Designing linguistic databases: A primer for linguists

Alexis Dimitriadis & Simon Musgrave

1. Introduction: What this is about

It is a commonplace, by now, to refer to the recent explosive growth in the power and availability of computers as an information revolution. The most casual of computer users, linguists included, have at their fingertips an enormous amount of computing power. Tasks such as writing a document or playing a movie now seem self-explanatory, thanks to the integration of computers into mass culture and more specifically to two related processes: On the one hand, to the development of sophisticated software specialized for such tasks; and on the other, to the emergence, even among casual users, of a common-sense understanding of what these tasks are and how they may be approached. It is the combination of advanced tools and an intuitive understanding of what they do that allows us to experience such software as nearly self-explanatory.

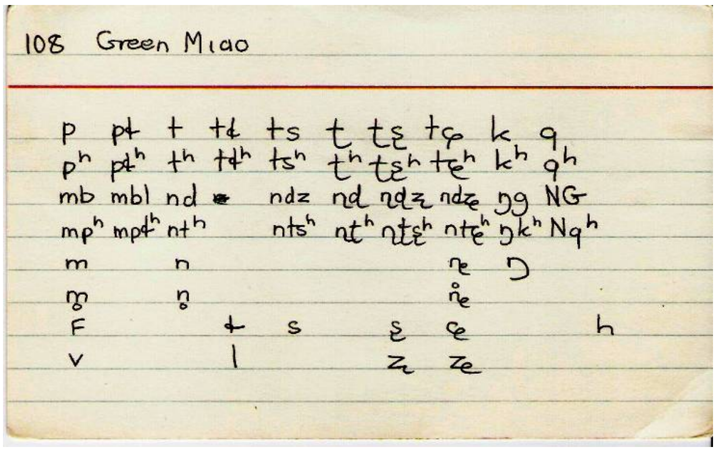
But the full potential of this new technology for linguistic research, or indeed for many other purposes, is still only beginning to be understood. Collecting and analyzing linguistic data is not like composing a text document – although many linguists, lacking a more appropriate paradigm, have no choice but to approach it as such. To fully realize the promise of linguistic databases, the subject of this book, it is necessary to understand the underlying concepts and principles; only then can the goals and problems associated with creating a linguistic database be properly identified, and appropriate solutions arrived at.

There are, of course, countless books on databases, at all levels of sophistication. But while a desktop database application is a standard component of "office" software, the world of databases revolves around commercial concepts such as inventories of merchandise, personnel lists, financial transactions, etc. A linguist who undertakes to understand databases is presented with documentation, textbooks and examples from this world of commerce, with few hints as to how such examples relate to the domain of linguistics. The principles, to be sure, are equally applicable; but while a cookbook approach suffices for adapting a sample database to a related use (for example, turning an example CD database into a personal book database customized for one's own needs), a linguistic database needs to be created from scratch, and (therefore) requires a real understanding of the principles involved. This chapter is intended to help provide this conceptual understanding, and to make its acquisition easier by using examples from linguistics and focusing on topics that are relevant to typical linguistic databases.

1.1 What are databases, and why do we care?

Technically, a database is defined as "any structured collection of data". The emphasis here should be on *structured:* a stack of old-fashioned index cards, like the one in the example below, is a database:[1] The cards are organized in a consistent way to indicate the language name, phoneme inventory, allomorphy rules, a code summarizing the size of the inventory, etc. In this day of digital computers, of course, the term *database* is generally reserved for digital databases; but while a digital database application has enormous advantages compared to old-fashioned pen and paper, it is still the *structure* of its contents, not the digital presentation, that is the essence of a database.

[1]The card is part of Norval Smith's Phoneme Inventory Database, which has been digitized and is available online as part of the Typological Database System at http://languagelink.let.uu.nl/tds/. (See Dimitriadis et al., this volume). Some of the information mentioned in the text is on the back of the card.



108 Green Miao

p	pɬ	t	tɬ	ts	ʈ	ʈʂ	tɕ	k	q	
pʰ	pɬʰ	tʰ	tɬʰ	tsʰ	ʈʰ	ʈʂʰ	tɕʰ	kʰ	qʰ	
mb	mbl	nd		ndz	ɳɖ	ɳɖʐ	ndʑ	ŋg	NG	
mpʰ	mpɬʰ	ntʰ		ntsʰ	ɳʈʰ	ɳʈʂʰ	ntɕʰ	ŋkʰ	Nqʰ	
m		n					ȵ	ŋ		
m̥		n̥					ȵ̊			
F			ɬ	s		ʂ	ɕ			h
v			l			ʐ	ʑ			

The stereotypical electronic database brings to mind tables of data, or a web form with fields for entering data or defining queries. But databases (the electronic kind) appear in many shapes and forms: Every bank withdrawal, library search, or airline ticket purchase makes use of a database. Less obviously, they are also behind every phone call, every train arrivals board, and every department store cash register. The ideal database is invisible:[2] It is a way to manage the information involved in carrying out a certain task. A desktop computer's calendar application, for example, stores appointments in a database; but the user sees a daily, weekly or monthly calendar, not a table with records, columns and keys. To return to linguistics, the Ethnologue website (www.ethnologue.com) provides a profile page for each of the world's more than six thousand languages, and language profiles for each country in the world. The site does not obviously look like a database: there are no complicated search forms, and not a table to be seen anywhere. But each report page is composed on the fly, from information stored in a database. It is an interesting exercise (after reading this introduction) to reconstruct the database design that must be behind the Ethnologue directory. The database of the World Atlas of Language Structures (Haspelmath, this volume) presents its data in the form of maps of the world, with the data for each language appearing as color-coded pins at the nominal location of the language.

1.2 The "database model"

Almost all computer applications engage in managing stored data of some sort. A text editor reads a document file and writes out an edited version, a computer game presents a series of encoded challenges and (if all goes well) records a high score for subsequent display, etc. Since this data is inevitably stored in computer files, the most straightforward approach to creating applications is to let them read and write data from files in some suitable, custom-designed or generic format. For example, a text editor can read a file containing a document in some recognized format, and later write the modified document in the same file (or a different one).

But this approach, called the "**file-based model**", has serious shortcomings for complex data-intensive applications, which need to process large amounts of information in much more demanding ways. The software system that keeps track of a bank's money, for example, must be able to:

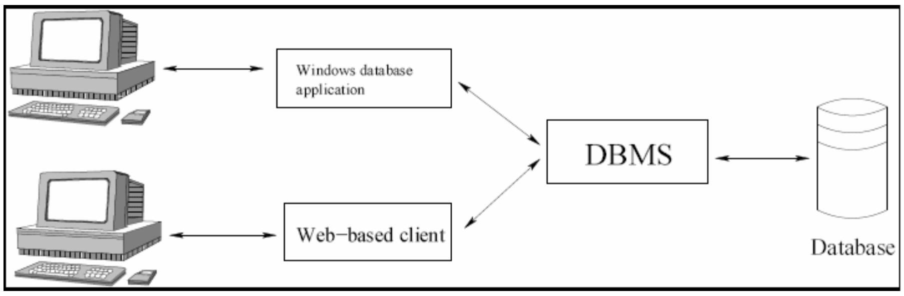
- store and retrieve large amounts of data very quickly
- carry out "concurrent" queries and transactions for multiple users at the same time
- allow various sorts of operations on the data, keeping an "audit record" of why each one was carried out. ("Who withdrew 1000 euros from my account last week?")
- allow different applications to manipulate the same data: (E.g., the software embedded in automatic teller machines, the computer terminals of bank tellers, management software for producing overviews statistics, systems for carrying out transactions with other banks, etc.)
- selectively grant access to different subsets of information to different users. (I should only be able

[2] The oft-quoted maxim "good typography is invisible" is applicable to functional design in general, and to database design in particular.

to see the balance for my own account; clerks should not be able to move millions of euros at will).

While the file-based model can be made to work, it leads to expensive duplication of programming effort, not to mention errors and incompatibilities at all levels. The many software applications involved need to understand the same file format, cooperate to deal with problems of simultaneous access to the same data, and somehow prevent unauthorized users from gaining access to the wrong data.

The solution, called the "**database model,**" is to delegate the job of storing and managing data to a specialized entity called the **database management system (DBMS).** Instead of reading and writing from disk, applications request data from the DBMS or send data to it for storing. All the complicated issues of storing, searching and updating, and even the question of who should be granted access to the data, can be solved (and tested) just once at the DBMS level.



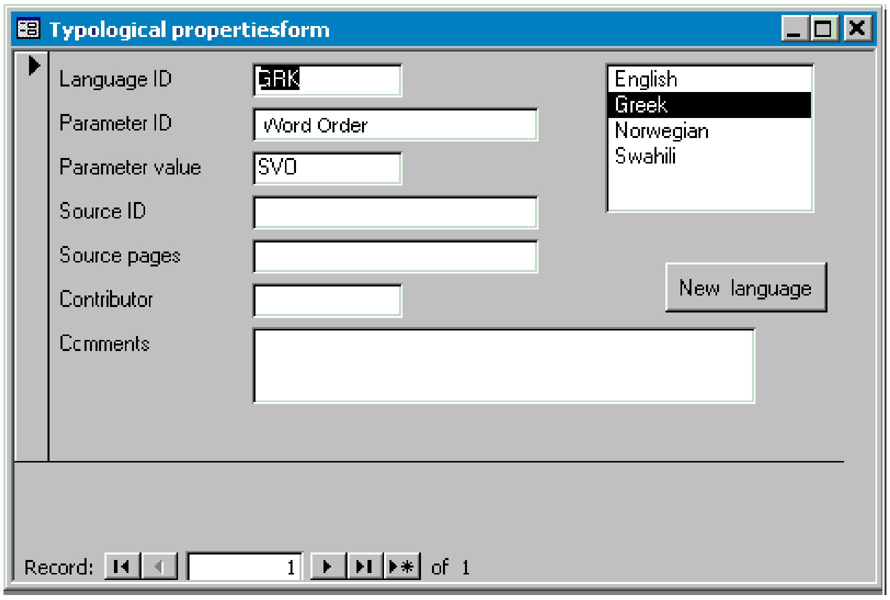
The DBMS might function as a software library that is compiled into a larger program, or as a service that is contacted over an internet connection. What is important is that it functions as the gatekeeper for the system's data: External applications request data from the DBMS (by means of a suitable *query* explaining what data they want), or submit data to the DBMS for storage in the database. The DBMS must meet all the challenges discussed above: handling concurrent requests, high performance, access management, etc. But it is much easier to address these issues in one module, which is then relied upon for all data-related tasks by other software.

Rather than create a database manager for the needs of each project or organization, today's DBMSs are general-purpose engines that can handle any kind of data management task. They do this by supporting a very general model of data organization, which can then be customized according to each project's needs. A bank or a linguistics research group, for example, can acquire such a database engine (perhaps Oracle or MySQL) and use it to create and run a database tailored to their particular needs.

1.3 Interacting with the database

The typical DBMS is never seen by the users of the database; it does not have a graphical user interface (GUI for short), but merely serves or stores data in response to requests expressed in a suitable command language. (The most common command language is SQL, for *Structured Query Language*; we'll come back to it later). Large commercial DBMSs such as Oracle and Microsoft's SQL Server, and free analogues such as MySQL and PostgreSQL are in this category; they must be used in combination with a **"client"** application, created in conjunction with the database, that provides the user interface. In the simplest cases, such an application is a so-called "thin client" that provides a view of the database quite close to the structure of the underlying tables. In other cases, the client application has significant functionality of its own, and should be viewed as the heart of the application; the database is merely used to store the persistent data that supports the application.

On today's computers, an interface application is almost certain to provide a graphical user interface, usually a network of **forms** with text fields for typing in data, and buttons or menus for various actions.



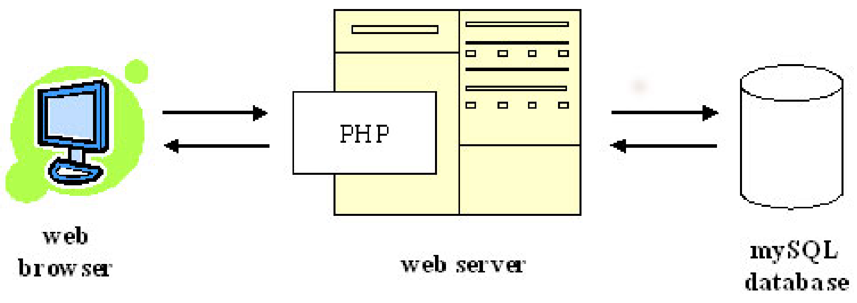
Typological propertiesform

Language ID	GRK
Parameter ID	Word Order
Parameter value	SVO
Source ID	
Source pages	
Contributor	
Comments	

Often, read-only views of the database do not look like structured forms but like **“reports,”** which present their contents in a more text-like format. (We already mentioned the Ethnologue website in this connection). The interface application might be a machine that sells train tickets, a cash register, a bar-code reader, or perhaps an eye-tracking device that collects data for a psycholinguistics experiment. The database model makes it possible to use any of these interfaces, or several of them at once, with the same collection of data. Note that the end-user’s view of the data can be very different from the tables and columns that the DBMS uses to store the data. Even when relying on forms like the above, the interface presents the data in a way that is useful and informative to the user: Data from several tables can be combined, and various fields can be selectively hidden when they are not relevant. The complete application, in short, can be a lot more than just a way of viewing and editing the tables of a database.

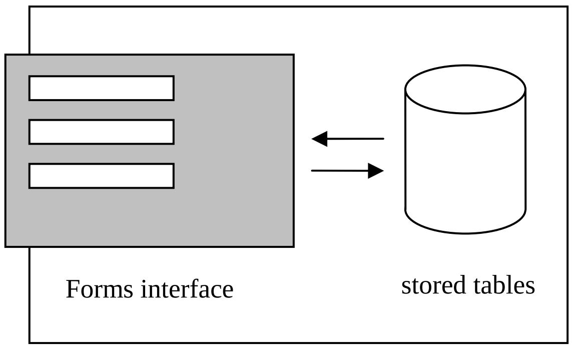
In this guide, we will not discuss the many issues involved in the design of the user interface. The problems are no different from those that arise for any software development task, and the topic is much too large to be addressed here. We will limit ourselves to the question of designing the underlying database, and give but a bare outline of the relationship between the database and the interface application it supports.

The interface application can be a Windows application, installed on the user’s desktop computer. But an extremely popular alternative is the combination of a DBMS with a way of generating web pages on the fly, providing a **web-accessible database.** The diagram shows a typical arrangement.



End-users employ a web browser to view pages provided by a web server somewhere. Behind the scenes, the web server communicates with a DBMS that manages the actual data; a computer program (written in PHP in this example) interprets user actions, requesting data as needed from the DBMS and formatting the results for web display. Web server and DBMS may be on the same server computer, or on different computers – it makes no difference, since the web server has access to the data only through the DBMS, and the user has access to the data only through the web server.

Desktop database applications such as Microsoft Access and FileMaker Pro combine a DBMS with a graphical user interface in a single package.



Such applications are sometimes mistakenly referred to as “off the shelf” databases, in contrast to “custom” DBMSs of the type just discussed. In fact, a database must be defined in FileMaker or in Access just as it must be defined in MySQL; the difference is that the desktop databases come with a graphical interface for managing the creation of tables, and a specific environment (also with a GUI) for creating the user interface of the database (a process that can be very easy or even automatic, but also has its limitations), whereas a pure DBMS does not provide integrated tools for the creation of its user interface.[3]

1.4 Types of databases

General-purpose database management systems are based on some formal, general model for organizing data. By far the most common type of database in use today is the so-called **relational** database. All the well-known DBMSs are relational databases, including Oracle, MySQL, Postgres, FileMaker Pro and Microsoft Access.[4] While we will not discuss non-relational databases further in this chapter, it is worth mentioning some alternatives in order to better understand what a relational database can, and cannot, do.

1) The simplest type of data model is to have a single table, or “file”. Each row corresponds to some object (e.g., a language) being described, and each column represents a property (“attribute”), such as name, location, or Basic Word Order.[5]
2) A **relational database** consists of several tables (“relations”) of this sort, linked to each other in complex ways.
3) A **hierarchical database** is organized not as a table but as a tree structure, similar to folders and subfolders on a computer disk drive. Each unit “belongs” to some larger unit, and contains smaller units. Think of a book divided into chapters, then sections, then subsections etc.
4) In an **object-oriented database,** data are modeled as “objects” of various types. Objects share or inherit properties according to their type; e.g., a database about word classes could let objects of the type *transitive verb* inherit properties of the type *verb.* While useful for very complex applications, this model need not concern us here.

[3] This does not mean that there is no GUI support for “pure” DBMSs. Database vendors or independent projects (in the case of open-source DBMSs) provide graphical tools for the management of each type of DBMS. For example, the application *PhpMyAdmin* is allows web-based administration of MySQL databases. Note that such applications are intended for the set-up and administration of databases, not for interaction with the end-user.
[4] FileMaker Pro has some unusual features, which somewhat obscure the fact that it is undoubtedly a relational database.
[5] Tabular information might also be stored in a format that does not *look* like a table; e.g., as a series of name-value pairs. Data files for *Shoebox,* the linguistic data management application, are in this format.

The hierarchical model was among the very earliest database models; although it was largely supplanted by the relational model, it has become relevant again today because it corresponds to the natural structure of XML data, and is suitable for managing heterogeneous data. All of the linguistic databases presented in this volume are relational databases, with the exception of the Typological Database System (Dimitriadis et al., this volume) which uses a hierarchical model to unify a collection of independently developed linguistic databases. All of the component databases of the TDS are, in fact, relational databases.

2. Choosing a database platform

Creating a linguistic database does not always come easy to linguists. For one thing, it involves technology and concepts that are not covered in the typical humanities curriculum. (Hopefully this chapter is already helping in this respect). For another, it involves making decisions and choices involving this technology, whose consequences are sometimes not felt (or understood) until much later. While this chapter is meant to be a conceptual introduction, not a technology guide, we will attempt in this section a very general discussion of some specific technical choices.

The first decision to be made is: Do you need to build your own database? Numerous ready-to-use applications now exist that can support linguistic data collection, from the linguistics-specific (the best known probably being SIL's Shoebox, and its successor the Linguist's Toolbox) to the general (such as Microsoft's Excel and other spreadsheet applications). Ferrara and Moran (2004) present an evaluation of several such tools. If an existing application meets your needs, it is not necessary to embark on designing and creating a new database from the ground up.

Our discussion here cannot hope to provide definitive answers: The issues and trade-offs are complex, and depend on the nature of the specific task as well as the available resources (human, financial and technological), now and in the future. If you are pondering the creation of a new linguistic database and are unsure of the technical choices involved, it is probably best to solicit some expert advice from someone with a good understanding of the technology, being sure to discuss the specific requirements and resources available to your project.

Having said that, most of the (custom) linguistic databases we know of can be classified in one of the following categories:

1. The **all-in-one desktop database,** created with either Microsoft Access or FileMaker Pro.[6] The simplest solution, it is most suitable for one-person data collection projects.

2. The small, do-it-yourself **web database.** While considerably more complex than a desktop database, it is the best solution if multiple people must be able to enter data in parallel, or if there are plans to eventually make the database publicly available.

3. A sophisticated database for a large project with professional programming staff.

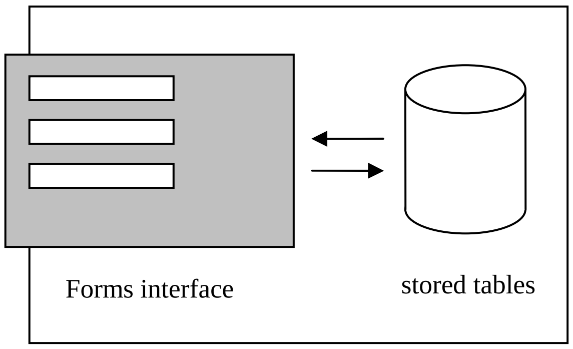
We will have little to say about the third category. While there is no sharp line between a "small" database project and a large one, our goal here is to address the concerns of linguists with limited technical resources; the professionals do not really need our advice.

2.1 The all-in-one desktop database

For a one-person research project without expert technical support, a desktop database application

[6] These two applications are the best-known in this category. The free office software suite Open Office provides an open source desktop database application that is (largely) compatible with Microsoft Access.

such as Microsoft Access or FileMaker Pro is often a very good solution. These applications store your entire database in a single disk file or folder,[7] allowing it to be copied, backed up, and moved about like an ordinary document. This arrangement greatly simplifies the initial set-up of the system, since no server or network configuration is necessary. This can sometimes be very important in institutional settings, where IT policies might forbid the operation of independent database servers.



The user interface of the desktop database application allows users to define tables and relationships for the database, and to create forms for its user interface. Some understanding of database principles (such as this chapter provides) should go a long way towards helping a new user understand how these programs are meant to be used. Each product includes a scripting language that can be used to extend the functions of the automatically generated forms – but only with a degree of arcane knowledge.

The advantages of using an all-in-one database can be summarized as follows:

1. A single product with a graphical user interface for both the database configuration and the user interface.
2. Automatic or interactive generation of the forms.
3. Everything fits in one file or folder, and can be backed up, sent by email, etc.
4. All that is needed is a desktop computer with the database application; software is easy to install or already present, and it is not necessary to set up a server.
5. Internet access is not needed.

The last point means that the all-in-one database can be used on a laptop computer without internet access – an important consideration for linguists considering data collection in the field.[8]

On the other hand, the approach also has certain disadvantages:

1. The desktop databases discussed are proprietary software.[9] This limits the ease of distributing copies of the database, since the recipient must own the application software. It also means that the data is not highly portable – it is possible to export data from, e.g., Access to a non-proprietary file format, but doing so adds extra work.
2. The form creation facilities have their limits, which cannot be exceeded without a lot of programming knowledge.
3. It is not possible for multiple persons to enter data in parallel.
4. It is now a common (and highly recommended) practice to make one's data available to other researchers over the internet. An all-in-one database generally needs additional effort in order to be made available on the web.

[7]Microsoft Access stores the entire database in a single disk file, while FileMaker stores each table as a separate disk file.
[8] While a web database can also be installed locally on a laptop for non-internet use, the process is considerably less trivial than simply copying a disk file.
[9] The database included in Open Office, which can read Access databases, is open source software.

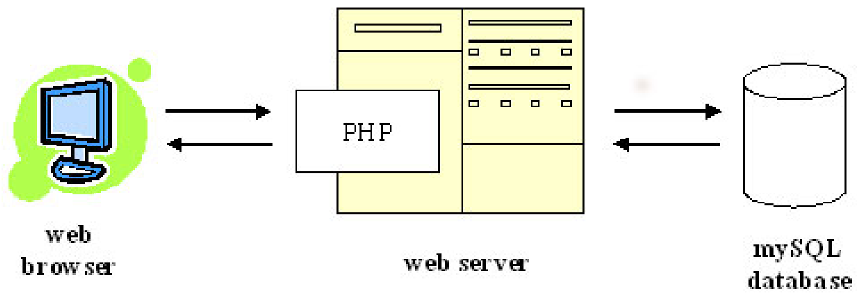
The third point is particularly important for collaborative projects: One of the goals of the database model is to support concurrency, i.e., simultaneous editing sessions by different users. But since this kind of database is stored in a disk file, it must be ensured that only one person at a time should edit it. In general, collaborative data entry with a desktop database raises issues similar to collaborative editing of a text document: Even if a common copy is kept in an accessible location (e.g., a network drive), only one person can modify it at a time.

These problems are not insurmountable, and FileMaker and Access each have mechanisms for addressing them. Although this chapter is not intended to be a review of software applications, we will comment briefly on two of them: FileMaker has an easy mechanism for creating a web server interface to a database; once this is set up, multiple users can work on the single copy of the database by connecting to it with a web browser. The approach does require a workstation that can be configured as a database server, and of course all users must have an internet connection (or at least be on a common intranet). Also, the automatically generated web interface does not support all the user interface features of the full-fledged application.[10] Microsoft Access, in turn, has a mechanism for "cloning" a database, so that modifications to the copies can later be merged together. But since there is no way to prevent two people from independently modifying the same data, the approach is not entirely safe unless project policies can guarantee that this will not happen.

There are other mechanisms and add-on products that can make a desktop database accessible through a web server. To the extent that they work properly, they have the effect of converting a desktop database into a web database of the kind included in our second category, to which we now turn.

2.2 The small web database

When multiple people must collaborate on data entry, the desktop database solution is inadequate. A way must be found for all users of the system to work with the same data store. As we have already seen, this means a DBMS that interacts with "client" application programs over the network. While the client programs could be stand-alone applications written in a variety of programming languages, a very popular solution is to set up the database on a web server, allowing ordinary web browsers to be used for display. We have already mentioned this arrangement, shown in the following diagram (repeated):



The web server, in the middle, includes a set of programs that generate web pages on the fly, using data retrieved from the database. The scripting language PHP is one of the most common languages for writing such programs (scripts); it is specialized for use in web applications, providing extensive support for connecting to databases and communicating with webservers and browsers. In a web-accessible database, PHP scripts interpret and carry out user actions, including requests to view data, to log on or off the system (if authentication is required), and to insert or update data in the database. The necessary data is fetched from the database (which may or may not be on the same computer as the webserver) and formatted into html pages. The web server then sends the generated pages to the user's browser for display.

[10] Bliss and Ritter (this volume) discuss their experiences with a FileMaker web database.

A very popular way to set up such a system is the so-called “LAMP stack”: Linux operating system, Apache webserver, MySQL database management system, and PHP. But many variations are possible: PostgreSQL can be used instead of MySQL; the entire Apache-MySQL-PHP combination can be installed on a computer running Windows or Mac OS X; a PHP module can be embedded in a Windows machine running Microsoft’s webserver, IIS; etc. One of our web databases consists of a PHP front end on a Linux machine, which talks to a Microsoft SQL Server DBMS running on a Windows server somewhere else. But the LAMP combination is a popular default because it is reliable, and the software is free and open-source. The popularity of the technology has another important consequence for linguists of limited means: It is relatively easy to find programmers, including amateur programmers (e.g., students) who know how to make a web database with PHP. For the technically inclined, introductory how-to guides are also available online.

The genius of this approach is that it relies on the user’s web browser to display the user interface of the database. A web browser, in addition to being already installed on every user’s computer, has the advantage of being an extremely sophisticated piece of software. For things that a web browser can do, it would be hard for a database project to create a stand-alone client application that does them equally well.

But a web database has one important limitation: A web page cannot provide the fonts required to display it properly; these must be already resident on the user’s system. Linguistic databases often need to use phonetic symbols, or text from languages with less common writing systems, for which the required fonts are far from universally available. In such cases, users of a web database may need to manually download and install a font before they can use it properly. Fortunately, this is much simpler (and less problem-prone) than installing a full-fledged application.

Some things are too complicated to do with HTML and a web browser. For example, we may want to display audio or video in a format that the browsers do not support, to draw syntax trees, to accurately measure reaction times, or to manipulate maps interactively. Modern browsers support a number of ways to extend their basic functionality; notably, they can execute javascript programs embedded in a webpage. For more demanding uses, a stand-alone client application is sometimes the only option.

In short, the web-based database has the following important advantages:

1. Fully supports collaboration.
2. Allows open access to the database over the web (if, and when, desired).
3. Can be built entirely with free software, and generally relies on open standards rather than proprietary protocols or file formats.
4. The program that generates the user interface can be arbitrarily complex. The capabilities of browsers can be further extended with javascript and other browser technologies, if desired.

Disadvantages:

1. Compared to the all-in-one desktop database, the main disadvantage of the web database is that creating one requires knowledge of several different technical domains: facility with setting up software and servers, PHP programming, HTML and CSS (for designing the pages to be generated) and SQL (for communication between PHP and database). Fortunately, the popularity of the LAMP suite means that it is relatively easy to find skilled help. For the technically inclined, there are many free primers and reference manuals.
2. A second problem is that a server computer is needed. For linguists that only have access to their desktop workstation, this can mean negotiations with their IT department and/or the cost of buying a server. At some institutions, IT policies prohibit the operation of an independent server.

It should be added that knowing how to build a database is only the beginning; to actually do so, a successful design must be devised and carried out. This is as true for a desktop database as for a web database; but because of the greater complexity of web databases, the time investment required and the consequences of errors are proportionately larger.

2.3 Some recommendations

So how should you go about creating a database? We can only offer general suggestions here, and even these are limited to the kinds of situations we have experience with. But it should be clear that designing a database is a complex undertaking. If possible, get help from someone experienced with databases.

If you do get help, be sure to be *actively* involved in all stages. Get as good an understanding as possible of the relevant issues (reading the rest of this volume should help), and meet at least weekly to discuss the design. Don't assume that your programmer understands how you, as a linguist, view the data you want to collect; they don't. Only through sustained discussion can there be a convergence of visions.

Some more general suggestions:

1. Plan ahead: design your database carefully before you start using it in earnest.
2. Plan for change: As you collect data, your understanding of the phenomenon and the best way to study it will evolve.
3. Keep it simple. (But make sure it meets your foreseeable needs).
4. Document your database: explain it in writing, to yourself and others.

3. The relational database model

In a relational database, data is formally represented as instances of one or more **relations** (the mathematical basis of this design was first set out in Codd 1970)**.** Concretely, a relation is a table with named columns:[11]

Language name	ISO code	Speakers	Area	SourceID
English	eng	309,000,000	Europe	Eth15
Italian	ita	61,500,000	Europe	Eth15
Swahili	swh	772,000	Africa	Ashton47
Halh Mongolian	khk	2.337,000	Asia	Eth15
Dyirbal	dbl	40	Australia	Dixon83
Mongol	mgt	336	Papua New Guinea	SIL 2003

Each row of the table is a **record,** corresponding to some object being described; in this case, to a language. Each column is an **attribute,** representing a property. It can be seen that the cells of the table contain the data; each cell gives the **value** of an attribute, for the object corresponding to that row. Each value represents one *unit of information* about the object being described by the record in question.

It can be seen from the above example that rows and columns are not interchangeable: Each column is given a name (and meaning) by the database designer, while rows are added as the database is used. A table could have columns but contain no data (hence no rows); but there could never be a table with rows but no columns. As a database grows, it can come to contain thousands or even millions of re-

[11] Data and citations in these examples are sometimes made up, and should not be assumed to be either correct or representative of what the cited sources actually write.

cords in some tables; there is in principle no limit. Most DBMSs, on the other hand, have comparatively low limits on the number of attributes that can be declared. Microsoft Access allows a maximum of 255 columns for each table.

There are quite a few alternative terms for these database fundamentals: A record (table row) is formally known as a **tuple.**[12] A table (relation) is also known as a **file.** We will avoid this term since it invites confusion with files on a computer disk; for example, Microsoft Access stores all tables of a database in a single disk file, while some DBMSs (such as MySQL) use several disk files for information belonging to one table. Attributes are sometimes called **fields** (because they correspond to input fields in the user interface), or just **properties**.

3.1 Keys and foreign keys

The notion of **key** is central to relational databases. A key for a table is a *set* of attributes (but usually just one attribute) that will always uniquely identify a record in that table. In the above example, the language name, ISO code, and number of speakers are all unique; but the number of speakers, and even the name, are not *guaranteed* to be unique (for instance, all extinct languages have the same number of speakers). Only the ISO code is, by design, guaranteed to be unique. The keys of a table are sometimes also called **candidate keys;** a table can have several.

Our real-world knowledge allowed us to identify a key in this table; since a DBMS cannot be expected to guess which sets of attributes can serve as keys, there is a way to declare them. The **primary key** is an attribute, or set of attributes, that the DBMS will use to uniquely identify records. The DBMS will enforce this uniqueness, refusing to create two records with the same key value. By convention, the primary key is indicated by underlying:

Language name	**<u>ISO code</u>**	**Speakers**	**Area**	**SourceID**
English	eng	309,000,000	Europe	Eth15
(etc.)				

A key that consists of more than one attribute is called a *composite key* (as opposed to a *simple key*). While a database table can have several different candidate keys, it will only have one primary key (which might, of course, be composite).

Keys are involved in expressing **relationships** between tables. (Note that a *relationship* should not be confused with a *relation*, which is just a table). A relationship in a database expresses a real-world relationship between the objects described. For example, our table of languages contains the attribute *SourceID*, which indicates the bibliographic source of the information. Our example database also contains another table, whose records are not languages but bibliographic sources (books, articles, and other publications). A record in the Language Details table[13] is now **related** to a record in the Bibliographic Source table, by a relationship we might describe as "contains information from."

To encode the relationship in the database, we store with each record in the Language table a value (in the attribute *SourceID*) that matches the primary key of a record in the Bibliographic Source table; in the example below, this is the value "Eth15". We say that the attribute *SourceID* is a **foreign key.**

More generally, a **foreign key** is an attribute (or set of attributes, if it's a composite key) within one table, that matches the primary key of some (other) table. A foreign key expresses a relationship between the two tables. The DBMS can ensure that every foreign key really matches the key of a record

[12]Abstractly, a table represents a *relation,* defined mathematically as a collection of "tuples" (triples, quadruples, etc.) of values for some attributes. For example, the triple (Dyirbal, dbl, 40) represents the name, ISO code and recorded population of a language.
[13] Now that we have more than one table, it is useful to identify them by name.

in the related table, e.g., by refusing to store non-matching values. In the following, we use a star after the attribute name to indicate that it is a foreign key. (This is not standard notation).

Language Details

Language name	**<u>ISO code</u>**	**Speakers**	**Area**	**SourceID***
English	eng	309,000,000	Europe	Eth15
(etc.)				

Bibliographic Source

<u>ID</u>	**Title**	**Author**	**Year**	**Publisher**	**…**
Ashton47	Swahili grammar (including intonation)	Ashton, E.O.	1947	Longmans	
Eth15	Ethnologue: Languages of the world, Fifteenth Edition	Gordon, Raymond G., Jr. (ed.)	2005	SIL International	
(etc.)					

3.2 Retrieving data with SQL

The names of several DBMSs have already been mentioned, and the reader may have noticed that more than one of these includes the string SQL in its name. This is indicative of the importance of SQL in the field of databases. As already mentioned, SQL stands for *Structured Query Language*, a language used interactively and by programs to query and modify data and to manage databases. With the partial exception of FileMaker, all commonly-used relational DBMSs support the use of SQL, and we recommend that anyone involved in a database project acquire familiarity with the basics of the language.[14]

Although SQL can be used for a variety of database operations, such as inserting data, deleting data and creating new tables, its most visible function (as the name suggests) is for expressing queries: specifying data to be retrieved from an existing table or tables. Queries minimally consist of two parts: a specification of the field or fields to be retrieved and a specification of the table in which those fields will be found. These two aspects are represented in SQL by a SELECT command containing a FROM clause. For example, the following SQL statement will retrieve all the language names recorded in the *Language Details* table shown above:[15]

```
SELECT “Language Name” FROM “Language Details”;
```

More than one field can be specified in the SELECT clause. Multiple fields are separated by commas, as in the following example which retrieves language names and speaker populations:

```
SELECT “Language Name”, Speakers FROM “Language Details”;
```

Typically, when we construct a query, we are interested only in records which match a certain criterion. Such restrictions are expressed in SQL using a WHERE clause. WHERE clauses usually express a condition on some field of the table being queried; this need not be one of the fields from which data

[14] SQL is recognized as a standard by ISO, the International Standards Organization; however, actual database implementations always have limitations, extensions or other differences from the standard, which are significant enough that complex SQL scripts are typically not portable from one DBMS to another. Nevertheless, the basics of manipulating tables are nearly the same, and an understanding of the workings of SQL allows one to work with any SQL implementation.

[15] The quotation marks (“) around table and field names are necessary for names that contain a space; otherwise, they can be omitted. A semicolon (;) signals the end of an SQL command, which can continue over several lines of text.

will be retrieved. The first of the following examples will retrieve the names of the languages in our table for which the number of speakers is more than 500,000, the second would retrieve both language names and speaker populations given the same condition:

```
SELECT “Language Name” FROM “Language Details”
        WHERE Speakers > 500000;

SELECT “Language Name”, Speakers FROM “Language Details”
        WHERE Speakers > 500000;
```

The above queries do not specify the order in which the matched values should be presented. In this case they will probably be presented in the order in which they are stored in the table, but the rules of SQL do not guarantee this. If we wish to display the results of our query in a particular order, e.g., alphabetically by language or ordered by size of speaker population, this can be accomplished by using an ORDER BY clause. Such clauses allow us to specify which field (or fields) will be used to sort the data, and whether the order should be ascending or descending. The following example will return a list sorted in ascending order of speaker population – ascending order is the default and need not be specified explicitly:

```
SELECT “Language Name”, Speakers FROM “Language Details”
        WHERE Speakers > 500000
        ORDER BY Speakers;
```

All the examples given so far have retrieved data from a single table. One of the important features of a relational database is that queries can be carried out which access more than one table; in SQL this is accomplished using a JOIN clause. In its simplest form, such a clause specifies a combination of two tables from which data will be retrieved and a **join condition** based on a field which the two tables have in common, typically the foreign key field in one table and its source in another table. The following example will retrieve the titles and authors of all sources used for the languages of Europe, along with the names of these languages. Because two tables might use the same name for some fields (e.g., “ID”), an extended syntax can be used to specify field names: *TableName.FieldName.*[16]

```
SELECT “Language Details”.“Language Name”, Title, Author
     FROM  “Bibliographic Source” JOIN “Language Details”
        ON “Bibliographic Source”.ID = “Language Details”.SourceID
     WHERE “Language Details”.Area = ‘Europe’;
```

Conceptually, a JOIN operation creates a new, transient table, created by combining the rows of its constituent tables according to the join condition (the ON clause). The main SELECT clause then operates on this resulting table.[17] The JOIN operation creates one row for each combination of records (rows) that satisfy the join condition; this means that any table row that matches multiple rows of the other table will be used multiple times. In our example, a single bibliographic source can be used for several languages, and the joined table will include rows like these:

[16] Prefixing the table name is optional if the field name appears in one table only.

[17] Note that this is a *conceptual* description; such tables exist only in the sense of representing the stages of the query. The DBMS can usually figure out which rows and columns of the join table are needed for the final result, and will construct those alone without generating the entire intermediate table. In any event, such tables are not permanently stored in the database.

From *Language Details* table				From *Bibliographic Sources* table			
Language name	**ISO code**	**…**	**SourceID**	**ID**	**Title**	**Author**	**…**
English	eng		Eth15	Eth15	Ethnologue: Languages of the world, Fifteenth Edition	Gordon, Raymond G., Jr. (ed.)	
Italian	ita		Eth15	Eth15	Ethnologue: Languages of the world, Fifteenth Edition	Gordon, Raymond G., Jr. (ed.)	
Swahili	swh		Ashton47	Ashton47	Swahili grammar (including intonation)	Ashton, E.O.	
Halh Mongolian	khk		Eth15	Eth15	Ethnologue: Languages of the world, Fifteenth Edition	Gordon, Raymond G., Jr. (ed.)	

The *Source* record with ID *Eth15* has been paired with each language that gives *Eth15* as its *SourceID.* From this temporary table, the SELECT query will retrieve only those rows that match the WHERE clause (in our example, those with *Area = 'Europe'*); finally, the result of the SELECT query is a new transient table that contains only the requested columns of these rows:

Language name	**Title**	**Author**
English	Ethnologue: Languages of the world, Fifteenth Edition	Gordon, Raymond G., Jr. (ed.)
Italian	Ethnologue: Languages of the world, Fifteenth Edition	Gordon, Raymond G., Jr. (ed.)
(etc.)		

This sort of table manipulation, explicit or implicit, is the essence of the "relational algebra" underlying the operation of relational databases: Operations on tables return other tables, which can themselves be operated upon. Multiple JOIN and SELECT statements can be combined in a single query in various ways, allowing extremely powerful manipulations of the data.

Data retrieval with SELECT is only one of the many functions of SQL; almost any database operation can be controlled with SQL commands. There are commands to create and delete databases, control access rights, and to add, delete or modify stored data. As already mentioned, we recommend that anyone involved in planning or implementing a database should acquire familiarity with the basics of SQL. This knowledge is very useful when considering the possible design of a database: one should be able to see what sort of queries would be of interest to the potential users of the database, and to express these queries in SQL using the names of tables and fields as specified in the design. If this proves difficult to do, then possibly the design needs rethinking.

4. How data is stored in databases

4.1 Data types

DBMSs allow us to specify what type of data we wish to store in any particular column of a table. While all data is in the end stored in some sort of binary format, the data type of a field determines how it will be treated by the DBMS. There is a temptation for beginning database creators to ignore this factor, and to define all columns as fields holding text data. However, there are several reasons why this is not good practice, and we will discuss some of these as we introduce the most important data types.

The most important data types are text, numbers, and Boolean (or "logical") fields; all databases have additional data types and, partly for technical reasons, numerous subtypes of the core data types. In particular, each DBMS will offer several types of integer fields, differing in their size and (therefore) the range of numbers they can store, and several types of text fields with options for controlling the maximum amount of text that can be stored in them. The specific types and subtypes offered depend on the DBMS, as do the names by which they are known.

Conceptually the simplest type of data is **Boolean** data, that is, a field which can only have the values *True* or *False.* Boolean data is efficient to store and use, since a Boolean field nominally needs only one bit of memory and can therefore be stored extremely compactly. This type of data is frequently used in typological databases, where many fields may contain information about whether a given language has a certain property or not.

Numeric fields are for storing numbers; linguistic databases generally make little use of numbers (except for automatically-generated IDs or as indices into a list of possible values), but they are of great importance for applications in business and the quantitative sciences. DBMSs provide a variety of numeric types, with different storage requirements. Typical examples are small integers ($x < 256$), which require one byte of memory, [18] large integers which use up to eight bytes, and one or more sizes of "floating point" fields, which store non-integer ("real") numbers.[19]

For a database that will include numeric data, there are immediate and obvious disadvantages to storing numbers in a text field: Such fields are not considered numeric by the database, but are treated as simple sequences of characters (which happen to contain digits). If sorted, they will be alphabetized like names, with 1, 10 and 1,000,000 appearing before 2 or 20. They cannot be added, multiplied, or compared for magnitude with other numbers; for example, the following query (given as an example in the preceding section) is only meaningful if the field *Speakers* has a numeric data type:

```
SELECT "Language Name" FROM "Language Details"
        WHERE Speakers > 500000;
```

The most complex core type, and the one that requires the most storage capacity, is **text** data. One strategy which DBMSs use to contain the demands on memory of fields containing text data is to allow the designer to specify the number of characters allowed in a given field. Databases historically provided fixed-length text fields, which always reserve space for a fixed number of characters whether it is needed or not. Today's DBMSs also provide variable-length fields, which only take up the amount of space actually used for each string (plus some overhead for encoding the length of the field). The database designer can specify a maximum length, up to a limit imposed by the design of the DBMS; the most efficient text type (known as *Varchar* in MySQL, and as *Text* in Access) is limited to 255 characters. Every DBMS also provides a data type for long strings; this might allow up to 65,535 characters. Short and long text types may come in fixed-length and variable-length variants.[20]

Because variable-length text fields only use as much storage space as they need, there is no need to severely restrict the maximum size of strings in the design of the database; we recommend making all

[18] The basic unit of storage for information in a digital computer is one binary state, known as a **bit**. These are then organized into groups of eight, known as **bytes**, which have 2^8 or 256 possible values.
[19] Floating point numbers are represented as an exponent plus a fixed number of "significant digits," and can store extremely large or extremely small numbers, with some loss of precision. E.g., the number 602,214,141,070,409,084,099,072 can be represented as 6.0221414×10^{23} (In a database, exponent and significant digits are in binary, not decimal form).
[20] Long strings are known as *Memo* in Access, and as *Text* in MySQL. (Therefore, "Text" means very different types in these two DBMSs). Access does not have fixed-length strings.

text fields long enough to contain any foreseeable data. The short-text type should be preferred to long text, for strings that are not expected to exceed 255 characters. (You should consult your database manual to check that the data type you are using is indeed variable-length).

The full inventory of database data types is quite a bit more complex than we have sketched here. Most DBMSs also have special numeric types for dates, times, and currency amounts. Access has a special text data type for hyperlinks. Specifying a field to contain data of one of these types has consequences for integrity at data entry (see below), and also allows a range of specialized operations to be performed on the data. For example, if a field is specified to contain only dates, then the exact format to be used can also be specified and the user interface will provide the user with a **mask**, a template which will only accept data of the required format. Many DBMSs also allow the database designer to declare the character encoding (see next section) used in a text field, or in an entire table or database. This can affect data validation, sorting alphabetically, and searching. Finally, many DBMSs provide a special data type for arbitrary large blocks of binary data, which the database will store without knowing anything about their internal format.

Another data type that is of particular interest to linguists is the **enumerated** type, which restricts a field to taking values from a list specified by the database designer. For example, the property *Basic Word Order* can be modeled as an enumerated attribute that takes values from the set (*SVO, SOV, VSO, VOS, OSV, OVS, Free*). This is an extremely useful construct, and one that is very frequently applicable in linguistics since a lot of properties take values from a fixed set of options. Relying on an enumerated type restricts the data which can be entered in a field, and therefore lessens the risk of input errors and inconsistencies (for example, if one types "*S-V-O*" instead of "*SVO*", which is recognizable to a human but will be treated as a distinct value by the DBMS). While some databases define an enumerated data type, it is also easy to model in the user interface, or by using foreign keys. We will see how in section 7.

We should not end our discussion of data types without mentioning that DBMSs typically allow a field to be null, i.e., to have no value at all. Nulls require care by the database designer, since their behavior in queries and arithmetic operations can be surprising. A text field whose value is the empty string is different from a null text field, and a null numeric field is different from any numeric value (including zero).

We have seen, then, some of the properties of the various data types and the benefits of utilizing them properly. One other reason for adopting the discipline of data typing is that it can minimize problems in the processing of the data at the user interface. When routines are defined in a scripting or programming language, variables will be used to hold data for processing. Such variables must be typed in many scripting languages, and it is always good programming practice to do so even if it is not required. If the types assigned to the data fields in the tables which supply data for processing do not correspond to the types assigned to variables, runtime errors can occur. Tracking down and correcting such errors is very time-consuming, but they can be avoided by the correct use of data typing. Finally, the benefit of data typing can also be viewed in a rather more conceptual sense. A theme which runs through this chapter is that being able to think precisely about the nature of one's data is an essential skill in working with databases. Data typing can be viewed as one aspect of that skill: if we cannot be precise about the type of data we intend to store in a particular field of our database, then we are not thinking about the problem with sufficient precision.

4.2 Character encodings

Linguists often require access to characters beyond those normally available on a standard Roman alphabet keyboard. The characters of the International Phonetic Alphabet (IPA) are needed for phonetic transcription, and different character sets are needed for the standard written representation of many languages. Many linguists will have had the experience of carefully ensuring that the necessary characters are included in a document only to see them vanish completely when the file is opened on a different computer. Such problems arise from the way in which character encodings are internally represented by computers. Each character is represented as a number, but the mapping between characters and numbers is arbitrary and may vary from one operating system to another, and even from one font to another. The basic characters of a Roman keyboard do have standardized codes, provided by the American Standard Code for Information Interchange (ASCII). Later systems based on ASCII, especially ISO 8859, provide standard mappings for other writing systems. ISO 8859 includes Cyrillic, Arabic, Greek, Hebrew and Thai characters, as well as various extensions of the core Latin character set. The IPA characters are not included.

However, all such schemes faced one basic obstacle, which is that they were designed to use one byte of data for each character. This limits the number of characters which can be encoded to a maximum of 256, the number of distinct values of an eight-bit binary number. Accordingly, ISO 8859 provides a *family* of separate encodings for the different alphabets it supports. In each of those, the first 128 values (expressible as a seven-bit number) are identical with the ASCII encoding, and contain the standard English letters (capital and lower case), punctuation and numbers, and a number of whitespace characters and control codes; the remaining 128 values, the so-called "high page" of the character table, are different for each encoding defined by ISO 8859. For example, ISO-8859-1 (also known as Latin-1) provides accented characters and other symbols needed for Western European languages; ISO-8859-5 covers Cyrillic, ISO-8859-7 Modern Greek, etc.

The problem with using a family of eight-bit encodings is that it is difficult to mix text from different alphabets. Since each encoding includes ASCII as a subset, a file that uses the-ISO 8859-7 encoding, for example, can contain a mixture of Greek and English text. But there is no simple way to mix Cyrillic and Greek, or Cyrillic and French: ISO 8859 does not provide a way to indicate which encoding is being used, or to switch between encodings. To manage multi-lingual text, a database or other application must provide its own way of keeping track of the character encoding used.[21] HTML pages and Microsoft Word documents each have their own way of accomplishing this; but desktop database applications are not designed to allow fine control over text encoding – database fields are generally meant to contain "flat" text, without invisible embedded codes, and the encoding can usually be set as a database-wide option, if at all. For a cross-linguistic database, this is a severely restrictive state of affairs.

Every font uses some encoding scheme to map character codes into character shapes ("glyphs"). Because ISO 8859 does not include an encoding standard for IPA characters, IPA fonts used various arbitrary mappings. In effect, each IPA font defined its own encoding, an alternative to the standard encodings of ISO 8859.[22] All this made a change of fonts a potentially disastrous affair, since a string of Russian, French or IPA text could suddenly turn into nonsense when displayed with an incompatibly encoded font.

Such limitations, as well as the proliferation of other incompatible encodings, led to the formation of a working group in the mid 1980s whose aim was to establish a comprehensive and universally recognized scheme for the digital encoding of character data. The outcome of the work of that group and its successor, the Unicode Consortium, is Unicode. Unicode is a single, universal character encoding scheme, originally based on two-byte codes (giving it 65,536 potential "codepoints") and later generalized to abstract codepoints that are independent of the number of bytes used to represent them. Unicode 5.1 covers over 100,000 symbols, and each is assigned its own codepoint.

Unicode makes a principled distinction between character symbols and **glyphs,** the visual shapes used to represent them. For example, the character *A* ("capital A") can be represented with any of the glyphs A, **A**, ***A***, **A**, *𝒜*, etc. A single glyph may also correspond to a combination of several characters:

[21] Another standard, ISO 2022, includes a means of switching between encodings; this still requires the application to keep track of the "current" encoding at any point in the text. This system is not very widely supported, and is increasingly being supplanted by Unicode (see below).
[22] SIL eventually adopted a consistent mapping for all their IPA fonts, which has also been used by some independent IPA fonts.

For example, many fonts include a glyph for the typographical “ligature” [ﬁ], which represents [f] and [i] together.

Unlike most earlier encoding schemes, Unicode in principle treats any element which can form part of a character as a separate symbol (“character”). For example, Unicode treats the letter [e] as one symbol and the acute accent as another one; each is assigned to a different codepoint. The character [é] is therefore made up of two symbols in Unicode.[23] This approach has admirable conceptual clarity, but it does also pose problems for rendering complex characters which are made up of several symbols. Fonts provide glyphs for common combinations of letters and accents (such as [é]), which often look better than the direct combination of the separate glyphs; some applications and display systems can automatically substitute such combined glyphs for the two-symbol combination, but others cannot.

The 100,000 plus symbols of Unicode 5.1 include coverage of all major scripts of the world: European alphabets, Middle Eastern scripts including Arabic and Hebrew, Indian scripts including Devanagari, Bengali, Tamil, Telugu and Thai, Chinese, Japanese and Korean characters, a number of historically important scripts such as Runic, Ogham and Gothic, and the symbols of the IPA. Unicode is therefore a development of great significance for linguists, providing a standardized scheme for encoding characters from most of the writing systems used by the world’s languages. In terms of the portability of data (Bird and Simons 2001), it is highly desirable that linguists should use Unicode for all of their work with language data.[24]

There are however some practical problems which arise in using Unicode. Firstly, there is a crucial difference between *encoding* and *rendering*. As we have said, Unicode provides an encoding for a huge range of characters, and that encoding is stable across hardware and software platforms. However, in order to make practical use of this capability, the computers on which files are opened must be equipped with fonts that include glyphs for the characters which have been encoded. Although all major manufacturers support Unicode in principle, availability of wide-coverage fonts is proving to be slow in coming. Currently, for Windows machines, Arial Unicode MS is included as part of Microsoft Office (and thus is *not* installed on all computers by default). It has coverage of 50,377 codepoints which equates to 38,917 characters, including IPA and most major scripts. Rendering is in general very accurate, but there is a known bug which affects the rendering of double-width diacritic characters: these consistently appear one character to the left of their true position. Such problems will eventually disappear as new software versions are released, but in the meantime they cause considerable inconvenience.

There are other Unicode fonts which are valuable for linguistic work, of which we will mention only a couple. Lucida Sans Unicode has well-designed IPA symbols, but limited coverage of non-Latin writing systems. The Doulos SIL font family, distributed freely by the Summer Institute of Linguistics (http://scripts.sil.org/TypeDesignResources), is designed specifically to render IPA characters. It covers all IPA symbols and diacritics, which are rendered very accurately, but otherwise has narrow coverage. Numerous other Unicode fonts are also available from SIL, and more are under development. The Gentium font, in particular, provides wide coverage but is still under development (it has no bold typeface yet, for example).

A valuable resource for all issues related to Unicode fonts is the webpage authored by Alan Woods (http://www.alanwood.net/unicode/), and some specialist advice on Unicode and IPA is provided by John Wells (http://www.phon.ucl.ac.uk/home/wells/ipa-unicode.htm). The SIL pages also provide a wealth of information.

[23] In fact, in order to maintain heritage encodings from ISO 8859, Unicode also includes [é] as a single character. Unicode defines “normalization” tables that decompose such characters into the corresponding sequence of simple characters. There are also tables for normalization in the opposite direction (wherever such combined symbols exist).

[24] For those seeking more information about Unicode, Gillam 2003 provides a detailed but accessible introduction.

The second issue which arises in using multi-lingual text, in Unicode or in any other encoding, is that of data entry. Standard software packages such as Microsoft Office provide only limited facilities for inputting data in non-Roman characters. The Insert-Symbol command in Word allows such characters to be accessed, but it is designed for inserting single characters and is far from adequate for working with sizeable bodies of text in non-Roman writing systems. There are two basic approaches to this problem, although in practice the two sometimes overlap. Firstly, it is possible to redefine the mapping between the keyboard and the characters it inserts; this is most useful when a fixed set of characters will be used extensively (e.g., for IPA input), and the user can memorize the new layout or at least learn to find needed symbols quickly.

For Windows computers, the best known keyboard-remapping utility is Keyman (http://www.tavultesoft.com/keyman/). This is a powerful tool but, at least in previous versions, it had some drawbacks.[25] Keyman is a system-level tool: changing to a different keyboard mapping should alter the behaviour of the keyboard for all software. But because of the way keyboard input is handled in Windows, this is not actually possible, and the versions with which we have had experience have not in fact operated consistently in this fashion. In particular, the keyboard remapping did not operate at all when using the Microsoft Access DBMS (for further details, see Musgrave 2002). In previous versions, it was also quite difficult to create new keyboard mappings; one was reliant on mappings which had been created by other users and made available online. The current version seems to address this problem.

The solution which we prefer, therefore, is to use a specialized tool to edit Unicode text and then to import the prepared text into the database with which one is working. This approach, in our experience, allows much greater control over the process of producing accurate Unicode text.

For projects that do not involve intensive use of a single keyboard layout, it is sometimes most convenient to use an on-screen keyboard application. The user enters the desired text by “pressing” buttons displayed on the screen, and then pastes it to the database or another destination. Because it is easy to switch between sets of symbols and the symbol inserted by each key is always visible on the screen, this is a convenient method for moderate or light use. The TDS IPA Console (http://languagelink.let.uu.nl/tds/ipa/) is a java application specialized for IPA text entry; the on-screen keys are arranged in the shape of the familiar IPA symbol charts, and the application can be easily customized with additional symbols not included in the standard layout. There are also webpages that provide on-screen keyboards with similar functionality, but these cannot be customized.

Two other tools which are valuable for this purpose, again for Windows computers only, are Sharmahd Unipad and ELAN. Unipad is a Unicode text editor. A basic version is available for free download, but registration and payment is necessary for full editing capability. The program offers two modes for entering data: one can either select characters from a tabular representation of the Unicode character set, laid out in planes, or one can use a keyboard redefinition. The editor comes with a large number of keyboards (approximately 50) pre-defined, and several others are available from the Unipad website. But it is also possible to create custom keyboard layouts very easily; all that is involved is dragging the desired characters from the Unicode planes and dropping them onto the desired positions on a keyboard layout on the screen. When selected as the active keyboard, a mapping affects the behavior of the keyboard, but only in the Unipad editor. The keyboard can also be viewed on the screen, and characters can be entered by clicking on them with the mouse. The great advantage of this tool is that it is so easy to create special keyboards. If one is entering phonemic transcriptions of data from one language, access is needed to some of the IPA symbol set, but typically only a subset of those characters are used. Rather than having to negotiate a keyboard mapping which provides access to all the IPA characters, it is possible to make a keyboard with only those that are needed, with a significant gain in efficiency.

[25] We do not currently use this program and therefore we cannot comment on the performance of later versions. We also note that the tool was previously distributed freely, but is now distributed commercially.

ELAN is a tool for creating time-aligned annotations linked to media files, and is increasingly being used by linguists for transcribing primary data. We use it as an example of an indirect scenario for data entry: In many cases, the data entered into the database does not come directly from off-line sources, but from another electronic resource such as typed field notes, or (as in this case) a transcribed corpus. ELAN was designed and implemented by the technical team of the Max Planck Institute for Psycholinguistics, Nijmegen, and is distributed freely (http://www.lat-mpi.eu/tools/elan/). ELAN files are a specialized type of structured text file, saved in Unicode format; various aids to entering non-Roman characters are included in the interface. The user can access various keyboard remappings directly from the interface, with seven languages supported and some in multiple mappings. Two methods for entering IPA characters are also available, one using the SAMPA computer-readable phonetic alphabet (http://www.phon.ucl.ac.uk/home/sampa/), and the other using the Roman Typographic Root method. This requires the user to type the Roman character closest to the desired symbol and then select the symbol from a list which appears. The capabilities of ELAN for entering non-Roman characters are less sophisticated than those of Unipad, but for linguists who already use it for transcription, it has the advantage of allowing data entry into the database to be incorporated in their existing workflow.

4.3 Multimedia in databases

Recent technical advances have made feasible the presentation of linguistic data as audio or video material, or graphics. This advance opens up significant new possibilities for the discipline, and it is desirable to include such material in linguistic databases. While it is possible to store such content directly in a database, the relatively large size of audio and video recordings (relatively to the rest of the data) often makes it convenient to store such materials separately, and access them by storing links to them in the database.

Desktop DBMS packages now typically allow the content of a field to be a Uniform Resource Locator (URL) which points to a multimedia resource. The interface must include the necessary functionality to access the resources and present them to the user as required. A web-based database can easily handle multimedia, since web browsers already support multimedia content (sometimes via helper applications): Again, fields in the database can be used to store URLs, and the user interface scripts are programmed to present the relevant resources as hypertext links in the generated web pages.

A simpler application is the dynamic generation of maps or other graphics (e.g., statistical graphs or charts) on the basis of data in the database. On a web database, this can either be done “server-side” (a graphics manipulation plug-in on the webserver creates an image on the fly, which is sent to the user’s browser for display) or “client-side” (a fixed image, such as a map of the world, is sent to the user’s browser along with a script that controls the placement of additional information on it). A common use of the technique are the familiar typological maps, which display the distribution of one or more linguistic features as color-coded dots at the canonical geographical location of each language.[26]

5. Designing a database

5.1 Preliminaries

The relational database model is a simple, but very powerful basis for organizing data in a database: A relational database is a collection of tables, related (linked) to each other by means of foreign keys.

[26] Currently, the state of the art for such maps is to embed in the database interface a service provided by Google Maps: The Google service creates and manages a map of the world (or the desired part of it), which appears as part of the database interface. Behind the scenes, the database supplies geographical locations and annotations which are placed on the map by the user’s browser. A skilled programmer can set up the complete system in a few days. The websites of the World Atlas of Language Structures (Haspelmath, this volume) and the Typological Database System (Dimitriadis et al., this volume) utilize this method.

This so-called **relational structure** of tables and relationships determines the essential nature of a database: The user interface can present data in a very different way (or in many different ways, as we have seen), and there are additional features at the database level, such as indexes and access control directives; but these only build on the core properties of the database, which is determined by its relational structure.

So what is the *right* relational structure for a database? This depends on the situation, of course: on the nature of the data, and on its intended uses. Naturally, a database must have a way to store all information it is meant to contain; and this information must be organized in a way that allows it to be entered, viewed and modified in a straightforward way. There is a voluminous literature on the topic of how to design a database, with countless competing methodologies, procedures, and rules of thumb. While the database specialists disagree among themselves on the details, the underlying principle is simple:

> The structure of the database should reflect the logical organization of the data.

In particular, it is not the end-application or the user interface that determines how the database should be organized. If a database has a relational structure that suits the data, it will support any desired use of it (possibly after technical elaborations such as indexes, etc.). Conversely, incorrect design can make it impossible to put the database to new uses, or even to store all collected data. As a simple example, suppose that we are collecting information on reflexives, and design a database that allows exactly one reflexive per language. This design will require awkward conventions and work-arounds as soon as we encounter a language with two reflexives we need to describe.

An appropriate database design will have the following properties:

1. It will represent the required data and appropriate relationships between the units of data.
2. It will provide a model of the data that supports the operations that will need to be performed: entering, searching, and modifying data in the database.
3. It should be the *simplest* way to accomplish these tasks with acceptable performance.

The last consideration has mostly escaped our attention until now, and to some extent goes against the first two: A database is not a theoretical exercise but a practical tool; it should not be more complex than it needs to be – but it should be complex enough. For example, a person's name may consist of one or more last names, given names, connectives such as *von,* and perhaps other modifiers or honorific titles (*Jr.*, *Professor*). How this should be represented in a database depends on its purposes; a telephone directory or employee database should probably represent each of these parts separately, while a table of contributors to a linguistic database can often get away with fields for just a "first" and "last" name (loosely construed), or even a single field *Name.* For linguistic applications, the question is complicated by the theoretical perspective and goals of the database. In general, a database designer will model properties of the subject matter of the database much more precisely than other, peripheral properties. For example, while it is common to classify languages as having *trochaic* or *iambic* feet, this classification does not exhaust the range of variation in foot construction. The Stress Typology database (Goedemans and Van der Hulst, this volume), whose subject is the mechanisms of footing and stress assignment, provides a much more fine-grained set of classificatory parameters, from which the types *iambic* and *trochaic* can be computed as summary properties.

A database, then, is a model of the real world – specifically, of the part of the world we are interested in. But a model is not the same as the real thing: the data in a database, and hence its structure as well, represent an idealization of the world that is sufficiently detailed for our purposes but sufficiently abstract to be put into practice. Finding the right balance is something that comes with experience, and is arguably a bit of an art.

5.2 Normalization

To better understand the process of relational database design, we first consider two concrete properties that a correctly designed database must have:

1. Each cell in a database table should contain only one value (i.e., one piece of information).
2. Each piece of information should be entered in the database only once.

The process of discovering and correcting problems of this sort (and many similar ones) is known as **normalization.** A database that meets the first of the above criteria is said to be in *first normal form.* (The second of our criteria is addressed by several different normalization rules).

Consider again our table of language details: Suppose that we want to add information about the basic word order for each language, and that our definition of "basic word order" allows a language to have more than one (or that we decide to record multiple categorizations in case of doubt); suppose, moreover, that we decide to treat German as having both SOV and SVO basic word order.

Language Details

Language name	**ISO code**	**...**	**Basic Word Order**
English	eng		SVO
German	deu		SOV, SVO
(etc.)			

The above table violates our first rule (is not in first normal form), since both word order values for German have been entered in a single cell. This practice creates all sorts of technical difficulties, because database searching and sorting operations are designed to treat each table cell as a complete value. This might seem harmless in this toy example, but the problems rapidly multiply as things get more complicated: if we add a field for the bibliographic source of the word order information, it may be necessary to add two sources in the cell for German, and to somehow indicate which source is responsible for the which word order classification. We might just give these details in plain text in the bibliographic source field; or we might decide to use special notation, perhaps a backslash, between multiple values, and to adopt the convention that the first reference corresponds to the first word order value, and so on. But these conventions are opaque to the DBMS (i.e., unknown and unusable); and as they multiply, they add complexity to the system which quickly grows beyond what can be reasonably managed.

The solution is to structure the database in a way that allows multiple values for the word order attribute to be recorded for a single language. We replace our previous table, which provided one row for each language, with a new table *Word Order,* in which each row represents a basic word order of one language; German is then allowed to occupy two rows, solving our problem. Suppose, however, that our new table looks as follows:

Word Order

Language name	**ISO code**	**Area**	**Basic Word Order**	**Reference***
English	eng	Europe	SVO	Smith75
German	deu	Europe	SOV	Jones84
German	deu	Europe	SVO	Jones84
(etc.)				

This table violates the second of our conditions: It contains three fields, *Language name*, *ISO code* and *Area*, that must be the same for all records that concern a German word order. In other words, all copies of these values give the same information, which should appear only once. (On the other hand, the two values of *Jones84* on the *Reference* column do not represent the same piece of information: the SOV and SVO values might have come from different publications, so the fact that they are the same could not have been predicted). The solution, here, is to have two tables: One for languages, containing information such as name, population etc. (as before), and a second one containing only information related to the basic word order. The two tables are linked by means of a foreign key, *Language ID,* that must match the primary key (*ISO code*) of the *Language Details* table.[27]

Language Details

Language name	**ISO code**	**Area**	**Reference***
English	eng	Europe	Smith75
German	deu	Europe	Eth15
(etc.)			

Word Order

Language ID*	**Basic Word Order**	**Reference***
eng	SVO	Smith75
deu	SOV	Jones84
deu	SVO	Jones84
(etc.)		

These kinds of concerns, then, are among the issues we must address when designing a database. In the following sections we consider the process of database design, beginning with the general principles before we turn to the specifics.

5.3 The database design process

Building a database is a complex undertaking. We present here a commonly used, top-down process following Connolly et al. (1999): First, we collect sufficient information to gain a proper understanding of the problem; we then construct the database in several stages, beginning with a high-level model of the data domain.

1. Work out your needs, in writing. Examine real data and/or any paper forms or printed materials used for the same purpose in the past.
2. Carry out the high-level **conceptual design** (abstract design) of the database. This means thinking about how our data is organized, without worrying about the specifics of tables, keys and attributes.
3. In the **logical design** stage, we define tables and relationships that reflect the conceptual design.
4. The **physical design** stage involves the actual programming of the database, taking into account the features and limitations of our specific DBMS and user interface environment (with which we build the database client).
5. When the design steps have been completed, build the actual database.
6. Try it out.
7. Clarify your needs, and revise steps 1-6 as necessary.

Databases are by tradition complex entities that support very large enterprises. Under such conditions, the cost of change increases enormously once a system has gone into use: imagine needing to take thousands of automatic teller machines off line in order to revise the database that keeps track of their transactions! While linguistic databases are much simpler, it is still essential that a database should be planned and built as carefully as possible before it goes into use. Testing will almost always reveal

[27] We have not discussed the choice of primary keys for the two tables; we return to this type of example in a later section.

problems that can be easily corrected – as long as they are noticed at an early stage. It should also be noted that database-creation applications and tools are not built with the assumption that the design of the database might change at any time. If you add a new attribute to a table in Microsoft Access, for example, it will not automatically appear in existing forms for that table; the forms will have to be re-generated and customized from scratch, or the new field will have to be manually added to existing forms (a more complicated process).

The importance of planning ahead presents linguists with a chicken-and-egg problem: A research survey database is intended to collect data about a phenomenon that is not yet completely understood, so it is almost certain that an ideal model for it cannot be constructed until after data collection has been completed. We cannot address this difficult question here, beyond recommending the obvious: Do as much advance preparation as possible, and plan for change. For example, if the values for a field come from a fixed list (menu) of possible values, make sure that the list can be easily extended during the course of the project (and that someone actively involved in the project knows how to do it).

5.4 Preparation

Many databases are not designed by the people who are going to use them. It is important to collect the information necessary for the database designer to make the right decisions:

1. What kind of data will be stored in the database? Collect some *real* data, in electronic or in paper form, of the kind that will be entered into the database. If a substantial body of relevant data exists already, it could reveal patterns that will not be apparent from just one or two sample records (or, worse, from fake “data” made up by the designer).

   If the data is to be collected by questionnaire, prepare one or two completed questionnaires (again, with real data), before the database design is even started. The questionnaire itself is also a big source of information; but be sure to distinguish clearly between what will be written in the questionnaire and what will be entered into the database. For example, some exploratory questions might be intended to guide the consultant, and do not need to appear in the database; while some properties are not provided by the consultant, but will be determined by a database analyst on the basis of the completed questionnaire.

2. What will be done with the data? Think about specific scenarios you want to be able to carry out, including data entry as well as searching and browsing: will people want to see a list of all the phonemes in a language? Should they be able to search by geographic region? Etc. Think about the research questions you hope to answer with the help of the database, and the kind of information you will need in order to do so. It is best to involve several people in discussing the use of the database.

3. What kinds of users will be using the database? Typically, there will be privileged users who can enter and edit data in the database, and external users who can only search or display the data. For larger projects, there may be external consultants that should be given editing access to a single language only, etc. For a web database, it is common to initially restrict all access to authenticated users only, and to open up read access for external users after the data has been double-checked, and perhaps utilized for a particular research paper or dissertation connected to the project.

4. What will happen to the database after the conclusion of the project period? The data may be archived or made public on a separate web database; such future considerations can be taken into account during the database design, for example by providing suitable export functions and ensuring that appropriate standards are followed.

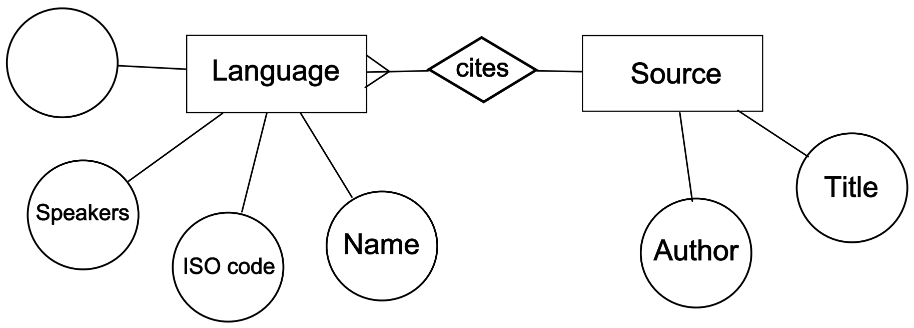
6. Conceptual design

In the **conceptual database design stage,** we determine the organization of our data at an abstract level. This is necessary, since the particular properties of relational databases are too low-level, in ways that we will presently see.

The core task in conceptual design is to identify the **entities,** i.e., idealized *objects,* that our database is about; and the **relationships** between them. More correctly, we must identify the **entity types** present in our database. An entity doesn't need to be a physical object. Rather, it is a notion or concept that we, in our project, consider to have "an independent existence", and can ask questions about. In other words, an entity is something that we collect data about. For a cross-linguistic survey database, we can start by identifying *languages* as entities that we will document: Our database might have information about German, Yoruba, Japanese, etc. We thus have our first entity type, *Language.* (Note again that our model must involve *types,* such as *Language,* rather than single objects such as *German).*

In our example we provided directory information about languages, such as name, ISO code, population, etc. These are **attributes** of the entity *Language.* (Entity attributes, as we will see shortly, are similar to the attributes of database tables but are more general). We also named a source for the information on each language. A book, or other bibliographic source, can also be said to have "independent existence": We can identify many attributes of a book, such as its title, author etc., independently of what we used the book for. We thus have a second entity type, *Bibliographic Source.* We also have a relationship between the two entity types: A bibliographic source can be *Cited* as the source of the demographic data on some language.

The entity-relationship model we have constructed can be displayed in a so-called **Entity-Relationship diagram (E-R diagram).** Each entity type is shown as a rectangle, with attached circles representing its attributes. The relationship is represented by a diamond, connected with lines to the two entities. The "crow's foot" on the language side of the relationship indicates that multiple languages could cite the same source.



Some heuristics for conceptual design

Entities are **things** (or **notions**) that our database is about. If we record information *about* something, that "something" is a candidate for being modeled as an entity.

To determine how to divide information into entities, and how to allocate attributes to them, consider:

1. What information goes together?
2. What information is dependent on other information?
3. What should be updated or deleted together?

For example, the vernacular text, gloss and translation of an example sentence clearly go together. It would make no sense to delete one of these while leaving the rest in the database (unless, of course, we are simply deleting an erroneous value until it can be entered correctly).

6.1 Relationships

The E-R diagram shown above also indicates the **cardinality** of the relationship between the two entities, which indicates the number of separate relationships that a single entity can participate in. A relationship is **one-to-one** if only one of each entity can be related. For example, the relationship *Is Capital Of* always relates a city to a single country (or with none), and vice versa. A relationship is **one-to-many** if one entity type can be related to several entities of the other type, but not vice versa. Example: A relationship *Location* between cities and countries, since each city is in only one country (if we ignore the possibility of divided cities, or treat such cases as two cities), but a country could be the location of many cities. Finally, a relationship is **many-to-many** if both entity types can be related to multiple entities of the other type. As an example, consider the relationship *Flies To,* between airlines and cities. For a linguistic example, consider a relationship *Occurs In,* between morphemic gloss labels and sentences. A particular gloss label can occur in many sentences, and each sentence can contain many labels in its gloss. The cardinality of a relationship determines (along with other factors) how it will be represented in the relational structure of our database, which we derive in the logical design stage.

A relationship can also be classified depending on whether it is obligatory for the participating entities. If every entity of some type must participate in a relationship, the relationship is **total** for that entity type; otherwise, the relationship is **partial** for that entity type. For example, every country has a capital, so *Is Capital Of* is a total relationship for countries; but some cities are not the capital of any country, hence *Is Capital Of* is a partial relationship for cities. Consider also the relationship *Describes,* between languages and reference grammars: every grammar describes *some* language, but there are languages for which no grammar exists; therefore the relationship is total for grammars, but partial for languages.

6.2 Entity attributes and relationship attributes

Entity attributes are a generalization of the attributes of tables. Like them, they express particular properties of the entity being described. Once we have identified the entities in our database, we solidify the design by identifying the attributes that we are interested in recording.

An attribute of an entity should depend *only* on the entity it will describe; for example, our language directory contains an entity type *Language* with information about languages, and an entity type *Source* with publication information (author, title, publisher etc.) about books and articles from which the information was drawn. In general, we also want to record the page number where a piece of information appears. Note that this is *not* a property (attribute) of the book itself; after all, we might have drawn many different pieces of information from different pages of a single book. Neither is the page number an attribute of the language. Rather, the page number is an **attribute of the relationship** *Cited,* which relates a language to a bibliographic source. In an entity-relationship model, relationships as well as entities can have attributes.

In defining an attribute, we should have some idea of the possible values it may take; these are the **attribute domain,** and might be a list of fixed strings, a number or monetary value (rare for linguistic applications), or perhaps free text. An attribute can also be classified as **simple** or **complex,** which refers to whether the attribute consists of multiple distinct parts. For example, an entity *Person* might have a complex attribute *Name* that we further subdivide into *First Name, Surname,* etc. An entity attribute is also allowed to be **multi-valued;** in our language directory example, *Alternate Name* is a multi-valued attribute. Similarly, *Person* might have an attribute *Phone Number* which we allow to take multiple values.[28] An attribute that can only hold one value (the usual case) is said to be **single-valued.** Exercise: Consider the entity *Book* (or *Bibliographic Source*). What kind of attribute is the

[28] Recall that table attributes can only hold one value for each record. This is why we say that entity attributes are a generalization (abstraction) of table attributes. In our discussion of logical database design, we show how complex and multi-valued attributes can be encoded in a relational database.

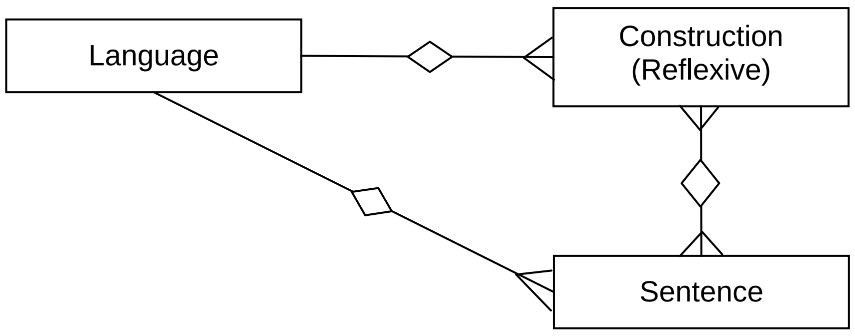
attribute *Author*?

We have just suggested that *Author* is an attribute of the entity *Book.* But isn't *Author* a separate entity, a special case of *Person*? The answer depends on our intended use for the database. Entities are things that we collect information about. Unless we intend to record biographical information about book authors, there is no reason to elevate them to the status of separate entities. An author is simply an attribute of *Book,* as far as we are concerned. On the other hand, a native speaker consultant for some language is generally of more interest: We may want to record their particular dialect, language biography, name and contact details, etc. Therefore we might decide to treat *Consultant* as an entity type in its own right.

6.3 A common pattern

Let us consider a linguistic example in more detail: A cross-linguistic database of reflexives includes data about reflexives in various languages; the entities in this database include *Language* and *Reflexive*; a language may have several distinct reflexives that need to be described, but a reflexive can only be considered as belonging to one language at a time (numerous European languages have a reflexive *se,* for example; but the *se* of each language has its own distinct properties, hence we must speak of separate, homophonous reflexives that are described independently of each other). Exercise: what is the cardinality of the relationship between languages and reflexives? And is it partial or total for each entity involved?[29]

Our database of reflexives will also include sentences exemplifying the use of reflexives; let us consider the status of these sentences in the database design. We can begin by thinking of them as "examples," which suggests that each sentence we record is obligatorily an example of some reflexive in our database. This is a common way of organizing a linguistic database. But we should point out that "example" is really a relationship between a sentence and a reflexive; hence we can consider *Sentence* to be our third entity type (or *Phrase,* or *Text*) if our examples are not exactly one sentence in length), and we let *Is Example Of* be a relationship between sentences and reflexives.[30] This point of view allows us to use the same sentence as an example of several things (of two reflexives, perhaps, or of several properties of a particular reflexive); this perspective also allows us to record sentences in our database that do not contain *any* reflexive (maybe we include the sentences for contrast, or we will identify a reflexive construction later). In other words, we have decided to treat the relationship between sentences and reflexives as many-to-many (since we can also give multiple examples of the same reflexive, of course), and as partial for both entities. We arrive at the following design:



[29] Answer: Every reflexive belongs to some language, hence the relationship is total for reflexives. However, we can conceive of a language with no reflexives (at least hypothetically), so the relationship is partial for languages. The relationship is many-to-one, since a language can have many reflexives, but a reflexive is by definition a property of a single language.
[30] More precisely, we can define a family of relationships of the type *Is example of property X,* where X is some property of reflexives that we are collecting information about (e.g., a morphological characteristic, a locality property, etc.)

This is an extremely common state of affairs for linguistic survey databases: The identifiable entities include *Language*, some form or *Construction* type that a language may have multiple varieties of, and *Sentences* that are used to exemplify each construction or some of its properties. To these three common entities we might add *Source* (for bibliographic references) and perhaps *Person,* for analysts, informants or other human contributors to the database (we might also treat each class of contributor as a different entity type).

If a sentence can only exist as an example of a reflexive, our database might allow users to browse sentences only by first selecting a reflexive, and then viewing the sentences that are linked to it. But if the relationship between sentences and reflexives is partial (for sentences), the user interface must provide a way to view or search for sentences which does rely on a reflexive that the sentence is an example of.

Exercise: Is the relationship between sentences and languages total or partial for each entity?

7. Logical design: From entities to tables

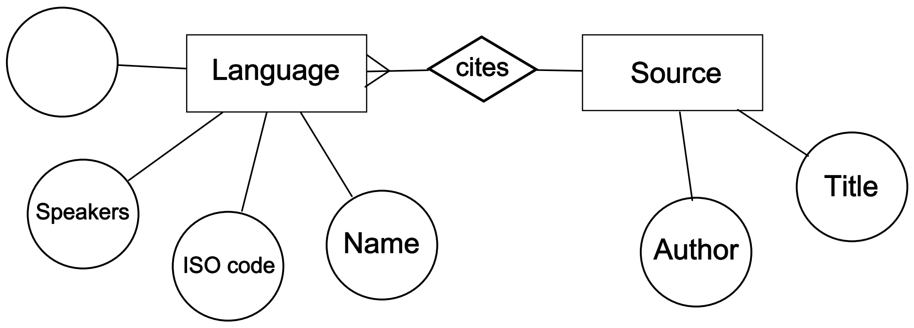
The conceptual design of our database, represented as an Entity-Relationship model, indicates the important characteristics of our database at an abstract level, allowing us to focus on the foundational issues of database structure without getting caught up in the technical limitations of relational database tables. But once the conceptual design has been completed, it must be used as the basis for a concrete relational design involving tables, keys and relationships. This is the function of **logical design,** the second stage in the database design process.

We can think of logical design as the process of converting an E-R model into a relational structure. In its simplest form, the conversion is as follows:

1. Each entity type normally becomes a table;
2. simple entity attributes become table attributes;
3. simple one-to many relationships between entities become table relationships represented by a foreign key.

More complex elements of the E-R model must be handled in other ways, which often involve the introduction of additional tables in the database; these additional tables, therefore, do *not* correspond to entities in the E-R model.

Let us consider the familiar example of the language directory; we have modeled it according to the following E-R diagram, which assumes that all information about a language will come from a single source. In other words, the relationship is one-to-many, with a given source (e.g. the *Ethnologue* directory) potentially providing information about several languages.



The logical design for this model begins with two tables, one for *Language Details* and one for *Biblio-*

graphic Source. Each entity attribute is represented simply as a table attribute.

We then **choose suitable primary keys:** For the *Language* table, we chose the ISO code which is guaranteed to be unique for each language (assume that we do not plan to identify any languages not recognized by SIL, the ISO code authority). For the *Source* table, we define as the primary key an arbitrary key chosen by the analyst entering the data, and expected to represent an author-year identifier in the customary style. However, the exact form of the arbitrary key is not important; we could have used an automatically assigned numeric value, which would require fewer decisions by the analyst but would have no mnemonic value.

Once we have the two tables, we want to indicate that each record in the *Language* table cites as its source a single record in the *Source* table. We do this by including a **foreign key,** *SourceID,* as an attribute of the *Language* table. This allows each language to be linked to just one source (since we are only allowed to enter one value for the *SourceID* attribute), with no restrictions on how many languages can be linked to a single source. In other words, this foreign key implements a many-to-one relationship between languages and sources: Many languages, one source.[31] All told, we have transformed the E-R diagram above into the following (already familiar) tables:

Language Details

Language name	<u>ISO code</u>	Speakers	Area	SourceID*
English	eng	309,000,000	Europe	Eth15
(etc.)				

Bibliographic Source

<u>ID</u>	Title	Author	Year	Publisher	...
Ashton47	Swahili grammar (including intonation)	Ashton, E.O.	1947	Longmans	
Eth15	Ethnologue: Languages of the world, Fifteenth Edition	Gordon, Raymond G., Jr. (ed.)	2005	SIL International	
(etc.)					

In this simple example, each entity corresponds to a table and each relationship corresponds to a foreign key relationship; there is no real difference between the entity-relationship model and the relational schema. But now let us consider a database on some phenomenon, e.g., a database of focus constructions, which includes example sentences. As we have seen, such a database will have the entities *Language, Focus construction,* and *Sentence.* In the logical design, we model each of them as a separate database table. But a single sentence might include instances of two or more focus constructions; i.e., the relationship is many-to-many. A foreign key in the *Sentence* table can only identify *one* construction, and hence is insufficient.

To encode a **many-to-many relationship,** we create a new table whose primary key is a combination of the primary keys of the two tables, *Focus Construction* and *Sentence.* (This is a *complex* primary key, consisting of at two attributes). Each record in this new table indicates a relationship between one construction and one sentence. Because the primary key (which must be unique for each table) is the combination of both keys, each sentence ID and each construction ID can be used multiple times.

[31] If we had instead added a foreign key to the *Bibliographic Source* table, the relationship would allow many sources to be linked to a single language – not what we want, in this case.

Focus construction

Name	**Language ID***	**…**
wa	jpn	
(etc.)		

Sentence

ID	**Language ID***	**Original text**	**Gloss**	**Translation**	**…**
101	jpn	hono wa, ...	book Top ...	As for the book, ...	
(etc.)					

Focus-sentence (relationship)

Focus ID*	**Sentence ID***
wa	101
wa	113
(etc.)	

Multi-valued attributes also require an additional table, which does not correspond to an entity in the entity-relationship model. Consider the language attribute *Alternate Name,* a multi-valued attribute of the *Language Details* table (since a language can have multiple alternate names). Each alternate name is a simple string, not an entity; but since we cannot store multiple values in a single attribute of the *Language Details* table, we create an additional table, *Alternate Name,* whose purpose is to store multiple names for a language. Each record in this table includes the language ID of the language that the record is about, declared as a foreign key that must match the primary key of the *Language Details* table, *ISO code*; because the language ID is *not* the primary key of the *Alternate Name* table, it is possible to have multiple records for the same language ID.

Language Details

Language name	**ISO code**	**Speakers**	**Area**	**SourceID***
Swahili	swh	772,000	Africa	Ashton47
(etc.)				

Alternate Name

ID	**Language ID***	**Name**
101	swh	Kiswahili
102	swh	Kisuaheli
103	swh	Arab-Swahili
(etc.)		

For illustration, the above table uses an arbitrary numeric primary key, *ID*; this allows us to enter any values we wish – even to enter the same alternate name multiple times for the same language. We could have instead declared the two attributes *Language ID* and *Name* as a complex primary key, which would make it impossible to accidentally enter the same alternate name twice for the same language:[32]

[32] It would still be possible to enter the same alternate name for *different* languages, of course.

Alternate name

<u>Language ID</u>*	<u>Name</u>
swh	Kiswahili
swh	Kisuaheli
swh	Arab-Swahili
(etc.)	

Note that (with either version) the table *Language Details* makes no mention of the *Alternate name* property; the relationship is encoded in one direction only, by declaring the field *Language ID* as a foreign key that matches the primary key of the *Language details* table.

A **complex attribute** (e.g., *Name*) does not need to be modeled in an additional table. We can simply model each component of the complex attribute as a separate table attribute (e.g., *First Name, Surname,* etc.).

We do need a separate table in the case of a **relationship with attributes,** even if the relationship is not many-to-many. Recall our example of a relationship with attributes, the *Citation* relationship in our *Language Details* example. We have seen that the page number on which a particular language property is discussed is not a property of the bibliographic source, but of the relationship between a language and a source. Therefore, we can create a table for the relationship; its primary key is a combination of the primary keys for *Language* and *Source,* and it contains the additional attribute *Page Numbers* where the appropriate page range appears.

Language details

Language name	<u>ISO code</u>	Speakers	Area
English	eng	309,000,000	Europe
(etc.)			

Bibliographic source

<u>ID</u>	Title	Author	Year	Publisher	…
Ashton47	Swahili grammar (including intonation)	Ashton, E.O.	1947	Longmans	
Eth15	Ethnologue: Languages of the world, Fifteenth Edition	Gordon, Raymond G., Jr. (ed.)	2005	SIL International	
(etc.)					

Citation (relationship)

<u>Language ID</u>*	<u>Source ID</u>*	Pages
eng	Eth15	45
deu	Eth15	42
swh	Ashton47	1-21
(etc.)		

7.1 Enumerated attribute values

There is another important use for tables that do not correspond to an entity: To hold *lists of possible values* for attributes whose value must be taken from a fixed set of choices (**enumerated** attributes). For example, a cross-linguistic database might assign each language to a geographic macro-area from the following pre-determined set of options:

Africa
Eurasia
SE Asia and Oceania
Australia and New Guinea
North America
South America

Using an ordinary text field to hold these values would be inefficient and error prone – the analyst would have to type the above values over and over, and nothing would prevent them from entering minor variations ("S. America") or even areas that are not on the above list (e.g., "Central America"). For this reason, it is much preferable for the user interface to show the above areas as a fixed set of choices from which the analyst must choose, in the form of a drop-down menu, "radio buttons", or perhaps in some other way.

Any system used to create the user interface will provide ways of creating menus, etc., that display a list of possible values. But where do these values come from? The set of possible values is part of the structure of the database, since it represents the domain of the attribute in question. The best approach is to build such information into the database itself, as follows: We create a table whose sole purpose is to represent an attribute domain, e.g., "macro-areas". This table could have a numeric primary key, or the names of the macro-areas could themselves be the primary key:

Linguistic Macro-area

ID	**Area**
1	Africa
2	Eurasia
(etc).	

To use this, we declare the *Area* attribute of our *Language Details* table to be a foreign key, pointing to this table. The DBMS can ensure that any foreign key value we enter matches a record in the *Linguistic Macro-area* table; the result is that our database is guaranteed to store only legal values for this attribute.

Language name	**ISO code**	**Speakers**	**Area***	**SourceID***
English	eng	309,000,000	2	Eth15
(etc.)				

It is important to realize that the table *Linguistic Macro-area* is part of the design of the database: The six records it contains are defined before data about any language has been entered into the database, and it will never grow beyond these six rows as we enter data for different languages – unless we decide to amend our list of macro-areas. In other words, this table *does not contain data* about any language or other entity; it is an auxiliary device representing an attribute domain.

This arrangement is not the simplest one imaginable: Any environment for building our user interface could allow us to store the list of possible values in an internal list, without involving a database table. We do not recommend this approach, for two important reasons: First, it separates the attribute domain information from the database definition, and embeds it in the interface application. If we later need to use the database independently of this interface application, the domain information will be lost. For example, we might wish to dump our desktop database and create a web interface for it, or to create an improved interface application to replace our first effort. In many cases, the database will contain only a small numeric value (between 1 and 6, in our example), which the interface application must interpret and display as the name of a macro-area. In the above example, we might end up with just numeric codes for the area, with no indication of which part of the world corresponds to area code 2 –

this information can only be retrieved by examining the original user interface, if that is still possible.

The second reason to avoid this approach is that it makes it harder to extend or revise the enumerated values during the course of the project. While we can reasonably hope to arrive at a list of macro-areas that will not need to be changed during the lifetime of our project, the same is not the case with classifications about which we do not have a clear picture in advance. For example, the *Sentence* entity in our database of reflexives might have an attribute *Clause Embedding Type,* for the type of complementation involved in multi-clause sentences. Whatever classification of embedding types we might initially arrive at, it is quite likely that we will wish to amend or extend it during the course of the project, in the face of unexpected constructions from new languages. Such changes should be undertaken with care, of course, to avoid invalidating data already in the database; but they are very often necessary in research-oriented databases. Hiding this kind of information in the user interface makes it harder to amend; the change typically requires a programmer, and since each interface environment is different, it can be difficult to locate someone with the necessary skills. It is easy, on the other hand, to arrange for a way to modify the contents of a database table (e.g., by providing a suitable form); even if no such provision was made in advance, database tables are well-understood and easier to modify than an unfamiliar collection of scripts.

The techniques we have discussed can be combined as needed: For example, we might want an enumerated attribute that takes multiple values: E.g., in our reflexives database we wish to indicate the alternative meanings that a reflexive construction might have: In addition to being reflexive, for example, the Italian reflexive *se* could have reciprocal, middle, impersonal, or certain other meanings. The possible meanings we identify are recorded on a list, as indicated in this section; but since we need to allow more than one alternative meaning to be selected, reflexives and meanings are in a many-to-many relationship. Accordingly, an intermediate table is used to relate reflexives and possible meanings, as described in the previous section.

7.2 Managing enumerated domains

Enumerated attributes are very common in linguistic applications; by creating a table to store the possible values of each domain, linguistic databases tend to become littered with dozens of small tables which can become difficult to manage – especially if we must make provisions for a form that will allow editing of the domain values, a means of activating this form, etc. While the approach we have described so far is quite commonplace, we will briefly present a refinement that can greatly simplify the management of large numbers of enumerated domains: Instead of creating a separate table for each domain, create a single table which will hold all enumerated values, listing the domain to which each value belongs alongside its ID:

Enumerated Value Definition

Domain	Value ID	Value Label
MacroArea	1	Africa
MacroArea	2	Eurasia
MacroArea	3	SE Asia and Oceania
(etc)		
Embedding	1	Tensed Complement
Embedding	2	Infinitival complement
Embedding	3	Paratactic construction
(etc).		

The user interface, then, will provide as possible values of the *Area* attribute not the entire contents of some table, but only the values of the above table that have the *Domain* “MacroArea”.

The above table has a complex primary key consisting of the attributes *Domain* and *Value ID* (as indicated by underlining). This ensures that we will not accidentally define the same value ID twice for the same domain. Other arrangements are also possible.

In addition to reducing the number of distinct tables, this approach has the considerable advantage that a single form, or group of forms, can be used to manage all the enumerated values – even values for domain types that have not been created yet. In addition, it is easy to add a column to this table for the documentation of each value; this makes it possible to document the meaning of each possible value in such a way that it is part of the database. (How this information is presented to the users, of course, is up to the designers of the interface).

8. Physical database design and beyond

Logical database design, when properly carried out, converts a high-level model of the information domain into a system of tables, keys and relationships that can be directly expressed in the "relational algebra" that underlies relational database management systems. The next step is to create an actual database that realizes this design (which is in any event preliminary, since it is certain to be modified down the road). This is not a trivial step; while all relational DBMSs are based on the same abstract mathematical model (the aforementioned relational algebra), they differ in a multitude of details that must be taken into account at this stage: The designer must choose from between several "table engines" provided by the DBMS for the actual storage of data, each of which with different features and performance benefits; each database attribute must be given an appropriate data type (and maximum size, as appropriate), from the many varieties of numeric, text and binary data types provided by the DBMS. (See section 4.1). Databases may also have limitations such as being unable to use a certain type as a primary key, or extra features not supported by most other databases. From our perspective, issues of character encoding are particularly important; for a cross-linguistic database created today, use of **Unicode** is an absolute requirement if the data will include non-Latin text. There is no excuse for using a different character encoding method for a cross-linguistic database today.

The database designer should also consider indexes at this stage. An index allows the DBMS to find records with a certain value without examining all records in a table, and can provide a noticeable speed-up for large tables and complex queries; many DBMSs will automatically index certain kinds of attributes (e.g., primary keys), but additional indexes are sometimes useful. Indexes do slightly slow down the performance of the DBMS during data entry, since they need to be updated whenever indexed data is inserted or modified. However, this time cost is minuscule; if your database does not run a busy airport, you will not notice any delay due to updating indexes.

More generally, this is the stage where performance issues should be considered. We have emphasized that the design process should focus on correct design, which is the most important factor in obtaining good performance. If this rule is observed, the comparatively small size of linguistic databases means that there is usually no need for further attention to performance: With a properly structured database, correct queries and a few indexes at the right places, few linguists are likely to encounter performance problems in the speed of database queries. If a problem does arise, it is almost certainly due to a flaw in the design of the tables or queries. Narrow attempts at "optimizing," such as storing numeric codes instead of text values in the hope that this will reduce processing time, are likely to produce negligible runtime improvements; but the added complexity they introduce will require extra time to code and debug, which will easily exceed the expected time savings from speeding up the database.

Based on all these considerations, the database developer will build the tables and relationships that constitute the database; depending on the DBMS and supporting applications used to manage it, the database might be defined with a series of SQL commands, or interactively through a graphical management interface provided by the database application.[33]

[33] For the open-source stand-alone DBMSs MySQL and Postgresql, there are independently developed applica-

An actual, working database is only one half of the data management system. The other half is the interface application, or applications, that will be used to get data into and out of the database. The interface application should provide forms, reports and/or other facilities for carrying out all the actions that the project needs: Entering data of various kinds, and searching for, displaying and summarizing information. We will not attempt even a cursory overview of the issues involved in designing the interface to a database, because these depend, much more than the relational design of the database, on the particular needs and procedures of the project; each project will have different needs and will therefore require different solutions.

The view of the data presented by the user interface can be quite different from that developed during database design, because the two are based on different considerations: While the relational design of the database should be based on the internal logic of the data, the design of the user interface should be based on what we want to do. In particular, the user interface will often group in one form information from different tables, or will present only parts of a table in a form, as appropriate for the use at hand. Our database of reflexives might show, next to each sentence, the name and language family of the language that the sentence belongs to; this information is drawn from the *Language* table, by means of a suitable query or view. For a questionnaire-based survey, it could be useful to have an application that presents each question of the questionnaire, in the correct order. Whether such features are worth the effort, or whether they are even a good idea, really depends on the specific needs of a project, including the degree to which the questions and answers can be expected to change in the course of data collection.

At a minimum, the user interface should display the data in a way that allows it to be easily interpreted, and should support the required operations (data entry, modifications, searches). Implementation features, such as numeric IDs or tables implementing many-to-many relationships, should be hidden from the user. Beyond this, the issues are no different from those that arise for any software development task; software development is a large and complex topic, and one that we have neither the resources nor the qualifications to offer concrete instruction on. In closing, therefore, we limit ourselves to the following general recommendations:

1. Get skilled help if possible; but stay actively involved if you do.
2. Consult several future users.
3. Test, and test again.
4. Anticipate the need for more revisions down the line.

9. Some problems, solutions, and limitations

While relational databases provide a powerful framework for data management, they are not equally well suited to all uses. In this section we touch on some potentially tricky issues, and perceived or real limitations of relational databases. Naturally the discussion in no way exhausts the possible sources of trouble.

9.1 Sequences and multiple values

A common source of difficulty involves multiple values. It is not uncommon, in linguistic description, to assign multiple values to a single property. For example, a certain construction may have more than one information-structure function, a language may be known by more than one name, a verb may have more than one meaning. Databases, as we have seen, should assign only one value to each attribute of a record; if we wish to allow more values, it is necessary either to create duplicate attributes (*Function2, Function3,* etc.) or to use an additional linked table for the multi-valued attribute. Both methods introduce complexity to the database that seems out of proportion with the modest goal of

tions that provide a graphical, web-based management interface.

occasionally recording multiple values for an attribute.[34]

Suppose that in our database we normally record one basic word order for each language, but a few tricky cases require more than one order to be entered. In this case it might be enough to add an extra field *Secondary Word Order*, or some such. But note that such a field is not interchangeable with the first word order field; any queries must explicitly take into account the fact that there are two (or more) places to store a word order. Moreover, if there are any attributes that represent properties specific to each word order, they will need to be duplicated as well. In more complex cases, it is not usually a good idea to create duplicate attributes.

A variation of this problem arises when it is necessary to store sequences of atomic elements, especially if the maximum number of elements cannot be specified. This is the case if our database is to be used for the analysis of words (as sequences of morphemes), or of sentences (as sequences of words, or of words and morphemes).

In designing a table to store the morphological structure of words, we might wish to store the entire decomposition of each word in one record; to do so we'd need to place each morpheme in a separate column, i.e., we'd need duplicate attributes *Morpheme1, Morpheme2, Morpheme3*, etc. Because columns need to be pre-defined when the database is designed, it is necessary to assign an arbitrary upper limit to the number of morphemes which can make up a word – but what do we do if a word with a greater number of constituent morphemes is found?

ID	**Form**	**Language***	**Morp1**	**Morph2**	**Morph3**	**Gloss1**	**Gloss2**	**…**
37	wa-li-on-an-a	swh	wa	li	on	c2	Pst	
(etc).								

In this example, this design is in any event inadvisable because each morpheme will need to be described by additional attributes, each of which *also* needs to be duplicated for the maximum possible number of morphemes; maintaining duplicates of so many attributes will quickly make it impossible to manage the database, or to write useful queries. The solution is provided by the design principles we have discussed: Morphemes should be treated as a separate entity type, in a many-to-one relationship with words; therefore there should be a separate table, in which each morpheme occupies a separate row. This eliminates the duplication of attributes and simultaneously removes the limit on the number of morphemes, albeit at the cost of considerable added complexity for the database programmer: The table of morphemes must include information about sequential position (rank), the morpheme itself and a foreign key pointing to the word being analyzed – all this in addition to the information we want to record *about* each morpheme. A single word is now spread over several rows of two tables (*Word* and *Morpheme*), which must be linked to other tables via foreign keys as necessary. The user interface must present this information in the familiar form of a string of morphemes, and convert in the opposite direction on data input.[35]

[34] For this reason, perhaps, Filemaker departs from the relational model and can handle multiple attribute values without requiring an additional table to be created.

[35] Note that this is a simplified example; in a real database of this sort, each morpheme might point to an entry in a dictionary of morphemes, so that the gloss and properties of the morpheme *–an* 'Recip', for example, need only be stored in one place. The words themselves would probably be linked to a text or sentence, and not directly to the language.

Word

ID	Form	Language*	…
37	wa-li-on-an-a	swh	
(etc).			

Morpheme

Word ID*	Rank	Form	Gloss
36	1	wa	c2
36	2	toto	child
37	1	wa	c2
37	2	li	Pst
37	3	on	see
37	4	an	Recip
37	5	a	FV
(etc).			

This design is not as simple as one might have hoped for, but it is necessary if we wish to manage detailed information about each morpheme in our text. If we only need to know where the morphological boundaries are, it may be simpler to store each complete word as a single string, and let the user interface manage and display it appropriately. For a concrete example, we turn to the equivalent task involving sentences. Consider a multilingual database that focuses on a certain phenomenon and includes glossed example sentences. The following structure is often sufficient for the sentences:[36]

Sentence

ID	Original	Morphemic tier	Gloss	Translation	…
307	Watoto walionana	Wa-toto wa-li-on-ana	c2-child c2-Pst-see-Recip-FV	The children saw each other	
(etc.)					

Instead of treating every morpheme as a separate value, we have allocated one value (attribute) for each tier of the sentence. (Note that each table cell contains one text string: they are shown on two lines for typographical reasons only). The database interface can parse each tier and display them in a word-aligned manner in the customary way; gloss labels used in the gloss tier can even be extracted and looked up in a table of glosses, for validation or to display their definition. If all this is handled by the user interface application, there is no need to burden the database schema by modeling each morpheme as a distinct value. In the web interface, the underlined gloss labels are hyperlinks that bring up a definition from a table of glosses:

Swahili (swh), strategy: -ana **ID 544**

(ok)	wa-toto	wa-li-sem-an-a
	c2-child	c2-Pst-speak-Recip-FV
	The children spoke to each other	

The downside of this approach is that we lose some of the power of relational databases. For example, we must use a full-text search to find sentences in which some morpheme occurs; while text searches are more than fast enough for a linguistic database on today's computers, they are slower than relying

[36] We suppress other properties of the sentence that we may be interested in.

on an index, and might lead to slower performance for very large or complex applications. More importantly, it is no longer possible to store detailed information for *each* morpheme in a word or sentence. For such uses, the more complex design presented above is necessary.

Note also that we have reduced the complexity of the database structure by shifting some responsibilities to the user inteface. Our interface application must ensure, for example, that the sentence tiers contain equal numbers of words and morphemes. And if a new user interface is needed at some point (perhaps to convert a desktop database into a web database, or to replace an obsolete web interface), this functionality will have to be implemented again.

The right design choice depends on the nature of our interest in the data: If our database is intended to support the detailed analysis of primary data, it is probably a good idea to fully represent the morphemic decomposition in the relational design. If our database is targeted to some morphosyntactic phenomenon (e.g., reflexives) and the sentences serve as supporting material, the simpler design just sketched may well be sufficient.

9.2 Boolean attributes in theory and practice

Boolean (true/false) attributes are frequently used in typological databases, where many fields may contain information about whether a given language has a certain property or not. It is also not uncommon in linguistic theory to model a multi-valued property as a combination of binary features; for example, Chomsky and Halle (1968) treat the three-way vowel height opposition *low, mid, high* in terms of two features (since two binary features can occur in four different combinations, one of the four must excluded from occurring); Goedemans and Van der Hulst (this volume) classify primary accent placement patterns in terms of seven binary parameters; etc. These analyses were motivated by good linguistic reasons (theoretical and/or empirical), and linguists who use them in their work may well decide to design their database along the same lines. Care should be taken to document the behavior and meaning of the novel parameters (especially for a publicly accessible database), since their meaning will not be obvious to a viewer unfamiliar with the framework in question. Often the data can be presented in terms of both the formal features and the traditional classification (which the database, or the interface application, should be able to compute from the formal features).

While decomposing a linguistic property into features can be desirable for linguistic reasons, the technique should never be used purely as an attempt to "optimize" the performance of the database: A property that can take a restricted number of values should be modeled as an enumerated attribute, using the methods we have already discussed. Working with a single multi-valued attribute is simpler, more convenient and more intuitive; and it is in fact faster than a query that must address several artificial Boolean features in order to reconstruct a multi-valued attribute value.[37]

A related technique is to encode a multi-valued property as a collection of binary "characters," Boolean variables that each determine whether the property has a particular value; e.g., the property *Basic Word Order* could be represented as the Boolean variables *"BWO = SVO"*, *"BWO = SOV"*, and so on for all seven possible word orders (we include the word order *"Free"* among the possibilities). In this case the database, or the analyst, needs to ensure that only one of the seven can take the value *True* for any language (unless, of course, the project defines "basic word order" in such a way that a language could have more than one). While characters are useful for certain types of statistical analysis, we recommend always storing such properties in non-decomposed form, as an enumerated attribute; if char-

[37] See Dunn et al. (2003) for an example of the decomposition of typological features as binary characters, and Wichmann and Kamholz, (in press) for additional comments on the dangers of decomposing multi-valued properties.

acters are needed, it is straightforward to generate them dynamically with an SQL query.[38]

The general principle we have advocated applies here as well: The database should be designed so as to model the organization of the data (to the best of our understanding of it). While it may be justifiable to simplify the design in respects that are not important to our needs, modifications that make the database *more* complex than required by its conceptual model are never a good idea.

9.3 Design changes

The most important characteristic of a database is that it is intended to manage a *structured* collection of data. While we have seen that a database consisting of tables and relations can support an interface that is not at all table-like, it is still true that a database must be carefully designed and structured before it is put into use. This presents problems for the creators of research databases, who might expect to work with the data for some time in order to develop an understanding of how it should be analyzed and structured. In such cases, it is important not to build restrictions into the software, and especially into the database model, that may conflict with the eventual analysis that will need to be expressed. (This is another reason that unnecessary “optimizations” should be avoided). Imagine a cross-linguistic database that includes phonotactic information; the database might have been built on the assumption that it is enough to characterize each language by its maximal syllable template (e.g., *CVN*), which can be stored as a simple attribute in the *Language* table. But suppose that the researchers, during the course of their work, find it necessary to record separate data about each phonotactic pattern in a language (e.g., to study the properties of *CV* and *CVN* separately). As we have just seen, this requires syllable templates to become a separate entity that is in a many-to-one relationship with languages. This will involve a fundamental change in the design of the database.

A simpler problem can arise if a database includes a hard-coded list of enumerated values; e.g., verb argument types such as *direct object, indirect object, prepositional phrase*, etc. In the course of data collection, the users of this database might encounter languages that require an argument type that is not on this list; if the programmer who created the database is gone and has not arranged for an easy way to extend the list, considerable inconvenience will result. (The design sketched in section 7.2 is an effective way to avoid this problem).

9.4 Hierarchical structure

An important limitation of relational databases is that, by the nature of the “relational algebra” they are based on, they are not very good at managing recursive hierarchical structure. Parsed syntactic trees, in particular, are not a good fit for the kinds of queries a relational database can support. In such cases the parsed tree might be stored in a database, perhaps along with indexing information, but many kinds of searches would have to rely on functionality outside the database proper. Other strategies are needed if representing hierarchical information is important on a project (see Van Halteren 1997:42-46 for discussion). Similarly, database queries are not the most convenient tool for large text corpora (parsed or unparsed). In such cases, especially for small projects, it may be most straightforward not to rely on a DBMS at all.

[38] If the basic word order is stored in a field called Order, we can generate characters named charSVO, charSOV, etc., as follows (consult an SQL manual for documentation on the CASE statement).

```
SELECT *,
        CASE Order WHEN ‘SVO’ THEN 1 ELSE 0 END charSVO,
        CASE Order WHEN ‘SOV’ THEN 1 ELSE 0 END charSOV,
        . . .
   FROM “Language Details”;
```

10. Conclusion

This very brief introduction could not hope to provide a complete guide to designing or building databases. We have focused on conceptual rather than practical issues, especially those which in our opinion are easily overlooked during the hands-on process of learning one's way around a database application. Our goal was not so much to allow the reader to be self-sufficient as to make it possible to think about the process of database design at the right level of abstraction. We hope that this will be of help to readers who can already create simple databases (e.g., with Microsoft Access or MySQL and PHP) but lacked the conceptual perspective developed here, as well as to those who need to discuss the design of their database with a programmer.

References

Bird, Steven and Gary Simons (2003) Seven dimensions of portability for language documentation and description. *Language* 79:557–582.

Chomsky, Noam and Halle, Morris (1968). *The Sound Pattern of English*. New York: Harper & Row. 1968.

Connolly, Thomas, Carolyn Begg and Anne Strachan (1999). Database systems: A practical approach to design, implementation, and management. 2nd edition. Boston: Addison-Wesley.

Codd, E.F (1970) A relational model of data for large shared data banks. *Communications of the ACM* 13:377–387

Dunn, M., Terrill, A., Reesink, G., Foley, R. A., and Levinson, S. C. (2005) Structural Phylogenetics and the Reconstruction of Ancient Language History. Science, 309(5743):2072–2075.

Ferrara, M. & Moran, S. (2004) Review of DBMS for Linguistic Purposes. Proceedings of E-MELD 2004. Online publication, at http://www.linguistlist.org/emeld/workshop/2004/proceedings.html.

Gillam, Richard (2003) *Unicode Demystified* Boston: Addison-Wesley.

Musgrave, Simon. (2002) Taking Unicode to the (Microsoft) Office. *Glot International* (9/10):316–319.

Van Halteren, Hans. 1997. *Excursions into Syntactic Databases*. Amsterdam: Rodopi.

Wichmann, Søren and David Kamholz. In press. A stability metric for typological features. *Sprachtypologie und Universalienforschung* 61.3: 251–262.