To extract a part of a string starting somewhere in the middle, we use `SUBSTR(string, start-pos, length)`, which extracts the substring of its first argument that starts at the second argument and has as many characters as the

[30] The Internet Movie DataBase, https://www.imdb.com.

third argument indicates. Thus, another way to extract the year (this time without parenthesis) is to use

```
SELECT SUBSTR(title, 9, 4)
FROM imdb;
```

If we want to combine string values instead of splitting them, we can use the CONCAT function. This function simply takes a sequence of strings and returns a single string that is the combination of all the arguments. The arguments can be a mixture of attribute values and constants; this can be used to put a value in a certain format. For instance, to separate title from year using a hyphen, we could use

```
SELECT CONCAT(LEFT(title, STRPOS(title, '(')), ' - ',
              SUBSTR(title, 9, 4))
FROM imdb;
```

Most systems allow the use of the two pipe characters (||) as synonym for concatenation.

Exercise 3.54 Consider the `Name` attribute in table `city` of database `world`.

1. Some values in this attribute include a second name in parenthesis. Display `Name` without such second names.
2. Some values in this attribute are compound; but the parts of the name are separated by hyphen (-), sometimes not. Display `Name` with all hyphens suppressed.
3. Display compound names always with hyphen (i.e. change whitespaces between words to hyphens).

Finally, many systems include a useful function that allows concatenating string values *across rows*. Because they work in sets of rows, technically these functions are aggregates, but they behave unlike typical aggregate functions, which are numerical. The string concatenation version (called `GROUP_CONCAT()` in MySQL and `STRING_AGG()` in Postgres) simply creates a new value by putting all its arguments together; usually, a character separator is used (comma in MySQL; specified as the second argument, in Postgres) and sometimes it is possible to specify an order. As any other aggregate, this one can be used with grouping.

Example: String Concatenation

Recall table `Country` in database `world`, where each row contains information of a certain country, including attributes `Continent`, `name`, and `Population`. Then the MySQL query:

```
SELECT Continent,
       GROUP_CONCAT(name ORDER BY Population DESC)
FROM Country
GROUP BY Continent;
```

will return one row per continent, and for each one a single string listing all country names, separated by comma, and ordered by their population. The result will look like

```
+-----------+-------------------------------------------+
| Continent | GROUP_CONCAT(Name ORDER BY Population DESC)|
+-----------+-------------------------------------------+
| Asia      | China,India,Indonesia,Pakistan,Bangladesh,.|
| Europe    | Russian Federation,Germany,United Kingdom,.|
| North America | United States,Mexico,Canada,Guatemala,.|
| Africa    | Nigeria,Egypt,Ethiopia,Congo, The Democr.. |
| Oceania   | Australia,Papua New Guinea,New Zealand,Fiji|
| Antarctica| Heard Island and McDonald Islands,Antarcti.|
| South America |Brazil,Colombia,Argentina,Peru,Venezu...|
+-----------+-------------------------------------------+
```

3.3.1.3 Working with Dates

Dates are some of the most problematic types of data in SQL. This is due to the fact that dates can be expressed in many different formats: months can be expressed by name (January, etc.) or by number; years are sometimes written in full (2019) or shorted by century (19); the order of elements can change (year-month-day, month-day-year, day-month-year,...). In many countries, the standard format is DD-MM-YYYY (that is, two digits for the day first, followed by two digits for the month, followed by four digits for the year), but in the United States, the format is MM-DD-YYYY (month goes first). The SQL standard form for DATE is defined as YYYY-MM-DD. The reason to adapt this format is that, if sorted, it yields chronological order.[31] The standard for TIME is HH:MM:SS[.NNNNNNN] (two digits for hours first, followed by two for minutes, followed by two digits for seconds, and optionally seven digits for fractions of a second).[32] The SQL standard for DATETIME (timestamps) is a DATE plus a TIME, separated by a space: YYYY-MM-DD HH:MM:SS. The standards for INTERVAL distinguish between two classes of intervals:

- the YEAR TO MONTH class, with format: YYYY-MM.
- the DAY TO FRACTION class, with format: DD HH:MM:SS.F.

In Postgres, DATE, TIME, and TIMESTAMP follow the standard. For instance, '2017-01-08 04:05:06' is a valid timestamp. Other formats are also allowed; examples of valid dates are: 'January 8, 2017'; '2017-Jan-08'; '08-Jan-2017'. The date '1/8/2017' is allowed, but note that it is ambiguous. Examples of valid times

[31] This format is also in agreement with the ISO 8601 standard.

[32] This format is almost the ISO 8601 standard; it differs in the fractional seconds.

are: '04:05:06,' '04:05,' '040506,' '04:05 AM.' Note that any date or time value needs to be enclosed in single quotes.

Interval values are written using the pattern:

`quantity unit [quantity unit...] [direction]`,

where `quantity` is a number (possibly signed); `unit` is one of: microsecond, millisecond, second, minute, hour, day, week, month, year, decade, century, millennium, or abbreviations or plurals of these units; `direction` can be "ago" or nothing at all. For instance, `1-2` denotes the interval 1 year 2 months; `3 4:05:06` denotes 3 days 4 h 5 min and 6 s.

Most systems support *arithmetic* on time data: '+' can be used to add two dates, or two intervals, or add an interval to a date:

- Date '2001-09-28' + integer '7' results in date '2001-10-05' by adding seven days to the first argument.
- Date '2001-09-28' + interval '1 h' results in timestamp '2001-09-28 01:00:00' by adding 1 h to the original date, assumed to denote midnight of that day.
- Date '2001-09-28' + time '03:00' results in timestamp '2001-09-28 03:00:00' by adding 3 h to the original date, again assumed to denote midnight.
- Interval '1 day' + interval '1 h' results in an interval '1 day 01:00:00,' that is, one day and 1 h.

Likewise, '−' can be used for subtraction.

The function `EXTRACT` is used to get information bits from a date or timestamp:

```
SELECT cleaned_date,
       EXTRACT('year'   FROM cleaned_date) AS year,
       EXTRACT('month'  FROM cleaned_date) AS month,
       EXTRACT('day'    FROM cleaned_date) AS day,
       EXTRACT('hour'   FROM cleaned_date) AS hour,
       EXTRACT('minute' FROM cleaned_date) AS minute,
       EXTRACT('second' FROM cleaned_date) AS second,
       EXTRACT('decade' FROM cleaned_date) AS decade,
       EXTRACT('dow'    FROM cleaned_date) AS day_of_week
  FROM Data;
```

Other convenient functions to manipulate temporal data include:

- `current_date()` returns the current date.
- `current_time()` returns the current time of the day.
- `now()` returns the current date and time.
- `make_date(year int, month int, day int)` creates a date value.
- `make_interval(years int DEFAULT 0, months int DEFAULT 0, weeks int DEFAULT 0, days int DEFAULT 0, hours int DEFAULT 0, mins int DEFAULT 0, secs double precision DEFAULT 0.0)` creates an interval.
- `make_time(hour int, min int, sec double precision)` creates a time.
- `make_timestamp(year int, month int, day int, hour int, min int, sec double precision)` creates a timestamp.

Casting can also be used to create times. In fact, going from strings to dates often involves some text manipulation, followed by a *cast*, an operation where the system is told to coerce a value into a certain type. This is indicated by '::' in Postgres, as the next example shows.

Example: Casting Strings into Dates

Assume the data contains a string-based attribute `thedate` with values like '15-04-2015' and we want to convert them into dates:

```
SELECT (SUBSTR(thedate, 7, 4) || '-' || LEFT(thedate, 2) ||
        '-' || SUBSTR(thedate, 4, 2))::date AS date
  FROM Data;
```

The string functions break down the original string value into parts (the year starts at position 7, and it takes 4 characters; the month starts at position 4 and goes for 2 characters; and the day are the first (leftmost) 2 characters) that the casting "::date" can convert into a date attribute.

Also, functions `TO_DATE(text, pattern)` and `TO_TIMESTAMP(text, pattern)` can be used for conversion. The pattern argument indicates how the conversion should interpret the string; for instance, `to_date('05 Dec 2000', 'DD Mon YYYY')` indicates that '05' is the day, 'Dec' the value of the month, and '2000' the value of the year.

In addition to these functions, the SQL OVERLAPS operator on intervals is supported in most systems:

- `(start1, end1) OVERLAPS (start2, end2)` is true if `start1` is earlier than `start2` but `end1` is later than `start2` but earlier than `end2`, or vice versa;
- `(start1, length1) OVERLAPS (start2, length2)` is true as before, with `end1 = start1 + lenght1` and `end2 = start2 + length2`.

MySQL is a bit more restrictive about temporal values than Postgres is. DATE values must have the standard format of year-month-day (for example, '98-09-04'). However, the year may be 2 or 4 digits (with two digits, values 70 to 79 are given to the twentieth century, and values 00 to 69 to the 21st). Also, when expressing the date as a string, one can use any punctuation character as delimiter (so '98/09/04' and '98@09@04' are also okay), and even no delimiters ('980904') as far as the system can make sense of the value as a date. It is also possible to express the value as a number (again, as far as the system can make sense of the value as a date). Finally, MySQL has a "zero" value of '0000-00-00' as a "dummy date," which can be used in place of NULL for missing values. The same is true of DATETIME and TIMESTAMP values: they should follow the standard, but they can be expressed as a string with whatever delimiter the user chooses (or no delimiter).

As for functions, MySQL uses `ADDDATE()` or `DATE_ADD()` to add a time value to a date, and `DATE_DIFF()` to subtract a date from another. Function `ADDTIME()` adds two times, and `DATE_SUB()` subtracts a time from a date.

Example: Date Functions in MySQL

Here is an example that uses date functions. The following query selects all rows with a date_col value from within the last 30 days:

```
mysql> SELECT something FROM tbl_name
    -> WHERE DATE_SUB(CURDATE(),INTERVAL 30 DAY) <= date_col;
```

An important function is `DATE_FORMAT(date,format)`, which formats the date value according to the format string. The format is expressed by using a sequence of *specifiers*, characters preceded by the percent (%) sign, which indicate how to lay out the date. For instance, specifier `%a` is used for weekday name, abbreviated (Sun..Sat); `%W` for weekday name, not abbreviated (Sunday . . . Saturday), and `%w` for weekday as a number (0=Sunday..6=Saturday). Similar choices exist for month, year, hour, minutes, and seconds.

Formatting Dates in MySQL

The query

```
SELECT DATE_FORMAT("2019-08-12", "%M %d %Y");
```

returns

```
August 12 2019
```

Specifier `%M` indicates that the month is shown in full name; `%d` that they day is shown as a number; and `Y` that the year is shown as a 4 digit number.

Creating Times in MySQL

When creating a table for the `ny-flights` dataset, times like arrival and departure were modeled as integers, since this is how the raw values were expressed (‘547’). However, this is not convenient for meaningfully manipulating the values (for instance, finding differences). To transform such values into times, we can use the function `MAKE_TIME(h,m,s)`, where argument `h` is an integer giving the number of hours; argument `m` is an integer giving the number of minutes; and argument `s` is an

integer giving the number of seconds. To transform '547' into time '5:47:00,' we can use

```
SELECT MAKE_TIME(arr_time DIV 100, arr_time MOD 100, 0)
FROM ny-flights;
```

A similar approach will work for other times in the dataset.

Exercise 3.55 Transform scheduled departure time and departure time attributes in `ny-flights` as shown and take their difference (as times); compare this difference with the attribute `dep_delay`. Do you get the same values?

Exercise 3.56 Combine attributes `year`, `month`, `day`, `hour`, and `minute` into a time (note that you will need a 'seconds' value) and compare the result to attribute `time-hour`. Do you get the same values? What seems to be the difference?

3.3.2 Missing Data

The first problem with missing values is identifying them. In some dataset, missing values are not explicitly identified: the value is simply not present. In csv files, for instance, a missing value is identified by a value not being in its place; this results in two commas being adjacent ('„') or a record (line) ending in a comma instead of a value. In some datasets, explicit markers are used; unfortunately, there are no 'standard' or 'typical' markers, so the dataset must be examined carefully for such markers. However, some approaches are common. When there is an identifiable range of values, it is common to use values that are outside the range to mark missing data, so that they are not confused with regular values. For instance, a -1 may be used in a numeric field that is supposed to have only positive values. For strings with delimiters (quotes), sometimes the *empty string* ("") is used. In those cases, it is necessary to eliminate those 'special' values: the -1 would confuse many statistical analyses, and the empty string would still be treated as a string by a lot of software. Another typical value in categorical fields is the strings 'n/a' (or its variations 'N/A' or 'NA'), which stand for 'not available.' Since inside the database there is only one legitimate way to identify missing data (with the *Null* marker), we must identify and substitute such values when data is loaded into the database (see Sect. 2.4.1) or right afterward, before any analysis.

We assume in the rest of this section that missing values have been located in the data and substituted by Null markers, if not already identified by them. This can be achieved with the UPDATE command; for instance, imagine a dataset *Patients* with information about patients in a medical study. One of the attributes is `Height`, and another one is `Weight`, representing, respectively, the height and weight of each patient at the beginning of a study. We load it into table `Patient`, and note that

some Heights and Weights are missing, indicated by a -1. We then 'clean' the table as follows:

```
UPDATE Patients
SET Weight = NULL
WHERE Weight = -1;
```

and similarly for `Height`.

Recall that SQL's Null is not a value, but an indicator that there is a 'hole' in a tuple, i.e. a Null denotes the *absence* of a value. Because of this, Nulls behave in somewhat idiosyncratic ways. For the purposes of Data Analysis, the most important characteristics of Nulls are: how they interact with comparisons in the WHERE clause, with aggregates in the SELECT clause, and with grouping when a GROUP BY is present.

With respect to conditions, the most important characteristic to remember is that all comparisons with Null fail. Technically, comparisons in SQL can return, besides a True and False result, and 'Unknown' result, and this is what happens when a value is compared with a Null—but since queries only return tuples that yield a True result in comparisons, we can think of comparisons with Null as returning False. As an example, assume that we are looking, in the `Patients` table, at patients who may be overweight, and we put a cut point of 220 lbs.[33] The query

```
SELECT *
FROM Patient
WHERE Weight > 220;
```

should retrieve all such patients. However, imagine that (for whatever reasons) we do not have the weight for some patients; the attribute `Weight` has some Nulls on it. All such tuples will `not` be retrieved, since the condition `Weight > 220` will return 'Unknown' on those tuples, and the system only uses, for answering a query, tuples that return 'True' to conditions in the WHERE clause.

The fact that comparisons with Nulls always fail extends to even comparisons of two Nulls (they also fail) and to calculations with Nulls. Assume, for instance, that we are trying to approximate a BMI (Body Mass Index) calculation; this can be done by dividing weight by the square of height (when using the metric system, with weight in kilograms and height in meters; a corrective factor is applied when using pounds and inches). We could write a query like (assuming metric measurements for simplicity)

```
SELECT *
FROM Patient
WHERE Weight > power(Height, 2) * 18.5;
```

with 18.5 considered the threshold of a normal BMI. However, if either `Weight` or `Height` or both have Nulls, this calculation will fail: the comparison with

[33] That is approximately 100 kgs.

> will return 'Unknown' because both sides of the > (expressions `Weight` and `power(Height, 2) * 18.5`) are Null, and even `Null = Null` returns 'Unknown.' This means that we could be missing patients with a condition but with unknown weight or height.

This behavior of Nulls extends to more complex predicates as follows:

- Conjunctions (AND) are true when both conditions used are true. Since a comparison with Null is not true, a conjunction involving a comparison with Nulls will fail, regardless of what other comparisons do. As an example, the query

```
SELECT *
FROM Patient
WHERE Weight > 220 and Height < 6;
```

  will fail on tuples where `Weight` is Null, regardless of what `Height` is. The whole comparison will also fail if both `Weight` and `Height` are Null or if only `Height` is Null.
- Disjunctions (OR) are true when at least one of the conditions used is true. When one of the conditions used involved Nulls, the whole comparison depends on what the other one returns. As an example, the query

```
SELECT *
FROM Patient
WHERE Weight > 220 or Height < 6;
```

  will return tuples where the `Height` attribute is indeed less than 6 feet, even if `Weight` is Null. But even an OR will fail if both `Height` and `Weight` are Nulls. This behavior leads to some strange situations; the query

```
SELECT *
FROM Patient
WHERE Weight > 220 or Weight >= 220;
```

  would seem to return every tuple of table `Patient`, since the condition is a tautology (whatever the weight of a patient is, it surely is greater than 220 or less than or equal to 220). However, tuples where the weight is Null will *not* be returned. This is important if we are *splitting* the dataset into classes or subsets using a predicate like this one.
- Negation (NOT) flips the result of a comparison: True becomes False, and False becomes true. However, negation leaves 'Unknown' unchanged, since we do not know what to flip. As a result, if the first example above were written as

```
SELECT *
FROM Patient
WHERE NOT (Weight <= 220);
```

  the query would still fail to retrieve tuples where the attribute `Weight` is null.

The fact that all predicates fail with Nulls leads to an interesting question: what to do if we want to actually find those tuples where we have Nulls? SQL provides

two special predicates, `IS NULL` and `IS NOT NULL`, that do exactly that. Either one of them takes an attribute name and returns True only for those tuples where the attribute has a Null marker. That is, the query

```
SELECT count(*)
FROM Patient
WHERE Weight IS NULL;
```

will tell us how many missing values there are in attribute `Weight`. Conversely,

```
SELECT count(*)
FROM Patient
WHERE Weight IS NOT NULL;
```

will tell us how many real (non-missing) values there are in attribute `Weight`. Using `IS NOT NULL` is akin to eliminating all rows of data where missing information is present.[34]

With respect to aggregates, we must be aware of the following:

- As a rule, *aggregates ignore Nulls*, with the exceptions below. As a result, if we use an aggregate in a query, like

  ```
  SELECT Sum(Weight)
  FROM Patient;
  ```

  the result will be the sum of all available weights, with nulls ignored. This seems commonsensical for `Sum`, but it is also the case for the other basic aggregates—`Count`, `Avg`, `Min`, and `Max`.
- An exception to the above is `Count(*)`. This aggregate essentially counts rows, without concern as to the contents of the rows. In particular, whether attribute values are present or are Null is irrelevant for this aggregate.
- If *all* values in an attribute are Null, aggregates return a *default* value: this is 0 (zero) for `count`, but it is Null for all other aggregates (`sum`, `avg`, `min`, and `max`).
- When operating in the context of a GROUP BY, the behavior is the same: within each group, aggregates ignore Nulls, with the exceptions noted—but keep on reading to see how GROUP BY itself handles Nulls.

With respect to grouping, Nulls behave differently than they do with comparisons. Recall that in comparisons, Nulls are not equal to each other. Thus, if we use an attribute A in a GROUP BY clause, and A contains one or more Nulls, it would seem that each one of them would generate its own, separate group. However, GROUP BY treats Nulls *as if they were the same*: all Nulls in an attribute generate a unique group. Suppose, for instance, that we are trying to generate a histogram of

[34] For those familiar with R, this is similar to the `na.rm = TRUE` optional argument in R, and to the `na.omit` predicate in data frames (but the SQL `IS NOT NULL` only operates on single attributes, not whole tuples).

weight values, and we write

```
SELECT Weight, count(*)
FROM Patient
GROUP BY Weight;
```

If attribute `Weight` contains *one or more* Nulls, one single separate group will be created. That group will contribute one tuple to the answer: `(Null, n)`, where `n` is the number of tuples with Nulls in `Weight`. If we write

```
SELECT Weight, avg(Height)
FROM Patient
GROUP BY Weight;
```

and attribute `Weight` contains *one or more* Nulls, we will again see a tuple `(Null, n)` in the result, where `n` is the average height of all tuples with Nulls in `Weight` (if `Height` itself has Nulls, they will be ignored when computing the averages).

Because the behavior of SQL is somewhat inconsistent, and because we want our data to be 'clean' for analysis, it is customary to try to get rid of nulls. The choices are to eliminate nulls or to substitute a value for them. In the first case, we can choose to delete from the dataset the rows where some null is present, or to eliminate from the dataset the attribute where nulls are present. Once nulls are detected and we know which attribute (or attributes) are at fault, both operations are quite easy in SQL—the first one calls for a command like

```
DELETE FROM (Table)
WHERE (attribute) IS NULL;
```

The second one calls for a command like

```
UPDATE TABLE
DROP ATTRIBUTE (attribute);
```

Note that, in both cases, we likely lose data with the nulls. If the proportion of nulls is very small, dropping rows (observations) may be the best way to proceed, but dropping the attribute could result in a substantial loss of data.

However, determining values to substitute for nulls is much more complex; it involves *imputing* or *predicting* the missing value from some other data. How to do this depends on the kind of missing value we are dealing with.

Several authors [4, 16] distinguish 3 types of missing values: let A be an attribute where some values are missing, and B be all other attributes in the table with A. Values missing in A can be

- *missing completely at random (MCR)* or *observed at random*: the values are missing independently of the underlying value of A, and of any values of B. In this case, the missing values can be filled in because the values of A that are present give us a good idea of the distribution of all values in A, so we can infer the underlying distribution and replace the missing values by the mean of the present values or by fitting a distribution to see which value of A is likely to be under-represented.

- *missing at random (MAR)*: the values are missing independently of the underlying value of A but may depend on other values B. This implies that that values of A are correlated with the value of (some of) the attributes in B. In this case, the missing values can be filled if we can find out what attribute(s) in B is (are) related to A, and the nature of the relationship. In such cases, we can apply a `predictor` method to the values of the relevant attribute(s) in B on rows where the value of A is missing. Two typical methods to use are *linear regression* and `k-nearest neighbors`, both explained in the next chapter.
- *non-ignorable missing values*: the values are missing independently of other values B but may depend on the underlying value of A. The problem with this case is that there may not be a way to replace the missing values meaningfully, since the values of other attributes do not help, and there is a connection with other values of A, which means that those other values of A that we have are not an impartial guide to the missing values, as in the first case.

An example from [4] makes the difference clear: assume a sensor for air temperature that runs out of battery at some point in time. The battery running dry is not related to the air temperature, or to any other weather variable, so we are in the first case. Here, we would be justified in replacing the missing values with the mean temperature, since the mean of the present values is (under the assumption that the missing values are random) an unbiased estimator of what the missing temperature is. Now assume a person is in charge of replacing batteries in the air sensor, but he or she does not do it when it rains. Then the battery is more likely to be dead when it is raining (one of the B attributes), although it has nothing to do with air temperature. This is the second case. Here, if we find out the connection between rain and temperature, we can try to infer an appropriate temperature from the value of the attribute 'rain.' We can, for instance, take the mean temperature of the raining days only or apply linear regression to the rain attribute in order to predict temperature (see next chapter). We would be justified in doing this because of the relationships between the attributes. Finally, assume that the sensor malfunctions at temperatures under zero degrees. Then other variables may be not related (i.e. it may rain or not rain at any temperature), but the sensor fails in a manner related to the missing value. In this case, imputing a value is very problematic: clearly, we cannot use the other temperatures, since they represent a whole range of possible temperatures, while the missing values come from a small subset of the range (below zero temperatures). If there is no connection with other attributes, they do not yield enough information to impute a value. Finally, note that even if we are aware of the situation (we know for a fact that there is a sensor malfunction at low temperatures) and we can infer that the missing values are low values, we still do not have enough information to determine a good value for each missing one.

How to tell the cases apart? One way is to try to determine whether there is some connection between A and some other attributes in the table. A first approach is to identify an attribute B that may be connected to A; using correlation, as explained in the previous section, is a first step (although, as we have mentioned, correlation only detects *linear* relationships). Using PMI may also be a good idea, as PMI is more

general. However, in both cases we do not obtain a direct answer to the question "Is B related to A (and hence a decent predictor)?" When B is numerical, a preliminary test can be to compare values of B when A is absent to values of B when A is present:

```
SELECT avg(B) as mean,
       CASE (WHEN A IS NULL THEN 'absent'
             ELSE 'present') AS Avalue
FROM Data
GROUP BY Avalue;
```

If the means are sufficiently different, we can deduce that there is a connection. If no connection is found with any attribute, we can rule out the second case and we are in either the first or third one. Unfortunately, it may not be possible to distinguish between these two from the data alone.

Note that much simpler methods are always available; for instance, in a numerical attribute, it may be tempting to get rid of nulls by using zeroes as this allows arithmetic to proceed without problems (in some systems, trying to divide by a null may cause an error). However, such approaches may introduce bias, in that it may disguise the true distribution of the attribute or its relationships to other attributes. Hence, this is not a desirable approach and should be avoided—and, if used, it should be properly documented in case it needs to be undone.

Finally, assume that have decided we are going to get rid of nulls by substituting them with some new value. To see what the attribute would look like, we can use

```
SELECT CASE(WHEN attribute IS NULL THEN (newvalue)
            ELSE attribute END) AS new-attribute
FROM Data;
```

However, there is a simpler way to achieve this using the `COALESCE` function:

```
SELECT COALESCE(attribute, newvalue) AS new-attribute
FROM Data;
```

The `COALESCE` function accepts a list of values and returns the first one that is not null. In this case, it will simply return the value of `attribute` when it does not contain a null, or `newvalue` otherwise, and so it accomplishes the same as the CASE construction in a more succinct manner.

3.3.3 Outlier Detection

An outlier is a value that is not 'normal,' in the sense that it is quite different from other values in a domain. It tends to be *extreme* (for numerical attributes) and *unusual* (for categorical attributes). While the notion is highly intuitive, there is no formal definition of what it means for a data point to be an *outlier*, since this depends on the context. Outliers may indicate a data quality problem (an error

in data acquisition/representation), or they may indicate a random fluctuation, or they may represent a truly infrequent, exceptional, or abnormal situation. In some applications, outliers are exactly what we are looking for; as an example, in credit card fraud, the fraudulent transactions are the outliers among a sea of legitimate transactions. In other applications, outliers are bad values that disturb the general nature of the data, and it is a good idea to get rid of them. Reusing an example from the previous section, assume we receive data from a temperature sensor. The average temperature so far is 75 degrees Fahrenheit, and the standard deviation is 5 degrees. Suddenly we receive a reading of 100. Is this the result of a heat wave, or a faulty sensor? There is usually no way to tell just from the data, and the difference is crucial for addressing the issue: a bad reading should be deleted and replaced by another value (or simply considered missing and ignored); an extreme, but legitimate value, should be kept—it is a crucial piece of information.

Outlier detection for single attributes depends on the type of attribute: for nominal attributes, we can calculate frequencies for each possible value; an outlier will be a value with a very low frequency. For numerical values, it is harder to decide what is an outlier without assuming an underlying distribution. A possible tactic is to try and find a distribution that fits the data well (see Sect. 3.2.3). When a distribution fits the data reasonably except for a few values, those values can be considered outliers. This is commonly done with the standard distribution: in this distribution, 95% of values are within 2 standard deviations from the mean, and over 99% of all values are within 3 standard deviations, so values beyond that can be classified as outliers. However, in other distributions, for instance exponential distributions (or any distribution which is very skewed or has a long, heavy tail), it is not be possible to tell outliers from regular values with this test. To make things more complicated, note that when outliers are present in a dataset, they influence the very statistics we are using (i.e. they can move the mean, and they may make the standard deviation much larger). Thus, it may be a good idea to use more robust statistics, like the trimmed mean or the Median Absolute Deviation (MAD) introduced earlier, to search for outliers.

Finally, in some cases it may be necessary to do a full analysis like *clustering* to decide if a value is an outlier.

Example: Finding Outliers in Names

Assume a dataset about people where one of the attributes is last (family) name. We suspect some names may be not entered correctly (typos, mispronunciations). One way to check for this is to focus on very rare values (since errors tend to be different, each typo may generate different results): the query

```
SELECT last-name, count(*) as freq
FROM Dataset
GROUP BY last-name
HAVING freq = 1;
```

will show names that appear only once. This, by itself, does not make the values 'bad,' but it makes them deserving of some further scrutiny.

Example: Finding Outliers in Numbers

Instead of simply finding numbers that are n standard deviations from the mean, we can use the trimmed mean and trimmed standard deviation introduced in the previous section to more reliably identify outliers.

```
WITH TrimmedStats AS
 (SELECT avg(A) as tmean, stdev(A) as tdev
  FROM Data, (SELECT max(A) as Amax FROM Data),
             (SELECT min(A) as Amin FROM Data)
  WHERE A < Amax and A > Amin)
SELECT A
FROM Data
WHERE A < tmean - (2*tdev) or A > tmean + (2*tdev);
```

Exercise 3.57 Write an SQL query to find outliers by using MAD. A common rule is to consider outliers values that are more than $1.5x$ from the MAD value, where x is the number of standard deviations we consider significant [8].

We will see later in Sect. 5.3 more direct ways to use MAD to find outliers.

3.3.4 Duplicate Detection and Removal

Identifying duplicates, in tabular data, is the task of examining two or more rows and determining whether they refer to the same observation, object, or entity in the real world. This task is also known as *deduplication, entity linkage, data linkage, data/record matching, entity resolution, co-reference*, and *merge/purge*.

The hardest part of duplicate identification is to determine when two rows refer to the same object or entity. In the basic case, we expect one or more attributes to be identical. That is, we determine a set of attributes that could serve as primary key (perhaps expanded with additional attributes for caution) and check to see if two rows have the same values for those attributes. Note that a primary key, if created artificially, is useless for this purpose. For instance, in a people database, assume that gave an 'id' to each person entered but now suspect that there may be duplicates (the same person may have been entered more than once). To check that, we may focus on (first and last) name, address, and date of birth, reasoning that two people with the same name, living in the same address, and having the same birthday are a strong indication of a duplicate. The 'id,' if system generated, is of no use to determine duplication. In this case, a simple query will do:

```
SELECT fname, lname, address, dob, count(*)
FROM Data
GROUP BY fname, lname, address, dob
HAVING count(*) > 1;
```

However, in many cases this simple approach will not be enough. Many times, repeated records do not have the exact same values for the attributes, but very close or similar ones. The reasons may go from typos to measure noise to having incorrect data in the database. A more sophisticated method that can deal with small mistakes is *fuzzy matching* (also called *approximate matching*). The idea is that if two values are very close, we can consider them the same for the purposes of duplicate detection. Note that what is considered 'very close' depends on the context: on a date of birth attribute, one day apart is a lot, but on a 'shipping date' attribute, one day apart may be an error in data entering. Likewise in numerical attributes: in a measurement of distance between two cities, 1100 miles (or kilometers) is almost the same as 1101 miles (or kilometers), while in a measurement of screw length, 11 inches and 12 inches can be quite different.

Implementing the intuitive notion of 'close' is usually done by using the idea of *distance*, which we study in Sect. 4.3.1. Intuitively, a distance between two values is a number that expresses how 'far apart' (how different) they are: a small distance means the values are similar; a large distance, that they are dissimilar. For the case of numerical values, the distance one typically uses is the absolute difference, sometimes 'normalized' by some value. This can be expressed quite simply in SQL.

Example: Approximate Distance

Recall the `Patients` table; the query

```
SELECT *
FROM Patient,
     (SELECT max(height) as maxh FROM Patient)
WHERE abs(D1.height - 6.0)/ maxh < alpha;
```

will return all individuals in the dataset whose height is 'close' to 6.0 (six feet), where the idea of 'close' is represented by being, as a percentage of the largest height, less than some cut point `alpha`.

For nominal values, the idea of fuzzy matching is to consider two strings as similar if they have more similarities than differences. Recall that SQL provides a `LIKE` operator that compares a string to a pattern, but the options of this operator are limited (see Sect. 3.1 for a description of LIKE).

For the case of comparing two string values, most systems provide some functions to implement fuzzy matching. Postgres, for instance, has two methods in its `fuzzystrmatch` module:[35]

- Function `difference(string1, string2)` returns a number that expresses the differences between the *Soundex* of two strings. The Soundex system is a

[35] Additional modules are activated in Postgres with the command `create extension` *`module-name`*.

method of matching similar-sounding names. Because of the way English is pronounced, the same (or very close) sound may end up being written in very different ways, depending on the context.[36] Soundex attempts to undo this; `difference` compares the results of Soundex and gives a number between 0 (no similarity) and 4 (total similarity). A query like

```
SELECT last-name
FROM Dataset
WHERE difference(last-name, 'Jones') > 3;
```

will return all last names that would sound very similar to 'Jones' in English. Obviously, this method works only for English language names. MySQL provides a similar method `string1 SOUNDS LIKE string2`, which is a short for `SOUNDEX(string1) = SOUNDEX(string2)`, with `SOUNDEX()` a function that generates the Soundex of its input.

- Function `levenshtein(string1, string2)` calculates the *Levenshtein distance* between two strings. This distance computes the total number of letter changes that would be necessary to transform one string into the other. In this case, a higher number means more difference: zero means the strings are identical, and the number can be as large as the size (number of characters) of the larger string. This distance applies to any language.

```
SELECT last-name
FROM Dataset
WHERE levenshtein(last-name, 'Jones') < 2;
```

Unfortunately, MySQL does not provide a similar function.

Of note, Levenshtein is only one of several possible distances for strings; more sophisticated matches against text are explored in Sect. 4.5.

When comparing two data records in the dataset, we first need to determine which attributes are likely to identify each data record; for each attribute, choose some distance that we are going to use to decide when two values are 'close enough' and, finally, pick a method to combine all distances to decide when the data records are indeed the same or not. The typical approach is to 'add' all the individual attribute distances. This can done in several ways. Let $A_1, \ldots, A_n$ be the attributes being compared, and $dist_i$ the distance applied to values of attribute A_i. Then, two rows or records r and s are compared using $dist_1$ on $r.A_1$ and $s.A_1$, ..., $dist_n$ on $r.A_n$ and $s.A_n$. We can combine all these distances to obtain a total distance between r and s, $dist(r, s)$, in several ways:

- Directly: if distances are comparable across domains (for instance, if they are all numerical and based on normalized values, or normalized themselves), with

$$dist(r, s) = \frac{\Sigma_{k=1}^{n} dist_k(r.A_k, s.A_k)}{\Sigma_{k=1}^{n} max(dist_k)}.$$

[36] As many students of English as a second language have discovered to their consternation.

Note that the denominator is the largest possible distance between two items and serves to normalize the result. Note also that this only works if all distances are on the same scale; otherwise, they need to be normalized individually:

$$dist(r,s) = \Sigma_{k=1}^{n} \frac{dist_k(r.A_k, s.A_k)}{max(dist_k)}.$$

Usually, the result obtained is compared to some threshold.

- By *Boolean combination*: assume that we have a threshold for each distance measure to determine, on an attribute-by-attribute basis, whether two values are similar enough or not. Let δ_k be 0 if the similarity of values in attribute A_k is not 'good enough,' and 1 if it is. Then the formula

$$dist(r,s) = \frac{\Sigma_{k=1}^{n} \delta_k(r.A_k, s.A_k)}{n}$$

is the proportion of attributes where there is agreement (out of all n of them).

Other approaches are possible. For instance, either one of the two approaches introduced can be modified with *weights*: if not all attributes have the same importance, we can use a sequence $w_1, \ldots, w_n$ of weights to indicate the weight w_l to give to $dist_l(r.A_l, s.A_l)$. For instance, the direct approach would result in

$$dist(r,s) = \frac{\Sigma_{k=1}^{n} w_k dist_k(r.A_k, s.A_k)}{\Sigma_{k=1}^{n} max(dist_k)}.$$

Example: Duplicate Removal in Patient Dataset

In the `Patient` table, we decide that two patients are one and the same if their last names and weights are similar enough:

```
WITH Similarity AS
(SELECT D1.Id, D2.Id,
     ((levenshtein(D1.lname, D2.lname) / (maxl * 1.0)
      +
      abs(D1.height, D2.height) / range) as sim
 FROM Dataset D1,
      Dataset D2,
   (SELECT max(length(lname)) as maxl FROM Dataset) AS T1,
   (SELECT max(height) - min(height) as range FROM Dataset)
   AS T2)
SELECT D1.Id, D2.Id
FROM Similarity
WHERE sim > 0.9;
```

Exercise 3.58 Give the SQL query to determine whether two people are the same or not in the Patient dataset by using Boolean combination of distances on `last-name` and `weight`.

Exercise 3.59 Modify your SQL query to give a weight of 0.75 to `last-name` and 0.25 to `weight` (it is common to use weights that add up to 1).

Finally, another task related to duplicate detection is the detection of inconsistencies. Two records in the dataset are said to be inconsistent if they refer to the same entity/event/observation and have contradictory information. Assume, for instance, that in the people dataset we discover that there are two records for the same person, but each one of them gives a different age. Since a person can only have one age, we know we have an inconsistency here. Clearly, not both data points (ages) can be correct at the same time; thus, we should correct one of the two. Unfortunately, in many cases (as in this example) it is impossible to decide, just from the data, which value is correct. In fact, dealing with inconsistencies depends largely on having domain knowledge, and it can become quite difficult. For instance, if two records about the same patient are inconsistent with respect to weight, we know that this is a value that can change over time, so we could assume that we simply ended up with information about the same patient taken at different time points. In a case like this, we would like to keep the more recent data (since this makes the data more likely to be accurate; recall Sect. 1.4). However, this does not tell us which value is the more recent one.

One way to detect inconsistencies is to enforce as many rules about the domain as possible; see Sect. 5.5 for some guidance on how to do this.

3.4 Data Pre-processing

Even after cleaning, data may still not be ready for analysis. This is usually due to the fact that values, even if correct, are not in the format that analysis tools (or algorithms) assume them to be. Hence, additional operations are needed to prepare the data for analysis. Typical operations used at this stage include aggregation, sampling, dimensionality reduction, feature creation, discretization, and binarization [16]. We explain each one briefly:

- Aggregation: this consists of combining two or more data records into one. This is especially common when data can be seen at several *levels of granularity* or detail. For instance, data on some event that is taken at regular time intervals of 1 min can be seen as 'by-the-hour' by aggregating all records within 60 min. Aggregated data has less detail but tends to have lower variability, and overall patterns may be more clear. This is usually implemented using the GROUP BY operator in SQL.
- Sampling: choosing a subset of the data ('sample') to work on. This makes computation less expensive, so it is common to sample when we want to carry out

complex analysis over large datasets, or when trying several tentative analyses of the same data. A sample chosen at random is expected to be representative of the whole dataset, hence it can help to establish the properties of the dataset. The tricky part is making sure that a good procedure (one where each data record has equal chance of being chosen) is followed. In databases, the task is sometimes accomplished by simply picking some rows of a table with the help of some pseudo-random number generator, as this provides a good enough approximation. The SQL standard provides for an operator, `TABLESAMPLE`, to accommodate this. In Postgres, the `TABLESAMPLE` keyword in used in the FROM clause as follows:

```
FROM table_name TABLESAMPLE sampling_method (percent)
```

This will result in the system sampling from the table `table_name` using the sampling method `sampling_method`, until a total of `percent` of all the rows in `table_name` are returned. This sampling precedes the application of any conditions in the WHERE clause. The standard PostgreSQL distribution includes two sampling methods, BERNOULLI and SYSTEM. A similar goal can be accomplished with the `random()` function:

```
SELECT * FROM Table_Name
ORDER BY random()
LIMIT n;
```

where `n` is a positive integer. The function `random()` generates, for each row, a pseudo-random floating point value v in the range $0 \leq v < 1.0$[37] that is then used by the limit to pick up whatever rows happen to have the first (lowest) n values of v. In fact, in Postgres,

```
SELECT * FROM Dataset TABLESAMPLE SYSTEM (.1);
```

is the same as

```
SELECT * FROM Dataset WHERE random() < 0.01;
```

MySQL has not implemented TABLESAMPLE, but the approach of using the pseudo-random function (called `RAND()` in MySQL) and LIMIT will work.

- Discretization: this is a transformation that changes numerical continuous values to ordinal ones. This is a very common transformation for classification tasks. As an example, in the people dataset the attribute 'height' may be transformed from a number to a categorical attribute with values 'low,' 'medium,' and 'high,' according to certain cut points. This can be accomplished in SQL, although it requires several steps: first, since each attribute has a type, if we are going to change 'height' to categories expressed by labels, we need to create a new, string-based attribute:

```
ALTER TABLE People ADD ATTRIBUTE height-category varchar(6);
```

[37] To obtain a random integer r in the range $i \leq r < j$, one can use the expression `FLOOR(i + RAND() * (j - i))`.

and then carry out the transformation:

```
UPDATE People
SET height-category =  CASE WHEN height > 5.8 THEN 'low'
                            WHEN height > 5.2 THEN 'medium'
                            ELSE 'low' END;
```

Note that we could represent the values 'low,' 'medium,' and 'high' with numbers (say, 0, 1, and 2). However, codes are always somewhat opaque. Note also that this procedure allows us to keep the old, original values of the attribute (since this operation loses information, that is generally a good idea).

- Binarization: this is a variation of discretization, in that it takes a numerical continuous attribute or a categorical one and transforms it into a binary attribute (or several binary attributes). Reusing the previous example, 'height' could be transformed into 3 binary attributes called 'low,' 'medium,' and 'high,' each one with possible values 'yes' or 'no' only (in some systems, a BINARY or BIT type exists that can represent 'yes/no' values with 1/0). This process is also called *creating dummy variables* in statistical contexts. This transformation is sometimes needed when an algorithm explicitly calls for such variables, and it can be performed similarly to discretization. We review it in depth in the next subsection.

Sometimes, any type of attribute may benefit from some additional treatment. For instance, in highly skewed distributions we may choose to keep a specified number of the most frequent values and create a single, new value to represent the long tail of remaining values. We have seen how to do similar manipulations by combining GROUP BY and CASE to create bins for both categorical and numerical attributes.

Exercise 3.60 Assume table `Income(PersonID, yearly-income)` and assume the table is quite large and attribute `yearly-income` has a long tail of small values. Create a new table called `NewIncome` with the same schema and data as `Income` except that all tuples in the long tail are gone (pick a value `cutpoint` where the long tail starts) and a new tuple `(id, v)` is added instead, where `id` is a new, made-up id and `v` is the average of all `yearly-income` values with size in the long tail.

We have left *dimensionality reduction* and *feature creation* for last, as they are complex operations, typically not expressed in SQL (although feature creation, in simple cases, can be done without difficulty). The best way to explain these is to think of a prediction task (classification or correlation) and consider all the attributes of a dataset as divided between the predictors and the outcome (usually, only one attribute at a time is considered for outcome). One question we face is whether the predictors, considered together, contain enough information to determine the outcome. There are three possible scenarios:

- Just enough predictors: all the predictors together can determine the outcome;
- Too many predictors: some of the predictors are actually redundant or unnecessary and do not help in predicting the outcome.

- Not enough predictors: even all predictors combined together do not have enough information to predict the outcome.

In the second case, we may want to identify and get rid of the useless attributes. The reason is that, for many algorithms, more attributes mean more parameters to consider, and this implies more work to do; in the worst case scenario, these useless attributes can confuse the algorithm. There is a substantial body of research on this issue; most of it uses sophisticated techniques that are difficult or impossible to implement in SQL, like PCA (Principal Component Analysis). We will not cover them in this book.

In the third case, we may want to create new, additional attributes by combining existing ones, in the hope that the new attributes will allow us to predict the outcome by making explicit some information that was implicit in the attributes. A typical example of this is a set of data points in 2 dimensions (given by x and y) that need to be classified into one of two classes (binary classification). The data resists our attempts, so we create a new dataset with three attributes $(x^2, y^2, \sqrt{2}xy)$, which allows us to apply a simple classifier and be successful.[38] A simple transformation like this is clearly doable in SQL:

```
CREATE TABLE NewDataset AS
SELECT x*x, y*y, sqrt(2)*x*y
FROM Dataset;
```

The tricky part here, of course, is to come up with a suitable transformation. This is also an advanced topic that we do not cover.

3.4.1 Restructuring Data

Sometimes data needs to be structured for further analysis. This is especially the case when the dataset comes from several files that go into different tables or when data is not in a *tidy* format (see Sect. 2.1.4), since most Data Mining and Machine Learning tools assume that all the data is presented as a single tabular structure and that this structure is tidy.

There are three types of situations where we may want to combine data from different tables into a single one or restructure a single table. The first one involves tables with complex structures (objects with multi-valued attributes, or different kinds of related objects) that we examined in Sect. 2.2. Such tables, when connected by primary key–foreign key connections, are put together with joins.[39]

[38] For those who have seen this before: with some datasets that are not linearly separable, the transformation yields a new dataset where the points can be separated with linear regression.

[39] Sometimes, especially when data does not come from a database but from spreadsheets, files, and similar sources, the primary keys and foreign keys may not be explicit. However, in most some cases some sort of identifier attribute is used to glue the data together. We can always use joins on

The second type involves similar data that is distributed into kinds. Assume, for instance, that we have data on university rankings, but with different rankings for different specialties: we have a table

`PsychologyRank(school-name, state, type, ranking-position)`

for schools ranked according to their Psychology programs; another table

`EconomyRank(school-name, state, type, ranking-position)`

where the schools are ranked according to their Economy programs; another table

`HistoryRank(school-name, state, type, ranking-position)`

and so on. As another example, assume that we have real estate sales information from New York, but we have different datasets for each one of the 5 boroughs that are part of the city: Manhattan, Brooklyn, Queens, The Bronx, and Staten Island. On each dataset, we have a similar schema: `(address, type, date-sold, amount-sold)`. What characterizes these *distributed* datasets is that all tables have the same or very similar schema. The way to deal with such datasets is by combining them with *set operations*, explained in Sect. 5.4.

The third type of situation is a bit more complicated; it involves data that is not *tidy* (see Sect. 2.1.4). Such data has to be changed to adjust to the format that is most appropriate for analysis. Many times, this involves *pivoting*, that is, transforming a schema that includes a series of values of an attribute as distinct columns into one where such values are part of the data—or vice versa: we may need to pivot rows to columns or columns to rows.

In essence, this involves tables where the schema contains $\texttt{name}_1, \ldots, \texttt{name}_n$, with each of those being the value of an (implicit) attribute `A`. The table may contain entries like

Attribute	**name**$_1$	...	**name**$_n$
a_1	value$_{11}$	...	value$_{1n}$
a_2	value$_{21}$	...	value$_{2n}$
...			
a_m	value$_{m1}$	...	value$_{mn}$

that we would like to be transformed into

Attribute	**A**	**Value**
a_1	name$_1$	value$_{11}$
a_1	name$_2$	value$_{12}$
...		
a_1	name$_n$	value$_{1n}$
...		
a_m	name$_1$	value$_{m1}$
...		
a_m	name$_n$	value$_{mn}$

such attributes, even if they are not declared as primary keys or foreign keys. The usual difficulty in such cases is *identifying* the foreign keys.

These transformations can be written in SQL, albeit in a cumbersome manner, best explained through an example.

Example: Pivoting in SQL

Assume a table `Earthquakes(magnitude, Y2000, Y2001, Y2002, Y2003)`, where each row has a magnitude n and the number of earthquakes of magnitude n that happened in year 2000, the number of earthquakes of magnitude n that happened in year 2001, and so on. We would like to transform this into a table with schema `(magnitude, year, number-earthquakes)`, which is more amenable for analysis. The following query accomplishes this:

```
CREATE TABLE earthquatesTidy AS
SELECT magnitude, year,
       sum(CASE WHEN year = 2000 THEN "Y2000"
                WHEN year = 2001 THEN "Y2001"
                WHEN year = 2002 THEN "Y2002"
                WHEN year = 2003 THEN "Y2003"
                ELSE 0 END) as numberquakes
FROM earthquakes, (values(2000), (2001), (2002), (2003))
                  as temp(year)
GROUP BY magnitude, year;
```

This strange-looking query does the following:

- It generates a table `TEMP(year)` with the values being $\texttt{name}_1, \ldots, \texttt{name}_n$ (in this example, 2000, 2001, 2002, and 2003). That is, this table contains all the values of the implicit attribute `year`.
- It takes the cross-product of the dataset (`earthquakes`, in this example) and this newly generated table, thus combining each row of data with all possible values of the implicit attribute. This makes it possible to generate, for each original row of data, as many rows as values there are in the implicit attribute.
- It uses a CASE to pick, for each case of the implicit attribute, the appropriate value in the data.

Example: Pivoting

To understand this transformation well, it is a good idea to see it in action with a toy example. Assume table

Earthquakes			
Magnitude	**Y2000**	**Y2001**	**Y2002**
1.2	3	4	5
1.5	5	6	7
2	4	8	9

Table `temp` is

Year
2000
2001
2002

Their Cartesian product is

Magnitude	Y2000	Y2001	Y2002	Year
1.2	3	4	5	2000
1.5	5	6	7	2000
2	4	8	9	2000
1.2	3	4	5	2001
1.5	5	6	7	2001
2	4	8	9	2001
1.2	3	4	5	2002
1.5	5	6	7	2002
2	4	8	9	2002

The CASE runs through this; on each row, when the value of attribute `year` is 2000, it picks the value from column "Y2000" (in other columns, it picks a 0, which has the effect of skipping them), and the same for each other value of attribute `year`. Finally, values are summed across the magnitude and year.

Exercise 3.61 Recreate the example above in Postgres. That is, create a table `Earthquake(magnitude, Y2000, Y2001, Y2002, Y2003)` and insert the data shown in it. Then run the query above to see the resulting table. Tie each value on the result to the data it came from in the original table.

Exercise 3.62 Repeat the previous exercise in MySQL. Note that the syntax is going to be a bit different.

If, for some reason, we actually want to reverse this process and go from the table `EarthquakeTidy(magnitude, year, number-earthquakes)` to the original table, this can be achieved with the following query:

```
SELECT magnitude,
       sum(CASE WHEN year = 2000 THEN numberquakes
                                 ELSE 0 END) as "Y2000",
       sum(CASE WHEN year = 2001 THEN numberquakes
                                 ELSE 0 END) as "Y2001",
       sum(CASE WHEN year = 2002 THEN numberquakes
                                 ELSE 0 END) as "Y2002"
FROM earthquakesTidy
GROUP BY magnitude;
```

Note that the aggregate SUM is not really adding anything: the table `EarthquakeTidy(magnitude, year, number-earthquakes)` has only one row per magnitude–year combination. Since we are grouping by magnitude and picking the years apart in the CASE statement, each SUM will only use one value. But using the aggregate allows us to use a GROUP BY clause to generate a single tuple per magnitude.

This approach is clearly burdensome, and it can become unpractical when the number of values of the implicit attribute is high. In some systems, there is a built-in pivot capability, usually called the `crosstab` operator, that can achieve similar results. For example, Postgres has this built-in capability.

Crosstab in Postgres

The query

```
SELECT * FROM crosstab(
          'SELECT magnitude, year, number-earhquakes
           FROM earthquakes order by 1',
          'SELECT distinct year
           FROM earthquakes order by 1');
```

would allows us to go from a table `Earthquakes(magnitude, year, number-earthquakes)` to a table with schema `(magnitude, Y2000, Y2001,...)` in Postgres. Note that `crosstab` takes 2 strings as arguments, each string being an SQL query (a SELECT statement): the first string/query defines the original data table, and the second one indicates which attribute is going to be 'spread' into the schema.

Exercise 3.63 Assume a table `GRADES(student-name, exam, score)`, where attribute `exam` can be one of 'midterm' and 'final.' Produce a table with schema `(student-name, midterm, final)` that is the cross-tab of the original one.

A closely related issue is the creation of a *dummy variable* (also known as an indicator variable, design variable, one-hot encoding, Boolean indicator, binary variable, or qualitative variable). Dummy variables are "proxy" variables, numerical values made up to represent categorical or ordinal values for analysis approaches that do not handle nominal values and require numbers (for instance, regression models). Given a categorical attribute with n different values $v_1, \ldots, v_n$, n dummy variables $A_1, \ldots, A_n$ are created. When the categorical attribute has value v_i, we set $A_i = 1$ and $A_j = 0$ for $j \neq i$. Note that it is not strictly necessary to have n different attributes; we could do with one less, since the nth case can be encoded by setting all other $n - 1$ dummies to zero. For instance, a binary categorical variable (like 'male/female,' 'indoor/outdoor') can be represented by a single dummy variable with 0 for one category (say 'male' or 'indoor') and 1 for the other ('female' or

'outdoor'). However, it is customary to use n attributes (in this example, an attribute `Male` or `Indoor` and another attribute `Female` or `Outdoor`), since many data mining and machine learning algorithms expect this format.

This a similar problem to pivoting, since we want to distribute the n values of a (categorical) attribute into a schema with n attributes; the only difference is that the new values for those new n attributes are 0 or 1. This can be achieved with the same approach as above: using the SUM aggregate on the categorical variable, but passing 1 and 0 as values (since the SUM is not really aggregating anything, this will the final values too).

Example: Creating Dummy Variables

Assume a data table like

Name	Category
Jones	A
Jones	C
Smith	B
Lewis	B
Lewis	C

This is transformed by query

```
SELECT name,
       sum(CASE WHEN (category = "A", 1, 0)) AS A,
       sum(CASE WHEN (category = "B", 1, 0)) AS B,
       sum(CASE WHEN (category = "C", 1, 0)) AS C
FROM Data
GROUP BY name;
```

into the table

Name	A	B	C
Jones	1	0	1
Smith	0	1	0
Lewis	0	1	1

Again, note that since each name appears only once, there is no real SUM. The aggregate is used so we can group by the name, thereby making sure we create a unique row for each name.

Exercise 3.64 Assume as before a table `GRADES(student-name, exam, score)`, where attribute `exam` can be one of 'midterm' and 'final.' Produce a table with schema `(student-name, midterm, final)`, where attribute 'midterm' is 1 if the student had a score > 60 and 0 otherwise, and similarly for attribute 'final.'

3.5 Metadata and Implementing Workflows

The process of exploring, cleaning, and preparing data is essentially *iterative*. After some EDA, we may discover attributes that need cleaning or other processing; after this is done, we may understand the data better and carry out some further exploration and processing. As we go on, we sometimes modify the original, raw data, generating new datasets. We need to manage all these datasets and keep track of how they were created and why. The step-by-step transformation of the data is sometimes called a *workflow*, as it corresponds to a sequence of actions applied to the data or to the results of earlier actions.

Most actions can be done in two modes: destructive and non-destructive. In destructive mode, the action changes some data, so that the old version of it is lost, and a new version put in its place. In non-destructive mode, data is transformed by generating a new version, but keeping the old data. Each action in the cleaning and pre-processing state can be implemented in a destructive and a non-destructive way. Also, all actions can be classified as *reversible* and *non-reversible*. A reversible action is one that can be undone; for instance, concatenating two strings using some character `c` as separator can be undone (if we register the fact that `c` was used as the separator, and this character does not exist in the original strings). A non-reversible action, in contrast, cannot be undone: for instance, trimming whitespaces from a string cannot be undone unless we note, for each string, how many whitespaces were deleted, and where they appeared—without this, we cannot recreate the original string from the modified one. It is clear that reversible actions can be done in a destructive mode, since we can always get back the old data. However, non-reversible actions can also be reversed if implemented in a non-destructive mode: for instance, if when we trimmed the whitespaces from a string by simply generating a new string and keeping the old one, we can choose to ignore that new string and revert to using the original one.

Implementing any actions in a non-destructive manner results in more data being added to the dataset, requiring additional storage. In choosing whether an action is done in destructive or non-destructive ways, one must balance the ability to undo actions if necessary with the extra storage requirements.

In databases, a destructive action is implemented by an UPDATE statement. We modify an attribute by using a command of the form:

```
UPDATE TABLE SET ATTRIBUTE = FUNCTION(ATTRIBUTE);
```

This is called an *in-place* update because the space used by the attribute is reused for the new value; the old value is gone.

In contrast, a non-destructive action is implemented by adding the result of a change as a new attribute in the table, or creating a brand new table. The former

involves a change of schema (we are adding a new attribute), so it must be done in two steps in SQL:

1. First, the schema is modified with an `ALTER TABLE` command:

   ```
   ALTER TABLE ADD ATTRIBUTE new-attribute-name datatype;
   ```

   Note that a new attribute is added to each existing row in the table; since there is no value for it, a NULL is put by default in this new attribute.
2. Next, the change is made but the result is deposited in the new attribute:

   ```
   UPDATE TABLE SET new-attribute-name = FUNCTION(ATTRIBUTE);
   ```

   The latter (creating a brand new table) is easy:

```
CREATE TABLE NEW-TABLE AS
SELECT (all attributes of existing table except
        the one being changed),
       FUNCTION(ATTRIBUTE) AS new-attribute-name
FROM TABLE;
```

Note that this creates a copy of the whole dataset, so it should not be every time we make a change to the data, as it would result in a proliferation of tables and a duplication of all unchanged data. However, it may be a good idea to do this at certain points (after very important changes, or after a raft of related changes).

All relational systems have an additional mechanism to evolve data: a *view* is a virtual table, one that is defined through a query. That is, a view is created as follows:

```
CREATE VIEW name AS
SELECT attributes
FROM TABLE
WHERE conditions;
```

The view's schema is defined, implicitly, by the query used. The view inherits the attribute names from the table used. If new names are desired, they can be specified with

```
CREATE VIEW name(first-new-name, second-new-name,..)  AS
SELECT attributes
FROM TABLE
WHERE conditions;
```

or as

```
CREATE VIEW name AS
SELECT first-attribute as first-new-name,
       second-attribute as second-new-name,...
FROM TABLE
WHERE conditions;
```

It is also possible to apply functions to the attributes, hence creating a view that is the result of doing some cleaning/transformation to existing data.

The view does not actually have data of its own; it depends on its definition. If we want to see what is in a view, the system simply runs the query that was used in the view definition. As a result, if the tables used in the query that defined the view change, the view itself changes accordingly. Note that, since the view is not actually stored, no extra space is needed for views. On top of that, we can define views using already-defined views, so that this approach can be used for complex workflows. Thus, using views instead of tables should be considered when manipulating a dataset (see next subsection for an additional advantage of views).

3.5.1 Metadata

Some datasets may come with metadata telling us some of these characteristics; typically, the type of dataset is known in advance, and sometimes the schema is too. In this case, EDA can be used to confirm that what we have in the dataset does indeed correspond to the metadata description. When the dataset does not come with any metadata, EDA is used to create such a description. *We should always have as complete as possible an idea of what the dataset is about* before we start any serious analysis. Most analysis will require us to make some assumptions; the closer these assumptions are to the true nature of the data, the better our analysis will be (conversely, the further away our assumptions are from the data, the higher the risk of generating false or misleading results).

Whatever way we implement our workflow, it is important to keep track of what is being done. This is what metadata is for. Ideally, after acquiring the data we should have some descriptive metadata for each dataset, which we should store. If no metadata is available, we should generate our own after EDA. Next, as we go on processing the data, we should keep track of each action taken. When an existing attribute is modified, or a new one created, or a new dataset is generated, one should register the action that was taken. If this is done, it should be possible to examine each dataset in the database and have a list of all the changes that led to it; this is what we called *provenance* or *lineage* in Sect. 1.4. As explained there, keeping track of changes is fundamental for repeatability of experiments and, in general, will help us understand what is being done to the data, making the process transparent and enabling us to revisit our decisions and pursue alternatives if needed. It is also extremely helpful if the data is to be shared or published.

One very nice thing about relational databases is that *metadata can be stored in tables, just like data*. In fact, the system does this. Whenever we create a table, the system registers this fact in some special tables, sometimes called *system tables* or *catalog*. For each table, the system keeps track of its name and its schema (attribute name, type, etc.). Also, information about keys (primary, unique, and foreign) is kept, as well as information about users and their privileges (i.e. which data in the database they have access to).

Example: Metadata in Postgres

In the Postgres command line, the following provide information about the database:

- `\l` shows all schemas/databases in the server.
- `\d` shows all tables or views in the current database. `\dt` shows tables only, `\dv` shows views only. `\dp` shows all tables and views and, for each one, who has access to them.
- `\d name`: for each table or view matching `name`, it shows all columns, types, and other related information.
- `\dg` (`\du`) shows all users (called 'roles' in Postgres) of the current database.

Example: Metadata in MySQL

In the MySQL command line, the following provide information about the database:

- `SHOW DATABASES` (also `SHOW SCHEMAS`) displays the names of all databases in the server.
- `SHOW TABLES (FROM|IN) db-name` shows all tables in the database `db-name`. Also, `SHOW TABLES (FROM|IN) db-name LIKE tablename` shows all data about any table with name `tablename`.
- `SHOW CREATE TABLE tablename` displays the `CREATE TABLE` statement that generated the table `tablename`. The statement includes all schema information, as we saw in Sect. 2.4. Alternatively, command `SHOW COLUMNS FROM tablename` shows column information for all columns in table `tablename`.
- `SHOW CREATE VIEW viewname` displays the `CREATE VIEW` statement that generated the view `viewname`. As we just saw, this includes the SQL query that creates the view.

This basic metadata gives only basic information. For a given dataset, we can create a table that describes additional metadata, as given in Sect. 1.4. Thus, we could have a table with schema:

`(attribute-name, representation, domain, provenance, accuracy, completeness, consistency, currency, precision, certainty)`

and each row describing one attribute of our table. The attribute `representation` should coincide with the data type used to store the data. The attribute `domain` should describe the underlying domain in terms that a person can understand; it is a good idea to make this attribute a long string or text type, so we can describe the domain in plain language. Information about good and bad values, 'typical' values and so on can also be included here. All other attributes are about the quality

of the data (again, see Sect. 1.4). Note that not all attributes will require all these features; `precision`, for instance, is associated with numerical measurements. We may expect the table, then, to have some nulls on it, for reasons similar to those seen in the *Chicago employees* dataset.

Each time an action is carried out, the effect should be registered. Thus, a second table for each dataset should be added to describe actions. For each action, we want to register

- the attribute or attributes affected by the action;
- the change made (and any functions applied);
- the values of any parameters used by the functions;
- when the action was carried out;
- *why* it was carried out;
- who carried the action out.

The first three records can be captured simply by copying the SQL command used; this can be easily stored in a long string attribute. The fourth one can be very useful in case we need to determine in which order actions were carried out and/or undo some of them; using the timestamp of changes helps see what was applied to what. The fifth one is both very important and rarely registered. In fact, it may be the most important and ignored part of all metadata. Decisions taken during EDA are many times based on a partial, sometimes faulty understanding of the data; some assumptions are usually made. If we later on discover that our assumptions were incorrect, we may want to undo those actions that were guided by newly revised assumptions (and, sometimes, we may have to undo any further changes, which is where the timestamp helps). Finally, the last record may be important in scenarios where there is a need to create `audits` because access to data is restricted or because there are other reasons (legal, regulatory) to keep this information.

One nice thing about using views is that the system does keep track of their provenance automatically. When a view is created, the system stores its definition in a catalog table that is devoted exclusively to views. Hence, with views it is not necessary to register how they were created. Still, we may want to capture *why* the change was done: the system does not do that and, as we argued above, it is important information.