Data-Centric Systems and Applications

Antonio Badia

SQL for Data Science

Data Cleaning, Wrangling and Analytics with Relational Databases

Springer

Data-Centric Systems and Applications

Intelligent data management is the backbone of all information processing and has hence been one of the core topics in computer science from its very start. This series is intended to offer an international platform for the timely publication of all topics relevant to the development of data-centric systems and applications. All books show a strong practical or application relevance as well as a thorough scientific basis. They are therefore of particular interest to both researchers and professionals wishing to acquire detailed knowledge about concepts of which they need to make intelligent use when designing advanced solutions for their own problems.

Special emphasis is laid upon:

- Scientifically solid and detailed explanations of practically relevant concepts and techniques
  (what does it do)
- Detailed explanations of the practical relevance and importance of concepts and techniques
  (why do we need it)
- Detailed explanation of gaps between theory and practice
  (why it does not work)

According to this focus of the series, submissions of advanced textbooks or books for advanced professional use are encouraged; these should preferably be authored books or monographs, but coherently edited, multi-author books are also envisaged (e.g. for emerging topics). On the other hand, overly technical topics (like physical data access, data compression etc.), latest research results that still need validation through the research community, or mostly product-related information for practitioners ("how to use Oracle 9i efficiently") are not encouraged.

More information about this series at http://www.springer.com/series/5258

Antonio Badia

SQL for Data Science

Data Cleaning, Wrangling and Analytics with Relational Databases

Springer

Antonio Badia
Computer Engineering & Computer Science
University of Louisville
Louisville, KY, USA

ISSN 2197-9723 ISSN 2197-974X (electronic)
Data-Centric Systems and Applications
ISBN 978-3-030-57591-5 ISBN 978-3-030-57592-2 (eBook)
https://doi.org/10.1007/978-3-030-57592-2

This Springer imprint is published by the registered company Springer Nature Switzerland AG.
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

Data Science (or Data Analytics, or whatever one prefers to call it) is a 'hot' topic right now. There is an explosion of courses on the subject, especially online: many universities and several for-profit and non-profit organizations (Coursera, edX, Udacity, Udemy, DataCamp, and many others) offer on-campus and online courses, certification, and degrees. The coverage of these offerings is quite diverse, reflecting the fact that Data Science is still a young and evolving field. However, many courses seem to coalesce around a few topics (Machine Learning, mostly) and tools (R, Python, and SQL, mostly). What few of these courses offer is a textbook.

There are already many books on databases and SQL, but almost all of them focus on the traditional curriculum for Computer Science majors or Information Systems majors (there are a few exceptions, like [11] and [17]). In contrast, the present book explains SQL *within the context* of Data Science and is more in line with what is being taught in these new courses. This book introduces the different parts of SQL as they are needed for the tasks usually carried out during data analysis. Using the framework of the *data life cycle*, it focuses on the steps that are given the short shift in traditional textbooks, like data loading, cleaning, and pre-processing.

This book is for anyone interested in Data Science and/or databases. It should prove useful to anyone taking any of the abovementioned courses, online or on-campus, as well as to students working on their own. It assumes very little from the reader; it just demands a bit of 'computer fluency,' but no background on databases or data analysis. In general, all concepts are introduced intuitively and with a minimum of specialized jargon. It contains an appendix (Appendix A) meant to help students without prior experience with databases, with instructions on how to download and install the two open-source database systems (MySQL and Postgres) that we use for examples throughout the book. All readers of the book are encouraged to install both systems and follow the book along with a computer in order to practice, do the exercises, and play around—simply reading the book alone is going to be much less useful than *using* it.

The book is organized as follows: Chapter 1 describes the *Data Life Cycle*, the sequence of stages, from data acquisition and ingestion until archiving, that data goes through as it is prepped for analysis and then actually analyzed, together with

the different activities that take place at each stage. It also explains the different ways that datasets can be organized, and the different types of data one may have to deal with. Many students have an intuitive understanding of the concepts in this chapter, but it is useful to have it all together in one place and to give a name to each concept for later reference. Chapter 2 gets into databases proper, explaining how relational databases organize data. The chapter also explains how data in tables *should* look like (what Hadley Wickham has called *tidy* data [19]), a point which is not traditionally emphasized and can lead to severe problems down the road. Non-traditional data, like XML and text, are also covered. Chapter 3 introduces SQL *queries*, the SQL commands that allow us to ask questions about the data. Unlike traditional textbooks, queries and their parts are described around typical data analysis tasks (data exploration, cleaning, and transformation). These tasks are vital for a proper examination of the data but are frequently overlooked in Data Mining and Machine Learning textbooks. Chapter 4 introduces some basic techniques for Data Analysis. Even though this is not the focus of the book, the chapter shows that SQL can be used for some simple analyses without too much complication.

After this part, which constitutes the core of the book, Chap. 5 introduces additional SQL constructs that come in handy in a variety of situations. This chapter completes the coverage of SQL queries so that readers get an overview of all the main aspects of this important topic. Chapter 6 briefly explains how to use SQL from within R and from within Python programs. This chapter is not an introduction to R (or to Python) and, unlike other chapters in the book, does assume that the reader is already familiar with at least the basics of R and Python. It focuses on how these languages can interact with a database, and how what has been learned about SQL can be leveraged to make life easier when using R or Python.

The book also contains another appendix (besides the one already mentioned), which introduces some basic approaches for handling very large datasets. The purpose of this appendix is to demystify the ideas behind the vague label *Big Data* and give the readers basic guidance on how to use their newly acquired skills in this world.

As in many textbooks, none of what this one contains is new. This book covers the same (or very similar) content to what can be found in many sources, especially online. What this book does is to put it all together under one roof and to give it some order and structure. In many blogs and sites, the material is presented as an answer to a particular question (how do you…?), which may be useful to someone with a specific need but gives the impression that learning SQL is about a bag of tricks. Here, the material is logically organized using the idea of the data life cycle so that all the concepts introduced can be understood as parts of a coherent whole.

Data Science itself is a relatively new and still changing field, but it has deep roots, as it uses approaches and techniques from well-established fields, mostly math (statistics, linear algebra, and others) and computer science (databases, machine learning, and others). As a result, the same concept is sometimes given different names by different authors in different textbooks. Whenever I am aware of this, I

have given a list of known names so that readers with different backgrounds can relate what is in here with what they already know.

The goal of the book is to introduce some basic concepts to a wide variety of readers and provide them a good foundation on which they can build. After going through this book, readers should be able to profitably learn more about Data Mining, Machine Learning, and database management from more advanced textbooks and courses. It is my hope that most of them feel that they have been given a springboard from which they are in a good position to dive deeper into the fascinating world of data analysis.

Louisville, KY, USA Antonio Badia
July 2020

Contents

Chapter 1
The Data Life Cycle

It is sometimes said that "data is the new oil." This is true in several ways: in particular, data, like oil, needs to be processed before it is useful. Crude oil undergoes a complex refining procedure as the substance that comes out of wells is transformed into several products, mostly fuels (but also many other useful by-products, from asphalt to wax). A complex infrastructure, from pipelines to refineries, supports this process. In a similar way, raw data must be thoroughly treated before it can be used for anything. Unfortunately, there is not a big and sophisticated infrastructure to support data processing. There are many tools that support some of the steps in the process, but it is still up to every practitioner to learn them and combine them appropriately.

In this chapter, we introduce the stages through which data passes as it is refined, analyzed, and finally disposed of. The collection of stages is usually called the *data life cycle*, inspired by the idea that data is 'born' when it is captured or generated and goes through several stages until it reaches 'maturity' (is ready for analysis) and finally an end-of-life, at which point it is deleted or archived. Data analysis, which is the focus of Data Mining and Machine Learning books and courses, is but one step in this process. The other steps are equally important and often neglected.

The main purpose of this chapter is to introduce a framework that will help organize the contents of the rest of the book. As part of this, it introduces some basic concepts and terms that are used in the following chapters. In particular, it provides a classification of the most common types of *datasets* and *data domains* that will be useful for later work. We will come back to these topics throughout the book, so the reader is well served to start here, even though SQL itself does not appear until the next chapter. Also, for readers who are new to data analysis, this chapter provides a basic outline of the field.

A. Badia, *SQL for Data Science*, Data-Centric Systems and Applications,
https://doi.org/10.1007/978-3-030-57592-2_1

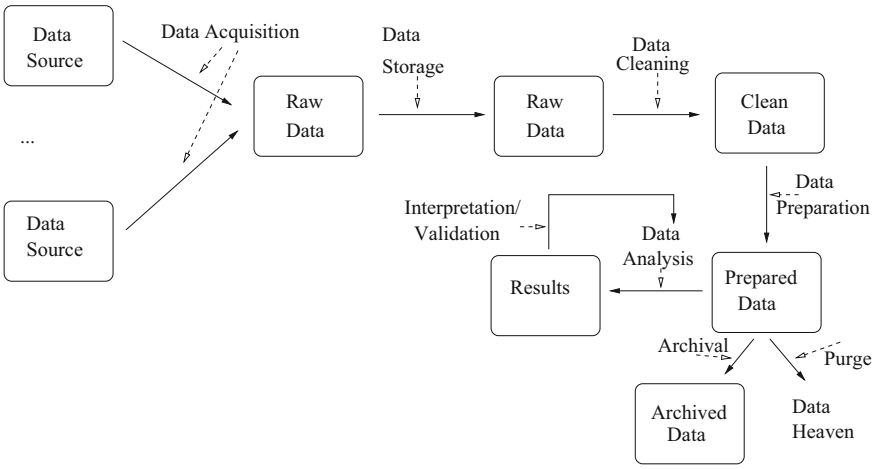


Fig. 1.1 The data life cycle

1.1 Stages and Operations in the Data Life Cycle

The term *data life cycle* refers both to the transformations applied to data and to the states that data goes through as a result of these transformations. While there is not, unfortunately, general agreement on the exact details of what is involved at each transformation and state, or how to refer to them, there is a wide consensus on the basic outlines. The states of the cycle can be summarized as follows:

$$\text{Raw data} \rightarrow \text{cleaned data} \rightarrow \text{prepared data} \rightarrow \text{data + results} \rightarrow \text{archived data}$$

The arrows here indicate precedence; that is, raw data comes first, and cleaned data is extracted from it, and so on. The activities are usually described as follows:[1]

$$\text{Data Acquisition/capture} \rightarrow \text{data storage} \rightarrow \text{data cleaning/wrangling/enrichment} \rightarrow \text{data analysis} \rightarrow \text{data archival/preservation}$$

Again, the arrows indicate precedence; data acquisition/capture happens first, followed by data storage, and so on.

The diagram in Fig. 1.1 shows the activities and stages together. We now describe each part in more detail.

The first activity in data analytics is to acquire, collect, or gather data. This happens in different ways. Sometimes existing sources of data are known and

[1] Some activities are given different names in different contexts.

accessible, sometimes a prior step that uncovers sources of relevant data[2] must be carried out. What we obtain as the result of this step is called *raw data*.

It is very important to understand that "raw" refers to the fact that this is the data before any processing has been applied to it, but does not indicate that this data is "neutral" or "unfiltered." In statistics, the domain of study is called the *population*, and the data collected about the domain is called the *sample*. It is understood that the sample is always a subset of the whole population and may vary in size from a very small part to a substantial one. However, the sample is never the population, and the fact that sometimes we have a large amount of data should not fool us into believing otherwise. For analysis of the sample to provide information about the population, the sample must be *representative* of the population. For this to happen, the sample must be chosen at random from elements of the population which are equally likely to be selected. It is very typical in data science that the data is collected in an *opportunistic* manner, i.e. data is collected because it is (easily) available. Furthermore, in science data usually comes from experiments, i.e. a setting where certain features are controlled, while a lot of data currently collected is *observational*, i.e. derived from uncontrolled settings. There are always some decisions as to what/when/how to collect data. Thus, raw data should not be considered as an absolute source of truth, but carefully analyzed.

When data comes in, we can have two different situations. Sometimes datasets come with a description of the data they contain; this description is called *metadata* (metadata is described in some detail in Sect. 1.4). Sometimes the dataset comes without any indication of what the data is about, or a very poor one. In either situation, the first step to take is *Exploratory Data Analysis (EDA)* (also called *data profiling*). In this step, we try to learn the basic characteristics of the data and whatever objects or events or observations it describes. If there is metadata, we check the dataset against it, trying to validate what we have been told—and augment it, if possible. If there is not metadata, this is the moment to start gathering it. This is a crucial step, as it will help us build our understanding of the data and guide further work. This step involves activities like classifying the dataset, getting an idea of the attributes involved, and for each attribute, getting an idea of data distribution through *visualization* techniques, or *descriptive statistics* tools, like histograms and measures of centrality or dispersion.[3]

We use the knowledge gained in EDA to determine whether data is correct and complete, at least for current purposes. Most of the time, it will not be, so once we have determined what problems the data has, we try to fix them. There are often issues that need to be dealt with: the data may contain errors or omissions, or it may not be in the right format for analysis. There are many *sources* of errors: manual (unreliable) data entry; changes in layout (for records); variations in measurement, scale, or format (for values); changes in how default or missing values are marked; or outdated values (called "gaps" in time series). Many of these issues can only be

[2]This step is referred to as *source discovery*.

[3]Readers not familiar with these notions will be introduced to the basics in Chap. 3.

addressed by changing the data gathering or acquisition phase, while others have to be fixed once data is acquired.[4] The tools and techniques used to fix these problems are usually called *data cleaning* (or *data cleansing*, *data wrangling*, *data munging*, among others). The issues faced, and the typical operations used, include

- Finding and handling *missing values*. Such values may be explicitly or implicitly denoted. Explicitly denoted missing values are usually identified with a marker like 'NULL,' 'NA' (or "N.A.," for "Not Available") or similar; but different datasets may use different conventions. Implicit missing values are denoted by the absence of a value instead of by a marker. Because of this variety, finding missing values is not always easy. Handling the absence of values can be accomplished simply by deleting incomplete data, but there are also several techniques to *impute* a missing value, using other related values in the dataset. For example, assume that we have a dataset describing people, including their weight in pounds. We realize that sometimes the weight is missing. We could look for the weight of people with similar age, height, etc. in the dataset and use such values to fill in for the missing ones.
- Finding and handling *outliers*. Outliers are data values that have characteristics that are very different from the characteristics of most other data values in a set. For example, assume that in the people dataset we also have their height in feet. This is a value that usually lies in the 4.5–6.5 range; anyone below or above is considered very short or very tall. A value of 7.5 is possible, but suspicious; it could be the result of an error in measurement or data entry. As this example shows, finding outliers (and determining when an outlier is a legitimate value or an error) may be context-dependent and extremely hard.
- Finding and handling *duplicate data*. When two pieces of the dataset refer to the same real-world item (entity, fact, event, or observation), we say the data *contains duplicates*. We usually want to get rid of duplicate data, since it could bias (or otherwise negatively influence) the analysis. Just like dealing with outliers, this is also a complex task, since it is usually very hard to come up with ways to determine when duplicate data exists. Using again the example of the people dataset, it is probably not smart to assume that two records with the same name refer to the same person; some names are very common and we could have two people that happen to share the same name. Perhaps if two records have the same name and address, that would do—although we can imagine cases where this rule does not work, like a mother and a daughter with the same name living together. Maybe name, address, and age will work? Many times, the possibility of duplication depends on the context; for instance, if our dataset comes from children in a certain school, first and last name and age will usually do to determine duplication; but if the dataset comes from a whole city, this may not be enough.

[4] The overall management of issues in data is sometimes called *Data Quality*; see Sect. 1.4.

The result of these activities is usually referred to as *clean data*, as in 'data that has been cleaned and fixed.' While cleaning the data is a necessary pre-requisite for any type of analysis, at this point the data is still not ready to be analyzed. This is because different types of analysis may require different additional treatment. Therefore, another step, usually called *data pre-processing* or *data preparation* is carried out in order to prepare the data for analysis. Typical tasks of this step include:

- Transformations to put data values in a certain format or within a certain frame of reference. This involves operations like *normalization, scaling*, or *standardization*.[5]
- Transformations that change the data value from one type to another, like *discretization* or *binarization*.
- Transformations that change the structure of the dataset, like *pivoting* or *(de)normalization*. Most data analysis tools assume that datasets are organized in a certain format, called *tabular data*; datasets not in this format need to be restructured. We describe tabular data in the next section and discuss how to restructure datasets in Sect. 3.4.

Data is now finally ready for analysis. Many techniques have been developed for this step, mostly under the rubric of Statistics, Data Mining, and Machine Learning. These techniques are explained in detail in many other books and courses; in this book we explain a selected few in detail (including an implementation in SQL) in Chap. 4.

Once data has been analyzed, the results of the analysis are usually examined to see if they confirm or disprove any hypothesis that the researcher/investigator may have in mind. The results sometimes generate further questions and produce a cycle of further (or alternative) data analysis. They can also force a rethinking of assumptions and may lead to alternative ways of pre-processing the data. This is why there is a *loop* in Fig. 1.1, indicating that this may become an *iterative* process.

Finally, once the cycle of analysis is considered complete, the results themselves are stored, and a decision must be taken about the data. The data is either *purged* (deleted) or *archived*, that is, stored in some long-term storage system in case it is useful in the future. In many cases, the data is *published* so it can be shared with other researchers. This enables others to reproduce an analysis, to make sure that the results obtained are correct. The publication also allows the data to be reused for different analyses. Whenever data is published, it is very important that it be accompanied by its metadata, so that others can understand the meaning of the dataset (what exactly it is describing) as well as its scope and limitations. If the data was cleaned and pre-processed, those activities should also be part of the metadata. In any case, data (like oil) should be disposed of carefully.

[5] Again, readers not familiar with these should wait until their introduction in the next chapter.

1.2 Types of Datasets

Our first task is to understand the data. Here we describe how datasets are usually classified and described.

There are, roughly speaking, two very different types of data: *alphanumeric* and *multimedia* data. Multimedia refers to data that represents audiovisual (video, images, audio) information. This data is usually encoded using one of the several standards for such media (for instance, JPEG for digital images[6] or MPEG for audio/video[7]). Alphanumeric data refers to collections of characters[8] used to represent alphabetic (names) and numeric individual datum. For instance, '123' represents a number (an integer) in decimal notation; 'blue' represents the name of a color. Such data is used to provide basic values, which are then grouped or organized in several ways (described below). Most methods for data analysis have been developed to deal with alphanumeric data, and that is the only data that we cover in this book. Handling multimedia data requires specialized tools: in order to display the image or play the video or music, a special program (a 'video/audio player') that understands how the encoding works is needed.[9]

An alphanumeric dataset (henceforth, simply a 'dataset') is a collection of *data items*. An item is usually called a *row* or *tuple* in database parlance; a *record*, in general Computer Science parlance; an *observation*, in statistical parlance; or an *entity*, *instance*, or a *(data) point* in other contexts. Each item describes a real-world entity, fact, or event; it consists of a group of related characteristics, each one giving information about some aspect of the entity, fact, or event being described. Such characteristics are called *attributes* in database parlance; *variables*, in Statistics; *features* in Machine Learning; and *properties* or *measurements* in other contexts. For instance, in our previous example of the people dataset, we implicitly assumed that the data was an assemblage of items, each item describing a person, and that the items were composed of attributes describing (among others) the name, address, age, weight, and height of each person. **Important note:** in this book we will use the terms *record* (although we will still use *row* for data in tables) and *attribute* from now on as unifying vocabulary; in formulas, we will use r, s, r_1, r_2, ... as variables over records and A, B, A_1, A_2, ... as variables over attributes.

The number of records in the dataset is usually termed its *size* (in databases, the *cardinality*). Conversely, the number of attributes present in a record is called the *dimensionality*. In some cases, all records in a dataset are similar and share the same (or almost the same) dimensionality, so that we can speak of the dimensionality of the dataset too.

[6] https://en.wikipedia.org/wiki/Image_file_formats.

[7] https://en.wikipedia.org/wiki/MPEG-1.

[8] In this context, a character is any symbol that can be produced by a key on the computer's keyboard: letters, digits, punctuation marks, and so on.

[9] There are books describing such tools and methods, although they tend to be quite technical.

Depending on the exact nature of the records, datasets can be classified as *structured*, *semistructured*, and *unstructured*. In this book, we will focus mainly on structured (also called 'tabular') data, because this is the kind of data that relational databases handle best, but also because this is the kind of data that is most commonly assumed when talking about data analysis, and the one targeted by most techniques. However, relational databases are also perfectly capable of handling semistructured and unstructured data, and we will also cover analysis of this data later in the book. Therefore, we start with a description of each kind of data.

The attributes that make up a data record or record are called the *schema* of the record. The value of an attribute may be a number or a label, in which case it is called *simple*, or it may have a complex structure, with parts that each has a value—such an attribute is called *complex*. Complex attributes have a schema of their own, too. For instance, in the people dataset, it could be the case that all records have a schema made up of attributes (name, address, age, weight, height) (it is customary to indicate schemas by listing attributes names within parentheses). An attribute like 'age' would have as values numbers like '16' and so on and thus would be simple. Conversely, an attribute like 'address' could be complex, with a schema like (street-number, street-address, city, zip code, state).

1.2.1 Structured Data

Structured data refers to datasets where all records share a common schema, i.e. they all have values for the same attributes (sometimes, such datasets are called *homogeneous*, to emphasize that all records have a similar structure).

Example: Structured Data

The following dataset is used in [20] (it comes originally from the US Bureau of Transportation Statistics, https://www.transtats.bts.gov/). It contains data about flights that departed from New York City in 2013; it is called `ny-flights` in this book. All the records have schema (flightid, year, month, day, dep_time, sched_dep_time, dep_delay, arr_time, sched_arr_time, arr_delay, carrier, flight, tailnum, origin, dest, air_time, distance, hour, minute, time_hour). The dataset has a dimensionality of 20 and a size/cardinality of 336,777 records. The first record is

```
(1, 2013, 1, 1, 517, 515, 2830, 819, 11, UA, 1545, "N14228",
"EWR", "IAH", 227, 1400, 5, 15, 2013-01-01 05:00:00)
```

The record is basically a list of 20 (atomic) values, one for each attribute in the schema. Thus, this is a structured, simple (tabular) dataset.

Example: Structured Complex Data

The following data record describes an imaginary employee; the schema is (first-name, last-name, age, address, department). Attribute 'address' is complex and has schema (street-number, street-name, city, state, zip code). Attribute 'department' is also complex and it has schema (name, manager). In turn, attribute 'manager' is complex and has schema (first-name, last-name).

```
("Mike", "Jones", 39, (1929, Main street, Anytown, KY, 40205),
("accounting", ("Jim", "Smith")))
```

Tabular data is structured data where all attributes are considered *simple*: their values are all labels or numbers without any parts. For example, the dataset `ny-flights` is considered tabular, while a dataset with records like the imaginary employee in the previous example would not be considered a tabular dataset, as some attributes are complex. As we will see later, this type of data is called *hierarchical*. It is sometimes possible to transform hierarchical data into tabular and vice versa.

When tabular data is in a file, each record is usually in a separate line, and inside each line, each attribute is separated by the next one by a character called a *delimiter*; usually, a comma or a tab. When using commas, the file is called a CSV file (for Comma Separated Values). If the schema itself (names of attributes) is included in the file with the data, it is usually in the first line—this is the reason it is called the *header*.

Tabular data is so-called because it is often presented as a table, organized into rows and columns. The rows correspond to data records and the columns to their attributes. Intuitively, each row describes an entity, or an event, or a fact that we wish to capture, with the attributes describing aspects of the entity (or event or fact). Not all data in tables follows this structure; data that does is called *tidy*, as we will see in Sect. 2.1.4.

Example: Data as Tables

The `ny-flights` dataset can be displayed as a table as shown in Fig. 1.2. It is customary to create a grid for rows and columns and to show the schema at the top, as the first row. In the figure we only show some of the schema (11 attributes) and 2 rows, for reasons of space.

Note that records (rows) fit neatly into the table because all records have the exact same schema. Also, values fit neatly in cells because they are all simple.

In most datasets, the size (recall: the number of records or records in the dataset) is much larger (at least one order of magnitude) than the dimensionality (recall: the number of attributes in the schema), but some datasets have high dimensionality

flightid	year	month	day	dep_time	sched_dep_time	dep_delay	carrier	flight	origin	dest
1	2013	1	1	517	515	2	"UA"	1545	"EWR"	"IAH"
2	2013	1	1	533	529	4	"UA"	1714	"LGA"	"IAH"

Fig. 1.2 NY-flights data as a table

(although what is 'high' depends on the context). As an example, the dataset **Chicago Schools**,[10] which contains school information for the city of Chicago during the 2016–2017 school year, has 661 rows and 91 columns (compare this with the `ny-flights` dataset). Analysis of datasets with a high ratio of attributes (columns) to data objects (rows) can be difficult, as we will see in Chap. 4.

Exercise 1.1 Get the Chicago Schools dataset and create a table for the first two rows. Just kidding! Pick your favorite dataset and create a table for two arbitrary rows.

1.2.2 Semistructured Data

Semistructured data is data where each record may have a different schema (also sometimes called *heterogeneous* data). In particular, some attributes may be *optional*, in that they are present in some records and not in others. Also, attributes may be a mix of simple and complex. Finally, some attributes may have as value *collections* of values as opposed to a single one (with each value in turn being simple or complex). As a consequence, records in semistructured data may have a complex structure.

In most real situations, semistructured data is used for datasets where attributes are not simple. Hence, it is very common to have datasets where the records, on top of being different from each other, have a complex structure.

Example: Semistructured Data

Assume a dataset describing emails. Each email has a *ID*, a *header*, and a *body*. The header, in turn, has attributes *timestamp*, *sender*, *receiver*, and *subject*. The timestamp attribute then has attributes *date* and *time*. Optionally, an attribute *CC* may be used; when used, this attribute may contain one value or a list of values. Such data is usually presented by showing both the attribute name and its value together, so as to avoid any confusion about what a value denotes. An example, with indentation used to display the structure and a colon (':') separating the attribute name from its value, would be[11]

[10] Available at https://bit.ly/30ZDJbf.

[11] This example is taken from the *Enron dataset*, a collection of emails from the Enron Corporation that has been used extensively by researchers for testing and analysis, as it is one of the few

```
ID: 19475126.1075855757890
Header:
        timestamp:
                date: Sun, Feb 4 2001
                time: 03:06:00
        sender: robert.benson@enron.com
        receiver: bsunsurf@aol.com
        subject: Rob-are you getting this?
        CC: peter.shipman@axiaenergy.com, gwadsworth@midf.com
Body: "How about lunch tomorrow?"
```

Note that attribute 'CC' has two values, separated by commas.

The above presentation mixes schema (attribute names) and data (values), while the tabular representation shows the schema once and does not repeat it. This is due to the fact that in tabular data, all records share the same schema, and therefore there is no need to repeat it for each record. On semistructured data, though, a record may be different from others; hence, we need to indicate, in each record, which attributes are present. Semistructured data is sometimes called *self-describing*, because each record contains both schema and data.

Semistructured data includes data in XML and JSON, two very popular data formats. They are very similar, differing only in how they present the data and a few other small details. XML describes the schema by using *tags*, labels that are enclosed in angular brackets. Tags always come in pairs, composed of an *opening* and a *closing* bracket. They can be identified because the closing bracket is exactly like the opening one but with the addition of a backslash. The value of the attribute goes between the tag pair.

Example: XML Data

The above email example is repeated here in XML format.

```
<email>
 <ID> 19475126.1075855757890 </ID>
 <Header>
        <timestamp>
                <date> Sun, Feb 4 2001 </date>
                <time>  03:06:00 </time>
        </timestamp>
        <sender>  robert.benson@enron.com </sender>
        <receiver> bsunsurf@aol.com  </receiver>
        <CC>
          <email> peter.shipman@axiaenergy.com </email>
          <email> gwadsworth@midf.com </email>
        </CC>
```

collections of real-life email datasets that is publicly available (see https://en.wikipedia.org/wiki/Enron_Corpus). Some fields are made up to simplify the example.

```
        <subject> Rob-are you getting this? </subject>
 </Header>
 <Body>  got it.       </Body>
</email>
```

Note that indentation is no longer needed, as the tags clearly indicate the data for all attributes, simple or complex; it is used here only to aid legibility.

JSON uses a different format for the same idea. Instead of tags, JSON uses labels for attributes, which are separated by a colon (:) from their value. Also, when an attribute denotes a collection, the values are enclosed in square brackets ([]), and when they denote a complex object, they are enclosed in curly brackets ({}).

Example: JSON Data

The XML data of our previous example can be written in JSON as follows:

```
{
 ID: 19475126.1075855757890
 Header: {
        timestamp: {
         date: "Sun, Feb 4 2001",
         time: 03-06-00
        },
        sender:  robert.benson@enron.com,
        receiver: bsunsurf@aol.com,
        CC: [
          email: peter.shipman@axiaenergy.com,
          email: gwadsworth@midf.com
            ],
        subject: "Rob-are you getting this?"
         },
Body:  "got it."
}
```

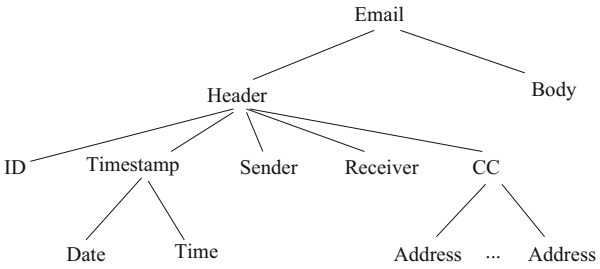
Note how the whole thing is enclosed in curly brackets, as it denotes a single record. Note also how some values are enclosed in quotes, so that it is clear where they end (there are no ending tags in JSON). All values (even complex ones) end with a comma, with the exception of the last value in the array.

When complex attributes are present, the schema of a record can be described by what is called, in Computer Science, a *tree*: a hierarchical structure with a *root* or main part that is subdivided into parts. Each part can in turn be further subdivided.

Example: Tree Data

In Computer Science, it is customary to draw trees upside down: the root is at the top, and each level down a 'sub-part' of the one above. The parts at the very bottom,

without any sub-parts, are called the *leaves* of the tree. In this example, the root is *Email* and the leaves are *ID, Date, Time, Address, Sender, Receiver, Body*. The node A above another node B is called the *parent* of B; B is the *child* of A. In a tree, each node has only one parent, but an arbitrary number of children. The parent of *Sender* is *Header*; the parent of *Header* is *Email*. The node *CC* has multiple children, indicated by the dots (. . .).



This tree represents the schema of the above examples; to add data, we would add a value to each leaf in the tree (since leaves represent simple attributes that have a simple value).

A semistructured dataset can be considered, then, as a collection of tree-like structures. It is important to note that in most practical cases the objects in the collection will have a good deal in common, that is, they all will share at least part of the schema. It is rare (albeit possible) to have an XML or JSON dataset where each object is completely different from all the others. This helps in dealing with semistructured data and makes it possible to put such data in a tabular format in many cases, as we will see in Sect. 2.3.1.

An important point to be made is that the border between structured and semistructured data is not rigid. It is possible to accommodate somewhat irregular data in tables, as we show next: give all records a schema with all attributes possible and leave empty cells for records that do not have values. Of course, this may yield a table with many empty cells, which makes analysis difficult, so this is only a good idea if data is more homogeneous than heterogeneous (items share many more attributes than not). As for attributes, whether an attribute is complex or simple is sometimes a design decision. For instance, an attribute 'address' could be expressed as simple (without parts) with values like "312 Main Street Anytown KY 40205" or the parts of it could be explicitly marked, as `street-number: 312, street-name: "Main Street", city: "Anytown", State: KY, zip: 40205`. As we will see, the choice to express the attribute in one way or another depends in part on what we want to do with the data. Even attributes whose value is a collection can be put in a tabular format, as we will discuss in the next chapter. In fact, one of the strengths of databases is that they make clear, in their design, what the data structure really is.

Example: Semistructured and Structured Data

The **Chicago Employees** dataset contains information about city employees in Chicago as of 2017, with a total of 33,693 records[12] The schema consists of attributes *Name, Job Title, Department, Full/Part Time, Salary/Hourly, Typical Hours, Annual Salary, and Hourly Rate*. Each record/row represents one employee. However, since an employee only has a salary if s/he is full time, and an hourly rate if s/he part time, and every employee is one or the other (an exclusive choice), not all employee records have values for all attributes. Two sample records are shown (names have been changed) in CSV format; attributes with no value are skipped (which creates the ',' followed by another ',' or by nothing if at the end of the line):

```
(Jones, Pool Motor Truck, Aviation, P, Hourly, 10,, $32.81)
(Smith, Aldermanic Aide, City Council, F, Salary,,$12,840,)
```

If we wanted to put this data in a table, we would have *missing values*. The same two records above are shown here; when there is a missing value in a cell, it is left empty.

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

(we have shortened "Full/Part Time" to "F/T," "Salary/Hourly" to "S/H," "Typical Hours" to "TP," "Annual Salary" to "AS," and "Hourly Rate" to "HR"). This data can be put in XML or JSON format, since we can use the flexibility of XML/JSON to get rid of empty attributes by only listing, in each record, attributes (tags) for which values exist (also, if we assume that all part-timers are paid by the hour and all full-timers have a salary, we could get rid of an additional field).

```
<Employees>
 <Employee>
  <Name> Jones </Name>
  <JobTitle>  Pool Motor Truck   </JobTitle>
  <Department>  Aviation   </Department>
  <Full-Part Time> P   </Full-Part Time>
  <Salary-Hourly>  Hourly   </Salary-Hourly>
  <TypicalHours> 10  </TypicalHours>
  <Hourly Rate> 32.81   </HourlyRate>
 </Employee>
 <Employee>
  <Name> Smith </Name>
  <Job Title>  Aldermanic Aide   </JobTitle>
  <Department> City Council </Department>
  <Full-Part Time> F </Full-Part Time>
```

[12]Available at https://data.cityofchicago.org/Administration-Finance/Current-Employee-Names-Salaries-and-Position-Title/xzkq-xp2w.

```
    <Salary-Hourly>  Salary </Salary-Hourly>
    <Annual-Salary> 12,840 </Annual-Salary>
  </Employee>
</Employees>
```

Semistructured data can be used for tabular data, since semistructured data can accommodate cases where all records have the same schema, although in this case repeating the schema for each record is highly redundant. The idea is to treat the table as an object made of a repeated attribute; this attribute in turn is an object representing the row, with attributes for each column.

Exercise 1.2 Put the data about New York flights in XML and JSON.

One particular, important case of semistructured data is *graph* data. Graph data (sometimes called *network data*) represents collections of objects that are connected (or linked, or related) to each other. A fundamental part of the dataset, then, is not just the objects themselves, but their connections. In many common situations, each connection links a pair of objects; the graph can be seen as a collection of objects and their pair-wise connections.[13] In the context of graph data, the objects are called *nodes* or *vertices*, and the links are called *edges*.

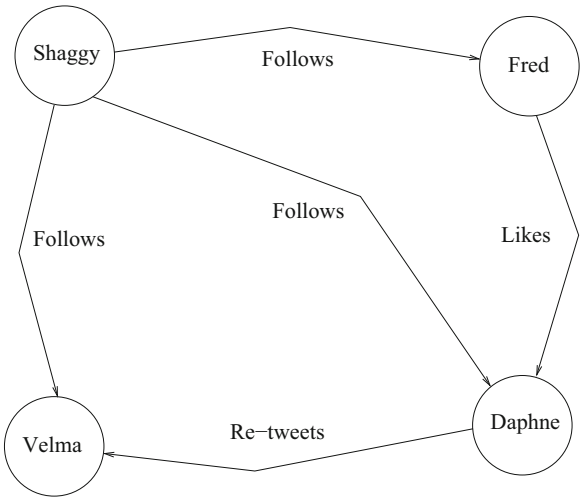
Example: Graph Data

The typical example of graph data nowadays is a *social network*, where nodes represent people and edges represent relationships between two people. In a scenario like Facebook, for instance, each node represents a Facebook user, and two nodes A and B may have an edge between them ('be connected') if A has declared B to be a friend, or if A has liked one of B's posts. In Twitter, nodes also represent users; two nodes there can be connected if A has re-tweeted or liked something that B tweeted, or A 'follows' B, or A and B share a hashtag.

Assume Shaggy, Fred, Daphne, and Velma are Twitter users, related as follows (we give a list of edges, with labels to indicate the type of relationship):
(Shaggy, Fred, follows)
(Shaggy, Daphne, follows)
(Shaggy, Velma, follows)
(Daphne, Velma, re-tweets)
(Fred, Daphne, likes)

This is usually depicted in a diagram, typically using circles for the nodes and arcs between them for the edges.

[13] It is possible for a connection to link more than two objects; the typical example is a relation SUPPLIES that connects suppliers, parts, and projects (3 objects). Graphs are limited to binary (two objects) relations, but are still very useful.



In many datasets, a node may have (regular) attributes to represent information about the object it stands for. In the case of Facebook data, for instance, each node may contain the information each user added to her *About* section, including work and education data. What makes this a graph, though, is the links to other users.[14]

One way to think of graph data is as records where the value(s) of some attributes are other records. For instance, in the previous example we can think of each node as a 'person' record where some attributes (like "follows") have as the value another person. Seen this way, graph data is semistructured because a node may have any number of such attributes—since a node may have an arbitrary number of edges to other nodes.

What makes graph data special is that the relationships between nodes are considered particularly important, so most analysis focuses on properties related to the edges, like finding whether two nodes N1 and N2 are *connected* (i.e. whether there is a sequence of edges, called a *path*, leading from N1 to another node N,' and from N' to another node N,"..., and so on until one last edge leads to N2), or how many connections there are from a given node to others. We discuss graph processing in Sect. 4.6.

A few variations of this idea are useful. Sometimes the edges between nodes have a *direction*, that is, they represent a one-way relationship. Such relationship is *directed*; a graph made up (exclusively) of directed edges is called a *directed graph*.

[14]This example is a simplification. Both Facebook and Twitter keep a very detailed dataset for each user, including every photo, video, or text that the user has ever posted, all their profile information, and everyone you have ever declared a friend (plus, in the case of Facebook, information about advertising categories that Facebook thinks you fit).

Other times, the edges have no direction. Such relationship is *undirected*; a graph made up (exclusively) of undirected edges is called an *undirected graph*. A good example of this is Facebook vs. Twitter. On Facebook, if person A is a friend of person B, then (usually) person B is a friend of person A, so this is an undirected relation. In Twitter, if person A 'follows' person B, it may or may not be the case that person B follows person A, so this is a directed relation.

Sometimes nodes in a graph represent different types of data. For instance, we may have a graph where some nodes represent people and other nodes represent books, and an edge between a person node and a book node represents the fact that the person has read the book. In such cases, nodes usually have a *type* attribute, and the graph is called a *typed graph*. Also, in some cases the vertices of a graph have *labels*, a value that gives some additional information about the connection represented by the vertex. Following with the example of books and people, suppose that some people may have read the same book more than once, and so it is necessary to indicate, when a person has read a book, how many times this has happened. Such information is usually represented by a label with a positive integer value. The graphs that use labels are called *labeled graphs*. The graph depicted in the previous example is directed and labeled.

It is interesting to note that, technically speaking, trees are special types of graphs. In a tree, edges are directed in the direction parent $\rightarrow$ son, and each edge must have only one parent (although it can have an arbitrary number of sons). Also, trees do not admit *cycles* (paths that end up in the same node that they started). This implies that no node can appear as its own *descendant* (a son, or a son of a son, or a son of a son of a son, etc.) or as its own *ancestor* (a parent, or a parent of a parent, or a parent of a parent of a parent, etc.). Hence, all information coming in XML and JSON can be considered 'graph data.' However, this is not how such data is treated. For one, trees are much simpler to deal with than arbitrary graphs, so it is important to note when data schema forms a tree and when it forms a graph. The reverse of this is that graph data can handle some things that are difficult to represent as a tree, as we will see in the next chapter.

1.2.3 Unstructured Data

Unstructured data refers to data where the structure is not *explicit*, as it is in structured and semistructured data (that is, no tags or markers to separate the records into attributes). Most of the time, unstructured data refers to *text*, that is, to sentences written in some natural language. Clearly, such data has structure (the grammar of the language), but the structure itself is not shown. Thus, in order to process such data, extra effort is needed.

Text data tends to come in one of the two ways: First, when we have a collection of documents (usually called a *corpus*), the text in the documents themselves is the main target of analysis. Second, we may have tabular data where one or more of the

attributes are textual in nature, as in the email dataset example, where the *subject* and *body* attributes can be considered text, as in the next example.

Example: Text Data

The **Hate Crime** dataset, from ProPublica,[15] consists of 2663 records, each one describing an article in a newspaper. All records have the same schema, composed of attributes *Article-Date, Article-Title, Organization, City, State, URL, Keywords, and Summary*. Each attribute is simple because its value is a name or something similar. The first record, with each value in a separated line (and the last two enclosed in double quotes, for legibility), is

```
  3/24/17 13:10,
Kentucky Becomes Second State to Add Police to Hate Crimes Law,
Reason,
Washington,
District of Columbia,
http://reason.com/blog/2017/03/24/kentucky-becomes-second-state
-to-add-pol,
“add black blue ciaramella crime delatoba donald gay hate law
laws lives louisiana matter police trump add black blue
ciaramella crime delatoba donald gay hate law laws lives
louisiana matter police trump”,
"Technically, this is supposed to mean that if somebody
intentionally targets a person for a crime because they’re
police officers, he or she may face enhanced sentences for a
conviction. That’s how hate crime laws are used in cases when
a criminal targets"
```

Note that the second attribute, *Article-Title*, and the last two attributes (*keywords* and *Summary*) can be considered text (*Keywords* is, as its name indicates, a list of keywords, and *summary* is a brief description of article content that is two sentences long).

We discuss how to store text in Sect. 2.3.3 and how to analyze it in Sect. 4.5.

Again, it is important to remember that the same dataset can be structured in different ways. For instance, even though Twitter data can be seen as a graph, Twitter actually shares its data using JSON: each Tweet is described as a JSON structure with fields like *text, user* (with subobjects `id, name, screen_name, location`), *entities* (with subobjects `hashtags` and *urls*, both of them collections). As another example, the `ny-flights` data introduced in a tabular way in Example 1.2.1 could also be presented as graph data by making the airports the nodes and

[15] This dataset can be obtained at https://www.propublica.org/datastore/dataset/documenting-hate-news-index.

considering each flight an edge from airport A to airport B, using the information in attributes `origin` and `dest`. Note that this is a directed graph, and that the edges do not have just a single label, but a collection of them (all other attributes). This format could be more appropriate than the tabular one for certain types of analysis. This lesson carries one very important consequence: when we acquire data, it will come in a certain format. This format may or may not be the appropriate one for the analysis that we want to carry out. Hence, it will sometimes become necessary to transform or *restructure* the data from one format to another. We will discuss this task in Sect. 3.4.1.

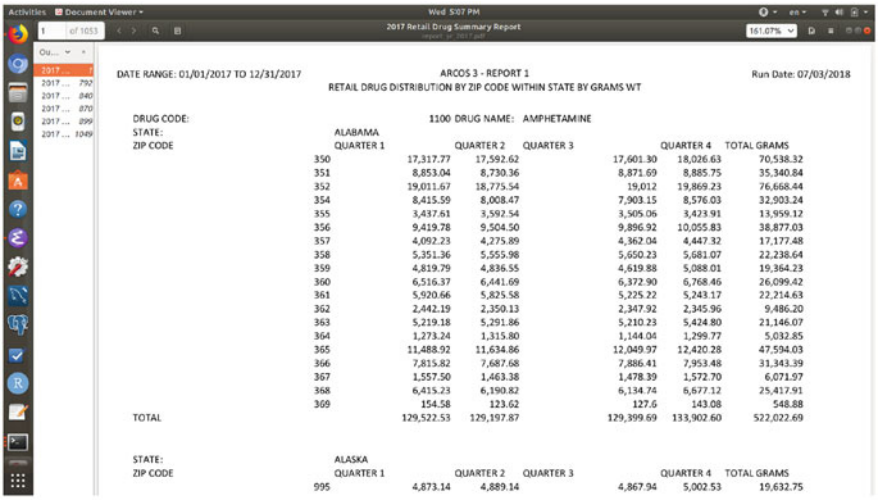
Exercise 1.3 A description of Twitter data in JSON can be found at https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/user-object.
Using that description, build a small JSON dataset with two imaginary users.

Exercise 1.4 Take a few rows of the `ny-flights` data from Example 1.2.1 and create a diagram to show it in graph format.

One final observation is that some datasets come as a mix of formats, which usually renders them useless for analysis. Typical examples include data from *spreadsheets*, very popular tools for dealing with data. Even though many datasets look like a collection of columns organized by rows, spreadsheets are structured differently—in fact, they are not structured at all. A user is free to put whatever she wants in any cell of the spreadsheet; as a result, columns many not contain the same type of data at all. Also, rows are not required to look like other rows. Spreadsheets and their problems are discussed in depth in Sect. 2.1.4.

Example: Spreadsheet Data

An example of data in spreadsheets is the ARCOS (Automation of Reports and Consolidated Orders System) report from the U.S. DEA (Drug Enforcement Agency). It is organized as a set of pdf documents, one per year. In each document, the report is broken down by drugs, and within drugs, it provides data per state (broken down by zip code and quarter of the year). A sample page is shown below. Such data is nearly impossible to use as provided, due to the format (we need the data in alphanumeric format, while the pdf is more like multimedia, in that it is encoded using a proprietary code) and to the lack of structure (even though the data *looks* tabular, it really is not, presenting issues similar to those of spreadsheets).



2017 Retail Drug Summary Report

DATE RANGE: 01/01/2017 TO 12/31/2017

ARCOS 3 - REPORT 1
RETAIL DRUG DISTRIBUTION BY ZIP CODE WITHIN STATE BY GRAMS WT

Run Date: 07/03/2018

DRUG CODE: 1100 DRUG NAME: AMPHETAMINE
STATE: ALABAMA

ZIP CODE	QUARTER 1	QUARTER 2	QUARTER 3	QUARTER 4	TOTAL GRAMS
350	17,317.77	17,592.62	17,601.30	18,026.63	70,538.32
351	8,853.04	8,730.36	8,871.69	8,885.75	35,340.84
352	19,011.67	18,775.54	19,012	19,869.23	76,668.44
354	8,415.59	8,008.47	7,903.15	8,576.03	32,903.24
355	3,437.61	3,592.54	3,505.06	3,423.91	13,959.12
356	9,419.78	9,504.50	9,896.92	10,055.83	38,877.03
357	4,092.23	4,275.89	4,362.04	4,447.32	17,177.48
358	5,351.36	5,555.98	5,650.23	5,681.07	22,238.64
359	4,819.79	4,836.55	4,619.88	5,088.01	19,364.23
360	6,516.37	6,441.69	6,372.90	6,768.46	26,099.42
361	5,920.66	5,825.58	5,225.22	5,243.17	22,214.63
362	2,442.19	2,350.13	2,347.92	2,345.96	9,486.20
363	5,219.18	5,291.86	5,210.23	5,424.80	21,146.07
364	1,273.24	1,315.80	1,144.04	1,299.77	5,032.85
365	11,488.92	11,634.86	12,049.97	12,420.28	47,594.03
366	7,815.82	7,687.68	7,886.41	7,953.48	31,343.39
367	1,557.50	1,463.38	1,478.39	1,572.70	6,071.97
368	6,415.23	6,190.82	6,134.74	6,677.12	25,417.91
369	154.58	123.62	127.6	143.08	548.88
TOTAL	129,522.53	129,197.87	129,399.69	133,902.60	522,022.69

STATE: ALASKA

ZIP CODE	QUARTER 1	QUARTER 2	QUARTER 3	QUARTER 4	TOTAL GRAMS
995	4,873.14	4,889.14	4,867.94	5,002.53	19,632.75

1.3 Types of Domains

As we have seen, structured and semistructured data are made up of basic building blocks: individual attributes that represent features or characteristics of data records. Complex attributes are made up of schemas of simple ones; in the end, each simple attribute represents a *domain* or set of values. For a given data record, the attribute provides a value from its domain. For instance, in the tabular dataset with people information, the attribute 'height' has values that provide the height of a particular person, expressed by numbers like 5.8 (if we assume that heights are measured in feet and inches) or 183 (if we assume that heights are expressed in centimeters). We would expect all such values to be numbers and to have a certain range. Thus, we can say that the attribute represents a *numerical* domain, expressed by real, positive numbers with one digit precision and up to a certain value. Conversely, an attribute like 'name' would have values like "Smith"; this domain is not numerical, and in principle it seems impossible to determine what values are in it (even something like "Xyz" could be a name in some languages). It would make sense to compare two heights and see which number is larger. It would not make sense to compare two names that way. It is clear that domains are of different types, and that different operations can be applied to different domains. Hence, recognizing different domains is an essential part of data analysis. Roughly speaking, domains for simple attributes fall into one of the three categories: *nominal* (also called

categorical), *ordinal*, and *numerical*. Some people call nominal data qualitative and numerical data quantitative.[16] We explain each domain in more detail next.

1.3.1 Nominal/Categorical Data

Nominal/categorical data refers to sets of labels or names (hence the 'nominal'): the domain is a finite set, with no relations among the elements. In particular, we cannot assume that there is any order (a first element, a second, etc.) or that elements can be combined. A typical example is an attribute 'Country' defined over the domain of all countries.[17] Note that we can put country names in alphabetical order; however, this order does not correspond to anything meaningful in the domain (it does not organize countries in any significant way). Another typical example is an attribute within a dataset of emails that gives the type of each email as one of 'work,' 'personal,' 'spam.' Here, the labels represent *categories* or classes of email (hence the 'categorical'). Again, we could order these labels alphabetically, but this order really carries no meaning. The most representative characteristic of nominal/categorical attributes is precisely that we cannot do anything with them except ask if a given label belongs to the domain or if it is present in the dataset, and whether the labels of two records are the same or not. We can also count how many times a given label is present in a dataset and associate a *frequency* with each label. Beyond that, very few operations are meaningful: finding the mode or calculating the entropy of the associated frequencies or applying the χ^2 test are the most common.[18]

Two things are worth remarking about this type of domain. First, values of a nominal/categorical attribute may be represented by numbers, but that does not make them numerical attributes. This is, in fact, quite common; it is called a *code*—for instance, '0' for 'married' and '1' for 'not married.' Again, even though they look like numbers, they do not behave like numbers. As another example, a zip code may look like a number (a string of digits), but it is not. To see the difference, ask yourself if it makes sense to add two zip codes or multiply them—operations that you would normally apply to numbers. Another characteristic worth mentioning is that sometimes there may be a hierarchical organization superposed on a nominal/categorical attribute. Consider, for example, a 'time' attribute that represents the months of the year. They can be organized into seasons (spring, summer, fall, winter) or quarters. Or a *location* attribute, where a point can be

[16] There is no total agreement on the details of this division. For some people, qualitative data includes both nominal and ordinal data.

[17] Underlying domains may be hard to define. For instance, in this example one may ask whether this refers to *current* countries, or any country that has ever existed. For more on this issue, keep on reading.

[18] Here and in the next few paragraphs we mention several statistical operations; for readers not familiar with them, these are all discussed in the next chapter.

located within a city, the city within a county, the county within a state, the state within a country. When looking at such data, it is possible to look at the dataset from several levels of the hierarchy; this is usually called the *granularity level*. Moving across levels is, in fact, a common analysis technique.

1.3.2 Ordinal Data

Ordinal attributes are, like nominal ones, sets of labels. However, a linear ordering is available for the domain. We can think of a linear order as arranging the elements in a line (hence the "linear"), with elements 'going before' others. Technically, a linear order is one that is irreflexive (no element goes before itself), asymmetric (if A goes before B, it cannot be that B goes before A), and transitive (if A goes before B and B goes before C, then A goes before C). As a consequence of having an order, we can always enumerate all elements in the domain and distinguish between a first one, a second one, ... and a last one.[19] The order also allows us to compare values with respect to their position in the order. Examples would be classifying symptoms of a sickness as very mild, mild, medium, severe, very severe; or opinions on a subject as strongly agree, agree, neutral, disagree, strongly disagree. Note that, since values can be ranked, we can compute a *median* (and, in the associated frequencies, a mode) and also percentiles and rank correlation, but we cannot do further operations. This type of attribute also includes codes; for instance, using '1,2,3,4,5' for 'very satisfied,' 'satisfied,' etc. These codes are still not numbers.

Ordinal attributes occupy a gray space between nominal and numerical attributes. They are labels, but the fact that they can be ordered means that we can, to a small degree, treat them as numbers. In fact, numbers are also ordinal attributes, as they can always be put in the usual linear order $<$ ('less than').

1.3.3 Numerical Data

Numerical attributes are representations of quantities, dimensions, etc. and are truly represented by numbers, be they integers or reals.[20] It is customary to further distinguish two types of numerical attributes:[21]

- interval: in these, the zero value (i.e. a point where all values start) is arbitrary. As a consequence, the distance between one value and another is (approximately) the same, so we can compare values and add/subtract them, but taking ratios

[19] At least for finite domains, which are the only ones of interest here.

[20] Note that reals can only be represented in a computer with a certain degree of precision. This will be important for numerical computations, and we will discuss it later.

[21] Some statistics textbooks distinguish between interval, ratio, and absolute types.

make no sense. We can take the mean, median, and value, as well as range and standard deviation/variance. Hence, we can apply the t-test and calculate Pearson's correlation coefficient. A typical example is a 'date' attribute: there is no 'zero' date, but the difference between the two given dates can always be calculated. Another example is the temperature measured in Celsius or Fahrenheit: while both scales have a zero degree mark, they both continue below zero: zero is not the absolute lowest value, and the place where the zero is somewhat arbitrary (as shown by the fact that the 'zero' in Celsius and the 'zero' in Fahrenheit are different).

- ratio: in these, there is an absolute zero, hence ratios make sense. A typical example is the temperature measured in Kelvin degrees (in this scale, zero marks the point where there is no thermal motion, and therefore it is impossible to go below it). In fact, most measurements (of time, distance, mass, money, etc.) fit here. For instance, a time of 0 s means no time has passed; a time of 10 s is twice as much as 5 s. With this domain, we can compare the values, add/subtract, multiply/divide, find mode, median, and mean (including arithmetic and geometric mean), as well as pretty much any mathematical manipulation. To note, the counts that result from associating frequencies to categorical values are in this category, which is why calculating frequencies is so useful and so commonly done.

Determining the type of a domain is extremely useful for analysis since it gives us an idea of what operations make sense for an attribute.

Example: Domains in Datasets

In the `ny-flights` dataset, all attributes are atomic. Each one represents a domain, but most attributes are temporal (`year, month, day, dep_time, sched_dep_time`, and so on). As already stated, dates and times are all numerical (interval type, to be precise), but by itself we can consider `month` a categorical attribute.

In the Chicago employees dataset, attributes `Name, Job Title, Department, Full-Part Time`, and `Salary-Hours` are categorical (attributes `Full-Part Time` and `Salary-Hours` can only have two values each; these are sometimes called *binary*). The rest of the attributes (`Typical Hours, Annual Salary, Hourly Rate`) are numerical—in fact, all of them are ratios, as their zeros are truly the 'zero' of each domain. The combination of categorical and numerical domains is extremely common in datasets for analysis.

Exercise 1.5 Classify the domains of all the attributes in the schema of `ny-flights`.

Exercise 1.6 A car database has an attribute called "Maintenance." Typical values are "60000 km/2 years," "80000 km/3 years." Is this a categorical or numerical attribute? What should it be for analysis purposes?

For practical purposes, it is a good idea to examine the domain from the point of view of the possible values allowed—equivalently, how membership in the domain is determined. Roughly, we can distinguish between:

- *closed* sets: also called *enumeration* domains. These are domains where the set of labels allowed can be precisely given. An example are the cities in a country or the months of the year. These domains tend to be static, i.e. do not change over time. There is usually some external resource that can help determine membership, which is simply a matter of looking the label up in the resource. Even though these simple sets can present issues, determining membership is usually not hard. For instance, the list of cities in a country may be different under different names—a list of German cities in German vs one in English, but once the language has been fixed, we can determine valid labels.
- *bounded* sets: these are domains where the elements are taken from a larger domain but somehow limited in some way. A typical example is *age*, which is expressed by a number—but not any number. For instance, the age of a person, expressed in years, can be any number between 0 and, say, 120. Negative numbers are excluded since they do not make sense; fractions and reals are also excluded due to the convention of expressing age in rounded amounts. Very large numbers (like 5,000) are out of bounds (this would be a typical example of an outlier). The difficulty of determining membership in these domains depends on whether the bounds are fixed or variable, strict or fuzzy.
- *patterned* sets: these are domains where all elements must obey a certain pattern (or one of a finite number of patterns). Typical examples are email addresses and phone numbers. The difficulty of determining membership here depends on the complexity of the patterns. For instance, for regular mail addresses in one country, the complexity tends to be low; however, mail addresses over the world follow a bewildering variety of patterns, and determining whether an international address is correct can be extremely hard.
- *open* sets: these are sets where it is not possible to give a definite criterion for membership. A typical example is person names. What counts as a person's name, even in a given language, is subjected to change over time (i.e. these sets tend to be dynamic) and is open-ended (in some countries, parents can give their children whatever they want as a name, i.e. they do not have to follow any rules). Consequently, determining membership may be very difficult or even impossible.

The importance of these distinctions is that one of the main tasks during EDA and cleaning is to make sure that all values in our data are correct for their domain. As we can see, deciding this depends on the type of the underlying domain, and it may be extremely simple or plain impossible.

Example: Simple Domains

In the Chicago employees dataset, attribute `Name` is an open set, while `Job Title, Department` are likely closed (that is, there is a finite list of possible job titles and possible departments). `Full-Part Time` and `Salary-Hours` are also close;

`Typical Hours`, `Annual Salary`, `Hourly Rate` are very likely bounded, in that we can determine minimum and maximum values for each. If this is correct, we can determine whether values in our data are within permissible bounds or they are the result of some error.

Exercise 1.7 Classify the domains of the attributes in the schema of `ny-flights` from this point of view.

1.4 Metadata

Metadata is *data about data*. It describes what the data refers to and its characteristics as a whole. Having metadata is highly beneficial for several reasons:

- To operate meaningfully in data, we must only carry out operations that make sense for that data: it makes no sense to add a temperature and a height, even though they are both numbers.
- In general, having metadata can guide our decision as to how to clean and preprocess the data.
- Sometimes data analysis will reveal that a transformation was not quite appropriate; we may want to *undo* the effects of the change and perhaps try something different. This is much easier to do if we recorded clearly what the changes were in the first place.
- If we need to share/export our data, we need to explain what the data are (what they mean or refer to) for others to be able to use them. Data storage and manipulation always happens in the context of a project that has certain goals. However, very frequently data collected for a certain purpose is reused later for different projects, with different goals. Hence, it is important to understand what the data was originally meant to represent, and how it has been transformed, in order to assess suitability for different purposes than the original one.[22]
- Because it provides a trace of how data came to be, metadata is crucial in helping with *repeatability* of analysis in order to generate *reproducible* results. This is becoming more and more important in all kinds of analysis.

In spite of its importance, metadata has traditionally been ignored in data projects. One reason is that it is typically considered overhead, i.e. work that gets in the way of obtaining results from the data. Another significant reason is that there is very little agreement among experts as to what constitutes good metadata, and how it should be generated and stored. However, given its potential positive impact, an attempt should be made to manage the metadata of any given dataset. In this section,

[22] This is a field of study on its own right, called *Data Reuse*.

we give some basic guidance as to what metadata should be kept and when it should be created; later in Sect. 3.5 we discuss how to store and manage it.

The concept of metadata is vague and it can be extended to cover many aspects of data and data processing. As a consequence, there are many diverse classifications of metadata in the literature. However, there are some basic parts that enjoy wide support:

- *structural metadata*: for both structured and semistructured data, metadata should include the schema or a given dataset[23] and also a description and classification of the domain of each attribute in the schema.
- *domain metadata*: Domain information is especially useful. Two aspects merit mention: *syntactic* and *semantic*. Syntactic metadata refers to how data is represented, what kind of values it can take. For numeric values, this includes appropriate range and typical values; in the case of measures, *unit*, *precision*, and *scale* are a must. For instance, a field giving salaries may have values in Canadian dollars, representing thousands (so that the value '85' actually means 'CAN $85,000'); a field giving people's heights (an example we have used before) should, at the minimum, describe which units are involved: metric ones (meters and centimeters) or English units (feet and inches).[24] For categorical values, this may involve (depending on the kind of domain) a list of valid values or valid patterns; this will allow us to check for correct data. For instance, a *name* field may come with a description of what a good value is supposed to look like: last name, followed by a comma, followed by a first name, with both names capitalized. For dates, a description of the format (for instance, 'month-day-year') is highly desirable, as this avoids (typical but painful) confusions.

  Semantic metadata refers to what the data is supposed to represent. For instance, knowing that an attribute is for people's names or is a measurement of people's height helps considerably when examining the data since in many cases the analyst can think of typical values, values that may not be correct, etc. Note that in such cases the name of the attribute ('name,' 'height') tends to be enough to point us in the right direction, but even when using meaningful names, this may not be enough. For instance, an attribute named 'price' may give the price of a product, but this may be before or after taxes, with or without discounts.[25] In the case of codes, semantic metadata is especially useful, as many codes tend to have cryptic names. For instance, in a list of products, a numerical attribute called *FY15* may be an enigma, until we find out that the name refers to 'Females Younger than 15' (and not, say, to 'Fiscal Year 2015'). The problem with codes may be in the values themselves; for instance, an attribute *Customer Satisfaction* may have values 1–5, which need to be interpreted (while this is a typical ordinal

[23] In the case of semistructured data, each record may have a different schema, but in most practical cases most records share a common schema with a few variations.

[24] Note that in cases where we have a good knowledge of the domain it may be easy to deduce unit and scale by looking at some data; however, this will not always be the case.

[25] In fact, a product may have many prices in a given dataset, because of these distinctions.

domain, it is important to know if bigger number means more satisfaction or less satisfaction). Thus, explaining what the data is supposed to represent in plain and clear language is an invaluable aid for any kind of analysis.

- *Provenance/lineage metadata*: as stated above, this describes where data comes from. Provenance refers to two different, but related aspects: for raw datasets, it refers to the source of the data; for clean/processed datasets, to the manipulations that resulted in the current state of the data. In the first sense, provenance is a way to explain how data was obtained. It should describe the source (for instance, a sensor, a web page, or a file), the date when data was obtained, and other relevant information (for instance, the original owner and/or creator of the data).[26] It is especially useful to list constraints and assumptions under which the data was generated. For instance, data from an ER (Emergency Room) in a hospital should identify the hospital, the schedule under which data was collected, and whether the original source was original medical transcripts, electronic records, etc. The second sense of provenance applies to datasets generated by manipulating data through the data life cycle process: by cleaning, pre-processing, analyzing, etc. This includes recording actions applied to the data and their parameters. For instance, an attribute may have been found to include many abnormal values; a decision is made to delete them and substitute them by the mean (average) of the remaining values. Whether this is a good decision or not depends entirely on the context. Thus, the best thing to do is to document this step; if later on it turns out to be misguided, we may be able to undo it and try something else.[27]
- *Quality metadata*: this is a description of how good the data is. This metadata may not be available at all when data is acquired and may have to be generated after the EDA step. Nevertheless, it is an important aspect to document, since many datasets are found to contain problems that affect their usefulness. In fact, this is such a common issue that a whole sub-field, *Data Quality*, has developed concepts and techniques that deal with problems in the data. While quality is a many-faceted concept, some key components have been identified by researchers [2] and should be present in the metadata, if at all possible:
  - Accuracy: how close the represented value is to the actual value. This is especially important for measurements and may require specifying how values were acquired. In the case of continuous domains, there is usually an imposed limit to the accuracy; this should be documented. Accuracy will help determine the *precision* of the values.
  - Completeness: this refers to "the extent to which data are of sufficient breadth, depth, and scope for the task at hand" [2]. In a relational database, we can talk of *schema completeness* (do we have all entities and attributes needed?), *column completeness* (how many values are missing in an attribute?), and

[26] Such information is very useful to, for instance, have someone to go and ask questions.

[27] Note that recording an action is no guarantee that it can be undone (some actions cannot be reversed); however, not recording it pretty much guarantees that the action cannot be undone and, worse, that trying to identify and mitigate its consequences will be almost impossible.

population completeness (do we have all values from the domain?). The latter is especially important since many datasets are *samples* from an underlying population, and often they have been obtained in a process that mixes opportunistic and random characteristics.

- Consistency: this measures whether the data as a whole is sound; it answers the question: are there contradictions in or across records? Sometimes a record may contain inconsistencies; for example, a person record with a *name* like "Jim Jones" and "female" for *sex*; or a record with '5' for *age* and 'married' for *marital-status*; or an order record where the date when the order has to be delivered is earlier than the date the order was placed. Sometimes two or more records may contradict each other (in a dataset of flight information, there may be two records sharing the same identifier but different destinations). Inconsistencies in the dataset need to be eliminated before analyzing the data, as they negatively affect pretty much any type of analysis.
- Timeliness: this refers to value changes along time. We want to know, mainly, two things: how often data changes (*volatility*) and how long ago was data created and/or acquired (*currency*); the two of them together determine whether the data is still valid (and, in general, how long it will remain valid). Volatility refers to the frequency with which data changes (rate of change). Data can be *stable* (volatility: 0), *long-term* (volatility: low), *changing* (volatility: high). Currency refers to when data was created, but may also refer sometimes to when data was acquired for the dataset. For instance, assume a sensor that takes temperature measurements every hour. Usually, a *timestamp* attribute will tell us when the values of the *temperature* attribute were taken. Assume, further, that the sensor sends information wirelessly every 24 h to a database. Thus, the acquisition time is the same for a whole set of data. Note that data may be current and not timely (for instance, a schedule of classes is posted after the semester starts); this is because timeliness depends on both currency *and* volatility. Note also that if data is updated, this affects its currency.
- Certainty: how reliable is the data (how much do we trust the source)? For many measurements, this is not an issue, but in certain analysis it can become a crucial element. For instance, when analyzing news reports, we may find out that reports on certain subjects (for instant, sudden, high-impact events) are highly uncertain.

It is clear that these aspects are related: if the data is uncertain, we cannot estimate its completeness, but if it is inconsistent, it cannot be certain. Also, accurate data tends to be certain; it is unlikely (although possible) that we have a very accurate measurement but are uncertain about its reliability. Thus, it may be tricky to evaluate all the aspects in isolation. Also, not all aspects apply to all datasets. Thus, one may want to create only essential metadata for a given project. However, all shortcuts taken will restrict our ability to reuse and publish the data.

Some authors will consider, beyond what we have outlined here, other aspects like *administrative metadata*—a description of rights/licenses, ownership, permissions, regulations that affect the data, etc. This may be advisable for data subjected to legal or ethical rules, for instance, private data that must be kept out of reach of the general public.

Exercise 1.8 Generate domain metadata to describe, as best as you can, the meaning of the attributes in the schema of `ny-flights`.

When is metadata generated? Ideally, metadata should be created at the acquisition/storage stage, when data is first acquired. This metadata should reflect what we know about the source of the data and the domains that the data is supposed to represent, that is, the *domain knowledge*. Having metadata that describes the data is very useful for data exploration, cleaning, and pre-processing. In particular, dealing with missing values and outliers is much easier when we have an idea of what data is supposed to be like. This initial metadata can be refined by EDA. For instance, suppose that we are gathering physiological data for a medical study, and one of the features we have is blood pressure. This is measured with a couple of numbers that give the *systolic* and *diastolic* pressure, and for most people is around 120 (for systolic) and 70 (for diastolic). Numbers up to 130 and 85 are still considered normal, but numbers above those indicate hypertension (high blood pressure). Conversely, numbers below 120 and 70 may indicate hypotension (low blood pressure). Knowing all this can help us determine if some values are outliers (for instance, numbers like 300 or 20 are probably the result of some error) or extreme values (for example, numbers like 180 and 110 indicate extreme hypertension, but are certainly possible). Note that this information comes from the domain knowledge and needs to be reconciled with what EDA extracts from the data. For instance, if our dataset is from people with heart disease, it could be the case that most of them have high blood pressure. Or, if the dataset is about infants or school-age children, the data will have different values. If EDA finds out that the metadata is not a good description of the data, we must reconcile the observed measurements with the assumed interpretation of the data.

During cleaning and pre-processing we may apply several changes to the data (see Sects. 3.3 and 3.4 for details). These changes should also be incorporated into the metadata, as already discussed.

As analysis proceeds, the methods used, together with any parameters and assumptions, should also be recorded. This will contribute to a correct interpretation of the results.

When it comes to archiving, metadata itself should always be a prime candidate to be kept. Having provenance metadata may allow us or others to recreate the original dataset (if not kept) as well as all the work done. Metadata is also the main tool to determine whether data can be reused for purposes different from the ones that motivated its collection—in particular, structural and domain metadata will ensure that the dataset is correctly understood. Finally, metadata tends to be much smaller than the data itself, and therefore it can be easier to store.

1.5 The Role of Databases in the Cycle

A database can be used in several roles in Data Analysis, depending on the exact situation. The first scenario is when the data already exists in a database; therefore, we have to go there to access it. After accessing the data, we have two options: one is to export the data to files (see Sect. 2.4 for details) and use R or Python or some other software to do all the work (see Chap. 6 for a brief description of how R and Python can interact with a database). Another option is to carry out some of the work (for instance, some EDA, some data cleaning and pre-processing) in the database and then export only the clean, relevant data to a file and continue with other tools. When the analysis does not include sophisticated Machine Learning or Data Mining techniques, we may be able to do all the work inside the database, avoiding the extra effort to export the data and use a different tool (see Chap. 4). Even when this is not the case, doing some of the work within the database still offers a number of advantages over tools like R and Python. First, when the dataset is very big, some preliminary work may allow us to extract only a part of the data (either a sample or a carefully chosen subset), which would be beneficial for further work, as neither R nor Python is particularly suited to work with large datasets. Second, databases offer strong *control access*: we can carefully monitor and limit access to the data, an advantage if there are concerns about data confidentiality. Third, if the data arrives periodically, or all at once but is later updated (that is, if data changes at all), the database offers tools to handle this *evolution* of data that do not exist in R or Python. Certainly, changes can also be managed at the file level, especially if one is knowledgeable about the power of the command line [9], but it is certainly nice to have tools that make life easier (see Sect. 2.4.2 for information about modifying existing data).

In a second scenario, data is to be captured yet and it is decided to use a database because of the size of the dataset or its complexity. In that case, we start by moving the data into the database (again, see Sect. 2.4 for details). Once this is done, we are in the previous scenario, and the same considerations apply.

In many cases right now data exists in files and the whole process takes place in R or Python. That is, databases are absent from the process. While this is totally justifiable in many cases (small datasets, complex, or ad hoc processing), this approach is manual-intensive and tends to produce results that are not well documented and are difficult to reproduce unless the markup tools available in R and Python are used.

All in all, databases can be a helpful tool in managing data through its life cycle, many times in combination with other tools. The next chapter goes through the process of putting data in the database, updating it if needed and exporting it. Later chapters discuss how to examine, clean, and transform the data for analysis.