2 A gentle introduction to data modeling

Fotis Jannidis and Julia Flanders

1 What is data modeling?

Imagine you find a handwritten note beginning with the following text:[1]

```
     1867
   Albany May 24th

 Mr Walt Whitman.
   You may
 be surprised in receiving
 this from me but seeing
 your name in the paper the
 other day I could not
 resist the temptation of
 writing to you.
```

You would be able to understand immediately that the text belongs to a specific text type or genre: it is a letter. A letter—today typically a personal communication between individuals—shows some specific features. Many letters bear a date and contain an indication of where they were written. They typically begin with the name of the person who is addressed, and the text is a direct communication making often use of the pronouns "I" and "you." Based on this knowledge, one can identify the text type of our example and also identify certain groupings of information within the text: "1867" belongs with "May 24th" as part of the date, while "Albany" is obviously the place of writing—or at least, obvious to us humans. To machines, the whole text is an undifferentiated sequence of alphanumeric characters and whitespace. If we want the computer to "understand" the text—for instance, to extract all dates from a collection of letters automatically, or sort the collection chronologically—we must either mark all occurrences of dates in a way that allows the computer to search for our marks and collect the text, or we need to describe the phenomenon of "date" with a series of rules (or train the computer to recognize dates through a process of machine learning)

which allow the computer to identify specific parts of the text as dates. In other words, we have a choice between an algorithmic approach (which will be explored in more detail in several of the contributions to this volume, below) or what we might call a "metatextual" approach, in which information is added to the text in some explicit form that enables it to be processed intelligently. This latter approach will be our focus in the remainder of this chapter.

In short, to make information accessible to the machine in a processable way, we have to provide a general model for this type of information and to apply this model to the instance in question. Applied to our example, this means that we have to provide a model for dates in general and apply it to our specific date. A very simple model could look like this: we identify the boundaries of the "date" information in the text using explicit markers, and we provide a regularized version of the date in the form year-month-day. Applying it to our text could look like this:

```
1867 Albany [date-start]May 24th[date-end 1867-05-24]
```

Now we can search for all sequences beginning with "[date-end" and ending with "]" and extract the ten characters before the final bracket which represent the regularized date. We also know that the two numbers after the first dash are the month, since this was specified in the model. This is a start, but a simple model like this shows its limitations quickly: for example, how would it handle a date before the year 0, or dates in which only the month or the year are given and the rest is missing? How would we handle dates that use different calendar systems? These contingencies all require additional provisions in the data model. In what follows we will introduce some basic concepts of data modeling (section 2) before we outline some foundations for all forms of data modeling: logic, set theory and formal descriptions of sequences (section 3). On this basis, we can talk about some of the basic concepts of established approaches in data modeling like relational databases, XML, and graphs (section 4). Although these processes are quite different, we can describe some common aspects of the modeling process (section 5) and the evaluation of data models (section 6). We conclude with the discussion of some of the perspectives of and challenges for data modeling (section 7).

Before we delve into these topics, we should have a closer look at the term "model." We already used it because we all have some understanding of it. At the basis of our understanding of the term *model* is probably a class of physical objects such as a "model of a plane" or a "model of a house," which can be defined as "a small copy of a building, vehicle, machine, etc., especially one that can be put together from separate parts."[2] But we can also use the term to mean an abstract concept and its relation to other concepts, as in a sentence like "each model of a modern state has to include institutions for the executive, the legislative and the judiciary." We will be talking about models in this more abstract sense in the following. With Stachowiak, we can distinguish three basic properties of a model:

1. A model is a model *of something*. A model is always a kind of mapping. It represents something, an object, a concept, and so on, by representing it using something else like clay, words, images, and so forth.
2. A model is *not the original* and it is not a *copy of the original*. Unlike a copy, a model doesn't capture all features of the entity it represents, only some of them. The choice of features selected to be present in the model is usually based on assumptions by the creator of the model concerning which features are relevant for the intended use of the model.
3. A model is meant to be used *by someone for something*. As we already mentioned, a model has a use for someone who can do something with it. The model can be used as a substitute for the original at least for some operations and at least for some time.[3]

So one could say that a model is a representation of something by someone for some purpose at a specific point in time. It is a representation that concentrates on some aspects—features and their relations—and disregards others. The selection of these aspects is not random but functional: it serves a specific function for an individual or a group. And a model is usually only useful and only makes sense in the context of these functions and for the time that they are needed.

Models are not intrinsically computer-processable. In order to operate meaningfully in a digital context, a model must be represented in language that is unambiguous and explicit, and that represents the salient features of the model in a processable way. In the following, we will be talking about *formal models*. These are models that use a specified set of rules that explicitly and exhaustively define the model's syntax and semantics. This explicit and formal specification allows this kind of model to be processed automatically. But the relation between models and formal models is intricate. In one view, a model can be understood as the rich, conceptually expressed context of a formal model, framed as a communication among humans but containing all of the intellectual work necessary to establish a formal model. On the other hand, a model can also be understood as a looser precursor to a formal model, one whose expression in human language necessarily entails vagueness and indeterminacies that will need to be clarified in the formal model. The first of these views tends to focus on how the concept of the "model" is itself embedded in more general concepts like theory and how they, theories and models determine or at least interact with the formal model. The latter view places greater emphasis on formal modeling itself, its mechanisms and dependencies, and treats the more general "model" as a kind of natural preliminary. Our approach in this volume leans more toward the latter.

Formal models can be expressed as a set of logical expressions or mathematical functions, but often there are specific notation systems that have been developed for specific types of models—for example, the Entity-Relationship notation for the conceptual level of relational databases, or tables for the logical level, or XML schemas to describe the model for a group of XML documents.

When we talk about a formal model, we refer to a specific structure or a set of structures defined by the model; quite often, we also imply that the components

of this structure have to conform to specific data types defined in the data model. For example, a simple data model for a date could stipulate that a date is a sequence of three components, either [day][month][year] or [month][day][year] and that the data type for day is an integer between 1 and 31.

Formal models which are defined by this application of logical/mathematical descriptions enable two functions: data constraints and data queries. During data entry or data creation, the model can be used to express constraints. For example, we might include in our model for a date of birth some constraints to ensure that only plausible information is entered (for instance, that the value entered is a date and that it is earlier than the date of death). The structure of the model can also enable us to query the data more precisely and intelligently: for instance, if the data "knows" that a specific string is a date, we can request search results that fall within a specific date range. Relational databases can be queried using relational logic (described in more detail further down), which is the foundation of the model; SQL (the Structured Query Language, which can be regarded as "an engineering approximation to the relational model" (Wikipedia, 2015)) is used as a query language for relational databases. XQuery, which can be used as a query language for XML documents, makes use of the formal structure of XML documents and enables much more complex retrievals than those based solely on an untagged string representation of the same texts. If we include one further function present in most models—that they enable and support communication between humans—we have the three main functions of formal models: adding constraints to improve data quality, enabling more complex and semantically rich queries, and supporting communication between humans and machines about data.

In computer science there are three main areas known for the application of formal models: *data modeling*, *process modeling* (including simulation) and *system modeling*, the design of software systems. Data modeling is concerned with the modeling of entities: documents, events, information systems, agents, data sets, and so forth. Process modeling is concerned with the modeling of events in time: for example, the amount of water passing through a river bed in a given time, the change of employment resulting from specific events, or the spread of a new scientific concept in scientific texts. For the modeling of more complex processes, *simulation* has become an important tool. *System* or *software modeling* is the design of software systems, usually an abstract view of the design of a piece of software, nowadays usually taking an object-oriented approach. An established tool in software modeling is the use of the Unified Modeling Language (UML) as a visual design instrument. As UML has a very abstract definition, its use in the other two fields has been proposed but is not very common. In the following and in this book in general we will talk mostly about data modeling. This form of modeling has a long history in the digital humanities; modeling has a long history in the humanities in general, and data modeling in the humanities builds on many earlier attempts to define concepts and their relations in a clear and unequivocal way, such as, for example, classification schemes in library science like the Dewey Decimal Classification or Ranganathan's Colon System.

There is also another use of the term "modeling" which is closely related to the data modeling we are chiefly talking about in this book—namely, the modeling of data using mathematical functions. For example, if we are studying the rent of flats in a city and the size of the flats in square meters, we can plot this data using x for the size and y for the rent, which would reveal that in general rent increases as the size of the flat increases. We can approximate this relation using a line, described by a mathematical function of the form $f(x) = ax + b$, and although none of the data points has to be exactly on the line, the line will provide a rough model of the relation between the two pieces of information. If we now add a new flat to our data set and look only at the size, we can make a rough prediction about the rent. (We may also observe that there are other factors influencing the rent, such as location, and we could expand the model to include these additional factors.) The mathematical function with its specific values for a and b is a model for the data. Both forms of modeling—data modeling and mathematical models—add information about the data, but they allow very different operations on the data. Data modeling using a metamodel like XML is usually adding descriptive information about the data; typical operations are querying the data for specific subsets. Mathematical models are usually used to make predictions about new data or answer questions such as whether specific factors correlate with one another.

2 Some basic concepts

Let us come back to our example of the letter, and let us talk about one of the alternatives mentioned above, the manual entry of information. The date of a letter is an interesting case: on one hand, any letter is written on a specific date (or a series of dates), and this date is a piece of information *about* the letter. On the other hand, very often dates may be found in the letter *as part of the text* (for instance, in references to events or discussion of future plans). In order for the date of the letter itself to function as a way of managing the digital item—for instance, to find letters written near a given date, or to sort a collection of letters by date—we need to treat that particular date in a specialized way: as *metadata*, or data *about* an entity. Typical metadata might include information about the entity being represented by the digital resource: the name of the creator(s), a title, a date of creation or publication. Metadata may also include information about the process of digitization, such as the name of the person who created the digital resource, the means of digitization—for example, OCR or manual transcription—the editorial methods used, and so forth. In order to be useful for discovery, metadata is highly structured information, and hence it may often be kept in databases or in some other format separated from the data. Another approach is to keep the metadata bundled together with the original data but clearly distinguishable from it, through markup or some other mechanism.

Metadata's explicit information modeling permits us to query and manage the digital object in very precise ways: it tells us that "this information component is the creation date." Without it, we might be able to infer that the first four-digit

number we encounter in a letter is probably the creation date, but there might be many cases where that assumption was false. Similarly, if we want to query the content of our digital letter, we can also add information to the text of the letter to make explicit what kind of information a word or a series of words contains. Information added to some part of a digital object like a text is called *annotation* or *markup*, and it permits the computer to extract information precisely, without relying on inference.[4] For example, using our earlier imaginary data modeling scheme we could add the information that "Albany" is a place name as follows:

```
[placename-start]Albany[placename-end] May 24th
```

When we added the date to the text above we already added an annotation:

```
[date-start]May 24th[date-end 1867-05-24]
```

In both cases, we have added a new layer to the text: now we have the text of the original letter and another text (aimed at the computer) consisting of the annotations. In order for the computer to distinguish the two, we need a clear signal that delimits the annotations. In our example we used the character "[" while the character "]" signals a switch back to the normal text. The character used to indicate these switches is completely arbitrary and depends only on the standard you use. XML—for example, uses the characters "<" and ">" and shortens the -start/-end syntax used above like this:

```
<placename>Albany</placename>
```

This kind of annotation is also called *inline annotation* or *inline markup*, because it is added to the data file containing the original data. Another method would be to use *stand-off annotations* (see Chapter 11) that leave the original data intact and—using some addressing schema like the character offset—refer to the start and end of the data annotated (please see Table 2.1).

As inline annotations can always be converted to stand-off annotations and vice versa, it is more a matter of convenience which to use; stand-off annotation is most commonly used in cases where the annotations will be generated by a machine process, whereas inline annotation is more common in cases where the annotations are being added or edited by hand.

The example we offer here is a text, but actually all kinds of digital objects can be annotated; with non-textual media, it is most common to use stand-off

Table 2.1 Stand-off annotations

Beginning offset	*Ending offset*	*Annotation*
6	12	placename
14	21	date

Table 2.2 Metadata of Rubens's "The Judgment of Paris"

painter	*title*	*year*	*id*
Peter Paul Rubens	The Judgment of Paris	1606	21436

annotation. The metadata for an image like Rubens' "The Judgment of Paris" could be stored like this (please see Table 2.2).

The column "id" provides a *unique identifier*: that is, a name that is guaranteed to be unique among all other identifiers and can be used to reference this object. This identifier could be the common name of the object like "The Judgment of Paris," but since it is possible that more than one object with this name may exist, it is more common to use arbitrarily constructed identifiers that can be guaranteed to be unique.

A simple system to annotate images, for example, would allow to specify two points that mark the upper left and the lower right corner of a rectangle in the image. Here, the annotation as it could be depicted by a graphical user interface (please see Figure 2.1).

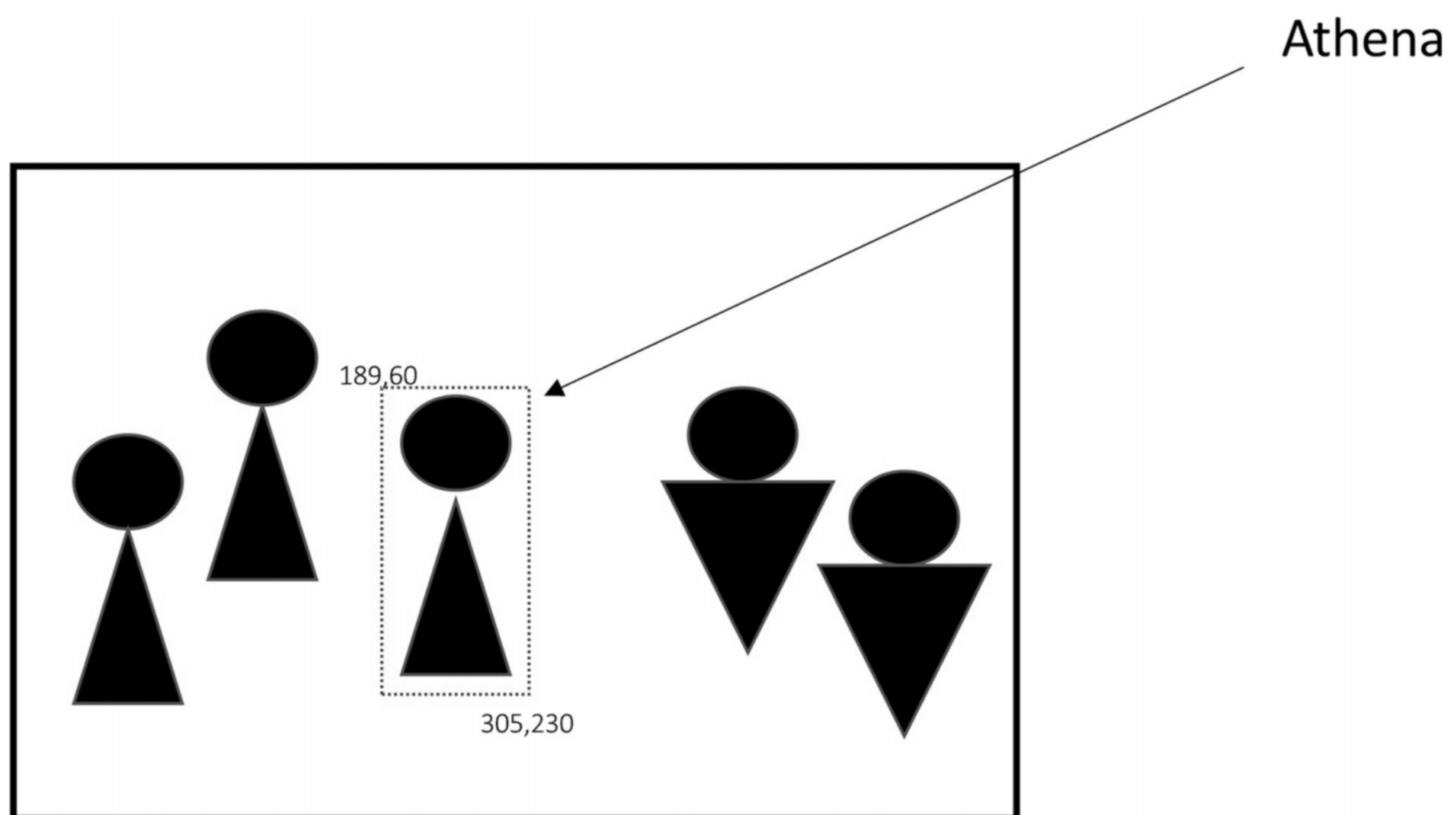


Figure 2.1 Image annotation.

And Table 2.3 shows a simple data model for this annotation.

Table 2.3 Data model of a basic image annotation

image id	*upper-left-x*	*upper-left-y*	*lower-right-x*	*lower-right-y*	*annotation*
21436	189	60	305	230	Athena

More complex annotation schemas would allow us to delineate any kind of form and add semantically more complex annotations like interactions.

Most of our examples in the following will be talking about the annotation of texts, which is mostly done by using *markup*: embedded notations that can be read and parsed by computational tools to assist in the analysis and processing of the text. Markup in this sense includes both metadata and the annotation of content, and in fact the distinction may not always be clear-cut. In addition, markup may be used to create data structures within the text that are not, strictly speaking, "annotation of content": for instance, we might use markup to create a timeline representing the events that occur in the letter. But for our purposes at the moment, we can set the more difficult cases aside and focus on the basic concepts.

Until now we have used the term "data model" rather loosely and have not distinguished between the different components of formal modeling: the *modeled instance*, the *data model* and the *metamodel*. The modeled instance is a digital model of a specific entity: for instance, a document, an art object, an event. All the metadata and the annotations related to the image example above are part of the modeled instance of this one image. The specific organization of the tables, their names and the column headings, constitute the data model which is used for all images in this collection. The concept of the *table itself*—a structure of rows and columns—as an organizational construct is the metamodel. We could also model this information in other ways: we might use a different model (a different organization of tables). But we might also use a different metamodel: for instance, instead of using a relational database with tables we could use XML. In the case of the letter example above, this might entail adding metadata and annotations to the text of the letter in the form of inline markup. The data model for our markup would be represented by a schema: for example, we might choose to use the TEI Guidelines and its schema, or we might choose to use HTML, or some other model. Here, the encoded letter is the modeled instance, the TEI (or HTML) is the data model, and XML is the metamodel. There are only a few metamodels widely used; we have already mentioned the relational model and XML. RDF, the Resource Description Framework, is used especially by cultural heritage communities. And the modeling of data in software, using for example an object oriented approach, is also very common.

On all three levels *data types* can play a crucial role. The term "data type" refers to the form in which one data point is stored, and it specifies the manner in which that data point will be interpreted: for example, as a date, or an integer, or a sequence of characters (string). In our image annotation table, the row labeled "image id" only stores integers, and the row labeled "annotation" only stores strings. Basic or "primitive" data types include different forms of numbers, such as integers (1, 2, 3) or floating point numbers (1.654, –23.91), Boolean values (True, False), characters ("a", "#") and strings ("history"). More complex data types, such as dates (2012-07-23), can be constructed out of these basic types. A data model usually also includes information concerning which kind of data type is allowed for a specific data element. Metamodels often include a list of predefined data types and also a way to define additional data types. In the

world of relational databases—for example, the specification for the data management and query language—SQL also contains data types that include numeric types, character strings, Boolean and datetime types, intervals, and XML types.

A *data structure* is the organization of data values (possessing specific data types) into a more complex form, which also defines a set of operations on the data in the data structure. Most programming languages offer some predefined data structures like "list" or "dictionary" and also mechanisms to construct more complex data structures together with their accompanying operations—for example, object-oriented modeling. A list offers a structure that allows us to address each of its elements by its position in the list. Here we have a list of country names:

```
countries = ["France", "Germany", "Japan", "Syria",
             "USA"]
```

Given this structure, we can address the first element of the list ("France") using a syntax which could look like this (most programming languages start to count with 0):

```
countries[0]
```

Programming languages usually also provide operations for modifying a list: to append or delete elements, and so on.

Somewhat confusingly, the term "data type" is also used in statistics to describe another quite different but important aspect of data modeling that affects the processing of data: the system or scale which confers significance upon the data and hence determines its behavior in contexts like statistical analysis. One very common distinction is between *discrete* and *continuous* data. We speak of "discrete data" if a data point can only have a value from a set of possible values: for example, the "country of birth" field can only have one value (such as "Canada" or "USA" or "Japan" or "Germany," and so forth) from a list of possible values. Similarly, "marital status" is usually defined using a closed list containing values like "married," "single," "divorced," or "widowed." The possible values for discrete data are either finite (as in the examples above) or infinite but countable. Examples of continuous data include information like height or weight, which can be measured to an arbitrary level of exactness. Another important dimension of statistical data is the characteristics of the underlying scale used for the data. In the classification of scales, "scale types are individuated by the families of transformations they can undergo without loss of empirical information" (Tal, 2015). By "transformations," here we mean things like changing the order of values, making comparisons, or performing computations. So, for instance, values for data such as country of birth or marital status simply divide the space of possible values into different segments, but without stipulating an order for the segments. These values have meaning only insofar as they can be distinguished from one another, and hence can be transformed without loss of information: instead of "divorced" and "married" one could substitute "D" and

"M" (or "1" and "2") without losing anything. This kind of scale of measure is called *nominal*; it simply names the values. If the values are ordered in some way, they constitute an *ordinal* scale, as with the possible results of some form of test: "failed," "passed," "passed with merit," "passed with distinction." An *interval* scale has even more information: the distance between different values can be known and interpreted. Temperature measured in Celsius is an example for an interval scale. Its zero point is defined by the freezing point of water and the value 100 is defined by the boiling point of water. The scale between these two points has been divided by 100. In an interval scale, certain kinds of computation are possible: for instance, we can say that the difference between 10 and 20 degrees Celsius is the same as the difference between 40 and 50 degrees Celsius. However, because the zero point is arbitrary, we cannot make other kinds of calculations; a statement like "this water at 60 degrees is twice as hot as this water at 30 degrees" is not meaningful. A *ratio* scale uses values with a meaningful non arbitrary zero point which allows values to compared to each other in the form of ratios like "this house is double the height of that house." The temperature measured in Kelvin (which starts at absolute zero) is an example of a ratio scale.

Table 2.4 shows some additional examples of data using the classification described above.

Continuous data like yearly income in dollars can always be converted to discrete data by creating so-called "bins" representing ranges of values: "less than 20,000," "20,000–100,000," "more than 100,000."

An important aspect of data and scale types is their direct influence on the type of operations which can be meaningfully done with the data. It doesn't make any sense to perform arithmetic on the numbers on football shirts, or to calculate the ratio of Cartesian coordinates. But you can sort numbers that express the choice on a rating scale from 1 to 5, or you can look at the differences between the hours measured using a 12-hour clock. This understanding of "data type" is rather specific to statistics, but the underlying concepts affect all kinds of data modeling because all kind of digitally stored data can and will be used for data analytics. But thus far the data type (in a statistical sense) of some types of data still has to be inferred from the label of the data—for example, the column headings of a table—and is not annotated in a machine-readable way.

Table 2.4 Examples of statistical data with different data types

	discrete	*continuous*
nominal	type of car, political party, number on a football shirt	n.a.
ordinal	choice on a rating scale from 1 to 5, sequence of cars reaching the goal in a race, sick vs. healthy	n.a.
interval	n.a.	time of day on a 12-hour clock, location in Cartesian coordinates
ratio	n.a.	income, mass

Finally, we should clarify one other aspect: Where does a data model live? In what concrete format is the data model expressed? We are talking here not about the data model as an abstract rule set (expressed, for instance, in an external schema), but about the shaping of the data itself. To answer this question, we can look at the different ways our digital data can exist—as a byte stream on some storage device, as a data structure in some memory cells in a computer's memory, as a serialized format meant to be read by machines and humans. The data model can determine the structure of all of these, but sometimes it is also only an abstract layer of communication between realizations of the data which—for some pragmatic reason—use another model. XML data, for example, are sometimes stored in relational databases where a piece of middleware translates all XML-related structures into a relational structure. Especially for the long-term preservation of data though, it is preferable to express the data in a serialized format that is readable by humans and is a direct representation of the data model.

3 Foundations

Before we take a closer look at some established approaches to data modeling, we want to review the foundations of *formal* modeling. Above, we said that a formal model is a model that can be expressed as a set of logical expressions or mathematical functions. And even if the data is expressed using specific notations (such as relational databases, XML, etc.), the underlying concepts of logic, set theory, sequences, and the rest form the deeper basis for the processability of the model. So, in the following sections we will offer a short overview of these concepts. A treatment in depth can be found, for example, in introductions into discrete mathematics. (Rosen, 2013), for example, is a very readable introduction for non-mathematicians.

3.1 Logic

Reasoning is an important way to produce new insights, and controlling reasoning to make sure it is valid has a long history; traces of its beginnings can be found in Classical Greece, China and India. For formal modeling, logic is important because the circuitry of a computer processor can be described using logic, and mathematical statements can be seen as logical statements, so the underlying mathematical structure of formal models can be represented as a system of logical propositions. Logic is still a thriving field of research and, as with all other material presented in this chapter, we will only be able to outline some of the very basic concepts. But this should be enough to give readers an idea how this approach works in general and to get started with more specialized literature if the need arises.

3.1.1 Propositional logic

Logic is divided in many subfields. The most basic one is propositional logic, which covers the truth values of simple combinations of propositions. The starting

point of this kind of logic is an abstraction: when considering combinations of statements, we can disregard the specific content of the statement and simply assume that it has a truth value (it is either True or False). The decision whether a specific basic statement is true or false falls outside of the realm of propositional logic, but once its truth value has been decided, logic can describe how it can be combined with other statements and what the truth value of these compound statements looks like. So, focusing purely on truth value, instead of statements like "the road is wet" or "In the year 2015 Obama was the president of the USA" or "The planet earth has one moon" (let's hope the truth value of this sentence does not change) we use variables like p or q.

Let us start with the first operator on propositions, the *negation*, which takes the proposition and states its negative; if the statement is "Today the sun is shining," then the negation is something like "It is not the case that the sun is shining today." If the initial proposition is represented with a variable "p" the negation is represented with an operator: ¬p ("not p").

To express the truth values of basic propositions and compound expressions in propositional logic, we create something called a truth table, which presents this information in an organized manner. Truth tables are an important instrument to describe the relation between the truth values of the basic propositions and the compound expressions, and, as we will see later on, we can use them to check whether two compound statements have the same truth values and are logically equivalent. A truth table for the negation operator looks like this (T = true, F = False).

p	$\neg p$
T	F
F	T

This table shows that if we know that p is true, then we also know that the negation of p is not true. This is at least the case in a logic with two truth values (True, False). So if the proposition "Today the sun is shining" is not true, we can infer that its negation, "It is not the case that the sun is shining today" is true.

The next basic operator is the *conjunction*: p ∧ q (p and q); it combines two basic propositions into a compound proposition, which is only true if all the basic propositions are true.

The truth table for a conjunction looks like this:

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

This table shows that if both p and q are true, then we also know that "p and q" is true, and that if one or both of p or q is false, the conjunction is also false.

The next operator is the *or* operator, which represents the disjunction, and it is a bit more complicated because there are two kinds of "or". One is the *inclusive or*, as in the following statement: "students who have taken the introduction to history class or the introduction to literary studies class can take the introduction to digital humanities course." It is enough that one statement is true for the compound statement to be true. This is the standard "or" and it is written like this: ∨.

The following shows the truth table of the inclusive or:

p	*q*	*p* ∨ *q*
T	T	T
T	F	T
F	T	T
F	F	F

Here, only one of the combinations of truth values of the basic propositions produces a false compound statement: if both p and q are false, the compound is also false. (Some say, if you have trouble remembering which symbol means what, you can use the following mnemonic: "and" (∧) in Latin is AUT while "or" (∨) is VEL; others say learning Latin to remember two symbols sounds like overkill).

The other kind of or is the *exclusive or*. If a restaurant menu offers as starters "soup or salad" it usually means you can have only one of them. Similarly, the exclusive or in a compound statement indicates that only one of the basic propositions can be true. There is no commonly used operator sign for the exclusive or, but a common notation is *xor*.

p	*q*	*p xor q*
T	T	F
T	F	T
F	T	T
F	F	F

Take, for example, the following propositions:

Proposition p: This car is blue.
Proposition q: This car is red.

The compound statement combining the basic statements with xor would be:

p xor q: Either this car is blue or this car is red.

If both sentences are true, the compound statement must be false. And if both propositions are false, the compound statement must be false also.

The *material implication* is probably the most problematic operator and in some ways the most counter-intuitive. It expresses a relation between p and q which can be expressed in a number of ways in English, for example:

"if p, then q"
"p implies q"
"q when p"

So let us take two propositions like these:

p: John is at the party
q: Mary is at the party

Then we can construct a compound proposition like this:

p → q: If John is at the party, then Mary is at the party.

This expresses a relation that is obviously falsified if we find John at a party and Mary is not there. And if both are at the party—that is, both basic propositions are true—then it makes sense to attribute truth to the compound statement.

Now if John is not at the party but Mary is, what can we say about the truth value of the compound statement? In real life our answer would probably be: How should I know? But this does not work for propositional logic which must assign one of two values to a statement. The truth table for the conditional operator looks like this:

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

There is long, intricate and controversial discussion in logic why and how the last two lines make sense (see Edgington, 2008). One of the better explanations points to the fact that if p is false and q is either true or false, then it still can be the case that the compound statement is not necessarily false and in a two-valued logic this means it must be true (see also Partee, ter Meulen, and Wall, 1990,

pp. 104f.). One can add a pragmatic point: In the context of argument analysis, these values yield the best results and fit into the rest of the architecture of propositional logic.

There is a variant of the material implication, the biconditional. It constructs a closer relation between p and q: p if and only if q, sometimes abbreviated as "p iff q." The compound statement is only true if both basic propositions have the same truth value. For example:

p: you can take the train.
q: you bought a ticket.
p $\leftrightarrow$ q: you can take the train if and only if you bought a ticket.

The truth table for the biconditional looks like this:

p	q	$p \leftrightarrow q$
T	T	T
T	F	F
F	T	F
F	F	T

The biconditional can be found very often in more formal definitions in logic or mathematics.

As mentioned before, the truth tables can be used to check whether two compound statements are equivalent. This may seem a trivial exercise, but even simple logical statements can be demanding to understand without them.

For example, if one tries to see whether two logical statements, $\neg$p $\vee$ q and p $\rightarrow$ q, have the same truth values, it definitely helps to look at the truth table. It shows for the statements the same truth values, so these statements are logically equivalent.

p	q	$\neg p$	$\neg p \vee q$	$p \rightarrow q$
T	T	F	T	T
T	F	F	F	F
F	T	T	T	T
F	F	T	T	T

Thus, truth tables can be a powerful tool, even if they are restricted to propositional logic.

3.1.2 Predicate logic

Predicate logic extends propositional logic by adding two new concepts: predicates and quantifiers. While in the last section we only looked at self-contained

statements in propositional logic (each of them having a truth value), here we are considering general expressions that have no truth value, but which constitute systems of potential truth value with variables representing the conditions we are interested in testing. For example, "x is president of the USA." This kind of expression is called a propositional function. It consists of a variable x and a predicate which is attributed to x, in our example "is president of the USA." The propositional function can also be written as P(x). With different values for x, this function has different truth values.

To take a simple example, let P(x) denote the statement "x is born in the year 1812" where x is the subject and "born in the year 1812" is the predicate. What are the truth values of P(Charles Dickens) and P(Friedrich Schiller)? We can check their values by substituting x with the specific names:

"Charles Dickens is born in the year 1812" is true
"Friedrich Schiller is born in the year 1812" is false

A propositional function can also have more than one variable, for example:

Let A(x1, x2) denote the statement "x1 + x2 = 10"

If $x_1 = 2$ and $x_2 = 8$, then we get the proposition

$2 + 8 = 10$

This proposition is true. If $x_1 = 1$ and $x_2 = 4$, we get the statement $1 + 4 = 10$ which is false.

Until now, we looked at statements with one variable ("x is born in the year 1812") and with two variables ("$x_1 + x_2 = 10$") and we used $P(x_1)$ or $A(x_1, x_2)$ as a notation for an expression with one and two variables respectively. We can add more variables in the form $A(x_1, x_2, x_3, \ldots x_n)$, in which case A is also called a n-ary predicate.

Those familiar with the basics of programming will recognize the propositional function as a part of the conditional "if then" statement which can be found in most programming languages:

```
if x > 80 then print("Warning: Line too long; more than
   80 characters.")
```

The call to the print function—which issues a warning—will only be executed when the condition "x > 80" is true. For example, if x equals 87, the statement "87 > 80" is true and the function call will be executed.

3.1.3 Quantification

Up to this point the propositional functions we have looked at consist of a simple schema: one or more variables are combined with one or more predicates.

Quantifiers allow us to express additional information about the functions; they allow us to specify whether something is the case for one, some or all members of a domain. In predicate calculus, two quantifiers are commonly used:

1. the universal quantifier $\forall x$ (for every x) and
2. the existential quantifier $\exists x$ (there is at least one x).

The quantifiers cannot stand alone; they have to be combined with a propositional function (or a proposition) like this:

$\forall x P(x)$: P(x) is true for every x

For example, let P(x) denote the statement $x - 0 = x$. The quantification $\forall x P(x)$ is true, because for all numbers this statement is true. Although the quantifier is called "universal," it is actually used in relation to a domain: in other words, the statement is true in a specific domain of discourse or universe of discourse. So the universal quantifier $\forall x$ applied to P(x) has to be understood to express something like

P(x) for all values of x in the domain.

A domain can be any kind of set—for example, all natural numbers or all people who are afraid of flying. The definition of the domain is up to the person using the quantification—and it is important because the domain often decides whether a quantification is true or not.

The quantification $\forall x P(x)$ is false, when there is a value of x in the domain which will produce a false P(x). For example, let P(x) be the statement "$x^2 > x$" and let the domain be $\mathbb{N}$ (natural numbers 1, 2, 3 . . .) then $\forall x P(x)$ seems to be true, because P(2) is true and P(10) is true, but we can find one counterexample: P(1) is false, and therefore $\forall x P(x)$ is false.

The existential quantifier expresses that the quantification is true for at least one element of a specific domain. $\exists x P(x)$ means:

There is an element x in the domain such that P(x) is true.

$\exists x P(x)$ is false, if there is no such element. For example, let P(x) be the statement "$5 - x > 3$" and let the domain be $\mathbb{N}$ then $\exists x P(x)$ is true, because P(1) is true ($5-1 > 3$).

These two quantifiers are quite important for logic and mathematics, but there are many more and we could even define our own. In logic, the quantifier $\exists! x$ means that there is exactly one x such that P(x) is true. As we will see later on, in some parts of computer science there are some other commonly used quantifiers: for example, in regular expressions and schema languages we find specific quantifiers for "zero or one" occurrence, "zero or more" occurrences

and "one or more" occurrences, all of which are useful in the context of searching and schema design.

3.2 Sets

Set theory was introduced by Georg Kantor in the nineteenth century as a way of thinking about the infinity of numbers, and it is now often seen as a foundation for mathematics in general (Pollard, 2015). Its concepts are also the basis for any data modeling, not least because it defines concepts like "set" and "tuple," which will play a role in most of our descriptions of data modeling approaches. This does not imply that set theory is especially deep or complex; general set theory is fairly straightforward, but it provides a very illuminating basis for the fundamental concepts of data modeling we want to introduce here.

A set, in the most general sense, can be informally understood as a collection of different objects. Anything can be part of a set: a word, a number, an idea, a real object, even a set. Usually, the members of a set have something in common, but that is not necessarily so. We call these objects the elements of a set. The elements in a set are not ordered. To express the information that a set A contains the element a, we write: $a \in A$ (a is an element of the set A), while $a \notin A$ tells us that a is not a member of A.

There are two ways used to define a specific set. One way, the list notation, is to enumerate its members:[5]

A = {John, Mary, lamb}
B = {2, 4, 6, 8, 10}

Even if the list contains the same element more than once, only the distinct elements are counted and considered to be part of the set. We can also use a notation like this, if the list is very long:

$$C = \{1, 2, 3, \ldots 99, 100\}$$

Sets do not have to contain a finite number of elements. The following defines the set of the natural numbers:

$$\mathbb{N} = \{0, 1, 2, \ldots\}$$

All the members of a set, taken together, are called its *extension*. So we can describe a set by enumerating all its members and we can think of this method of defining a set as being data-driven: we define the set by making note of its actual members.

The other way to define a set is by defining the conditions under which an entity is an element of the set; this is called the *intension*. By contrast with the previous method, this approach could be thought of as being theory-driven, or pattern-driven: it defines a set by establishing a logical set of conditions that

governs the members of the set. Very often the "set builder" notation is used, which allows us to specify the common properties of the set. So, if we want to specify a set with all even numbers between (and including) 2 and 10, we could write it like this:

A = {x | x is an even number larger or equal to 2 and smaller or equal to 10}

We might read this as "A is the set of values x where x is an even number larger . . . " Usually, set-builder notation is much more formal and compact. For example, we used the term "number," but what kind of number did we mean? Numbers like –2, 1.232 or the square root of 2? Or only natural numbers? To be more precise, we can use clearly defined notions and use the logical "and" we have seen above:

$$A = \{x \mid x \in \mathbb{N} \wedge 2 \leq x \leq 10 \wedge x \text{ modulo } 2 = 0\}$$[6]

We can see here how set theory and logic come together in the set-builder notation. The part after the | is a predicate and states that x is part of the set A only if x is substituted by something in such a way that the resulting proposition is true. Any predicate defines two sets in relation to a domain: one whose members will, if inserted into the propositional function, create true propositions and one whose members will create false propositions. So, it makes sense that the first set, in our example A, is also called the *truth set* of the predicate.

There is also a specific set that can be thought of as the zero of the sets, the empty set: Ø. The empty set is a set without elements, so we also could refer to it like this: { }. Sometimes, we describe the intension of a set, only to discover that it defines an empty set—for example:

D = {x | x is president of the US ∧ x is chancellor of Germany}

The concept of data types, which is so important for programming languages and data modeling, can be understood in terms of set theory: a data type is a set and there is also a set of operations that can be performed on this data type. For example, in many programming languages there is a data type called "string" and the members of this set are all character sequences. The set of operations on strings usually includes the plus operator (+), which concatenates elements of the set string and creates a new element of this set.

Many endeavors in data modeling can be understood as seeking an exact description of the intension of a set such that all potential elements will be included as intended, even those we haven't yet seen. This becomes a crucial element of schema design, in cases where the goal is to create a schema that can anticipate and accommodate the structure of a large set of documents whose full extent and nature cannot be known in advance, such as the articles to be published in a journal, or the documents contained in an unfamiliar archive. The description

of the intension doesn't have to be and often is not short; it can be and often is a specification that describes in formal terms the rules by which an element of this specific set—for example, the set of well-formed XML-documents—is defined. We will explore this in more detail below.

Let's now consider some of the ways in which two sets can be related: equality, subset/superset, and disjoint sets. Two sets are equal if they have the same elements: for example, these three sets are all equal:

$$A = \{2, 3, 5, 7\}$$
$$B = \{7, 2, 3, 5, 2, 3, 7\}$$
$$C = \{x \mid x \text{ is a prime number} \land x < 10\}$$

A = B, because the repeated items only count once for the purposes of determining the set membership, and the sequence of the elements is also not important for the definition of a set. C uses the set-builder notation, but this does not affect the assessment of the set's contents.

We say that a set A is a subset of a set B, if all elements of A are also elements of B. This can be written using the following notation:

$$A \subset B$$

This defines a *true* subset, where B also contains elements that are not members of A. If A could be equal to B, the following notation is used: $A \subseteq B$. It means that A is either a subset of or equal to B. As mentioned before, sets can be elements of sets. The empty set is a subset of all non-empty sets, so the statement

$$\emptyset \subset A$$

is always true, if A is not empty. And because the definition says that if all elements of A are also elements of B, then A is a subset of B, it follows that B is a subset of B too, because all elements of B are also elements of B:

$$B \subseteq B$$

If $A \subset B$, then B in turn is called the superset to A:

$$B \supset A$$

Obviously, there are many sets that do not have any elements in common, and these are described as *disjoint* sets.

There is a subtle but important difference between $A \in B$ and $A \subset B$. The first expression can only be true if the set B has an element that is the set A, while any set can be a subset of B. An example can clear this up:

Given the sets $A = \{1, 2\}$ and $B = \{1, 2, 3, 4\}$

A is a subset of B because all its members are also elements of B. But the set A is not an element of the set B. That would be only the case if B looked like this: $\{1, 2, 3, 4, \{1, 2\}\}$.

Sets can be combined in many ways. The most common are the set operations *union*, *intersection* and *difference*. The union of the sets A and B creates a new set containing all elements that appear in either A or B or in both. To use a more formal notation:

$$A \cup B = \{x \mid x \in A \vee x \in B\}$$

Probably more interesting is the intersection. The intersection of the sets A and B creates a new set containing all elements that are in A and also in B. Here the formal notation is helpful to avoid ambiguities:

$$A \cap B = \{x \mid x \in A \wedge x \in B\}$$

And finally, the difference between the sets A and B creates a new set containing all elements of A which are not in B:

$$A - B = \{x \mid x \in A \wedge x \notin B\}$$

The difference is sometimes also written as $A \setminus B$.

Our final set operation will seem a bit more unusual, but if we understand that we often want to understand something about all possible subsets of a set, it is clear that we need a shortcut to express this collection. The *power set* offers just that. The power set of a set A, usually written $P(A)$ or $\wp(A)$, creates a new set consisting of all the subsets of A including the empty set and A. So, if $A = \{1, 2, 3\}$ then $P(A) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$. It is generally the case that if A has n elements, then $P(A)$ has 2^n elements.

Relations between sets—like equality, subset/superset or disjoint—and also set operations such as union and intersection can be visualized using diagrams. There are two slightly different forms, called Euler diagrams and Venn diagrams, but in the following we will draw on features of both to express our concepts as clearly as possible.

Figure 2.2 visualizes the case where $A \subset B$ (A is a subset of B). Figure 2.3 visualizes the three basic set operations: union, intersection, and difference.

We emphasized above that sets are unordered collections of items. However, as we will see shortly, order is a crucial aspect of data modeling: for instance, one of the key differences between XML nodes and database fields is that XML nodes are explicitly ordered. We can add the concept of order to set theory using the set theoretic primitive *ordered pair*. An ordered pair is usually written with parentheses: (a, b); a is the first *entry* (or *coordinate*) of the pair and b the second.

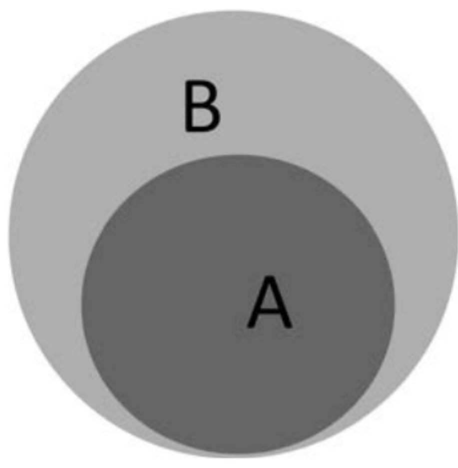


Figure 2.2 A set operation diagram: set A as a subset of set B.

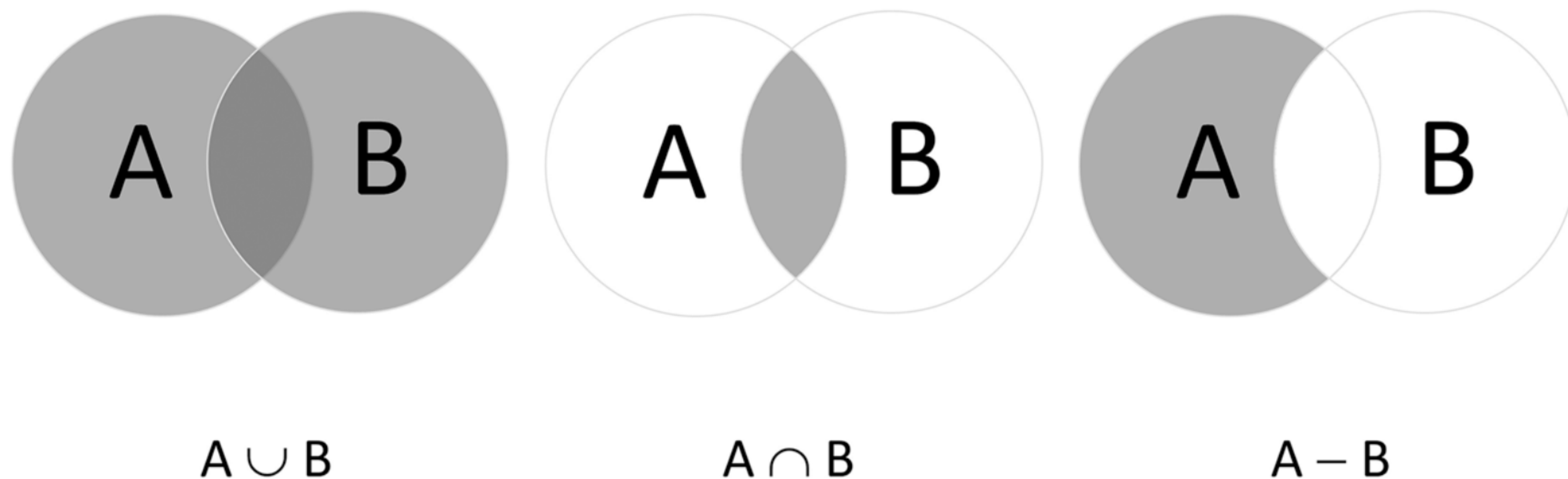


Figure 2.3 A diagram of the basic set operations: union, intersection, and difference.

The pair (a, b) is not equal to (b, a) unless a = b. But how can we derive this notion from the concepts we have seen thus far? We can express this ordered pair using our set notation thus:

(b, a) = {{b}, {a, b}}

The set {{b}, {a, b}} includes all those sets that are defined by going through the ordered entries from left to right and at each stage enumerating which elements we have seen, including the new element under focus. It is important to note here that the order of elements within this set—for instance, the fact that {b} comes first—is not significant and does not affect how this set expresses the order we are seeking to represent. Instead, the order is represented by the number of elements in each element of this set. The two-item set {a, b} represents the second item in the original ordered pair, and the item "a" that it adds (when the two-item set is compared with the one-item set {b}) is thereby identified as the second item in the original ordered pair. For a fuller and more formal explanation, see Halmos, 1960, Section 6.

In mathematics, an ordered pair is also called a 2-tuple, a special case of the more general concept of a *tuple* or an *n-tuple*, where n can be any positive integer and refers to the number of members. All of the following are tuples:

A = (1, 2)
B = ("a", "b", "c")

C = ("John," "Mary")
D = ("Mary," "John")
E = (32, "Mary," 1.3, "a", ⋈)

A is an ordered pair. Because order is significant for an ordered pair (2-tuple), C is not identical with D. Tuples can contain different kinds of elements as in E; we will see more of this when we talk about relations and relational databases below.

Now we have all the concepts needed to introduce the next term of set theory. The "Cartesian product" (named after the French philosopher René Descartes) describes an operation on two or more sets, which creates a new set consisting of n-tuples, where n is the number of sets involved in the operation. If we begin with two sets X and Y, then we can formally define the Cartesian product like this:

$$X \times Y = \{(x, y) \mid x \in X \wedge y \in Y\}$$

Essentially, this statement means that we create a set of tuples (x, y) in which we take each member of set X in turn and pair it with each member of set Y. The result set will contain all possible combination of elements. So, if we use the sets A and B we described above, the Cartesian product looks like this:

A × B = {(1, "a"), (1, "b"), (1, "c"), (2, "a"), (2, "b"), (2, "c")}

Remember that the sequence of our elements in a list notation of a set is irrelevant, but the sequence of elements in a tuple is relevant. So, while the inputs to the Cartesian product are unordered, the outputs are ordered. Therefore, A × B is not the same as B × A, because it would create different tuples:

B × A = {("a", 1), ("b", 1), ("c", 1), ("a", 2), ("b", 2), ("c", 2)}

It is also possible to create the Cartesian product on a single set: A × A, also written as A^2:

$$A \times A = \{(1, 1), (1, 2), (2, 1), (2, 2)\}$$

Descartes introduced this concept of a product to represent each point on a geometric plane as a tuple of x and y coordinates, and because they are usually real numbers this is often written like this: R^2. But a Cartesian product isn't limited to two sets. It can combine n sets and each of its resulting tuples will then have n elements. If we combine our three sets A, B, C defined above into a product set A × B × C, its tuples look like this:

A × B × C = {(1, "a", "John"), (1, "b", "John"), . . . (2, "c", "Mary")}

In the context of data modeling, the Cartesian product is important because it provides the basis for one of our core concepts: the *relation*. A relation is formally defined as a subset of a Cartesian product—in other words, any subset of the Cartesian product of one or more sets is a relation:

$$R \subseteq A \times B \text{ with } (x,y) \in R$$

And we can now say that x is related to y by R (short form: x R y).

To explain this in more detail, let us define two new sets:

N = {“Tina”, “Tom”, “Alex”}
T = {55523, 66619}
T × N = {(“Tina”, 55523), (“Tina”, 66619),
(“Tom”, 55523), (“Tom”, 66619),
(“Alex”, 55523), (“Alex”, 66619)}

Now let us define another set,

M = {(“Tina”, 55523), (“Tom”, 66619), (“Alex”, 66619)}.

M is a subset of T × N; it is one of many possible subsets and there is nothing special about this subset in comparison to all the other possible subsets. Because M is one of the possible subsets of T × N, we can say M is a relation from N to T. And we can say “Tina” is related to 55523 or (more compactly) “Tina” R 55523. Looking at it in this way allows us to apply all operators which can be applied to sets in general and also a specific operator—the operator “join”—which can only be applied to relations. We will discuss this in more detail below in the section on the relational model.

The meaning of this relation could be anything—for example, extension numbers in an office, or tax identification numbers. There are different ways to visualize a relation. We can use a table:

Or we can use a graph (please see Figure 2.4).

Table 2.5 A relation, represented as a table

	55523	*66619*
Tina	x	
Tom		x
Alex		x

Or we could use an adjacency matrix, which we will discuss below in the section on graphs. Relations on two sets are also called binary relations to distinguish them from n-ary relations (that is, relations based on n sets). An n-ary relation is a subset of $A_1 \times A_2 \times \ldots A_n$. We will have a closer look at them below in the section on relational models and databases.

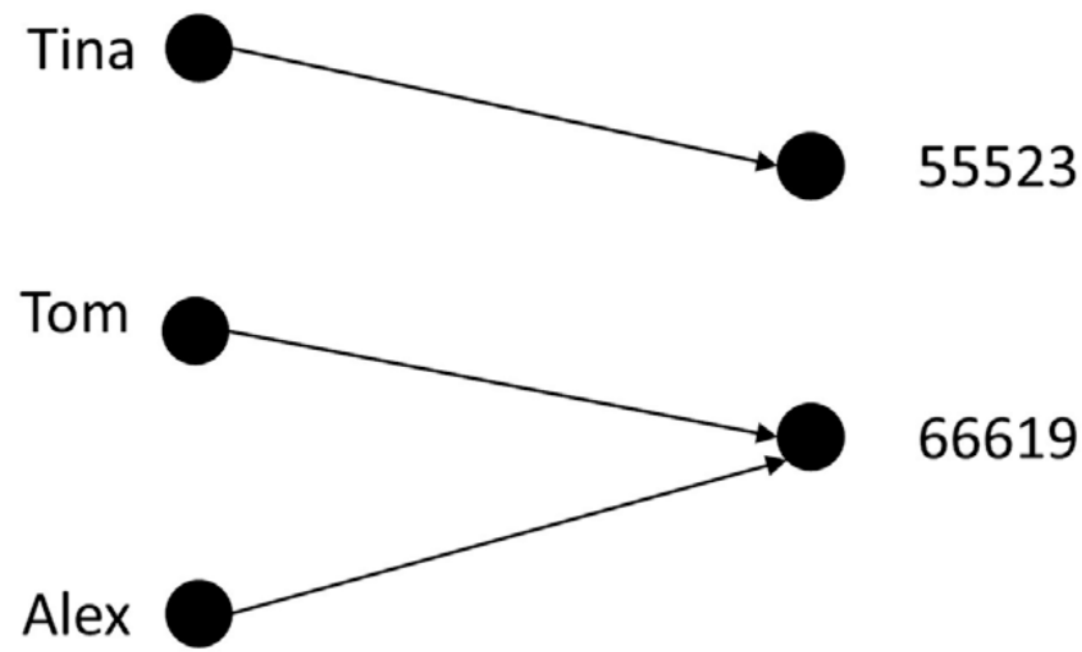


Figure 2.4 A relation between sets, represented as a graph.

Set theory as presented here is a powerful tool that allows to understand very different approaches in data modeling as doing basically the same thing—that is, offering a formal way to define sets and operations on sets. Even without any prior familiarity with formal set theory, most of it will seem intuitively plausible, an expression of common-sense concepts. The same is true for logic: it simply formalizes concepts that we use every day. The formal notation helps to avoid ambiguities and offers a compact way to express even complex relations clearly and precisely. On the other hand, we should point out that the version of set theory presented here is also called the "naive" set theory, in contrast to axiomatic set theory, which is more abstract. Axiomatic set theory came into being because naive set theory allows us to produce some rather baffling paradoxes. Probably the most well known is Bertrand Russell's famous paradox about the set of all sets that are not members of themselves. The intension of this set X is that all its members are sets and all of them are not an element of themselves. Now, if X is not a member of itself, then it clearly falls under its description: it is a set and not a member of itself. But if we say that X is a member of itself, it obviously doesn't fit the description of its members any longer. It is like the story about the difficult situation of a barber who only shaves those who don't shave themselves—if he does not shave himself, then by rule he ought to shave himself, but if he shaves himself, then he violates this rule. Axiomatic set theories like the one by Zermelo-Fraenkel define their premises in such a way that paradoxes like Russell's are avoided. But for our purposes, the naive set theory is good enough in most cases.

3.3 Sequences

Intellectual tools like logic and set theory allow us to specify sets and talk about the properties of these sets in a formal way. The concept of n-tuples extends the properties of sets by enabling us to describe an ordered collection or sequence. But tuples have their limits as a way of describing sequences, because in a typical set of n-tuples (as in the case of a relational database) all of the tuples are the same length. We can also generate open-ended sequences in a formal manner—

or instance, with a function that takes each element x_n as input and creates the next element x_{n+1}. But many of the types of sequences digital humanists must work with require open-endedness of a different kind. Humanities data is full of sequences that differ in length and form, but for which we can nonetheless identify a set of underlying rules or governing structures. For computer science, this area is of specific importance, because programming languages are a prominent example of this kind of formal communication. There are two common ways to specify this kind of sequence in a formal way: *regular expressions* and *grammars*. After outlining some of the basic concepts of regular expressions we will talk about one approach to describing a grammar, the extended Backus-Naur Form. A full coverage of regular languages or context-free grammars (which provide the theoretical background of regular expressions and grammars) is out of scope for this book, but will be helpful for anyone interested in a deeper understanding of the topic; see (Hopcroft, Motwani, and Ullmann, 2013) for a detailed introduction.

Regular expressions are a powerful tool to describe sets of strings. Essentially, they consist of a set of specially defined symbols that allow us to describe sets of characters and quantifiers. The following expressions describe one character and specify the set it belongs to:

. (i.e. a period)	any character
\d	any number
\s	any whitespace characters (space, tab, return, etc.)
\w	any word character plus numbers and underscore "_"
[xyz]	a self-defined set of characters consisting of the characters "x", "y" and "z". So [xyz] means one character, which is either x or y or z.

Quantifier operators indicate how many of the designated characters or sequences should be present. If there is no quantifier, it is assumed that the character or class in question appears only once. Quantifiers usually refer to the character or character class immediately preceding the quantifier. Examples of quantifiers are:

?	occurs 0 or 1 time
+	occurs 1 or more times
*	occurs 0 or more times

The notation takes a while to get used to; it is very compact but longer expressions can be hard to read. And it is not entirely standardized, although the syntax used by the scripting language Perl has been picked up by many other languages. Modern versions of regular expression engines usually work with Unicode; in early versions of regular expressions, the expression "\w" (any word character) referred to A-Z and a-z, but it now refers to all Unicode characters which are marked as word characters. The or-bar or pipe character (|) means

"or"—that is, either the symbols to the left or to the right must be part of the pattern. Here are some examples:

`[A-Z][a-z]+\s`	The first character is a capital letter between A and Z, followed by any combinations of lower-case letters, with any single whitespace character at the end.

This regular expression would match "House" or "Zug" but not "house" (no capital letter at the beginning) and not "Über" (while the first letter is capitalized, in Unicode the Ü does not fall between A and Z).

`U[Ss][Aa]?`	Matches "USA", "US", "Usa" but not "usa" and not "USsa" (only one "s" allowed at the second position).

We can also specify the number of occurrences using {n} or {n, m} where n is the minimal number of occurrences and m the optional maximum number. Take, for instance, a list of 4- and 5-digit numbers like this:

```
0123 2134 2200 34232 5390 9939 12129 2014 3911 3141#
```

We can extract with the following expression all 4-digit numbers, not including any number above 4:

```
[0-4]{4}[ #]
```

In the next example we are looking for all numbers not containing a 9 (the ^ denotes a negation). This expression says "any character except a 9, allowing 4 or 5 characters in a row":

```
[^9 ]{4,5}[ #]
```

All of these symbols (+, *, \, [], ^, etc.) can be classified as metacharacters, because they have a special operational meaning in the context of a regular expression. But what if we want to look for one of the metacharacters in the patterns we describe? In that case, we can use an escape character to signal that the next character is to be treated as a normal character in the pattern. This escape character in regular expressions is usually the backslash. So if we are looking for an international phone number, which usually starts with a "+", we could use a pattern like this:

```
\+[0-9]+
```

The first "+" has the escape character in front of it, so any number starting with + like +4955112345 will be found.

Regular expressions are part of most programming languages, of text editors, and of powerful Unix tools like grep, and they have even found their way to some degree into word processors.

Another way of describing sequences outlines a kind of grammar and views the rules of the grammar as productive or generative: they describe how valid structures—valid in the context of the language described by the grammar—can be produced. Regular expressions offer this generative quality in a very limited way: they permit us to express an open-ended set of possible patterns which the expression will match. But they do not provide a method for defining a grammar in the full sense: an interdependent and exhaustive set of rules by which a language may be defined and against which it may be tested. The system we are outlining in the following, extended Backus-Naur-Form (EBNF) was devised by two members of a committee working on the programming language Algol in the early 1960s and has been in use ever since. It describes a context-free language. A context-free language is a language in which all possible statements in the language can be described by a set of "production rules" or replacement statements: rules for replacing one symbol with another.

Ordinary language, the language of everyday life, depends heavily in its production and reception on context, and attempts to describe it with context-free grammars have been futile, so this approach works best in cases where we are processing very formalized communications without many implicit assumptions: in other words, when communicating with computers. The principal idea of the EBNF is easy. An EBNF grammar takes the form of a set of production rules that specify definitions or substitutions that may be performed to generate symbol sequences. Here is a sample:

```
A = B, C;
```

This production rule indicates that the symbol "A" may be replaced by the sequence "BC" (the comma indicates that the elements must appear in the order given). There are two kinds of symbols: non-terminal and terminal symbols. A non-terminal symbol may be replaced by its definition (i.e. the part that follows the equals sign) in any production rule. A terminal symbol defines any kind of string that cannot be replaced by something else (in other words, it terminates the definition or substitution process), so terminals cannot occupy the left side of a production rule, whereas non-terminals may appear on either side. The documents described by an EBNF consist only of terminal symbols, and the possible sequences of these symbols are fully described by the EBNF.

To generate expressions using an EBNF grammar, we take each production rule in turn and make the substitutions it stipulates, until we are left with only terminal symbols and the substitution process comes to an end. In some substitutions there will be only one possibility, but in others we may be given a choice. The following example defines a very simple language that allows you to produce sentences like "The dog bites the man".

```
sentence    = subject, predicate, object;
subject     = article, noun;
predicate   = "bites" | "loves";
object      = article, noun;
article     = "a" | "the"
noun        = "child" | "dog" | "cat" | "woman" | "man"
```

Everything in quotation marks is a terminal symbol. Everything not in quotation marks is either a metacharacter like the comma in the first line or a non-terminal symbol like "sentence." We can read the first line as follows: to produce a sentence you create a sequence beginning with a subject, then a predicate and then an object, all of them non-terminal symbols (the comma serves a concatenation symbol that also indicates sequence). The second line indicates that a subject is produced by an article followed by a noun. The predicate is created by either the string "bites" or the string "loves," both terminal symbols. We can also indicate optional elements using square brackets—for example, the following definition would also allow a sentence like "dog bites man":

```
subject     = [article], noun
```

Notice that this handling of optionality is quite similar to the "?" in regular expressions. In fact, there are also equivalents to the other elements of regular expressions: for example, with {} we can allow zero or more repetitions of the symbols between the brackets:

```
subject     = article, {adjective}, noun
adjective   = "small" | "big" | "cruel" | "beautiful"
```

These rules allow the production of sequences like "a small cruel man" and "a big beautiful small cat."

A complete description of the EBNF can be found in the ISO standard ISO/IEC 14977. It works well with any kind of context-free language which is linear, but it does not work with any other kind of patterns, such as electric circuit diagrams. And as usual, although EBNF is well defined, many variations can be found. The XML specification, for example, uses a simplified version which is actually an easy-to-read mix between EBNF and regular expressions.

4 Established approaches

From one perspective, it can appear that there exist infinite approaches to data modeling, since for each new data set there may be specific requirements for processing it, and also distinctive modeling requirements to foreground specific aspects of the data. But while this may be almost true on the level of the specific

data model, at the level of the metamodel things are much simpler. For most practical purposes, there exist three main metamodels: the relational model, XML, and graphs (including RDF). Data that cannot be modeled using one of these approaches is typically handled by creating a data model specific to the situation (which will also typically require software written for this specific data)—for example, using an object-oriented approach.

Even apart from the questions of design involved, the practical requirements for a workable metamodel are very significant: it must be a standard, in order to provide a stable and dependable basis for specific modeling efforts, and in the current information ecology that means it must also be an international open standard in order to enjoy wide adoption. It must typically also be accompanied by a suite of other standards that describe processes such as data creation, data manipulation, and data retrieval. The family of XML-related standards is a good illustration of the many aspects involved, as is the SQL standard with all its subdivisions. Nonetheless, the field of metamodels is still evolving, albeit slowly. In the last few years, for example, the demands of processing terabytes of data at high speed has not only produced new buzzwords like "big data," but has also yielded new data modeling concepts that are the foundation of the NoSQL databases. These new techniques still lack new standardized metamodels and hence currently lock users into specific software environments, but over time those standard metamodels may emerge, together with the accompanying families of supporting standards.

In the following, we will give a short introduction into some approaches. Our introduction will not be a replacement for a more detailed introduction into the specific metamodels, but it should provide a fundamental understanding of some of their core concepts. In particular, we will try to highlight the underlying logical and mathematical concepts and the kinds of processing they enable. And we will also try to give some hints when a specific metamodel is useful and when not, even if in most cases pragmatic considerations will also come into play in the choice of a metamodel. A short introduction into relational databases is followed by sections on XML and on graphs and RDF.

4.1 Relational models and databases

The relational model is one of the oldest metamodels and is used practically everywhere where computers are used. Many programming languages provide modules to access relational databases, and software companies developing relational databases like Oracle or IBM are among the biggest players in the market. The introduction into data modeling with the relational model is standard fare for computer science students. The process of developing a relational data model is usually divided in two steps: first, the creation of a conceptual model, and then based on that the creation of a logical model.

A conceptual model for a relational database is usually described by an entity-relationship model. It identifies all important entities of the universe of discourse—that is, the segment of the world which is to be modeled—and the important

attributes of these entities. The entity-relationship model (ERM) is a way to describe the structure and semantics of data which was first proposed in a now famous paper by Peter Chen (Chen, 1976) and since then developed—with some important changes and additions—into a design tool especially for relational databases. Its basic assumption is that "the real world consists of entities and relationships" (Chen, 1976, p. 9). To be exact: in the real world we identify entities like Charles Dickens or Frida Kahlo, which belong to a given entity type—for example, "artist." An entity type and each of its entities usually have one or more attributes—for example, the entity type "artist" could have the attributes "year of birth," "name," and "year of death." A relationship describes the relation between entities—for example, between artists and works of art. Chen also proposed a graphical representation for the ERM which has become very popular, because it provides a relatively easy and accessible way to visualize ERMs. Entities are referenced by rectangles and relationships by diamonds. The lines between these building blocks contain additional information—for example, the possible number of entities being part of the relationship. The relationship between artist and work of art would be modeled like this ("m" and "n" indicate any positive integer).

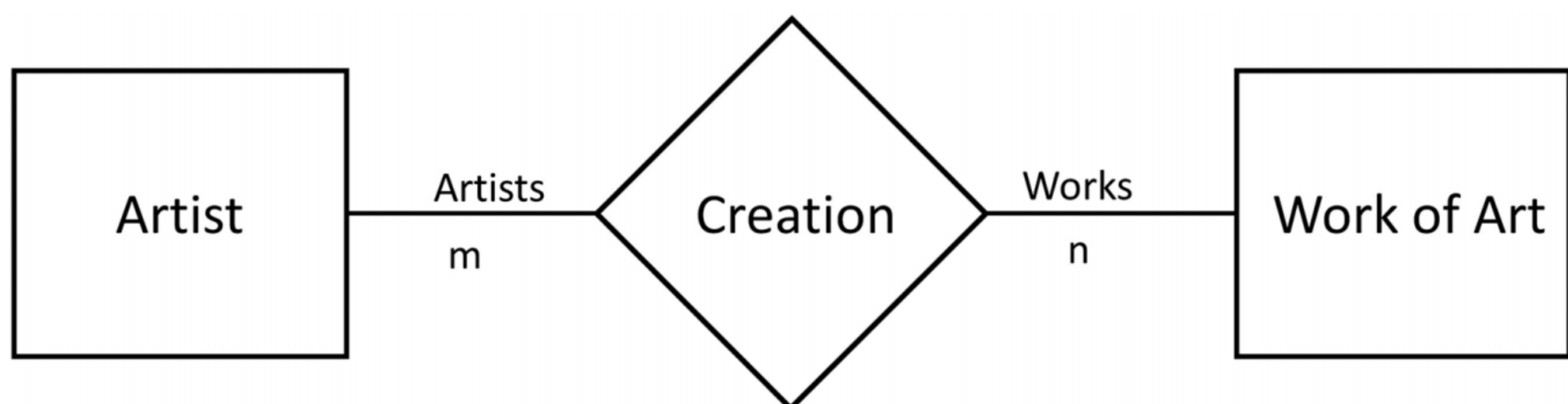


Figure 2.5 A sample entity-relationship model.

Modern variations of Chen's ER diagram often substitute the relationship symbol with a line. The so-called "crow-foot" notation, which is especially widespread, would describe the relationship in this way.

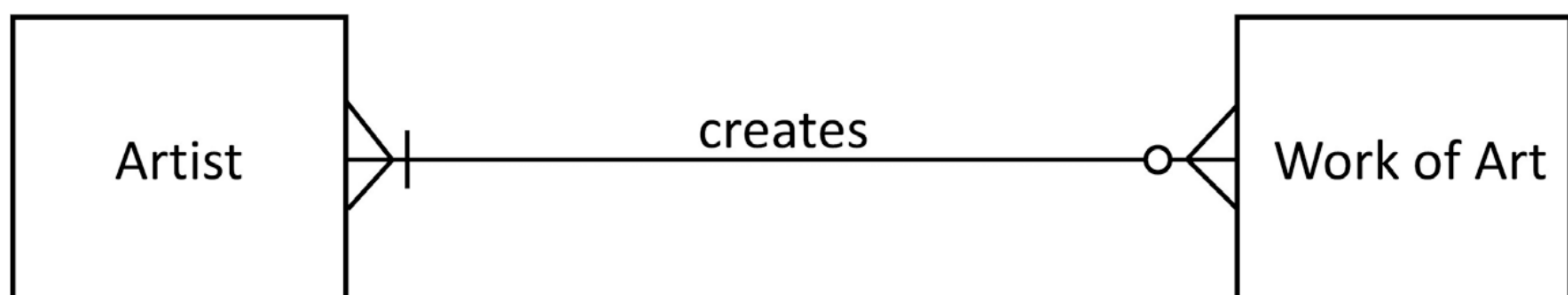


Figure 2.6 The same entity-relationship model, using "crow-foot" notation.

The vertical line indicates "at least one," the triangle with a middle line is the notation for "many," and the circle means "zero." So this diagrams says: "One or many artists create zero or more works of art." This aspect of ERMs is called the cardinality of a relation and it constrains the relation. For example, if we wish to model the relationship between the entity set "artist" and "place of birth," we would use cardinality to include the information that an artist can only have one birthplace.

There is no real standard for ER diagrams (the existing ISO standard has not established itself in practice as such) and there are many more or less similar implementations.

If one adds all attributes to this conceptual model, we have a visual representation of the logical model that can be transformed to a representation of the same model using tables. Every entity type in our ER diagram becomes a table on its own and every attribute of an entity becomes a column of the table. Now we can populate the table with entities described by their attributes (please see Table 2.6).

In 1970, E.F. Codd published a now famous paper outlining a mathematical model for this kind of information structure. At that time databases were ad hoc products with no coherent theoretical underpinnings; Codd's work provided a theoretical basis and an important field of application for a whole branch of mathematics, the *relational algebra* (Codd, 1970). Codd modeled the information in the column of a table as a set, so our table "Artists" would consist of three sets which are labeled Name, Year of birth and Year of death. And he conceptualized a row in a table as a tuple over these three sets: for example ("Charles Dickens", 1812, 1870). The table is thus a relation—that is, a set of tuples which are a subset to the Cartesian product of the three sets involved (Name × Year of birth × Year of death).

Table 2.6 A table with instances of an entity and its attributes

ARTISTS		
Name	*Year of birth*	*Year of death*
Charles Dickens	1812	1870
Frida Kahlo	1907	1954

The labels of the sets are called *attributes*—for example, the relation Artist has the attributes Name, Year of birth and Year of death. And for each attribute there is a set of possible values which are called the *domain* of this attribute. Each tuple represents a relationship between the elements of the tuple, in the context of a database we can also call them attribute values. At a given point in time, all tuples of a relation together specify all relationships.

We can view a table as a visualization of a relation, but it is important to remember that the ordering of the rows is of no consequence because a relation

is simply a set (hence unordered) of tuples. These tuples can be visualized in the table as rows. Each tuple represents an ordered sequence of information (describing one entity) and the ordering of the columns is significant because it corresponds to the order of the sets Name, Year of birth and Year of death and therefore to the structure of the tuples. But in the tabular visual representation, the order of columns is insignificant.

Obviously, there are many mathematically possible tuples that don't make any sense from a semantic perspective—for instance, all of the dates on which Charles Dickens was *not* born. One can view the attributes also as a logical expression; something like this:

$$\exists(x,y,z)P(x,y,z) \text{ for } P(x,y,z)\text{: } x \text{ is Name} \wedge y \text{ is Year of birth} \wedge z \text{ is Year of death}$$

This translates to mean that there are elements x, y, z in the domain such that P(x, y, z) is true, where x is a name, y is a year of birth and z is a year of death.

So the database represents the selection of mathematically possible tuples that also yield true statements, while all the other mathematically possible tuples (which do not result in true statements) are excluded from the database.

The relational model requires each tuple—each row in the table—to be unique. For many purposes we need to be able to identify each grouping of information (e.g. "Frida Kahlo -1907–1954"). We could use the field "name" as an identifier, but as mentioned above, entity names are often ambiguous (since there may exist in the world more than one entity of the same name), and therefore it is common either to use a combination of fields to construct a unique identifier or to introduce a new column with a unique identifier, the "primary key" (see Table 2.7).

Table 2.7 The same table with the addition of the primary key AID

AID	*Name*	*Year of birth*	*Year of death*
0	Charles Dickens	1812	1870
1	Frida Kahlo	1907	1954
2	John Smith	1662	1717
3	John Smith	1781	1852

If we have a table of works of art, we can now use the new identifier AID to refer unambiguously to the artist (and all information associated with that entity) instead of using the name (AID in the following table is called a "foreign key" in SQL) (see Table 2.8).

Because relations are sets, all set operations (such as intersection or union) may be used on relations as well. Additionally, the relational algebra allows powerful operations on the data and has become a core component of the Structured Query Language (SQL), the standard for relational databases. SQL describes

Table 2.8 Using a foreign key instead of the artists' names

WID	*AID*	*Title*	*Year of publication*
0	0	*A Christmas Carol*	1843
1	0	*Bleak House*	1852
2	1	*The Suicide of Dorothy Hale*	1938

how to query or manage data—for example, creating tables, adding or deleting information, retrieving information) in a declarative way: that is, it does not describe the procedure for achieving the results, but only the properties of the results. SQL provides a good example of the power of the separation of concerns: SQL offers an abstract data retrieval and data manipulation language, while the specific database systems take care of the way these instructions are implemented on the "physical" level of the model in the most efficient way—for example, the indexes that enable fast access to the information in the tables. From a data modeling perspective, this aspect is of no concern to us, so we can concentrate on the question how to work on conceptual and logical level. In the following, we will look at the way in which the relational algebra works and how this is realized in SQL.

The relational operator "join" allows us to combine two relations into a new one which contains all the columns of both input relations. An important prerequisite for a join is the existence of one attribute that is the same in both relations. We can join our two example relations because they share the attribute AID (the following describes what is called a "natural join," but other variants of join also exist). The resulting relation looks like this (Table 2.9).

Table 2.9 A natural join of the two relations

WID	*AID*	*Name*	*Year of birth*	*Year of death*	*Title*	*Year of publication*
0	0	Charles Dickens	1812	1870	*A Christmas Carol*	1843
1	0	Charles Dickens	1812	1870	*Bleak House*	1852
2	1	Frida Kahlo	1907	1954	*The Suicide of Dorothy Hale*	1938

Clearly, a lot of information in the resulting table is redundant. But often we need this kind of table for specific queries—for example, if we want to retrieve all works of art that were created when the artist was younger than 30. Entering and maintaining the data in this redundant format would be expensive and error prone, but performing a join between our two different tables enables the data to live in a non-redundant form and be combined only when needed for a specific query.

Because the relational algebra is such a powerful concept, a whole field has developed around how to model data in the most efficient way for a database.

Many SQL expressions can be reformulated as logical expressions (Date, 2011, Ch. 11). And logical expressions also play an important role as part of SQL expressions. For example, with the following expression we select those names from ARTIST who died before their 60th birthday:

```
SELECT Name
FROM ARTIST
WHERE Year_of_death - Year_of_birth < 60
```

As we have seen above, we can define a statement in predicate logic like this:

Let A(x1, x2) denote the statement "x2 – x1 < 60"

Then we can create propositions with different values for x_1 and x_2, taking these values from our tuples in ARTIST. And each proposition has a specific truth value, which decides whether the name belonging to the dates is part of the output or not.

Relational databases are the best tool, if your data is well structured and you need efficient queries based on different views of the data. Because there are so many tools to design databases and the existing database systems are among the most solid software systems in existence, they are often the first choice if data has to be systematically organized. But they are not very well suited for partially structured data like texts, and we will turn in a moment to modeling systems that can accommodate these less systematically structured data sets.

The advent of NoSQL databases has loosened to some extent the belief that only data in a relational database could be understood as truly structured, but nevertheless the domain of relational databases has produced the deepest and most systematic literature analyzing the data modeling process and trying to make it smoother and less error prone.

Data modeling of relational databases has been refined over the last five decades to achieve the most compact and less redundant representations. The motivations for this process have been twofold: for one, a compact data model will save storage space. When storage was expensive this was a crucially important argument, while nowadays storage is just one factor to be balanced with others such as the complexity of queries (since reducing redundancy tends to increase that complexity). The second argument is the probability that a change to the database—for example, updating or deleting data—will leave the database in an invalid state. If there are many redundancies in a database structure, there is a higher probability that a change at some point will lead to inconsistencies by changing an attribute in one place but not in others. Let us assume our relation is as shown in Table 10.

Table 2.10 Redundancies in a database structure

WID	*Name*	*Title*	*Year of publication*
0	C. Dickens	*A Christmas Carol*	1843
1	C. Dickens	*Bleak House*	1852

If we now decide to update the name of the artist from "C. Dickens" to "Charles Dickens," or even to "Charles John Huffam Dickens," we have to make sure that we do this for each of the entries with his name. If there are many entries, the probability of missing some of them is higher. But if in these entries, instead of the name, there is only a number pointing to another relation where the number is associated with a full name, then we would only have to change the name in that one place.

The process of eliminating all redundancies is called database normalization and is usually described as taking at least three steps. We will only discuss the first step here, because a complete discussion of normalization is beyond the scope of this introduction.

In the first normal form, an attribute should only have atomic values (and no compound values that contain multiple pieces of information).

Let us assume our relation looks as in Table 2.11.

Table 2.11 Using compound values

AID	*NAME*	*ACTIVITY*
0	Charles Dickens	Writer, journalist
1	Johann Wolfgang Goethe	Writer, scientist, politician

Here, the attribute ACTIVITY has compound values: it designates more than one activity for each entity. If we would like to retrieve all tuples that describe writers, we would have to parse each entry to see whether it contains somewhere the information "writer". What the first normal form requires is an attribute value as shown in Table 2.12.

Table 2.12 An attempt at a solution for the compound values

AID	*NAME*	*ART*
0	Charles Dickens	Writer
1	Charles Dickens	Journalist

Table 2.13 Splitting of the two relations to eliminate the redundancy

AID	*NAME*
0	Charles Dickens
1	Johann Wolfgang Goethe

Table 2.14 Splitting of the two relations to eliminate the redundancy

AID	*ART*
0	Writer
0	Journalist
1	Writer
1	Scientist

This cannot be the final solution, because it creates another redundancy, but it is a first step in the right direction. Now we split the relation into two separate relations to eliminate the redundancy (please see Tables 2.13 and 2.14).

When pursued thoroughly, normalization expresses each relationship with a single formal relation, and results in data that is as economically and elegantly expressed as possible, not only in conceptual terms but in mathematical terms as well.

4.2 Trees and XML

Texts have structure, but typically not of the closely defined, repetitive form that we associate with a table. The structure of text is governed by pattern that permits more variation, allowing alternatives, omissions, repetitions, and recursions. Texts are sequences in the sense defined above, and a grammar-based instrument like the extensible Markup Language (XML) is the most effective approach to modeling this form of open-ended structure. At the very beginning of the research into this kind of structure, some believed that it would be possible to create one general model usable for all kinds of texts: a single grammar that could describe all of the features observed in texts of all kinds. But it quickly became clear that such a system would be much too large to be attractive and would still leave many requirements unfulfilled. So the task had to be redefined: to develop a common notation system (to enable the development of a shared tool set), but to leave the specifics of an annotation schema to be defined within particular contexts of usage. The solution was found quite early in a three-tiered system which matured into what we know now as SGML/XML.

1. The *metamodel* is essentially an agreement (expressed as a formal standard) about notation and syntax, and a protocol for defining descriptive systems or "languages" that use this syntax and for defining their vocabularies and grammars. The shared syntax and notation permits the creation of tools to process the data,

while the freedom to create individual languages supports the necessary diversity of descriptive approaches. XML, like its predecessor SGML, thus specifies the delimiters that distinguish the markup from the content, and also defines a notation and syntax via the rules of well-formedness: the markup must be delimited using the correct notation (described below), all elements must nest without overlap, and there must be a single root element that encloses the entire XML document. Finally, the XML metamodel also provides a way of defining a markup language through the mechanism of the schema: a set of rules describing the features of a genre or class of texts.

2. The *data model*, or schema, defines types of documents by specifying the data elements they may contain and the sequences and structures those elements may form: in other words, the "grammar" of the document. The most widely used schemas for humanities research data are the Text Encoding Initiative (TEI), XHTML, the Encoded Archival Description (EAD), and (to a more limited extent) DocBook. Because XML can be used to express any kind of grammar, it is also used to express other information structures besides documents—for example, vector graphics with SVG, or the Resource Description Framework triples (RDF) discussed below. There are several different languages in which the schema itself may be expressed—for example, DTDs (which are also used in the XML specification), XML Schema, and Relax NG.

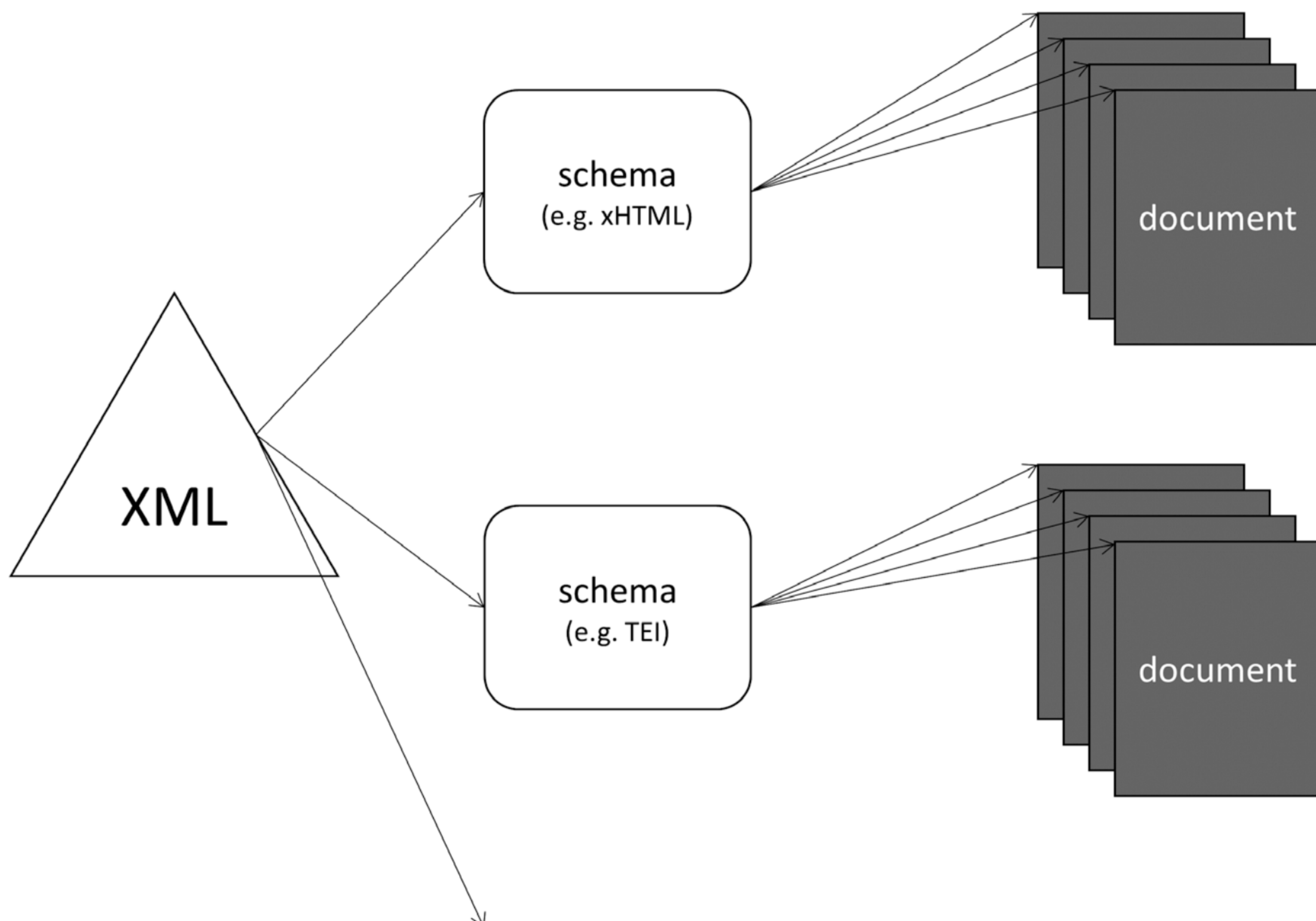


Figure 2.7 The relationship between the XML metamodel, a specific schema, and a schema-conformant document.

3. The modeled instance is the individual document, modeled using specific markup elements.

Figure 2.7 explains this relationship between the XML metamodel, a specific schema, and a document which has been marked up in conformity with that schema.

As noted above, XML data can also be created and used without a schema, and this approach makes sense in cases where some other constraint system is available (such as a data entry form), or where the data is simple and regular enough (or informal enough) that a schema is unnecessary. In the absence of a schema, XML documents must still follow the general rules of well-formedness (such as the nesting of elements), but there is no formal mechanism for describing and constraining the desired pattern of the data.

At the notational level, every XML document is identical—that is, it uses the same characters to delimit the markup, and it obeys the fundamental rules of well-formedness that are defined by the XML metamodel. Conformance with these rules is what enables XML software to operate on all XML data regardless of the specific schema. But at the level of vocabulary and grammar, every XML language—and the data it governs—is unique: this is what enables users of XML to accommodate the diversity of their data.

With this understanding of how XML is designed, we can now take a closer look at its building blocks. From the perspective of an XML parser, working through the document character by character, there are two basic information flows or states. Each character is either part of the “content” text or part of the annotation, the markup. In XML there is one character which functions like a switch: all following characters until further notice are part of the markup. This switch is the angle bracket “<”. And if the switch is on, the character “>” works like an off switch: the following characters are normal text. These characters function as markup delimiters and give the XML texts their typical look, with markup “tags” enclosing chunks of content. The informational effect of the markup is to create data “elements,” components of the text whose boundaries are demarcated by the markup. The following example shows how these delimiters are used in the start-tag and the end-tag which create the element “year” with the content “1867”:

```
<year>1867</year>
```

Opening and closing tags surround the text segment. The closing tag uses the same angle bracket delimiters with the addition of a slash to differentiate it from the start-tag. (In cases where the element is empty, a special compact notation may be used in which the entire element is expressed by a single tag—for example, a line break could be expressed as <lb/>.) The string enclosed within the tag delimiters is the name of the element. The meaning of an element, its semantics, may be suggested by its name and is formally defined in the documentation that accompanies the schema. The schema itself only describes the structural context of an element: what elements can come before it or after it, what elements it may contain, and so on. Elements are the basic building blocks of an XML annotation.

The characters between a start-tag and an end-tag are called the *content of the element*. This content can consist of text or other elements or a mix of both. As noted above, the rules of XML well-formedness require that elements must nest completely inside one another, with a single outer element enclosing the entire XML document.

Let us look at a slightly more complex example bringing all the above-mentioned aspects together:

```
<letter>
  <year>1867</year><lb/>
  <place>Albany</place><lb/>
  <text>Mr <name>Walt Whitman</name> . . . </text>
</letter>
```

We can see here two important aspects of XML documents. First, their structure is hierarchical, because elements nested within another element can be understood to be at a level subordinate to the enclosing element. This relation is usually described using metaphors from family trees, so <year> in this example is called a “child” of <letter> and <letter> is a “parent” to <year>. The <year> and <place> elements are “siblings” and <letter> is an “ancestor” of the <name> element. Second, the elements are ordered and the order is significant: <year> is the first child of <letter>, <lb> the second, <place> the third, and so forth. Thus, an XML document can be visualized as a tree. The small document above would look as shown in Figure 2.8.

Any XML document can be understood as a tree similar to this one, even if most of them are much more complicated. When we say “tree,” we mean a

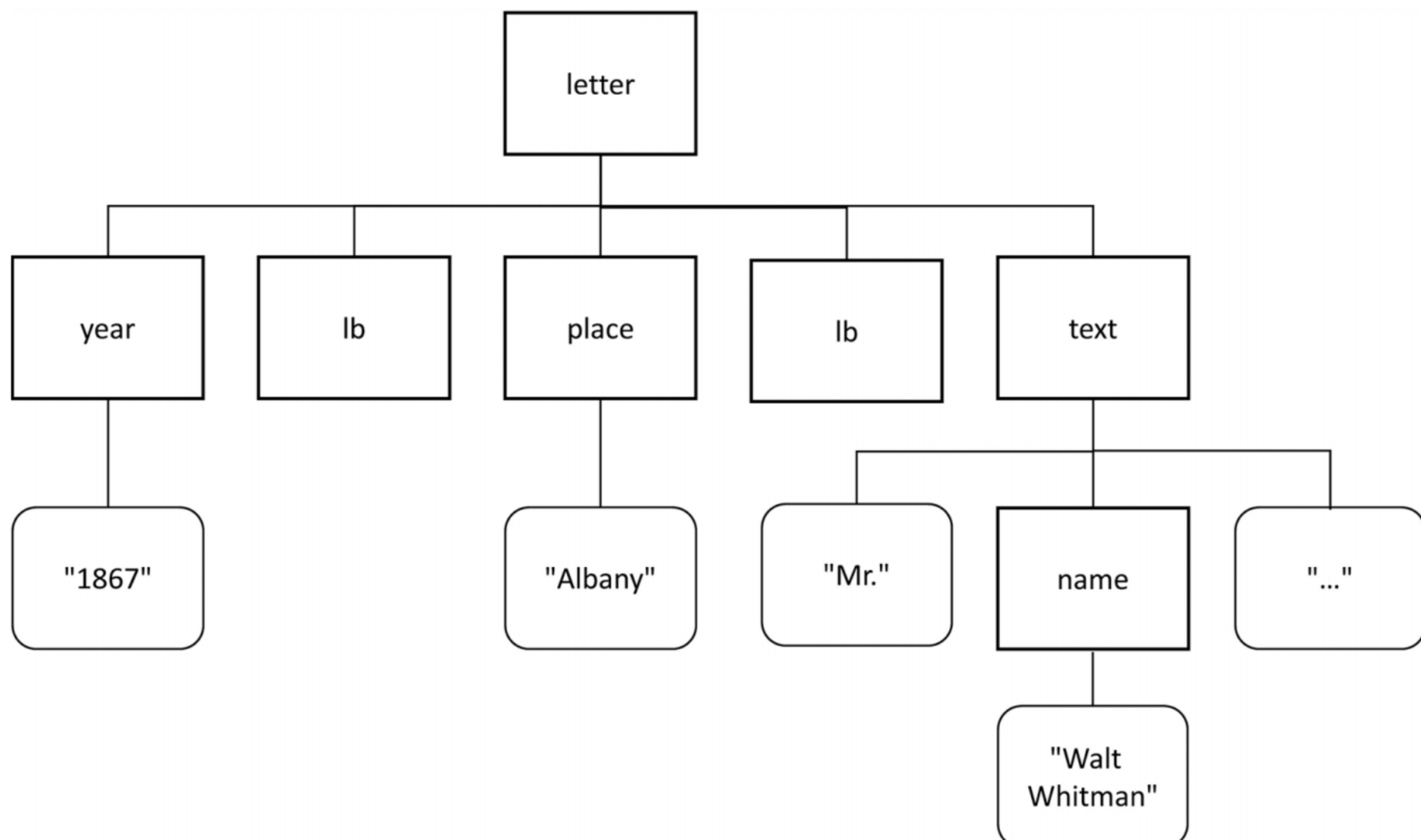


Figure 2.8 An XML document represented as a tree structure.

structure where all elements are part of one formation, and all components are either the root element or are connected with a parent component. And obviously the ordering of the branches of this tree is important, since it represents a text whose components are ordered. This structure enables a whole set of operations—for example, it allows us to restrict a search to specific parts of a tree, such as the part which contains the metadata, or the parts that are within footnotes. And it allows us to specify conditions for a search—for instance, to find only the <chapter> elements which contain a <quote> element.

Elements can be given further expressive power with *attributes*, which are a mechanism to attach name-value pairs to an element, providing additional descriptive detail. For example, here we specify the referent of a personal name by adding a reference to VIAF, the Virtual International Authority File, in the id attribute:

```
<name id="http://viaf.org/viaf/2478331">Walt
  Whitman</name>
```

An element can have multiple attributes, separated by white space. In the following example the <place> element has two attributes:

```
<place
gnis_id="977310"
coordinates="42.6525793 -73.7562317"/>
```

The attribute gnis_id refers to an authority file for place names, the Geographic Names Information System (http://geonames.usgs.gov) and the coordinates specify the latitude and longitude of the place. A more unambiguous approach would be to provide separate attributes for latitude and longitude, instead of a combined attribute that leaves the ordering and the meaning of the numbers unclear.

Above, we talked about how the angle bracket “<” works in XML like a switch between text and markup. As with all systems that use specific characters to delimit the markup, there is a problem in cases where these characters also appear in the text. For example, our letter writer might want to use a mathematical expression such as “3 < 5.” There are two common solutions to this problem. One has already been mentioned in the context of regular expressions, but is not used in XML: the special character can be preceded by an escape character that disables its specialized function. The second approach is a general mechanism which is a feature of XML and can be used flexibly for any character that cannot be included directly in the text. In cases where the character “<” is to be part of the normal text it is replaced by a character sequence “<” and any XML processor knows that this is a stand-in for the “<” character. These replacement sequences are called entities. XML defines five of them, but allows users to specify their own. Because the length of the replacement text is not specified, entities can also be used for boilerplate text. All entities start with the ampersand and end with a semicolon. The characters between them are typically a mnemonic: in this case “lt” stands for the “less than” sign or angle bracket. Since the ampersand works

as a metasymbol too—it signals the start of an entity—it is one of the other built-in entities and hence if we want to include a literal ampersand in the content of our text it must be expressed using an entity, as “&”. The other three predefined entities are not strictly necessary, but sometimes help to avoid confusion: > (greater than: >), " (quotation mark: "), ' (apostrophe: ’). A similar but more general mechanism, called a “numeric character reference,” exists by which we can refer to any Unicode character using its numeric code point. For example, the Greek capital sigma (Σ) can be represented by the numeric character reference Σ. Numeric character references may use decimal notation (as in this example), or hexadecimal notation; in the latter case, the reference includes an x preceding the code point (e.g. Σ).

The notation for XML also provides ways for the XML file to be parsed, processed, and interpreted as data in the environments where it will be used. To indicate specific processing-related information, XML may contain processing instructions; like XML tags, these are delimited (with <? and ?>) so that the parser can distinguish them from other parts of the markup. The name of the processing instruction defines its target application; for instance, PHP (as in <?php . . . >. In particular, processing by an XML stylesheet or schema is very common. The following instruction indicates that a Cascading Stylesheet (CSS) file named “common.css” is available to format the XML document:

```
<?xml-stylesheet href="common.css" type="text/css" ?>
```

Another crucial piece of information concerning the interpretation of the XML data is its character encoding: the mapping of the computer’s internal data coding onto specific character sets. The standard character encoding of XML documents is Unicode, UTF-8, which is essential for the circulation of data in an international context and across hardware and software platforms, but other character encodings are permitted, in which case the character encoding must be specified. This is done in the XML declaration: the first line of the file (not in fact a processing instruction although it is similar in form), which also specifies the version of XML being used. The following line specifies that the document makes use of ISO-8859-1, which has been used to encode different characters from western European languages, and the XML version:

```
<?xml version="1.0" encoding="ISO-8859-1"
  standalone='yes' ?>
```

As we mentioned above, XML files can be linked to schema files that describe their permitted structures, and the schemas can be used to inform us whether a document is conforming to the data model as expressed in the schema. The XML specification describes two states of XML documents: well-formed and valid. As we have already observed, a document is well-formed if and only if it conforms to a small set of general rules stipulated by the XML specification: elements have to be nested and the syntax for elements, attributes, processing instructions

and other XML structures must be correct. Data that is well-formed may be processed by XML-aware software, whether or not a schema is present. A document is additionally *valid* if there is a schema that specifies the grammar of the elements and attributes and the document conforms to this rule set as well. Software programs called XML parsers can be used to test whether an XML document is well-formed and valid, and XML editing software usually integrates such parsers to support authors in the process of data creation.

We can now look briefly at the schema itself. As mentioned above, there are different schema languages for XML and in the following examples we will use two in particular: the document type definition language (DTD), which is also used in the XML specification, and the RelaxNG format (RNG), because the latter is more modern and widely used and allows a tighter control of the schema. The following example shows the definition for an element called "place." It specifies that the content of a <place> element can only be text, with no internal markup allowed:

```
(DTD)  <!ELEMENT place (#PCDATA)>
(RNG)  <element name="place">
          <text/>
       </element>
```

It is easy to see that this is just a variant of the notation for grammars we looked at above. The #PCDATA (i.e. "parsed character data" or text) and <text/> components of the element definition can be understood as terminal symbols: when we process the grammar and make our substitutions, nothing further can be substituted for these patterns. The next example shows a very limited schema for a letter, which allows only a small set of elements and only one possible order in which they may appear:

```
(DTD)  <!ELEMENT letter (year, place, lettertext)>
       <!ELEMENT year (#PCDATA)>
       <!ELEMENT place (#PCDATA)>
       <!ELEMENT lettertext (#PCDATA)>
(RNG)  <element name="letter">
         <element name="year"><text/></element>
         <element name="place"><text/></element>
         <element name="lettertext"><text/></element>
       </element>
```

In EBNF this would be expressed as:

```
letter = year, place, text
```

For many transcriptional purposes, such a schema would be too minimal (because a typical letter contains other features) and also too strict (because many

letters contain different sequences of date and place); the following structure accommodates these realities more flexibly, by allowing zero or more occurrences of <year> or <place> before <lettertext>:

```
(DTD)  <!ELEMENT letter ((year | place)*, lettertext)>
(RNG)  <element name="letter">
         <zeroOrMore>
          <choice>
           <element name="year"><text/></element>
           <element name="place"><text/></element>
          </choice>
         </zeroOrMore>
         <element name="lettertext"><text/></element>
       </element>
```

We can also define attributes for the elements. Here we specify the attributes @gnis_id and @coordinates for <place>:

```
(DTD)  <ATTLIST  place gnis_id CDATA #REQUIRED
                 coordinates   CDATA #REQUIRED>
(RNG)  <element name="place">
         <attribute name="gnis"><text/></attribute>
         <attribute name="coordinates"><text/>
          </attribute>
       </element>
```

(In the DTD language it must be made explicit if an attribute is required, while in RNG it must be made explicit if an attribute is optional.)

CDATA is one of ten different keywords to specify the content of the attribute. That sounds a large number, but in fact one of the weaknesses of DTDs (one of the earliest schema languages) is that they do not support the wide range of data types that have emerged with the rise of XML as a standard: for instance, URIs, floating-point numbers, and many others. More modern schema languages such as XML schemas or RelaxNG provide much greater power in this respect.

The schema is usually a stand-alone file because it is meant to be used by multiple documents, so we need a mechanism for linking the document to the schema or schemas against which it is to be validated. This is done by using a processing instruction:

```
<?xml-model href="[file-location]" ?>
```

File-location can either be a link to a web-accessible file or a reference to a local file.

It is possible for a document to be valid against more than one schema, either simultaneously or at different stages of the workflow. In the former case, there

may be multiple sets of constraints in play: for instance, an internal schema representing the project's own constraints, and also a schema representing the data requirements of a collaborator or a repository to which the data will be submitted. In the latter case, there may be multiple schemas supporting different stages in a work process: the initial transcription with very basic structure, a later stage of annotation by content experts with more complex markup, a final publication stage in which metadata and details of publication are added. In these scenarios, the different schemas represent different perspectives on the same markup language: they each enforce or test for different subsets of the same language. In such cases we can simply reference all of the schemas in use at the top of the document, using the processing instruction mechanism we noted earlier. But there are also cases where a given document contains markup from two or more different markup languages, each with its own distinct vocabulary of elements. This is increasingly common as well-documented reference models are developed for specific domains: for instance, representing vector graphics, mathematical notation, music, or chemical formulas. A language like the TEI does not seek to duplicate those models, but rather assumes that a project needing to model such information will use the specialized languages developed and maintained by others. In these cases, we need a way to distinguish these languages and to indicate which elements in our markup come from which language.

To solve this problem, XML uses something called *namespaces*, which define distinct vocabularies of elements and allow them to be distinguished from one another. Usually they are declared in the root element of a document. Here we declare two namespaces, TEI and SVG, for our document:

```
<tei:TEI xmlns:tei="http://www.tei-c.org/ns/1.0"
         xmlns:svg="http://www.w3.org/2000/svg">
```

Each namespace declaration includes a prefix to be associated with elements in that namespace, so that in the document we can now explicitly state which namespace an element belongs to:

```
<tei:p>A rectangle looks like this:
<svg:svg x="100" y="100" width="500" height="200"/>
</p>
```

The namespace is especially crucial in cases where two XML languages use the same element name, but with a different meaning—for instance, the TEI language includes a <head> element which is used for section headings, and the XHTML language also includes a <head> element which is used for metadata. In a document that used both languages, the namespace prefix would make clear which element was being used in a given case, and hence how to assess its meaning.

Because one schema may predominate over others in our document, we can also specify this as the default namespace:

```
<TEI xmlns="http://www.tei-c.org/ns/1.0"
    xmlns:svg="http://www.w3.org/2000/svg">
  <p>A rectangle looks like this:
    <svg:svg x="100" y="100" width="500" height="200"/>
  </p>
```

Once a namespace has been declared for an XML document, it becomes an integral part of the XML language being used in that document, to the extent that a query without explicit reference to the TEI namespace (in the example above) would not show any results.

Thus far, we have considered XML mainly from a more technical perspective, focusing on its syntax and its implementation of the model of a context-free grammar. But XML and its predecessor SGML were invented to solve a very practical problem: the markup of text in all its variety and for many different purposes. In this process of developing SGML and XML (which have their beginnings in the 1960s), some basic concepts were introduced which are still important aspects of most models of digital text. One issue that emerged early on and has remained important is the question of how tightly the markup is aligned with specific tools and outcomes. In the early emergence of SGML, one of its important innovations was the fact that unlike many of its predecessors, its information was not aimed at instructing any specific application to produce a specific kind of output; instead, SGML focused on describing the structure and content of text in ways that could be used by a variety of applications for a variety of purposes. For this new role for markup, researchers developed the term "descriptive markup," used in contrast with "procedural markup" (e.g. PostScript or TeX), which communicates some directive to an application. Furthermore, in the context of descriptive markup, it became clear that if structural information is separated both from the procedural information that tells an application what to do, and also from renditional information that describes the intended output (e.g. a formatting stylesheet), the resulting data is exceptionally powerful. Markup languages like TEI take full advantage of both forms of separation, using XML's application independence to the fullest, placing their emphasis on structural and content features and allowing presentational details of output to be expressed as secondary information structures (in CSS or XSLT stylesheets, or more recently by embedding processing information in the TEI schema specification document).

Early articulations of this philosophy of descriptive markup (such as the line of debate stemming from DeRose et al. 1990) link the pragmatics of this scenario—the resulting ease of processing, flexibility of output—with its intellectual salience, arguing that descriptive markup necessarily focuses on what is informationally essential about the text. They take the point further to assert that texts always possess an identifiable informational essence that can be so represented and that hence serves as the natural pivot format from which other outputs can be derived. Languages like TEI have demonstrated quite thoroughly that our understanding of such an essence needs to be plural (to accommodate alternative ways of understanding the text as, for instance, a linguistic structure, an editorial

structure, or an expository structure). But nonetheless, for many purposes there is substantial consensus about the core information components of documents that are recognizable regardless of specific formatting and that seem therefore to transcend specific instantiations: in the terms of the Functional Requirements for Bibliographic Records (FRBR),[7] operating at the level of the "manifestation" rather than the "expression." However, over time it has also become clear that there are also classes of documents within which the concept of structure is unknowable or meaningless—for instance, a single word on a manuscript page—or which deliberately seek to destabilize our collective literacy in conventional document structures, as in the case of artists' books. Similarly, editorial approaches in which agnosticism about authorial intentions (and hence document structure) is important at certain stages of the editorial process have caused the TEI to introduce new forms of encoding (such as the <sourceDoc> element and its contents) in which the markup represents documentary appearance as its primary structure.

These approaches have historically been presented as counterarguments to the theory of descriptive markup articulated above, because they treat the identification of an ideal document structure as the key aspect of descriptive markup, and they offer a critique in which something else—document appearance, documentary materiality—is given primacy. However, from a data modeling perspective, it may be more useful to understand a deeper homology between the two. Of the two key insights of SGML/XML, the first—that markup should be *declarative rather than procedural* (in other words, should not speak directly to specific applications)—is now universally uncontroversial and almost beside the point here. The second—that "presentational" information can and should be separated from structural information and handled as part of the output processing—is more complex because of the assumption that appearance and presentational details are necessarily secondary to informational structure. But if we reframe that separation to mean that markup should *model what really interests us rather than what we plan to do with it*, things become clearer. With this perspective, we can acknowledge that information about materiality or appearance is not "presentational"—that is, not aligned with "what we plan to do with it"—if it constitutes the primary scholarly and interpretive perspective on the document in question. As the TEI's <sourceDoc> markup demonstrates, information about the physical zones and marks on the page is at times the primary information construct we need for our research. And the much longer history of usage of the @rend attribute to capture important details of the source—data about font choices, alignment, use of specific delimiters, and the like—has been substantially motivated by similar research needs. The thinness of the line separating <hi rend="bold"> and <bold> (and the difficulty of explaining it using the terminology of "procedural" vs. "descriptive") has always been a small embarrassment for the classic theory of descriptive markup, but if we understand it as a question of *what is being modeled* and the scholarly motivations for that modeling, that distinction is no longer necessary to draw. The legitimacy of such markup can be assessed not by deciding whether it is "descriptive" (good) or "procedural" (bad), but rather by determining whether it successfully "models what really interests us"

(a question of information design) and whether it successfully preserves our maneuverability with respect to the usage of the data (a question of workflow).

Like most strong metamodels, XML is surrounded by other standards that offer solutions for frequent tasks such as retrieval, conversion or rendering of documents. XPath provides a mechanism for navigating the XML tree structure and identifying sets of elements or other information nodes based on that structure. XQuery is a powerful and complex query language based on XPath with capabilities quite similar to SQL. XSLT is a programming language that can be used to transform and convert XML documents into other forms of XML or other data formats entirely; it is often used to convert XML documents into HTML. XSL-FO is a device-independent formatting language; XML documents can be converted to XSL-FO and rendered with an XSL-FO renderer to create a specific output, such as a PDF file. Most of these standards use XML internally as their own data format, and take advantage of its tree structure to support the identification and manipulation of document elements, using XPath.

XML is very widely used but has also received critiques, two of which are particularly prominent and are worth addressing here. First, XML is not only used for the markup of documents, but also has found a niche as an information exchange format in application programming interfaces. In these contexts some programmers prefer formats like JSON, as being less verbose than XML, and perhaps also because JSON more closely resembles data structures programmers are already familiar with like associative arrays (also called maps or dictionaries). Until recently, JSON had no provision for anything like a schema, giving XML an advantage in contexts where data constraint was needed, but the advent of JSON-schema may fill that gap: as formats for representing and interchanging highly structured data consisting of attribute-value pairs, JSON and XML may at some point be essentially equivalent. However, for representing semi-structured documentary data, JSON is inapplicable; it is not designed for this purpose and cannot be embedded into mixed-content contexts (where markup is embedded in text segments) in the way that XML can. Broad statements indicating that JSON renders XML obsolete tend to ignore this difference; XML and JSON should be understood as having specialized applicability.

The second common critique refers to the tree structure of XML, which some regard as a serious shortcoming; the most common version of this critique has to do with the difficulties arising around the representation of overlapping structures, which are common in humanities documentary contexts. This is a legitimate critique in an absolute sense: the hierarchical structure of XML poses design challenges for markup languages like TEI which are strongly document-oriented, although for more highly structured data this is generally not a significant concern. But as a comparison between XML and other possible data representation systems, the critique has less force. For one thing, an analogous limitation also exists in other systems. In databases, any content object can only be represented inside one data element at a time; XML provides greater flexibility here by permitting content objects to be nested inside one another, which databases do not. Formats like JSON suffer the same limitation as XML in this respect. The only data

formats that permit the representation of overlapping structures are either highly experimental—for instance, LMNL, the Layered Markup and Annotation Language—or obsolete—for instance, COCOA. The reason such alternatives have not become more widely adopted is that they change not only the data model but the whole ecosystem of data entry, data serialization, data processing etc. An approach which uses XML for data entry and data representation but not for all forms of processing—allowing overlapping hierarchies in some processes—keeps the strengths of XML and works around its main weakness, and therefore seems far more promising.[8] For the time being, while the critique of XML at a philosophical level remains a useful impetus to that research, as a rationale for or against the use of XML it is not useful. In practice, in the contexts where this is a pragmatic problem, such as TEI, well-established workarounds exist that permit additional information structures to be represented and processed.

In this section we have covered the basics of XML and briefly mentioned some of its related standards. For those seeking more detail, there are many good books and websites explaining them in more detail—for example, Harold and Means, 2004).

4.3 Graphs and RDF

Recently, networks seem to be everywhere. But it took a while to develop this unified way of looking at society, at streets, at Facebook groups, at internet connections, at letter writers in the eighteenth century, and much more. Important figures in the development of what is now called *social network analysis* were Jacob Moreno who led a small working group in the 1930s, and Harrison White in the 1970s (Freeman, 2004). They paved the way for a relational view of society and they were involved in developing the mathematical tools to describe these networks. Nowadays, these concepts are also applied to other kind of networks, like telephone lines, neural networks, streets and many more (Newman, 2010). The complexity of networks often poses a challenge for the calculation of aspects like the shortest path between two nodes or the probability of two nodes being connected in a simplistic manner, so the invention of fast algorithms, new ways to characterize aspects of the network, or efficient ways to visualize them are still a lively research field. In the following we will introduce some of the very basic ideas of the field, including graph theory (the mathematical basis of network analysis), and then discuss two applications in the field of digital humanities.

Graph theory provides the logical model for any kind of network analysis. Its basic building blocks are *vertices*, also called nodes, and *edges*, also called links. The railway system of a country could be modeled as a graph, with the cities as the vertices and the railway connections between them as edges. In the following we will depend heavily on figures to visualize graphs, but it is important to understand graphs as abstractions and the figures as tools meant to help to understand them. Some properties of the following figures are informationally insignificant—for example, the size of the nodes or the length of the edges) but, as we will see later, these properties can be used to convey additional information.

A *simple* graph like the one in Figure 2.9 can be used to model relations which are reciprocal, like friend relationships. Its edges are *undirected*: the nodes are connected but no additional information about the direction of the connection is given. The length of the edges is just an effect of the chosen visualization and has no meaning.The information contained in this graph can be fully expressed by a list of the connected nodes.

There are different ways to add complexity to graphs. One is to include information concerning whether the edge is *directed*. Another step is to allow more than one connection between the vertices, which is called a *multigraph*, and a third step is to allow vertices to connect to themselves by loops. Figure 2.10 shows a) a directed graph and b) a directed multigraph with a loop at vertex 4.

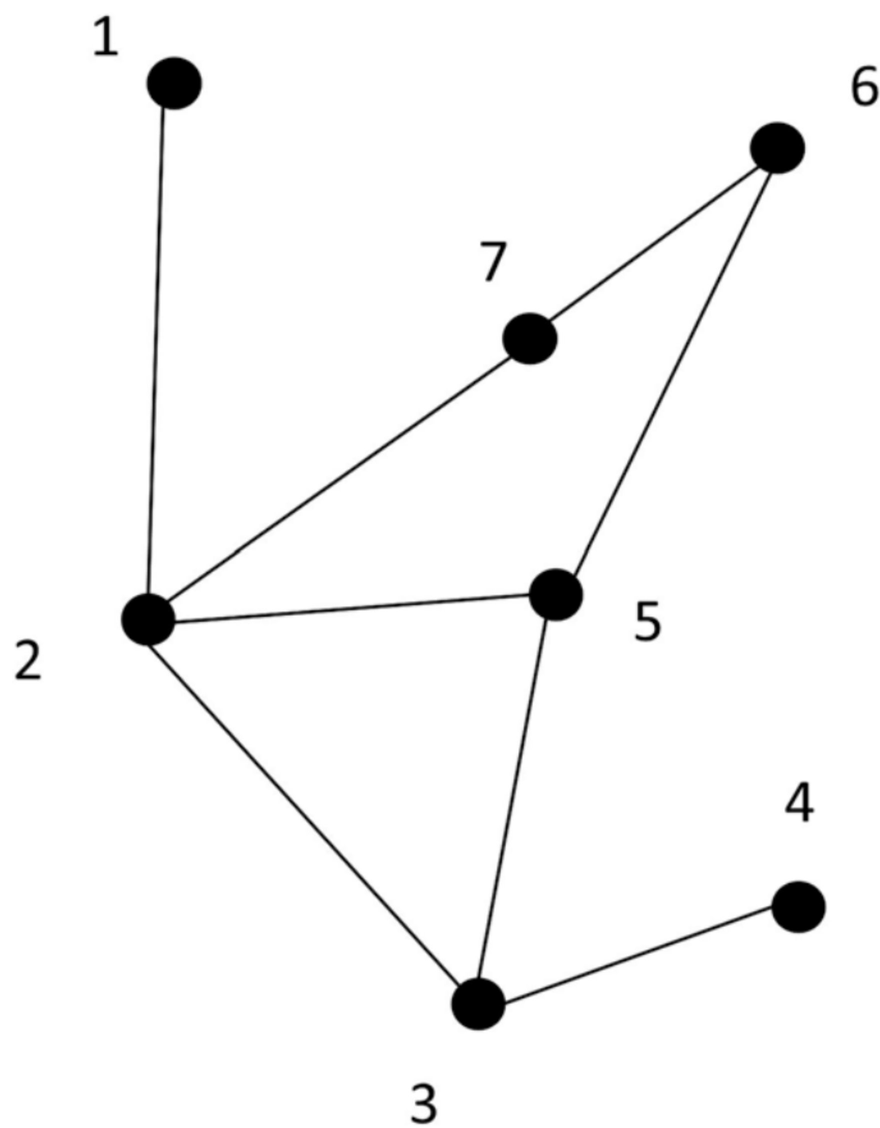


Figure 2.9 A simple network with 7 vertices and 8 edges.

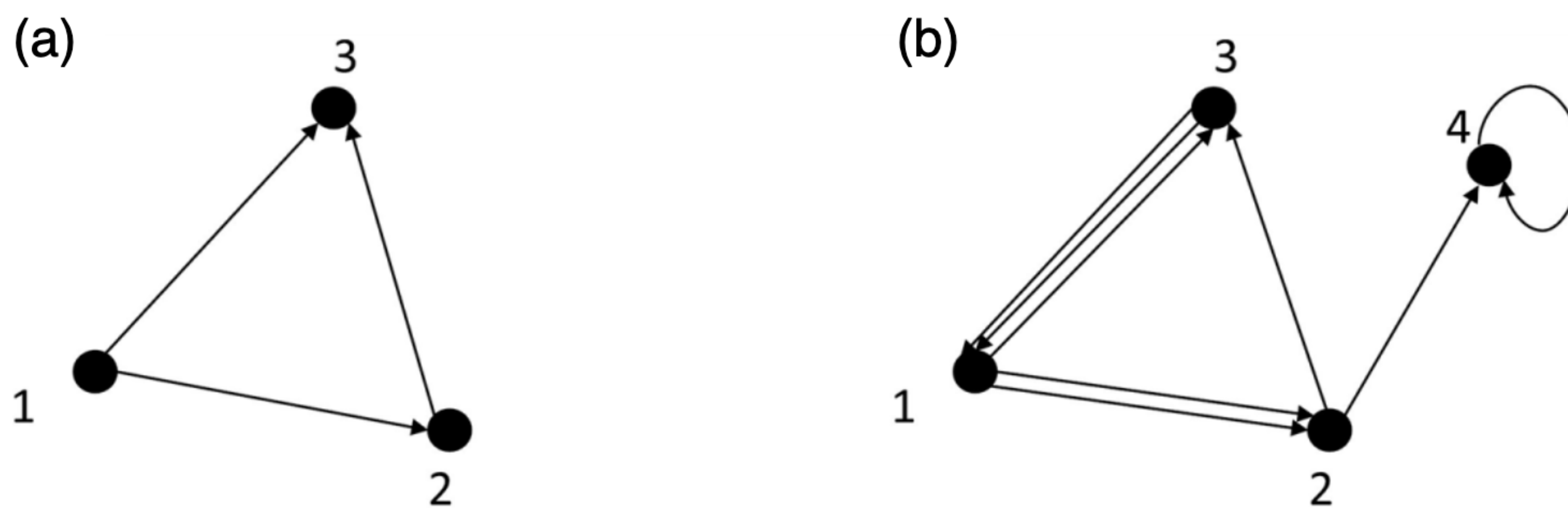


Figure 2.10 a) A directed graph, b) a *directed multigraph* with a loop.

What may look at first like a pedantic categorization—the labeling of a graph as a simple graph, a directed graph, and so forth—is important because behind each of these categories lies another formal definition of the graph, and each entails other operations on the graph. A directed graph D, for example, can be formally defined like this: D consists of a non-empty finite set V(D) of vertices and a finite set E(D) of ordered pairs of distinct vertices called edges. V(D) is the

vertex set and E(D) the edge set of D. So a graph can be defined by these two sets: D = (V, E). The second set consists of ordered pairs of vertices to express the directedness; if we replace it by an unordered pair, we have a description of an undirected graph.

In many applications of graph theory it is assumed that all vertices represent the same kind of entity—for example, an individual twitter user or a neuron in the brain. But sometimes we want to model relations between different kinds of entities—for example, authors and the literary societies they belong to. To represent these more complex networks we can use *bipartite graphs*. A bipartite graph is a graph that consists of two independent sets of vertices, where each edge only connects a vertex from one set with a vertex in the other set. We can then infer relationships between entities in one set, based on their shared connections to entities in the other set—for example, a group of authors who belong to the same club. In the following bipartite graph we have two sets of vertices, U and V, and each edge only connects vertices in different sets (see Figure 2.11).

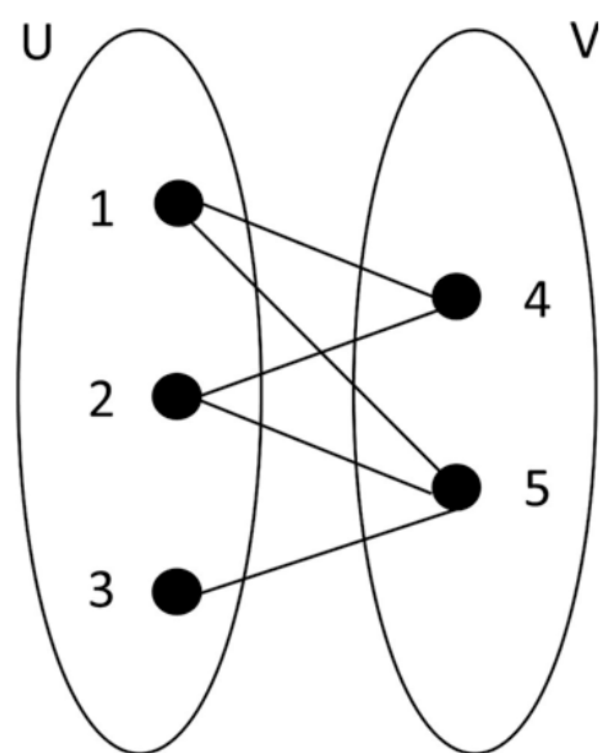


Figure 2.11 A bipartite graph with two sets of vertices.

Graphs can be used as data models because we can add additional information to this simple structure. A very common extension is to add weight to the edges. These weights can be used to indicate the strength of the connection. For example, in a network of people exchanging letters, the weights can be used to add

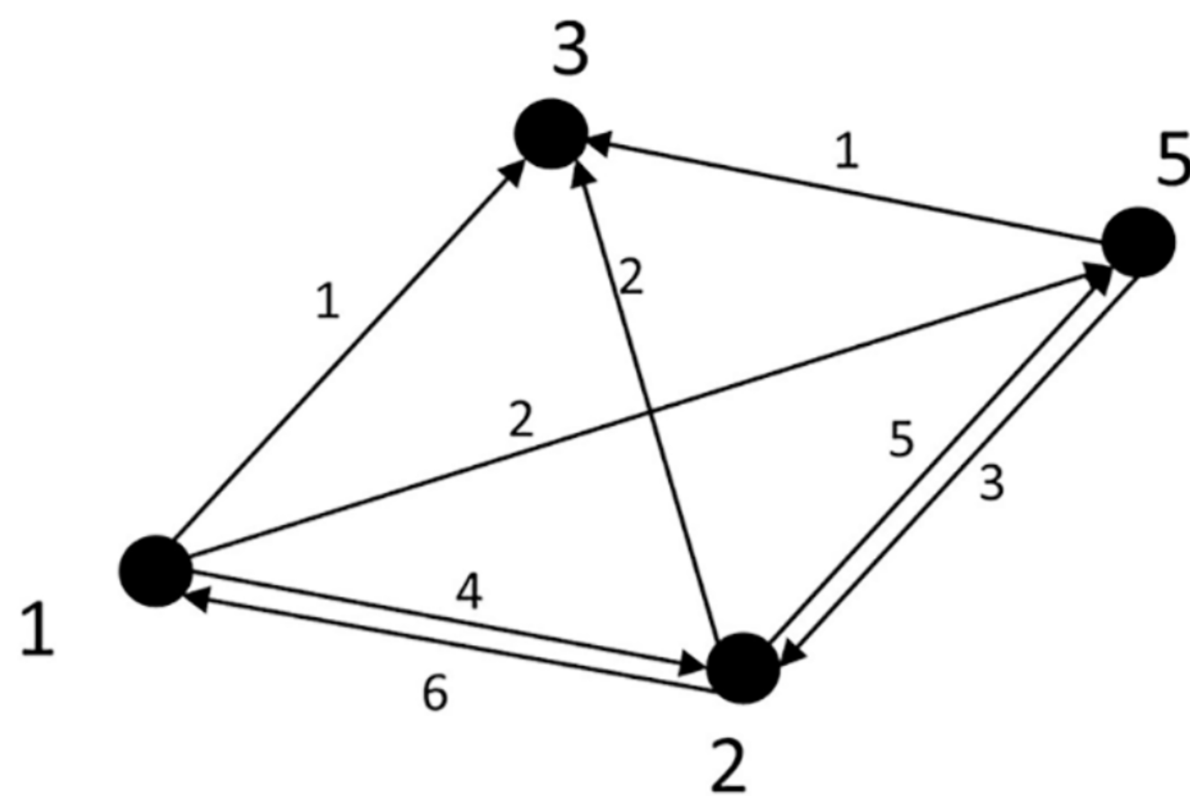


Figure 2.12 A directed multigraph with weighted edges.

information about how many letters have been exchanged. Figure 2.12 shows such a directed multigraph with weights.

So far, we have been assuming that the graph can represent different kinds of entities, but that information has not been an explicit part of our graph. We can add attributes to the vertices and edges to do exactly that. So we can add an attribute to a vertex with the name of a person or, even better, with a reference to an authority file like VIAF pointing to the name of a person.

There are many different kinds of graphs and many ways to classify graphs, but we will only mention a few specific graph attributes because they allow us to take a fresh look at something we already know. If we take a look at the three graphs in Figure 2.13, we can find many ways to describe how they are different from each other, but one aspect of particular interest is the path by which a vertex can be reached.

If you can find a path in a graph which starts at some vertex and then traverses a sequence of edges without using any edge a second time, coming back to the starting vertex, you have found a cycle. For example, in Figure 2.13a you can travel from 2 to 3, then to 4 and then back to 2. An acyclic graph is a graph without any cycles, like the graph in Figure 2.13b. If we use directed edges and allow only one root vertex we have a very special case of a rooted directed acyclic graph (Figure 2.13c). If, additionally, we claim that the ordering of the branches is important, we have an ordered, rooted, directed acyclic graph. This is the specific case of a tree we already know from our discussion of XML trees. Be aware that each feature we discussed in the context of graph theory usually only defines one specific attribute. The term "tree" in the context of XML is a compound statement implying a set of attributes like "having a root," "being ordered."[9] In other words, at this point we can connect our discussion of a tree as the logical structure of XML with the more general and more sophisticated discussion of graphs. Directed rooted trees are just a special case.

As useful as diagrams are for understanding graphs, they cannot be used as computational tools. So we must use some representational system which is computationally more tractable. A common representation of a network that serves this function well was developed before computers were used. It is an *adjacency matrix* which has a row and a column for each node. The number at

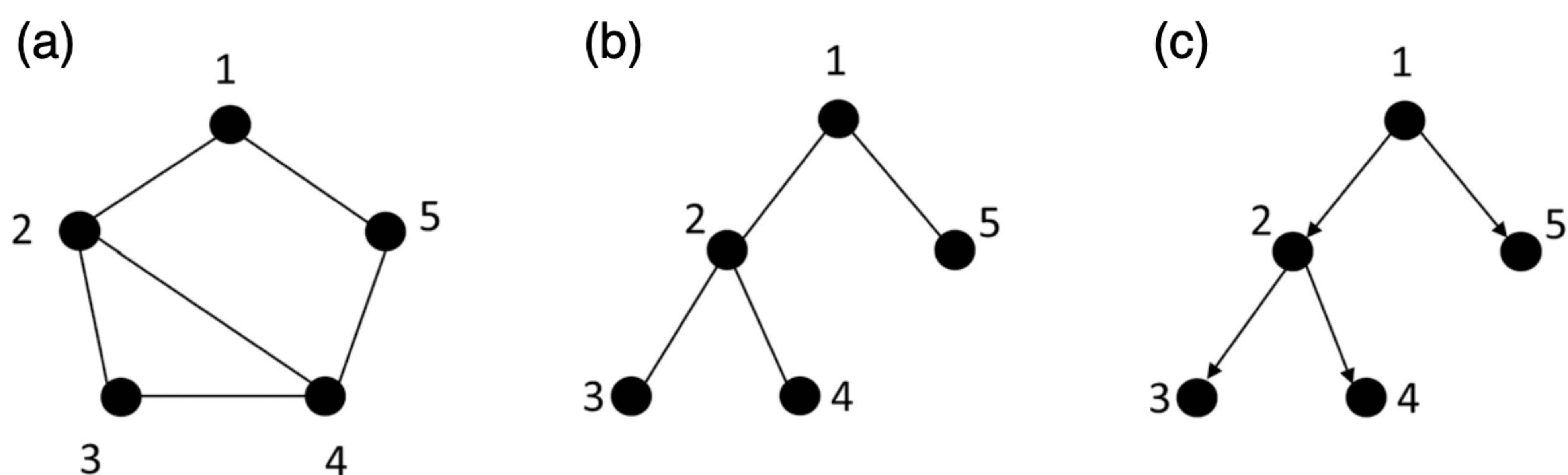
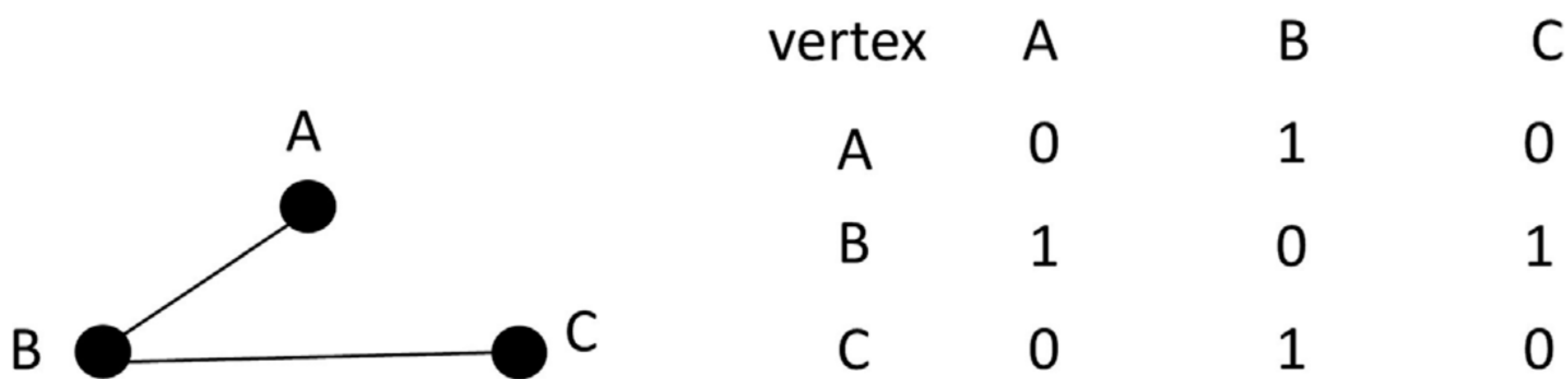


Figure 2.13 a) An undirected cyclic graph, b) an undirected acyclic graph, c) a directed acyclic graph.

the intersection indicates whether there is a connection between the nodes and optionally also gives information about the weight. For a simple graph—that is, a graph that is undirected and has no loops—the matrix contains a zero where the nodes are not connected and a one where they are connected. And because the edges are not directed in a simple graph, the matrix is symmetrical. The adjacency matrix for a simple graph like the following looks as shown in Figure 2.14.



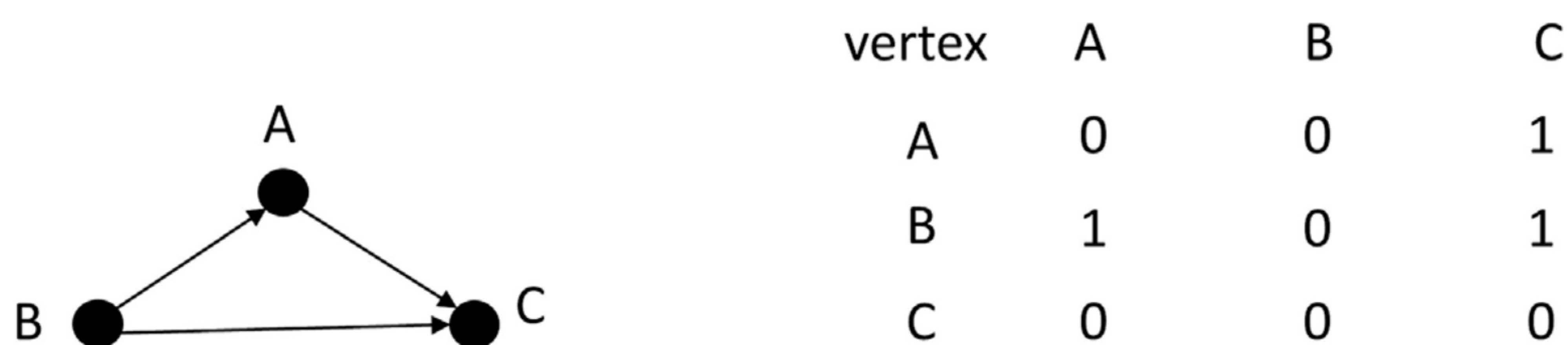
vertex	A	B	C
A	0	1	0
B	1	0	1
C	0	1	0

Figure 2.14 A simple graph and its adjacency matrix.

To describe an adjacency matrix a bit more abstractly: for a graph with n nodes, an adjacency matrix has n rows and n columns. An entry in the row *i* and the column *j* informs about the edge from vertex *i* to vertex *j*. So, in our example above, the row 2 and the column 3 show the edge between vertex 2 (B) and vertex 3 (C).

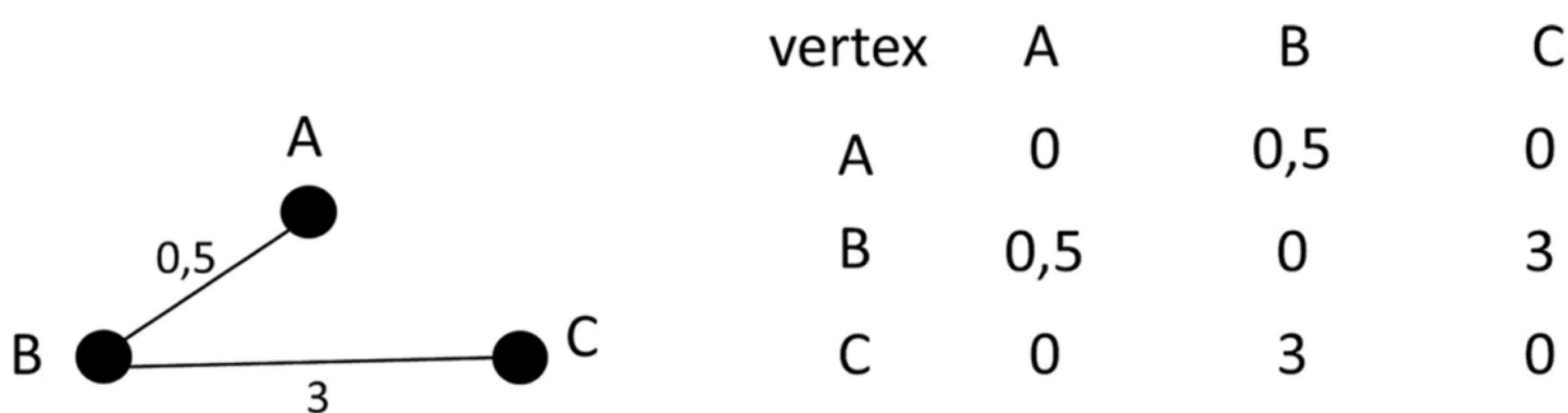
In an undirected graph, an edge connects vertices in both directions, so in our example A is adjacent to B and B is adjacent to A. In a directed graph, each node is only considered adjacent to the nodes you can reach by following the arrows, so the adjacency matrix shows fewer adjacencies; in the diagram below, A is adjacent to B, but B is not adjacent to A.

Figure 2.15 shows a directed graph and its adjacency matrix.



vertex	A	B	C
A	0	0	1
B	1	0	1
C	0	0	0

Figure 2.15 A directed graph and its adjacency matrix.



vertex	A	B	C
A	0	0,5	0
B	0,5	0	3
C	0	3	0

Figure 2.16 A weighted graph and its adjacency matrix.

The row determines the start of an edge and the column the end point, so row 2 connects to column 1 (B to A). As we saw above, a directed graph can also be used to represent a relation. And as the matrix represents the graph, we can also use it to represent a relation.

In a weighted graph, the matrix contains the weights instead of just the binary information that there is a relation (please see Figure 2.16).

Based on these very simple basic notions, a complex set of intellectual and computational tools has developed. Algorithms which can determine the shortest paths between two vertices are nowadays in use in all kinds of applications, such as navigation systems in cars where the map is represented as a network of cities linked by roads. In the digital humanities domain, an important application of network theory is the ability to model social networks and to use some of the available algorithms either to "describe" the network characteristics of specific vertices like persons or institutions, or to compare networks based on these characteristics. For example, *degree centrality* is one of the measures that enables us to express in formal terms our intuition concerning the most important nodes in a network: those that are most richly connected to others. It is essentially a count of the edges going to or coming from a specific vertex. In a network that models a group of letter writers, some people communicate with many, while others with only very few. One can, given the right circumstances, interpret those vertices with a higher degree centrality as being more important. In a simple graph, the *degree* is computed by counting all edges connected to a vertex. To permit comparison with other graphs, *degree centrality* is often normalized by dividing it by the number of possible connections, which is the number of all vertices minus one.

Other measures of centrality capture different aspects of the complex concept of importance. For example, *closeness* measures how close a vertex is to all other vertices, formalizing our intuitive understanding that a vertex is more important when it has many short connections—that is, when it has only few edges to traverse to connect to many other vertices. The Pagerank measure, made famous by Google, defines centrality for a vertex based on the importance of its neighbors; importance is based on the links to a vertex, but Pagerank also takes into account that a link from an important vertex is less important when it links to many other vertices at the same time.

The matrix notation is an important basis for these algorithms, not only because it allows fast computation of the results, but also because matrix techniques have been fruitful for the development of new network measures and algorithms.

We have given a very short outline of how graphs can be used as the underlying mathematical concept for networks. Even when the process of modeling networks as graphs seems sometimes almost natural, it is important to be aware that vertices and edges are abstract information units that can be used for any kind of modeling, and that any mapping from a conceptual entity and relation to a vertex or an edge is an intellectual activity by the modeler. A vertex has an obvious application for modeling self-contained entities like persons or neurons, but it can also be used to model any kind of concept—for example, events (such as the meeting of two

persons), or pieces of text (as in a hypertext), or concepts (as in ontologies and linked data).

An important application of graphs is the Resource Description Framework (RDF) developed by the W3C consortium, which is also responsible for the specification for XML and XHTML. The purpose of RDF is to express information about any kind of entity in a way that is readable by both humans *and* machines. This is done through formal statements consisting of a subject, a predicate and an object. Together they form a structure called a *triple* (please see Figure 2.17).

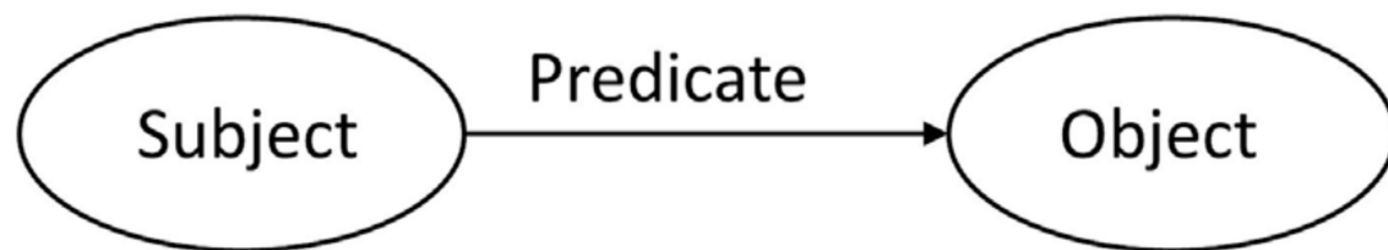


Figure 2.17 A graph representing an RDF triple.

Each statement is a small directed graph that ascribes one particular piece of information to the subject, for example:

[Walt Whitman]$_S$ [is born on]$_P$ [31.5.1819]$_O$
[Walt Whitman]$_S$ [is creator of]$_P$ [Leaves of Grass]$_O$
[Leaves of Grass]$_S$ [is first published in]$_P$ [1855]$_O$

The triples together form a labeled directed multigraph (please see Figure 2.18).

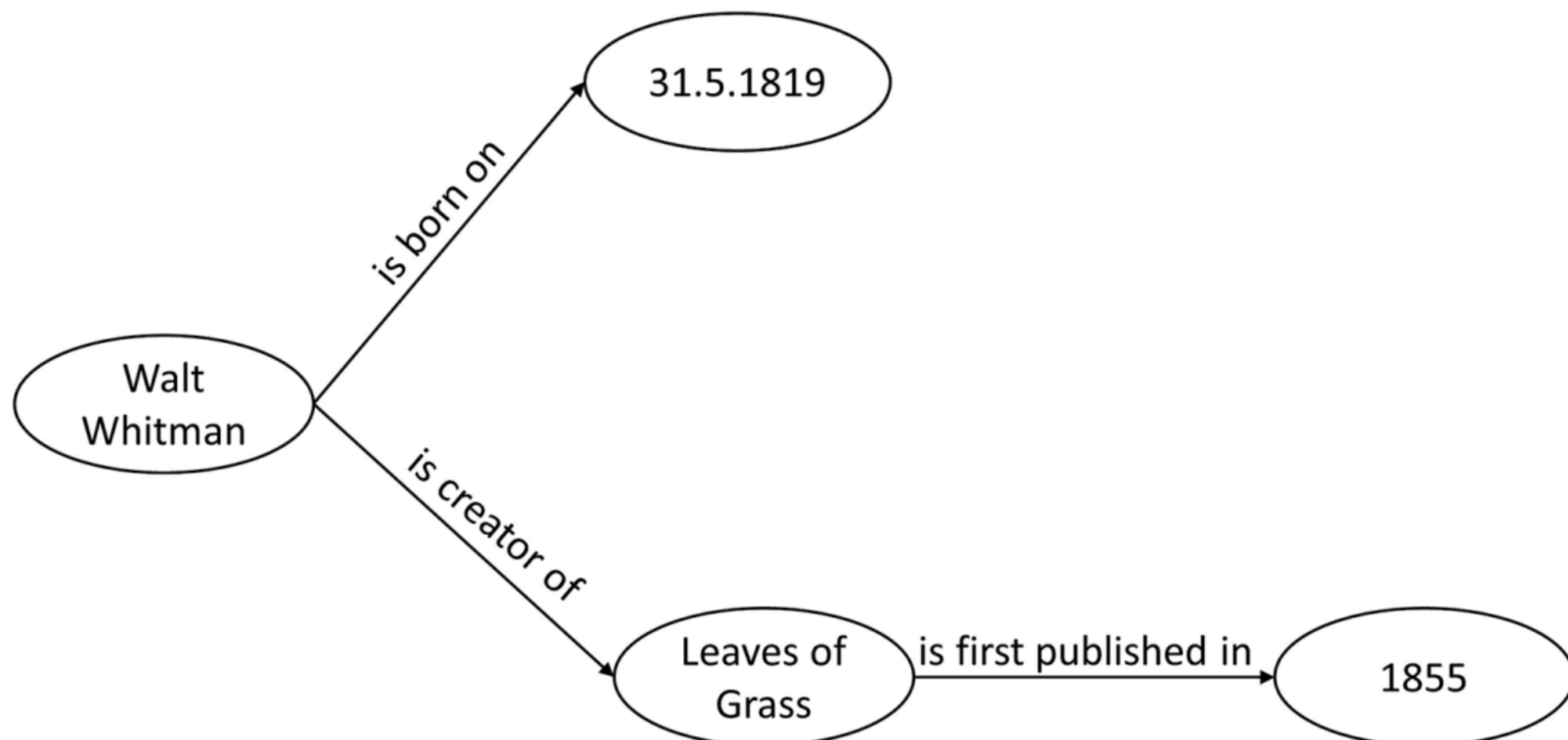


Figure 2.18 A graph representing the information in the three RDF triples about Whitman.

Because RDF is meant to be the language of a Semantic Web—that is, a collection of web resources which is not merely understandable by humans but can also be meaningfully processed by machines—the information about the subject, the predicate and the object is as far as possible expressed as an Internationalized Resource Identifier (IRI), an extension to URIs that extends

the set of permitted characters to include all of Unicode. So, instead of the string "Walt Whitman," we can use a unique identifier based on an authority file. Instead of the string "is born on," which could vary in many ways across different encoders in different languages, we can use a defined predicate from a schema registry like schema.org that provides formal definitions of concepts like "birth" and "death." And instead of our free-form date which might look different depending on the country the encoder is living in, we can use a string and attach the information that it is a date following the structure and formatting specified in a particular schema. So, our original statement that "Walt Whitman was born on May 31, 1819" can be expressed in formal terms thus:

```
[http://viaf.org/viaf/2478331]S [http://schema.org/
  birthDate]P ["1819-5-31"^^<http://www.w3.org/2001/
  XMLSchema#date>]O
```

Until now, for explanatory reasons, we have used our own system to notate the RDF triples, but the RDF specification by the W3C provides a series of formal systems, ranging from the very simple N-Triple's syntax up to rather verbose XML-serialization formats. The N-Triple is quite similar to what you have seen up to now. Re-expressed in that notation, our example would look like this (the triple would be on a single line in the file). The period at the end signals that the triple is complete:

```
<http://viaf.org/viaf/2478331><http://schema.org/birth
  Date> "1819-5-31"^^<http://www.w3.org/2001/
  XMLSchema#date>.
```

The vocabularies that provide the semantics of an RDF statement, such as birth or authorship, are not predefined; the RDF specification only defines a small set of information properties that specify abstract relationships between entities. So it is possible to express a given semantic concept in a variety of ways. But some vocabularies have been found to be especially useful—for example, Friend of a Friend (FOAF),[10] schema.org,[11] and the Simple Knowledge Organization System (SKOS),[12] which is also a W3C specification. RDF is meant to enable users to merge collections of RDF statements from different sources into one larger collection and mine this new collection for useful information, making use of simple inferences. There is also a retrieval language called SPARQL, which allows a user to query RDF data in a manner quite similar to SQL.[13] One of the largest RDF collections available is DBpedia, which extracts structured information from Wikipedia.[14] In particular, libraries have picked up the idea that their catalogues can be treated as a rich information set about books, and some have published these records as linked open data—that is, RDF data sets which are freely accessible on the web.[15]

An important further use of RDF, which extends and formalizes the establishment of specific vocabularies as described above, is in the description of ontologies. The term "ontology" has a long history in philosophy where it means

something like "the study of what there is," including the discussion of "problems about the most general features and relations of the entities which do exist" (Hofweber, 2014). But since the 1990s, the term has been appropriated by computer science to refer to something different:

> an ontology defines a set of representational primitives with which to model a domain of knowledge or discourse. The representational primitives are typically classes (or sets), attributes (or properties), and relationships (or relations among class members). The definitions of the representational primitives include information about their meaning and constraints on their logically consistent application.
>
> (Gruber, 2009)

An important aspect of ontologies in this sense of the term is that they express an interdependent set of concepts that constitute the important aspects of some information domain, from the perspective of a specific community. An ontology provides a basis for sharing large collections of data within the community that subscribes to its definitions. However, precisely because they express a specific community perspective, they are also often challenged on grounds that they fail to accommodate individual or alternative views. In some cases, they may also be quite complicated to use because community processes can produce specifications that integrate many different points of views in more or less elegant ways.

The above is not meant as a complete introduction to RDF, but is supposed to outline some of the basic ideas behind RDF with an emphasis on their relation to the underlying graph model. A good introduction into the details of RDF can be found in the W3C tutorial on RDF.[16]

5 The modeling process

When we turn our attention to the modeling process itself, we can view it as a series of steps with the ultimate goal of meeting a set of requirements specified by future users of the data, or people who act on their behalf. These steps have been formalized early on in the domain of relational database design, but they also apply more generally. The first step is *conceptual data modeling*: the identification and description of the entities that make up the model and their relationship in the "universe of discourse"—i.e. that part of the world a modeler is modeling—and notation of the findings, for example in an entity-relationship diagram. So, for instance, if we are developing a data model for a database of letters, the entities in question might be the document, its creator, its creation date, the repository that holds the source document, and the country in which that repository is located. The entity-relationship diagram might describe these relationships by showing that every letter has a creator, a date of creation, and a source repository, and that the repository in turn is related to its country of location. The second is *logical data modeling*: defining the tables of a database according to the underlying relational model. Following our simple example, we might have one table

representing the letters themselves (with their creation dates, creators, and repositories), and a second table might represent the repositories (with their locations and perhaps other information). The third is *physical data modeling*: optimization of the database for performance, in an actual implementation. There seems to be a consensus that this third step is usually done not by the data modeler but by an expert in database design. Ideally, both the conceptual and the logical model should be designed without any reference to the implementation, so that the implementation can be optimized or even replaced at a later time.

Although the distinction between the logical and the conceptual level may to some extent arise from specific database modeling techniques, it captures an important general aspect of data modeling. The logical model provides a structure for the data which allows the user to use a set of algorithms to answer questions of interest in relation to the data. For instance, in order to find all the letters held in a specific repository, we search for rows in the table containing the identifier for the repository in question. This computability is usually achieved by using a mathematical model: relation, in the case of databases. In this case, the logical model is a powerful formal abstraction, but it fails to represent most of the semantic information—that is, what we mean by "repository" and "creator" and "document." The conceptual model addresses this lack: it captures semantic information and offers an integral and embedded view of the data, organizing the information in such a way that the logical model can either be derived automatically or is at least very easy to derive. The distinction between the conceptual, the logical and the physical data model can be and has been applied to other areas of data modeling. We have followed this convention in our discussion of other forms of modeling, even if in most contexts it only highlights the fact that an equivalent to the conceptual model is missing.

In creating the conceptual model, an especially important aspect is abstraction. In its general meaning of identifying basic rules and features from specific examples, abstraction is a core element of all kinds of research and reflection. In computer science, it has some additional, more specific meanings. In relation to programming it refers to the principle of abstraction: the avoidance of repetition of code in two or more functions, by generalizing one function in such a way that it can be used in different contexts. In relation to data and programming, it is imperative to separate the abstract view of the data or program from the details of its implementation, and the approach to data modeling which distinguishes between the conceptual model, logical model and physical model is driven by this imperative. The separation of these layers allows us to abstract from the infinite complexity of the real world and use the conceptual model to outline the essential entities, attributes and relations, while deferring consideration of the details and constraints of the implementation. But, as with all forms of abstraction, this process is not innocent—there is no "natural" abstraction. What we deem to be essential is determined by the user requirements, but also by our world view, including our prejudices and our blind spots. And the tools of conceptual modeling have their share, as we all know, in forming the model, by making it easier to include one type of information and harder, if not impossible, to include another.

The goal of the modeling process, to fulfil the *user requirements*, also determines the relationship between the object and the data model. To understand this better, let us once again come back to our example of the letter. The text rendered at the beginning of this chapter is already a specific view of this object: a view that includes some kinds of information and disregards others. If we imagine sitting in an archive looking at the letter, we can perceive much more than just the characters on paper. We might note the size of the paper—perhaps it seems to be some standard size—and the stains on the paper which are visible at about the same place on both sides of both pages. The smell of the paper, its color, the way it has been folded and many other aspects could be part of our experience of the letter in an archive. In deciding what to preserve and what to disregard, we act on an understanding of the requirements of those who will use the modeled instances. If, for example, we are interested in creating a network visualization that will reveal the exchange of letters in a specific area at a specific time—for example, letters in Europe during the Enlightenment—then we will need some metadata about the author and recipient (such as their names and place of residence), and some metadata on the letter (when it was sent and received). If we are interested in a linguistic analysis of the language used in the letters, we will need the text of the letter itself and possibly also some linguistic annotation of components like tokens, sentences or part-of-speech information; if we are serving those who study the documents as material objects, we will need to capture information about the size, folding, and physical composition of the artifact itself. So, the intended usage of a digital entity like the transcription of a letter is the single most important factor to determine the selection, the amount and depth of the annotations and, more generally, the complexity and richness of the data model. Thus, a clear analysis of the requirements of the digital entities in question is an important step in data modeling. This is even true if the data being modeled will be only used by a single researcher interested in answering a specific research question, but it is critically important if the data will be used by a larger community of researchers or users in general.

In many ways, the process of creating a new data model or adapting an existing one is always unique, determined by the specific user requirements and the objects that are modeled. However, there are also some typical problem constellations and well-established solutions or design strategies that become familiar as one gains experience with data modeling. For example, we've already observed that by creating a distinct grouping of information *about* a data object (its metadata) we can facilitate retrieval of individual objects and management of groups of objects. Another basic modeling practice is the concept of *identifiers*: strings of alphanumeric characters that can be associated with data objects and used to locate and distinguish them uniquely. One further strategy in particular bears brief discussion here, because it enables us to create highly economical data structures: the concept of *indirection*. Imagine first our earlier example of the letter, which contains words that we know to be a name: "Mr Walt Whitman." Researchers often find names significant, so it would be of interest to add an annotation that makes explicit the fact that this is a name—for instance, thus:

```
<persName>
  <foreName>Walt</forename><surname>Whitman</surname>
<persName>
```

This simple encoding enables us to identify and extract all names from our text, but it does not tell us anything about the person named: their gender, birth and death date, and so forth, which is often crucial to the study of documents of this kind. We could expand our encoding of names to include this information:

```
<persName birth="1819-05-31" death="1892-03-26"
  gender="male">
  <foreName>Walt</forename><surname>Whitman</surname>
<persName>
```

However, if this name appears several times in the course of the document, it will be cumbersome and inelegant (and error-prone) to repeat all of this contextual information with each reference. Instead, we can introduce an element of indirection and create a data structure elsewhere that contains this information, assign each entry a unique identifier, and then use that identifier to associate the name in the text with the biographical entry we have created.

Within the text:

```
<persName ref="#WW01">Walt<persName>
```

Somewhere else:

```
<person  xml:id="WW01"
  birth="1819-05-31"
  death="1892-03-2"
  gender="male">
  <persName><forename>Walt</forename>
    <surname>Whitman<surname>
  </persName>
</person>
```

This mechanism is directly analogous to the creation of separate tables for distinct categories of information in a relational database, using an identifier to "join" the tables. And in our networked world this reference can also be a pointer to an external authority file which provides unique identifiers for entities like persons collected by experts somewhere else—for example, VIAF:

```
<persName ref="http://viaf.org/viaf/2478331">Walt
  </persName>
```

Indirection is an important design strategy for digital information, because it improves the cleanliness of data and its usability.

The spectrum of possible usages of a digital artifact is large and it is impossible to cater to all needs in equal detail. So how in practice do we choose the requirements on which a model is based? One can observe that there are mainly two approaches arising from different digitization communities. On one hand, when digitization is undertaken by archives, libraries, and publishers, the data model is typically understood as an interchange format aimed at serving the needs of a diverse (and largely unforeseeable) user population. We might characterize this as a *curation-driven* approach to modeling, which emphasizes the open-ended usefulness of the data rather than a specific research goal. On the other hand, in cases where data is being created to support the creator's own research needs, the data model functions to express specific research ideas; this approach is more common among individual scholars and projects, and we might characterize it as *research-driven modeling*. Curation-driven modelers must make assumptions about what features of digital objects are of interest for most users and in most use cases, while research-driven modelers typically concentrate more (though not exclusively) on the needs of their own project. Curation-driven modelers in particular should keep one point in mind: the empirical study of digital resources shows that data models based on needs expressed by future users tend to be overly complex; users do not always judge realistically how they actually use digital resources. In practice, most users tend to perform only very simple keyword searches, even if the search interface offers more complex options based on specific aspects of a data model (Connaway and Dickey, 2010). So a requirement analysis in larger projects should consider basing its model on empirical studies of user behavior.

Thus, we have in practice two different approaches to the task of modeling. One seeks to anticipate and synthesize very different views on digital objects in order to establish standards, and this involves very specific processes for deciding on these user needs and connecting these new models with existing traditions of modeling—for example, those arising in library science. The other is interested mainly in expressing as exactly as possible the theoretical assumptions and research interests of one or more scholars. Curation-driven modelers often use a data model which represents a small selection of generic features that will be important to most users, and that will also enable the objects to be rendered and published in a way that will satisfy many users. Research-driven modelers, on the other hand, tend to model with greater semantic specificity and complexity, making more precise distinctions between concepts and using a larger descriptive vocabulary, enabling more specific research activities.

Thus far, we have looked at data modeling as if from the perspective of someone creating the very first data model for a given domain, but in fact that will rarely be the case. After many decades of data modeling in computer science, information science and digital humanities, most domains have one or more established modeling systems for their central research objects. Approaching the question of how to model our own data, then, may be as much a matter of appropriation and adaptation as of creation. Sometimes we find a model that exactly fits our use case. More often, we find a model that we can adapt, or which is even designed to be

adapted and supports this by an explicit mechanism. Increasingly nowadays we can even find a *reference model*. Reference models are "generic conceptual models that formalize state-of-the-art or best-practice knowledge of a certain domain" (Becker et al. 2007, p. 2). They are often the results of many person-years of work and represent the knowledge of many practitioners in the domain. Thus, they often exhibit a complexity that can be daunting to a newcomer, who may wonder whether it is worthwhile to master the details of the reference model or better to simply design a model from scratch. However, it is worth thinking twice before building a completely new model. Most of the time the complexity of the models reflects the complexities of the data. What may look easy or even trivial in the beginning will become much less so if one had a closer look at the data with all its special cases and exceptions. The same is true for uses of the data, some of which become evident only further along in the design process. A well-designed reference model embodies detailed knowledge of how data in the wild can vary, and of how it is likely to be used. Those coming from outside the humanities fields may sometimes underestimate the complexity of historical and literary data, and prefer to start from scratch even if there is a reference model. Often, this approach seems to be justified by fast advances in developing an initial model, but as edge cases and less common examples come into view, the process reveals the challenges and difficulties that necessitated the complexity of the reference model. Before long, one finds oneself reinventing a system like TEI or CIDOC-CRM, but without the years of prior research and broad community input, and reworking the initial data that had seemed adequate at the outset. There are certainly cases where creating a new model is the right thing to do: for instance, in cases where a new theoretical approach is being proposed, or where the existing models suffer from known limitations. But in these cases, the responsible path to developing the new model is likely to carry the same responsibilities and degree of challenge: the need to involve domain experts, to reconcile diverse usage scenarios, and to consider the wild heterogeneity of real-world data.

There are two closely interconnected reasons why anyone doing data modeling should know about reference models: they serve as both the social and the technical aspects of formal information exchange. Reference models are not simply disembodied technical specifications: usually, there is also a community of experts working with them, developing them, exchanging ideas and spreading knowledge about them. Tapping into this font of expertise can be very useful for anyone interested in data modeling. Corresponding to this social dimension is a technical dimension: the possibility of *interoperability*. Basing a model on an existing reference model helps ensure that the resulting data can be connected as easily as possible to other data from the same domain—that is, it makes sure that the features the model has in common with the community's understanding are explicitly aligned with that understanding, via the reference model (which serves as a kind of hub). Adaptation of a reference model in this way is a common procedure in data modeling, and there are also other ways of reusing and referencing existing models. There is a useful literature that covers different approaches in detail (see, for instance, Becker et al., 2007).

6 Evaluation

There are many contexts where we need to evaluate data models. As part of their developmental workflow, digital projects have to decide on which data models to use, and if they adapt an existing model or develop their own, they need to evaluate its effectiveness. In a pedagogical context—for instance, in seminars on specific data modeling techniques like XML or relational databases, students will propose different modeling solutions, and teachers need to show the strengths and flaws in these approaches. Professional organizations seeking to recommend good practice to their members—for instance, the MLA's Committee on Scholarly Editions—need to assess available models not only in relation to their practical fitness, but also in relation to their long-term viability for the specific community.

Any evaluation of data models will take two different kinds of factors into account. First, there are internal factors that concern the properties of the data model itself—for example, whether it is consistent and free of contradictions, whether it covers the topic domain fully, whether it is scalable to different levels of use (including both simple and advanced applications), and whether it is complete and stable, or still under development and likely to change. In cases where there is a choice of data models, these considerations would inform that choice, but even in cases where there are really no alternatives, the assessment process may reveal areas where a weakness in the model can be compensated for, through adaptation or additional documentation. Along with these practical considerations, there are also some that have more to do with usability and aesthetics: Is the model succinct and economically designed? Is its architecture easy to understand and learn? These are probably not factors that would dissuade us from using a model that was the right choice in all other respects, but in identifying such weaknesses we can protect our work from their ill effects.

Second, there are external factors to consider. How does the model fit the user requirements in our specific situation? How well is the model supported with existing software? What are the costs of its application, in training, documentation, tool development, and maintenance? How is it situated in relation to other data models? Does it make use of existing standards whenever possible, or does it duplicate those standards? Does it support the creation of linked open data? Is it well supported by a strong user community? If the model is part of an ongoing research effort (and hence likely to change over time), are there open mechanisms for participation in that work?

Evaluating data models is from one perspective a highly practical matter. In this view, data models have to serve functions specified by the user requirements, and the key issue for their evaluation is how well they serve these functions. Success or failure in this case will be closely linked with the effectiveness of communication between a data modeler and the domain specialists who are supposed to work with the model at the end. However, with "curation-driven" or "archival" data models (which are developed with deliberate deferral or generalization of specific user requirements), we face a more complex situation: the dilemma between standardization and expressiveness, or, put another way, the fact that the better a model suits one specific case the worse it will fare in the

general case, and vice versa. Given the prevalence of these more "archival" data modeling efforts in the digital humanities, we need to consider how to evaluate such data models in other less practically driven ways.

Data models become more robust the greater the diversity of user requirements being considered in their design; in these cases, data models will cover more use cases and will be applicable to more situations. But making provisions for a broad range of specific user needs usually increases either the complexity of the model (the TEI Guidelines offer a striking illustration) or the level of generality at which the model operates (as in the case of a standard like Dublin Core). Though a more complex model will be more likely to cover more of the user requirements of any given project in its domain, there is still a theoretical limit to that likelihood; no model can cover all conceivable needs simply through added complexity. However, resorting to generalization carries its own risks. The more general a model is in the way it represents the world—that is, the more it relies on broadly defined concepts like "creator" rather than narrowly defined concepts like "translator" or "editor"—the greater the risk that users will find it lacking in the semantic specificity needed for meaningful communication. With very generally framed models such as Dublin Core, the model fails to express differences that are considered essential to specific research domains. Interestingly, neither over-complexity nor over-generality prevents users from working with an ill-fitting model. Instead, they will try to find workarounds: by appropriating categories manifestly meant for a different purpose ("tag abuse") or by compensating for overly general semantics through local usage conventions and documentation. These behaviors can themselves be used as a kind of evaluative index to assess the fitness of a model.

The distinction between what we called above "curation-driven" and "research-driven" modeling provides an important framework for evaluation, since these two modeling approaches entail different kinds of user requirements and overall goals for relationship between the model and the data. In the case of curation-driven modeling, long-term harmonization of the model with existing reference models is a high priority, and a looser fit between data and model is usually tolerated in the interests of achieving uniformity of workflow and consistency across the data sets being curated. With research-driven modeling, it is much more important to achieve a close fit between the data and the model (such that the model needs to be more complex and less general), and the tolerance for idiosyncrasy and even experimentalism is much greater; research-driven models may be based on (or may later be harmonized with) reference models, but the motives for doing so are curatorial. Similarly, the two approaches also entail different mechanisms for revising their models based on evaluation: an individual researcher might adjust a personal research schema incrementally based on new observations about a document set or the results of a test analysis, whereas a library digitization group or standards body would typically have a formal procedure for identifying changes, assessing backwards compatibility and long-term impact, requesting public comment, documenting the changes made, and disseminating the results.

While fulfilling user requirements seems to be the most important evaluative criterion, there is also another criterion that we would call tentatively the issue of "truthfulness" or "adequacy" of a model. The consensus in the digital humanities is that data modeling makes an interpretation of an object explicit—see for example, the TEI P5 chapter, "A Gentle Introduction to XML." Modeling does not simply mirror an external reality, but is an active process that depends on the social construction of a segment of the world. If we imagine a continuum with a radical subjectivist position at one end and a radical objectivist position at the other, digital humanities data modeling activities generally occupy a position in the middle that focuses on the role of social consensus and context-dependent negotiation of meaning. It is important to emphasize that this middle ground is as importantly distinct from radical subjectivism as it is from radical objectivism: it is incorrect to imagine that if we abandon the latter we are necessarily adopting the former. The role that social consensus plays in establishing our models is precisely to move them out of the grounds of purely private meaning into a space where meaning must be negotiated and must be intelligible and plausible to others in order to be useful.

For research-driven modeling, the argumentative emphasis is on the role of interpretation as a way of demonstrating the close ties between data modeling and other activities of scholarship, and evaluation of this kind of modeling will rest on whether the modeling reveals something unexpected or novel in the source materials. On the other hand, in curation-driven data modeling there is a stronger motive to emphasize shared modeling expectations, since the goal is to create data that will serve a broad and future constituency. The CIDOC-CRM, for example, describes its goal as being "to promote a shared understanding of cultural heritage information" (CIDOC-CRM, 2016). In these cases, the modeling of an object thus has to conform to the social construction of this object, and often a fruitful way to access this social construction is a closer look into older codifications of descriptions, such as standards of book cataloguing. In this perspective, we evaluate the model not by its truth-value, but with respect to how well it captures a shared understanding of some aspect of the world, independent of more ambitious theories of truth.

A third important dimension of evaluation for data models is their robustness—their ability to perform well and retain their expressiveness across different usage environments and applications. As we have already noted, good design practice attempts as much as possible to produce data models independent from specific processing contexts, although in practice it can be difficult to avoid some dependencies or anticipation of those contexts. This might be because there is currently only one context in which we can imagine the data being used, or because the usage requirements for the model revolve around a specific application (for reasons which may be social or institutional) in ways that exercise an overriding pressure on the design process. Bearing these pitfalls in mind, we need to be especially careful to distinguish information that is needed to support a specific workflow from information that will operate more generally regardless of the context of usage. The former can be represented as part of a processing model

(of which there might eventually be more than one to accommodate different processing contexts) while the latter is more properly part of the core data model that we expect to operate more universally. To evaluate the robustness of a model in a context where only one processing context currently exists, we may need to construct hypothetical scenarios: for instance, imagining future possible processing tools, adoption of our data model by different communities of users, or aggregation of our data within contexts that we do not control (such as institutional repositories). A final dimension of robustness which our evaluation should take into account is the ability to survive the rapid evolutions and revolutions of the field within which our models must operate, including changes to operating systems, software applications, data standards, and metamodels. In order to assess the robustness of our models in this respect, it may be helpful to look at historical examples and case studies to understand the vulnerabilities of specific modeling approaches.

On the first glance, choosing a data model may look like a primarily technical problem, but it soon becomes evident that it is embedded in social practices and relations. By using the framework of some larger standard like TEI or CIDOC-CRM, or the metadata scheme embedded in a tool like Omeka, one chooses a specific way of looking at digital objects, a way to discuss and even to evaluate strategies, and also a community of practice for those activities, even if one is not immediately aware of the full implications of that choice. And the reverse is also true: the decision to develop a project-specific modeling approach is also a decision concerning one's relationship to standards, and carries with it certain practical and social consequences concerning data longevity, shareability, and so forth.

There are also organizational factors that often play a decisive role in the choice of a data model—for example, the expertise of the people involved in a project. The best choice, considered abstractly, might nonetheless be a poor solution if the people who are supposed to perform the modeling lack the expertise to use it. In this last point, there may be a more complex cost-benefit analysis to be done: for a short-term project with few documents, the effort of familiarizing a scholarly team with a complex standard like the TEI might not be justified, but for a longer term project where data longevity is an important outcome, the cost of training is probably outweighed by other considerations such as sustainability and interchange.

7 Perspectives and challenges

Data modeling is not yet fully understood as a unified field with common underlying principles and concepts. Such a view would help us in the digital humanities to better apply and compare data models from different subject domains: for instance, to adapt a modeling approach (such as critical apparatus) from one domain for use in another, or to perform a comparative analysis of modeling approaches. But it would also yield a more deeply theorized understanding of our models that would translate directly into a stronger understanding of the distinct research value of our work. This book is a step towards such an understanding, but a significant amount

of intellectual work has yet to be done to achieve a full theory that brings all aspects of data modeling into this integrated view.

An important resource for this fuller understanding lies in ongoing research outside the immediate domain of digital humanities. There is ongoing research in specific fields—for example, on how to improve certain aspects of relational databases, XML or graphs, so as to extend the presentational effectiveness of the models, or the ability to translate between the models. Some of this research is taking place in standards bodies that are seeking to establish new guidelines or extensions to existing standards to solve problems that have become visible through usage. The W3C is an important venue for research and development in XML-related standards, and the working groups of the TEI regularly produce new additions to the TEI Guidelines which have broad significance for data modeling as well as immediate practical import for text encoding. Computer science, with all of its applied forms, is another important source of research impetus and produces general modeling approaches in advance of specific applications. It provides a stock from which others can choose solutions.

And there is also a steady stream of new problems for data modeling created by new perspectives on our research objects, which raise the question of whether new standards may be necessary. As cultural heritage institutions expand their mission briefs to curate an ever-wider range of challenging cultural objects—for example computer games and records of game play—new data models have to be devised to support this work. And current research on innovative forms of resource and content description using text mining will produce new metadata schemas integrating these new forms of information.

These three perspectives—the research on an integrated view on data modeling, the theoretically driven research on new options in data modeling, and the research driven by new needs in handling digital objects—will create new insights into the field of data modeling. This process has been going on for some decades now and has produced many of the concepts discussed in this chapter and this book in general. But recently there have also been shifts in the pragmatics of data modeling which may change the field profoundly—at least in parts. For a long time data modeling was done by humans who formalized expert knowledge. But with huge amounts of data available, and more and more sophisticated methods of mining them for structures by using machine learning techniques, algorithmic approaches to modeling offer better and cheaper results and scale more efficiently. However, although some of these models solve specific tasks very reliably, the kinds of data on which they operate are quite difficult to align with our human understanding of the data. For instance, in word embeddings, vectors in n-dimensional space represent words, and their proximity to one another expresses similarity, but it is not clear what the vectors themselves "mean," apart from the purely comparative analysis they yield. Furthermore, the resulting modeled data doesn't yield a higher-level understanding of the data (for instance, at the documentary level) that can be applied to the analysis of other problems. These approaches thus withdraw the data model from human analysis in a way that poses a challenge to data modeling as an intellectual (as opposed to computational) field.

The trend to data-driven data modeling has been mirrored by a similar shift in the specification of user requirements. Traditionally, user requirements were defined by domain experts, who were assumed to know best how information resources would be used. But as mentioned earlier, empirical research on the usage of digital resources showed a significant trend: the real use is most often much simpler than the usage anticipated and described by domain experts. In other words, domain experts have difficulties in predicting how much work their colleagues (both experts and novices) are ready to invest in learning interfaces and query languages. This insight has led to a shift to a data-driven analysis of the requirements in curation-driven data modeling.

Last but not least, we expect new impulses from data-driven data modeling for an old discussion: the semantics of a data model. Traditionally, the semantics of a data model are conveyed by the column headers of a table, the names of an entity in an entity relationship model, the name of an XML element, its description in a handbook and its content model in the schema, or the use of an element of an ontology in an RDF triple. Even if there is a formal ontology it doesn't really describe the semantics but aligns different objects (the set of alignments can be understood as a semantic representation on its own). This approach relies at some point on humans understanding the meaning of a concept as expressed in human language in these descriptive contexts. Some argue (as Stephen Ramsay does later in this volume) that the semantics of an element in a data model are constituted by the processes attached to this element. But only few would say that the semantics, let us say, of the concept of a "name" are really described by the ability to search for it in the contacts list in our phone and then press "dial." Processes using data models foreground the functions of individual data elements, but functions are not full descriptions of the use of a term, let alone its full range of possible use. Large data collections now provide new possibilities to describe semantics using ideas from the field of distributional semantics by describing words as vectors based on the analysis of many co-occurrences. Although these approaches have yet to solve some problems like the handling of word disambiguation, they do allow for a deeper understanding of concept relations—but they have the severe drawback that the data representation is still meaningless to humans. So one of the main cornerstones of data modeling from its very beginning would be threatened by these developments: until very recently, data modeling was understood as a matter of course to be an activity which produces formal descriptions usable by machines and humans. But now we start to see models which are more efficient for machines and incomprehensible to humans. How we handle this challenge will be the basis for most work done in the realm of data modeling in the future.

Notes

1 Cited after Folsom/Price: Walt Whitman Archive. Available at: www.whitmanarchive.org/biography/correspondence/tei/loc.01920.html Line breaks and other whitespace added using the reproduction of the letter. Available at: www.whitmanarchive.org/biography/correspondence/figures/loc.01920.001.jpg.

2 "Model" in: *Longman Dictionary of Contemporary English*. Available at: www.ldoceonline.com/dictionary/model_1
3 Available at: https://modelpractice.wordpress.com/2012/07/04/model-stachowiak/
4 If we choose instead to identify entities such as place names algorithmically, we are essentially relocating the modeling into the detection algorithm, which represents our understanding of what a "place name" is and how to recognize one.
5 Usually capital letters are used for the name of a set, while small letters are used for members of a set. Curly braces are used to enclose the members of the set.
6 The result of "a modulo b" is the remainder of the division of a by b—for example, 8 modulo 3 = 2. If x modulo 2 = 0, then x is an even number. Strictly speaking, the expression "x modulo 2 = 0" makes "$x \in \mathbb{N}$" superfluous.
7 See the Glossary for more information.
8 An example of this kind of approach to data modeling is the linguistic data model Text Corpus Format, which enables multi-layered annotations of text, where each layer is in XML. See https://weblicht.sfs.uni-tuebingen.de/weblichtwiki/index.php/The_TCF_Format.
9 There is also the term "tree" in graph theory which refers to an undirected graph without loops where each vertex is connected to any other by one path. Obviously, this is more generic than the term "tree" used in XML contexts.
10 Available at: www.foaf-project.org/
11 Available at: http://schema.org/
12 Available at: www.w3.org/2004/02/skos/
13 Available at: www.w3.org/TR/sparql11-overview/
14 Available at: http://wiki.dbpedia.org/
15 In the US, see for instance the Linked Data Service at the Library of Congress at: http://id.loc.gov; in Europe, see, for example, http://labs.europeana.eu/api/linked-open-data-introduction or the Bavarian State library, one of the largest libraries in Germany at: http://lod.b3kat.de/doc/download/
16 Available at: www.w3.org/TR/rdf11-primer/

References

Becker, J., Knackstedt, R., Pfeiffer, D., and Janiesch, C., 2007. Configurative Method Engineering—On the Applicability of Reference Modeling Mechanisms in Method Engineering. In: *AMCIS (Americas Conference on Information Systems) 2007 Proceedings*. Paper 56. Available at: http://aisel.aisnet.org/amcis2007/56 (accessed August 20, 2016).

Chen, P., 1976. The Entity-Relationship Model—Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1), pp. 9–36.

CIDOC (International Committee for Documentation), 2016. *The CIDOC Conceptual Reference Model (CIDOC CRM)*. Available at: www.cidoc-crm.org/.

Ciula, A., and Eide, Ø., 2007. Modeling in Digital Humanities: Signs in Context. *Digital Scholarship in the Humanities*, 32(suppl_1), i33–i46.

Codd, E.F., 1970. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*. Available at: www.seas.upenn.edu/~zives/03f/cis550/codd.pdf.

Connaway, L.S., Dickey, T.J., 2010. *The Digital Information Seeker: Report of the Findings from Selected OCLC, RIN, and JISC User Behaviour Projects*. OCLC Research. Available at: www.jisc.ac.uk/media/documents/publications/reports/2010/digitalinformationseekerreport.pdf.

Date, C.J., 2012. *SQL and Relational Theory. How to Write Accurate SQL Code* (2nd ed.). Sebastopol: O'Reilly.

DeRose, S.J., Durand, D., Mylonas, E., and Renear, A.H., 1990. What is Text Really? *Journal of Computing in Higher Education*, 1(2), pp. 3–26.

Edgington, D., 2008. Conditionals. In: E.N. Zalta (Ed.) 2008. *The Stanford Encyclopedia of Philosophy*. Stanford, CA: Stanford University. Available at: http://plato.stanford.edu/archives/win2008/entries/conditionals/ (accessed August 20, 2016).

Freeman, L.C., 2004. *The Development of Social Network Analysis: A Study in the Sociology of Science*. Vancouver, BC: Empirical Press.

Gruber, T., 2009. Ontology. In: L. Liu and M.T. Özsu (Eds.) 2009. *Encyclopedia of Database Systems*. Berlin: Springer-Verlag.

Halmos, P., 1960. *Naive Set Theory*. Reprint 2015. Oxford: Benediction Classics.

Harold, E.R. and Means, W.S., 2004. *XML in a Nutshell* (3rd ed.) Sebastopol: O'Reilly.

Hofweber, T., 2014. Logic and Ontology. In: E.N. Zalta (Ed.) 2014. *The Stanford Encyclopedia of Philosophy*. Stanford, CA: Stanford University. Available at: http://plato.stanford.edu/archives/fall2014/entries/logic-ontology/.

Hopcroft, J.E., Motwani, R., and Ullman, J.D., 2013. *Introduction to Automata Theory, Languages, and Computation* (3rd ed.) Edinburgh Gate: Pearson.

Kastens, U. and Kleine Büning, H., 2014. *Modellierung. Grundlagen und formale Methoden*. Munich: Carl Hanser Verlag.

Makinson, D., 2008. *Sets, Logic and Maths for Computing*. London: Springer.

Newman, M.E.J., 2010. *Networks: An Introduction*. Oxford: Oxford University Press.

Partee, B.H., ter Meulen, A., and Wall, R.E., 1990. *Mathematical Methods in Linguistics*. Dordrecht: Kluwer.

Pollard, S., 2015. *Philosophical Introduction to Set Theory*. First printed 1990. Mineola, NY: Dover.

Renear, A., 2005. Text from Several Different Perspectives: The Role of Context in Markup Semantics. In: Nicolas, C. and Moneglia, M. (Eds.) *Proceedings of the 2003 Conference on Computers, Literature, and Philology*. Florence: University of Florence, pp. 25–33. Available at: http://ebooks.mpdl.mpg.de/ebooks/Record/EB000323452.

Rosen, K.H., 2013. *Discrete Mathematics and its Applications*. New York: McGraw-Hill.

Tal, E., 2015. Measurement in Science. In: E.N. Zalta (Ed.) *The Stanford Encyclopedia of Philosophy*. Stanford, CA: Stanford University. Available at: http://plato.stanford.edu/archives/sum2015/entries/measurement-science/.

Text Encoding Initiative (TEI), 2016. *P5: Guidelines for Electronic Text Encoding and Interchange*. Text Encoding Initiative. Available at: www.tei-c.org/release/doc/tei-p5-doc/en/html/.