Chapter A02. Understanding Database Design

João Eduardo Ferreira[1] and Osvaldo Kotaro Takai[2]

Created: September 12, 2007.

Context

Today, more than in any other moment in history, public and private institutions depend on the ability to keep precious, up-to-date data regarding their activities in order to manage business and research, as well as to continue being competitive in market. In particular, bioinformatics applications often generate very large data sets that are stored through flat files and spreadsheet formats. In recent years, with the advance of bioinformatics and biologic techniques, the amount of data stored in computer systems has grown exponentially. Hence, it is essential that bioinformatics researchers employ tools to simplify the management of data, thereby allowing for the quick extraction and integration of useful information. In particular, the use of data mining techniques requires data to be stored in structured databases. In a database, information is more readily accessible and easy to query, good examples in the Tropical Diseases area are PlasmoDB and ToxoDB. However, data stored in simple files is still the rule rather than the exception in the bioinformatics area. This chapter will present the main concepts of database design and query, and will help the reader to understand the power of the database approach to storing data, as well as how queries can be constructed from a specification of database architecture.

The set of tools designed and employed for this purpose is known as a **Database Management System (DBMS).**

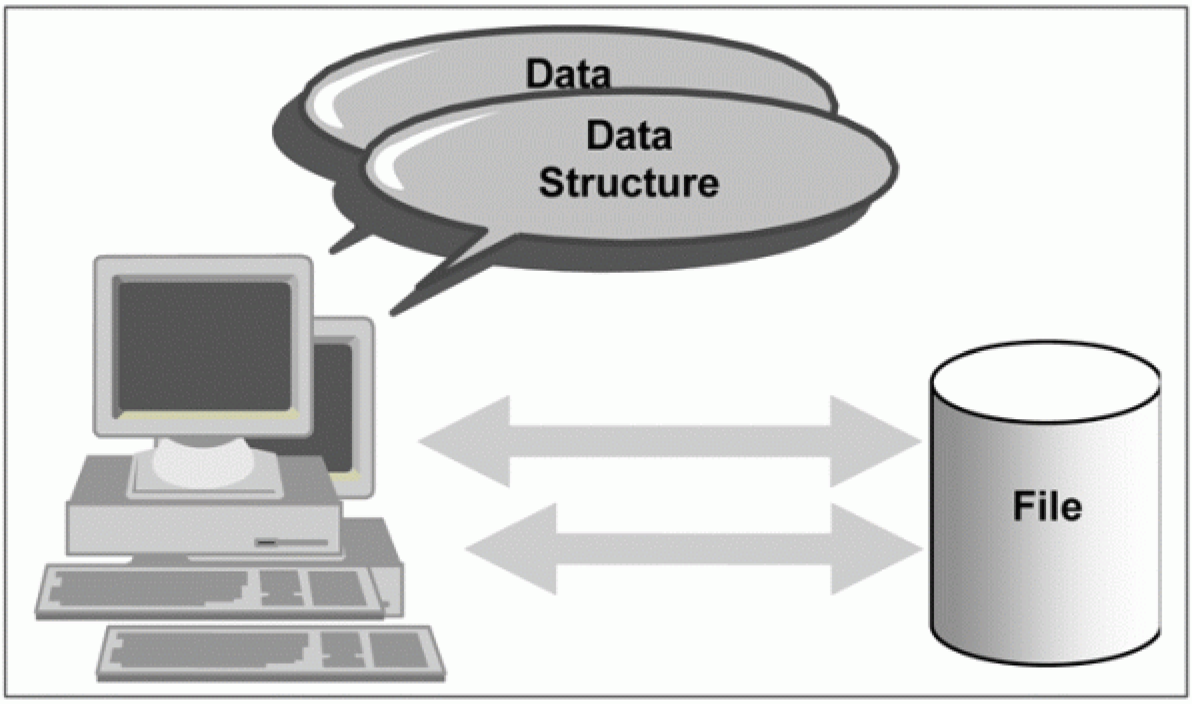
We will use a simple and practical example of a database of clinical data to illustrate the main concepts related to the creation and the maintenance of a database. In Section 1, we will give a brief introduction to DBMS's. Section 2 discusses the conceptual design of databases. Section 3 presents the *Structured Query Language* (SQL), a standard language used to create and manipulate databases. In Section 4 we present the Relational Model of data that is used to implement the database design.

1. Introduction

In the early years of computer technology, the main purpose of computer programs was to store and manipulate data. These programs recorded their data on disk according to their own structures. All programs that were not familiar with the structure of the data were unable to use them.

If various programs needed to share data from the same file, they would have to know and manipulate the same structures. The figure below gives a good picture of these situations.

Author Affiliations: 1 Institute of Mathematics and Statistics; University of São Paulo; São Paulo SP; 05508-900; Brazil; Email: jef@ime.usp.br. 2 Institute of Mathematics and Statistics; University of São Paulo; São Paulo SP; 05508-900; Brazil; Email: otakai@gmail.com.



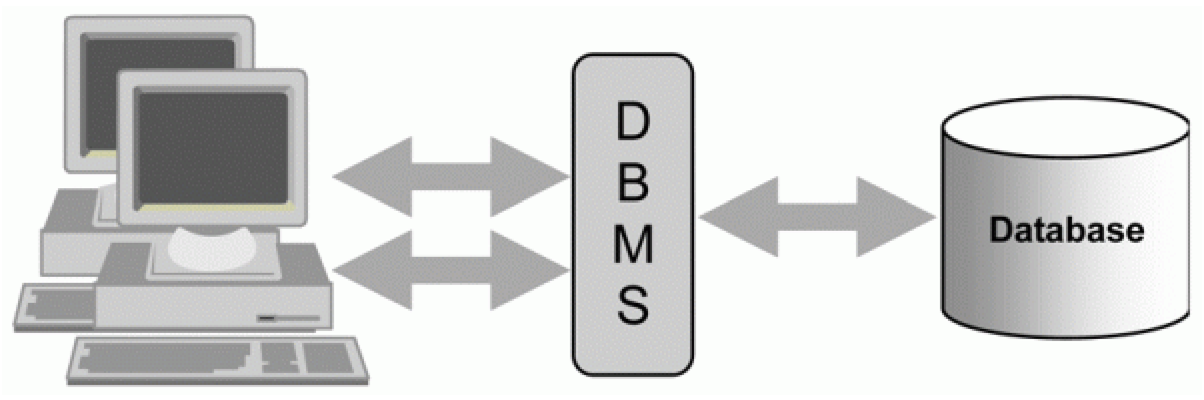
If any one of these programs needed to change the data structure, all other programs (which were accessing the same file) would have to be changed, even if the alteration took place in data that was not manipulated by all the programs. This led to a major problem: *guaranteeing the unity of the data structures among the different programs due to the existence of redundancies*.

In order to avoid this problem, an intermediate system was set up to convert the data from the format they were recorded in the file to the specific format recognized by each program.

With this intermediate program, the following occurs:

- The programs "see" only the data that they have to interact with;
- The programs don't need to know the details of the physical recording of their data;
- The programs don't have to be modified when the data structure is modified;
- The alterations are concentrated in this intermediate system.

Eventually, these intermediate systems were able to manage many different files. This group of files was named **Database** and the intermediate system became known as **Database Management System** (or **DBMS**). The next figure shows how a DBMS accesses data.

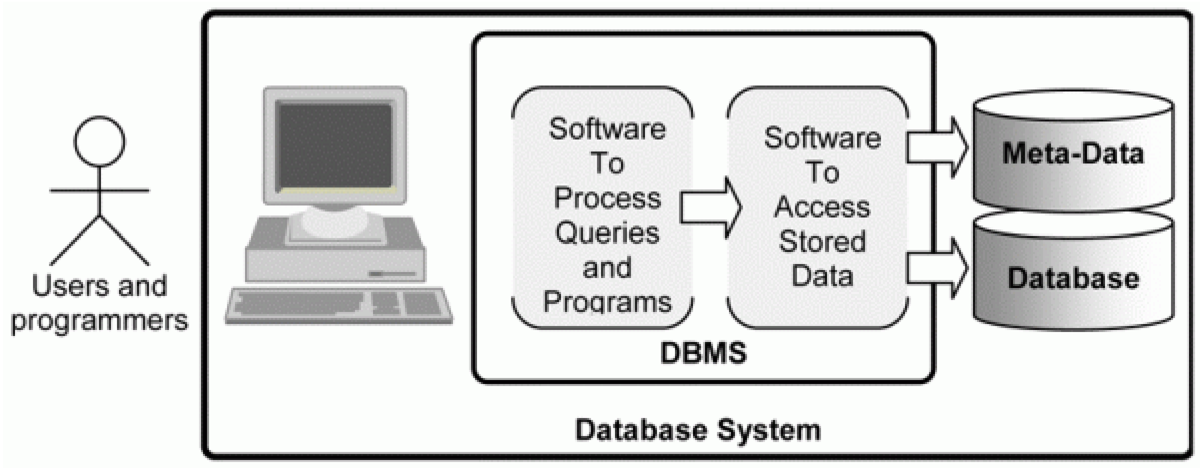


A **database** is currently defined as *a collection of coherent data that is logically related with some associated meaning*. A chaotic collection of data cannot be called a database. A database is designed, constructed and populated with data to meet a specific purpose and public. Furthermore, a database represents some aspect of the real world, sometimes referred to as **miniworld**. Changes in the miniworld are reflected in the database.

In other words, a database has a source from which all the data originate, some degree of interaction with events in the real world and a public that has genuine interest in the contents of the database. The DBMS is the software that incorporates functions that define, recover and change the data within a database.

- Designing a database involves specifying the types of data to be stored in the database;
- Setting up a database is the process by which the data is stored in some means of storage controlled by the DBMS;
- Manipulating a database means using functions such as queries to recover specific data, making changes to the database to reflect changes in the miniworld (insertions, updates and removals), and creating reports.

The first commercial Database Management System was released in the late 1960's and was based on primitive archiving systems available at the time. These DBMS's did not control the competing accesses of different users or processes. The DBMS's evolved from these archiving systems with on-disk storage, creating new data structures with the aim of storing information. Sometime later, the DBMS's began to use different types of representation, or **data models**, to describe the structure of the data found in their databases. This description of databases according to a data model is called **meta data**. The group of application programs which use a DBMS is called a **Database System**. The figure below presents a general diagram of a Database System and how it interacts with users.


Users and programmers
Software To Process Queries and Programs
Software To Access Stored Data
DBMS
Meta-Data
Database
Database System

1.1 Describing and storing data in DBMS's

For a DBMS to be able to store and recover information from a database system, this information must have a structure. The description of this structure, or **scheme**, is maintained by the DBMS in a database referred to as Meta Data.

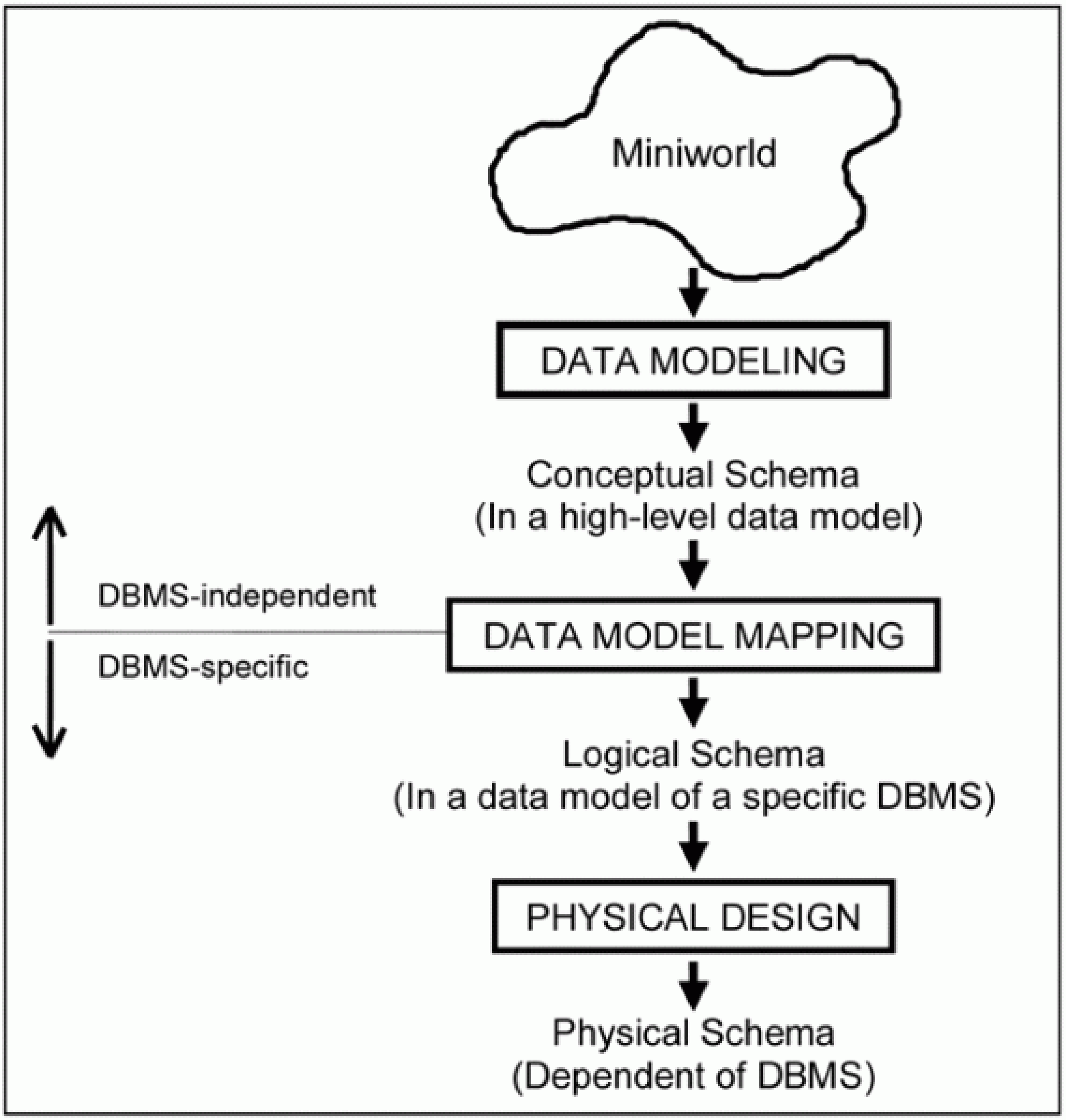
Applications in the real world are usually rich in concepts and there are various associations among these concepts. In a university's academic system, for example, there will likely be concepts such as: Courses, Outlines, Classes and Students; and associations between these concepts: Courses have Outlines, Outlines are made up of Lectures and Students enroll in Courses. These concepts and associations, which constitute the **application scheme**, should be captured from the miniworld (the university) and properly stored in a DBMS.

Unfortunately, the database technology that is currently available does not allow high-level representations of an application scheme, such as the one in the example above, to be directly represented in the DBMS's, because these can only be understood by human beings. A DBMS requires a high-level scheme (or conceptual scheme) to be converted into a representation that can be manipulated by the computer (the logical scheme).

The application scheme used by the DBMS is obtained through a three-step approach, known as a Database Design:

a. Data Modeling: the process used to conceive the Conceptual Scheme;
b. Mapping the Data Model: the process by which the Conceptual Scheme is transformed into the Logical Scheme;
c. Physical Design: the process of building the Physical Scheme (the application scheme that is manipulated by the DBMS) from the Logical Scheme.

These steps are illustrated below:



The first step is executed by database designers, which make use of a group of concepts and rules in order to guide them along the task of structuring the information from the miniworld. This group of Concepts and Rules is the **Conceptual Scheme.**

The second step, which is also performed by the database designer, transforms the Conceptual Scheme into the Logical Scheme according to the **Logical Data Model** employed by a specific DBMS (such as the Relational Model, which will be discussed in section 3).

The third step involves adding aspects of implementation to the Physical Scheme (indexing and file structure, transactions and concurrency control, optimization, recovery in case of failure, protection mechanisms, partitioning and data grouping) in order to obtain the Physical Scheme. This chapter will not cover the Phisical Schemeas it is the subject of advanced database courses.

1.2 The need for the Data Modeling Approach

In order to illustrate the concepts regarding database design, let us use a hypothetical application of a clinical database. The clinical database should store, for example, patient information, samples and DNA sequences. Let us suppose that the designers initially came up with the following description of the miniworld:

- The following patient information should be controlled: identification code1, gender, date of birth, city of birth, country;
- From the samples, the most important information is: sample code, country of origin of the sample, body part from which it was taken, number of sequences generated by a given sample; the sequence data is represented by the sequence's identification, the region of the genome, the length of the sequence and its respective DNA sequence;
- A patient may have more than one sample associated to him/her. On the other hand, a sample is associated to only one patient.
- A sample may have many sequences associated to it; however, a sequence is associated exclusively to one sample.

In the next sections, we will present the database concepts with which database designers must be familiar in order to create an application's conceptual model.

A first, straightforward way of representing our problem is shown in the Table below, which depicts all data belonging to patients, samples and sequences:

Code	Gender	Birthday	Country	Code Sample	Body Organ	Collection Date	Code Sequence	Genomic Region	DNA Sequence
0001	M	01/01/1970	Brazil	001	Liver	10/10/2005	1111	ENV	'atgcctgacttgctacccttggaaatcga
0001	M	01/01/1970	Brazil	001	Liver	10/10/2005	1112	POT	atgcctgacttgctacccttggaaacttaaa'
0001	M	01/01/1970	Brazil	001	Liver	10/10/2005	1113	null	aaatcga...tactttactttaccataacttaaa'
0001	M	01/01/1970	Brazil	002	spleen	05/05/2005	2222	null	accataacttaaa accataacttaaa
0001	M	01/01/1970	Brazil	002	spleen	05/05/2005	2223	POT	tacttactttactttttactttaccataacttaaa
0001	M	01/01/1970	Brazil	002	spleen	05/05/2005	2223	POT	tacttactttactttttactttaccataacttaaa
2040	F	02/02/1980	Angola	005	lung	02/02/2006	3333	ENV	'atgcctgacta...tactttactttaccataacttaaa
2040	F	02/02/1980	Angola	005	lung	02/02/2006	3334	Null	aaatcga...tactttactttaccataacttaaa'

As we can see in the table, one patient has one or more samples. Each sample has one or more sequences. In this table we have data replications for patients and samples for each sequence. For example, the patient 0001 has six

1 In clinical databases, patient ID is normally done indirectly. Hence, instead of giving explicit reference to the patient by name and last name, a code is used to indirectly identify the patient.

replicates and two samples. His samples 001 and 002 are replicated three times, for each sequence data. The patient 2040 has three replicates for one sample that has one sequence.

This table is a non-optimized data structured, which present many disadvantages related mostly to the redundancy in the storage of data (partially repeated rows). In particular we can mention:

- larger disk storage
- consistency control – repeated information has to be the same, a potentially complex task
- difficulty in the updating process, in particular when updates must be propagated to different rows (for example, correcting a mistake in the patient's gender)
- clarity on query results: for example, a query of which patients have samples collected from the liver will return patient 001 three times.

The next table summarizes the main characteristics of a system that is developed using a DBMS as compared to a system that is developed using the traditional archiving system.

Traditional File Processing	DBMS	Advantage of DBMS
Data definitions is part of applications programs code	Meta-Data	Redundancy control
Application-data Interdependence	Application-data Independence	Redundancy control
		Maintainability
Data representations in physical level	Conceptual representation by data and programs	Maintainability
Specifics modules implements each vision	Multiple vision	Query facilities

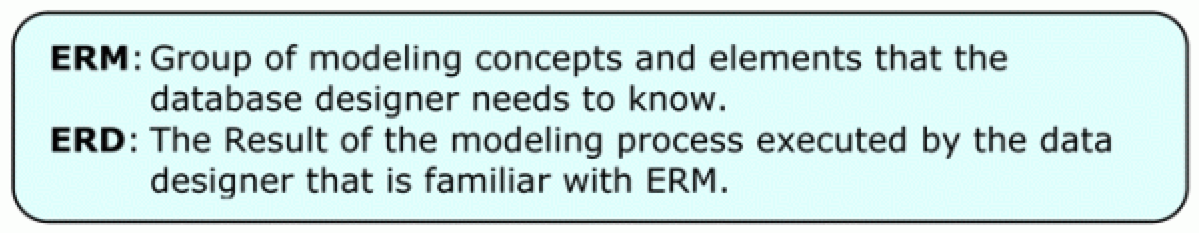
Although the advantages of using DBMS's are attractive, there are two situations in which they are not recommended:

- When the data and applications are simple and stable – for example, in the case of an address book;
- When an extreme access speed is required. In this case, the applications should store the data directly in the RAM (*Random Access Memory*).

Section 2 will show the correct way to partition this table through the Entity-Relationship data modeling approach and a more complete case study of patients, samples and sequences.

2. Data Modeling Using the Entity-Relationship Model

The **Entity-Relationship Model** (ERM) is a data model designed for the purpose of representing the semantics associated with data from the miniworld. The ERM is used in the **conceptual design** stage, where the **conceptual scheme** of the application's database is conceived. Its concepts are intuitive, which ensures that the database designers capture the concepts associated with the application data without the interference of specific database implementation technology. The conceptual scheme that is created based on the ERM is called **Entity-Relationship Diagram** (ERD). The basic concepts of ERM are essential to understand how a database can be effectively queried by users.



ERM: Group of modeling concepts and elements that the database designer needs to know.
ERD: The Result of the modeling process executed by the data designer that is familiar with ERM.

2.1 Entities, Attributes and Groups of Entities

Entity

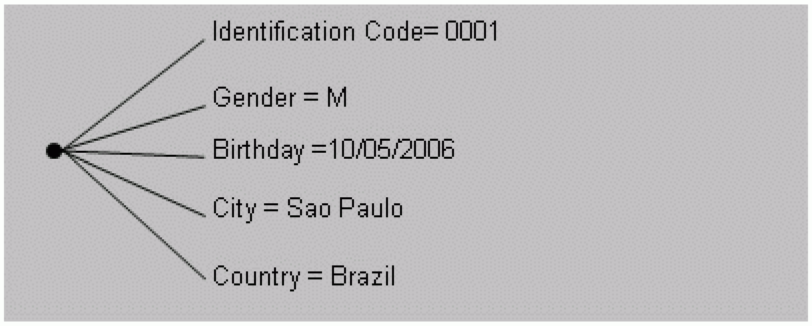
The most basic object represented by the ERM is the **entity**. An entity is something in the real world that exists independently.

For example individual patients, samples and sequences in a clinical database are Entities.

Attribute

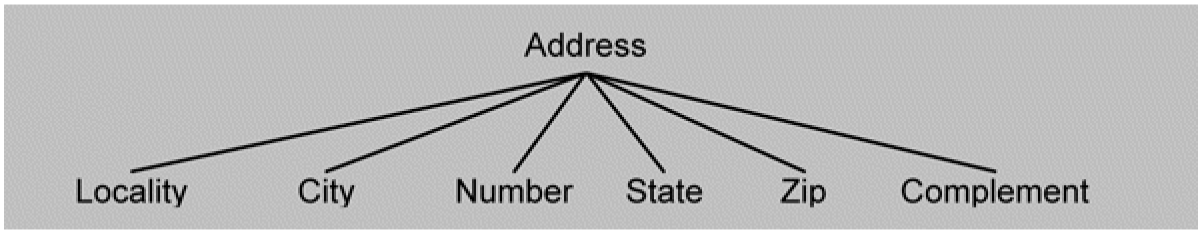
Each Entity contains specific information known as **attributes**.

For example, a patient from the clinical database can be described by his attributes: identification code, gender, data of birth, city of birth and country. Each attribute has a value, (attribute value). For example attribute gender can have the values "male" or "female". Below, we can see Patient entity (e1) with its attributes.



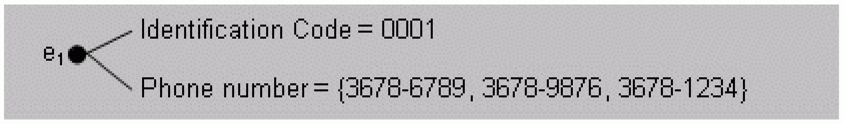
Composite Attributes

Some attributes may be composed of many parts, or sub-attributes. For example, the attribute *Address,* depicted next, is composed of other attributes, such as city, number, locality, state, zip and complement. Such attributes are known as **Composite Attributes**.



Multi-valued Attributes

Many attributes have only one value (**uni-valued**). However, there are some attributes that may be composed by a group of values (**multi-valued**). For example, as you can see next, a patient (e_1) has more than one telephone number, so *Telephone Numbers* is a multi-valued attribute.


e1
Identification Code = 0001
Phone number = {3678-6789, 3678-9876, 3678-1234}

Derived Attributes

Some information about an entity can be obtained by processing the attributes. For example, we can obtain a patient's age from its Date of Birth. This information is called derived attributes. Thus, age is a derived attribute where:

- age = Current Date – Date of Birth.

Null Value

Sometimes an attribute of a specific record may have an undefined value. In these cases, a **null value** is given to this attribute. For example, not all patients have addresses with an apartment number. In that case, the attribute apartment number will have the value *null*. Another case where null value can be applied is when the value is unknown. For example, a patient may not know exactly when he/she was born. Hence, a null value is used.

Null Value is used whenever the value of an attribute **does not apply** or when its value is **unknown.**
Attention! Null Value is not the same as zero (0).

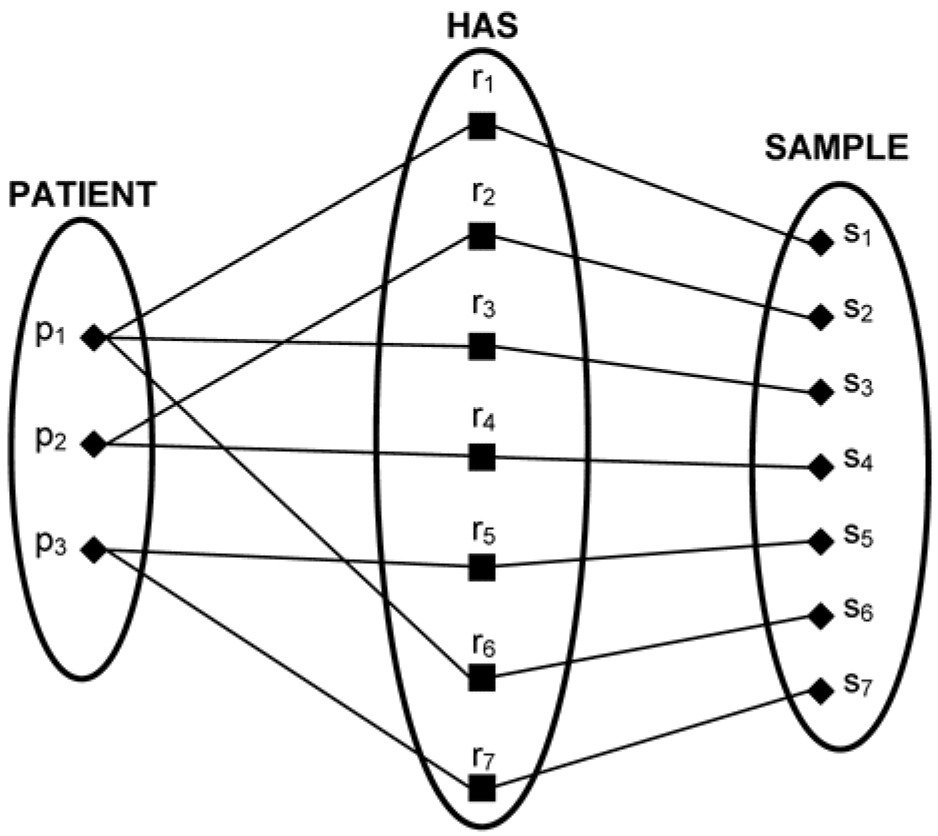
Key-Attribute

Entities must have an attribute whose value that make them unique in the database. We call these attributes key-attributes. All Entity Types must have at least one key-attribute; it doesn't matter if this attribute is simple or composite.

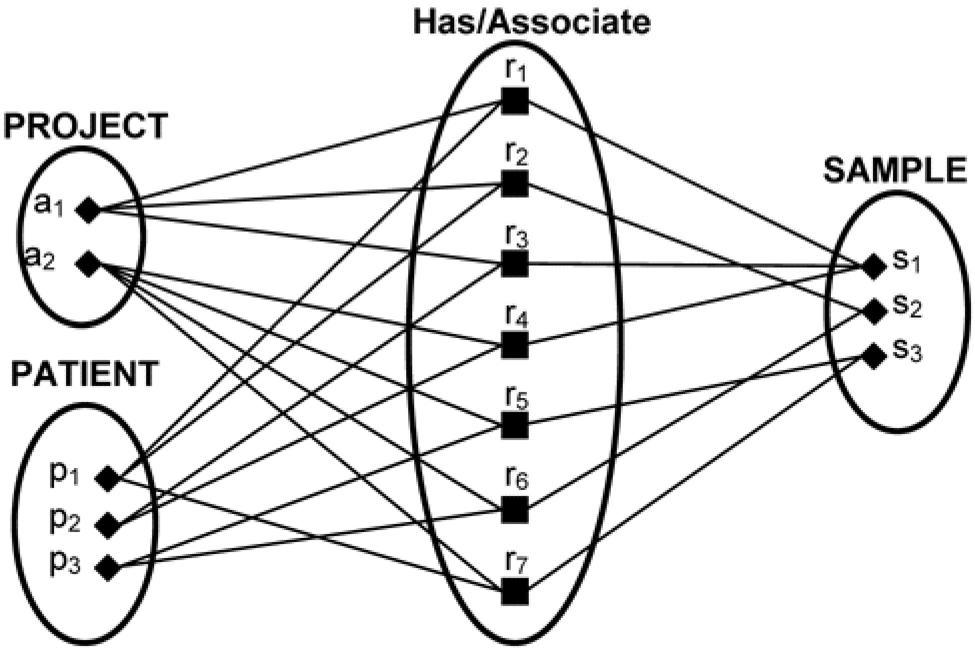
The values of a key-attribute must be unique. For example, one key-attribute of entity type Patient would be the identification code.

2.2 Relationships and Groups of Relationships

What makes the Entity-relationship model so flexible and powerful is the fact that we can establish relations between entities. For example, we can associate samples to patients. To see how this can be expressed in our model, consider the figure below:



Above, you can observe that entity p_1, from the entity type "Patient", is related to entity s_1, from the entity type "Sample", by way of the relationship **r_1,**. A *relationship type* is just a group of similar relationships. In our example, we have the relationship "Has" that associates sequences to patients. Relationships can be established among more than just two entities. For example, Patients may have certain Samples associated with certain Projects. The next figure shows a ternary relationship type (we say relationship type of level 3) that relates patients, samples and projects: Has/Associate.



Be Careful! A ternary relationship type cannot be substituted by three binary relationship types.

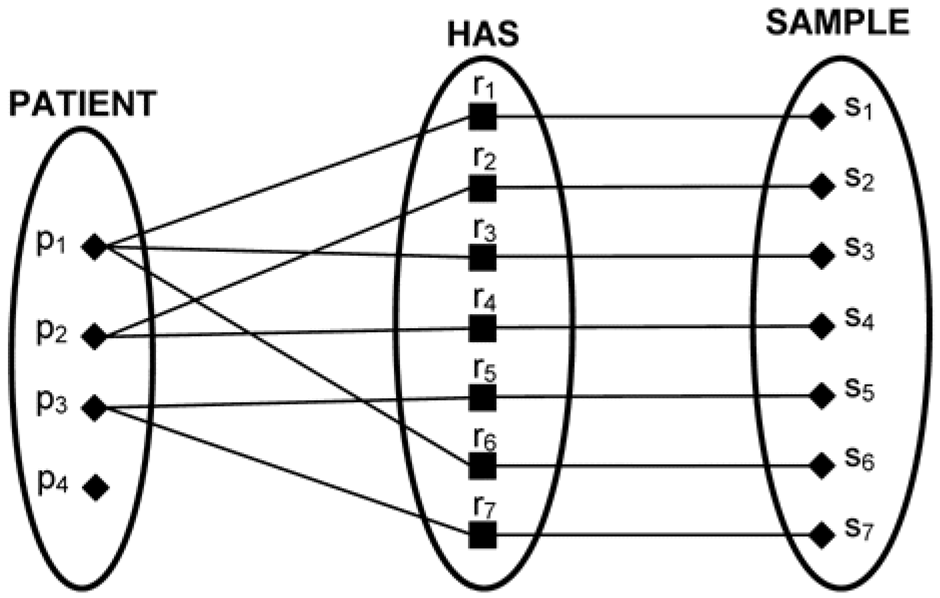
There can be relationship types of any level, but level 2 relationship types are more common.

Cardinality Ratio

The cardinality ratio specifies the number of relationship instances in which an entity can participate. The binary relationship type Patient HAS Sample, from Figure 2.6, has a cardinality ratio of 1:N (read: *one-to-"N"*). This means that for every Patient entity, we can associate any number of Sample entities. However, a Sample entity can only be related to a single Patient. The most common cardinality ratios for binary relationship types are 1:1 (one-to-one), 1:N (one-to-"N") and M:N ("M"-to-"N").

Constraints on Relationship Types

The cardinality ratio of a relationship establishes two types of constraints on these relationships: *participation constraint* and *structural constraint*. A nice feature of DBMS software is that they automatically enforce these constraints, that is, entities and relationships cannot be entered into the database if the constraints are not satisfied. The next figure illustrated a relationship with cardinality 1:N:



Constraint on Participation

The constraint on participation specifies whether the existence of an entity defines its participation on a relationship. There are two types of constraints on participation: *total* and *partial*. For example, if the clinical database establishes a rule that every registered Sample must belong to a Patient, then the entity Sample can only exist if it participates in a relationship instance HAS. The participation of Sample in HAS is called total, which means that every Sample entity must be related to a Patient entity via the relationship type HAS. The total constraint on participation is sometimes referred to as existential dependence. Also, it is unreasonable to impose that every Patient be obliged to have a Sample when he/she is registered on the clinical database. Hence, the patient's participation in the HAS relationship is partial. This means that some entities in the group of entities Patient may be related to an entity Sample via HAS, but not necessarily all of them.

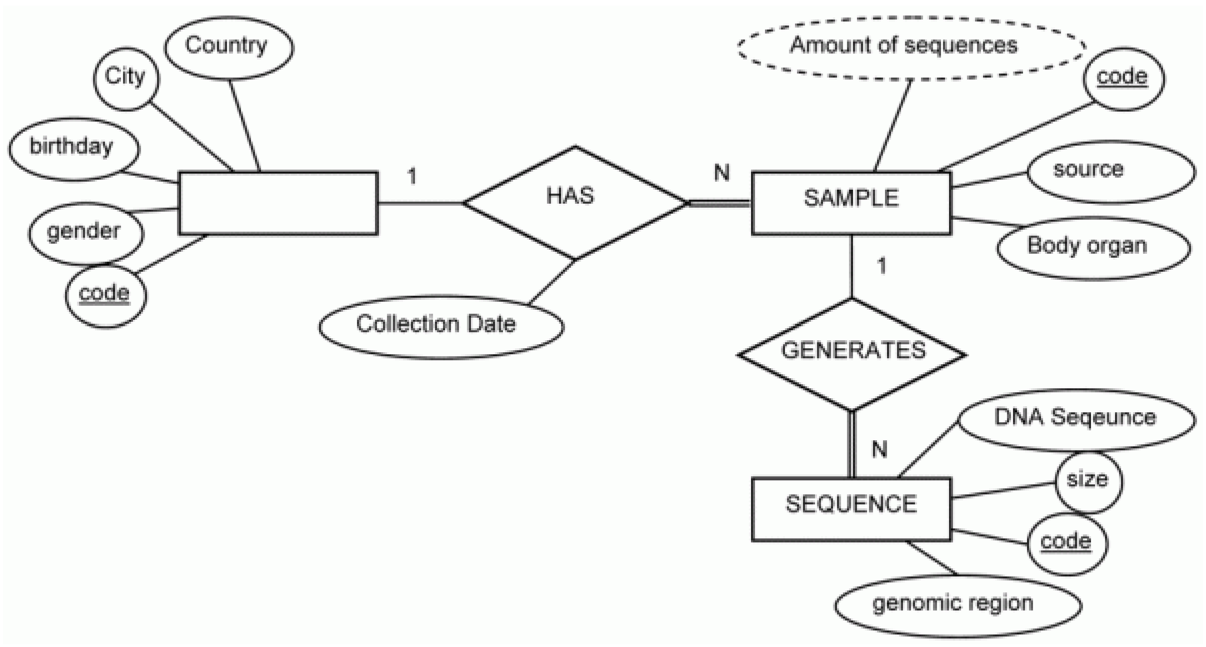
The **Cardinality Ratio** and the **Constraint on Participation** depend on Rules from the miniworld.

Structural Constraint

The **Structural Constraint** specifies the **minimum** and **maximum** number of relationships on which an entity should participate. For example, in the relationship type "Patient HAS Sample", a patient may not have a sample; therefore, minimum = 0. However, a Patient can have as many samples as necessary depending on the project. We say that by stating that maximum = N. Hence, the structural constraint of the Patient in relation to the relationship HAS is (0, N). On the other hand, a Sample can only exist if it belongs to one single Patient; hence, minimum = 1 and maximum = 1. Therefore, the structural constraint of Sample in relation to the relationship type HAS is (1,1).

Relationship Attribute

Relationship Types can also have Attributes. For example, the date when the sample was collected (Collection Date) from a given patient should be represented in the relationship type HAS. Also sequences are associated with blood samples in the relationship GENERATES, as illustrated next:.



In this case, the relationship GENERATES does not have attributes.

2.3 Conceptual Design of databases with Entity-Relationship Models

The previous section introduced some of the main concepts involved in the Entity-Relationship Model. The creation and specification of the Entity-Relationship model is called the *Conceptual Design.* During this process we identify entity types, their attributes, the key-attributes, the relationships their cardinalities and constraints.

In our example we have three entity types (key attributes are underlined):

- **PATIENT** -identification code, gender, birthday, city, country.
- **SAMPLE** -sample code, country of origin, body part, number of sequences.
- **SEQUENCE** - sequence code, region of the genome, size, DNA sequence.

The relationship types were also identified, together with the cardinality ratios, constraint on participation and attributes:

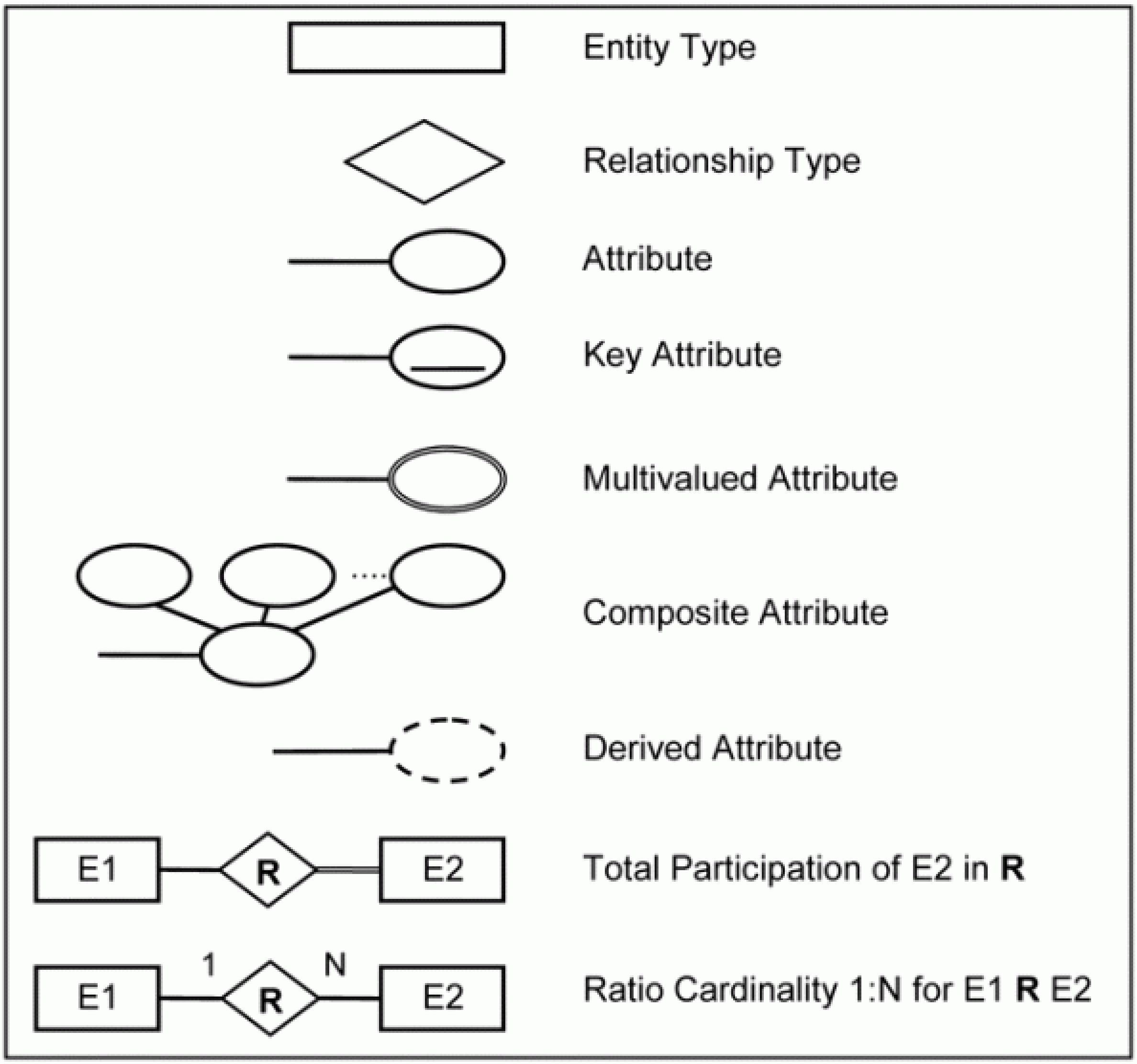
- PATIENT **HAS** SAMPLE:
    - Cardinality Ratio (**1:N**), as:
        - 1 Patient can have **N** Samples.
        - 1 Sample belongs to **1** Patient.
    - Constraint on Participation:
        - For the Patient: **Partial**, as the patient is not obliged to have a sample in order to be registered in the database.
        - For the Sample: **Total**, as a sample necessarily must be associated to a Patient in order to be registered in the database.
    - Attributes: collection date of the sample.
- SAMPLE **GENERATES** SEQUENCE
    - Cardinality Ratio (**1:N**), as:
        - 1 Sample can have **N** Sequences.
        - 1 Sequence belongs to **1** Sample.
    - Constraint on Participation:
        - For the Sample: **Partial**, as a Sample can be registered although there may be no Sequence to associate it with.
        - For the Sequence: **Total**, as there must be an existing Sample in the database from which the sequence have been generated.
    - Attributes: none.

Entity-Relationship Diagram (ERD)

The Entity-Relationship Diagram (ERD) represents the Entity-Relationship model of a database. It is one of the most common forms of documenting a database's structure. The ability to read ERDs can help biologists understand the information and relations that are stored in any database and is also the key to understanding which are the querying possibilities of that database. The previous figure illustrates an ERD for the clinical database scheme. The entity types Patient, Sample and Sequence are depicted inside rectangles. Relationship types REALIZE, GENERATE are shown in diamonds linked to the participating entity types. Attributes are shown in ovals connected to the entity types and relationship types. The key-attributes are underlined. Derived attributes are depicted in ovals with dotted lines.

Also, the figure shows the cardinality ratios for each relationship type. The cardinality ratio of Patient **HAS** Sample is 1:N and of Sample **GENERATES** Sequence is 1:N. The partial constraints on participation are specified by simple lines. The parallel lines denote total participation (existential dependency).

Finally, this section ends with the next figure, which shows a notational graph summary of Entity-Relationship Models. Further details can be found in Elmasri and Navathe 2004.



In this section, the basic concepts and notations of the ERM were introduced taking into account a specific application, clinical databases, for an easier illustration of the concepts and also to show that an application's conceptual model depends on the rules and constraints of the miniworld, as established by specialists in each respective area.

When designing a database, the next phase is the logical design of the database, which makes use of the application's conceptual model in order to build the application's logical model. The logical data model selected in this case is the Relational Model, which will be explained in section 4. Next, we will explain how to query a database based on an ER diagram, since this is the ultimate goal of a database, and assuming that most readers will not be involved in designing databases, but rather in exploring the information stored in them.

3. Querying in the Relational data Model

Using the ER diagram presented above a physical database was created to illustrate the use of queries. Details of this database creation is shown in Section 4. In the current section we will explore the benefits and flexibility of queries on a database that was carefully designed. We will use an actual database populated according to the values depicted in the three tables below:

PATIENT.

Code	Gender	Birthday	City	Country
0001	M	1980-01-21	São Paulo	Brazil
1000	F	1990-08-09	Manaus	Brazil
9876	M	1975-04-30	Cuiaba	Brazil
0101	F	1960-02-02	São Paulo	Brazil

SAMPLE.

Code	Source	BodyOrgan	DateOfCollect	PatientFk
0001	Brazil	Null	2006-01-01	0001
0011	Brazil	Null	2006-01-02	1000
0015	Angola	Null	2006-01-03	1000
1010	South Africa	Liver	2006-01-04	0001
1050	Mocambique	spleen	2006-01-05	9876
2069	Brazil	head	2006-01-06	0101
9876	USA	lung	2006-01-07	9876

SEQUENCE.

Code	GenomicRegion	Size	DNASequence	SampleFk
1111	Null	535	'AGCCACCAGCGCAACATGA...CTCAGCTGAAGAAG'	0011
2222	ENV	1035	'TGCGTTACTTTAAATTGTA...GAGACCTTCAGACCA'	0015
3333	POT	1018	'GGCGAACGGGTGAGTAAC...TTATCGCTAGTTACC'	1010
4444	ENV	1044	'TGCGTTACTTTAAATTGTA...AGACCTTCAGACCA'	0015
5555	Null	1193	'TACTTCAACTACGAGAACC...ATAAAAATTACTTTG'	0015

We will begin by studying the simplest type of query. In this type of query, all you have to verify is whether the tables were filled out correctly in the relational database. The SQL language has a basic command to recover information in a relational database: the SELECT command. In general, the results of registration queries in SQL are tables.

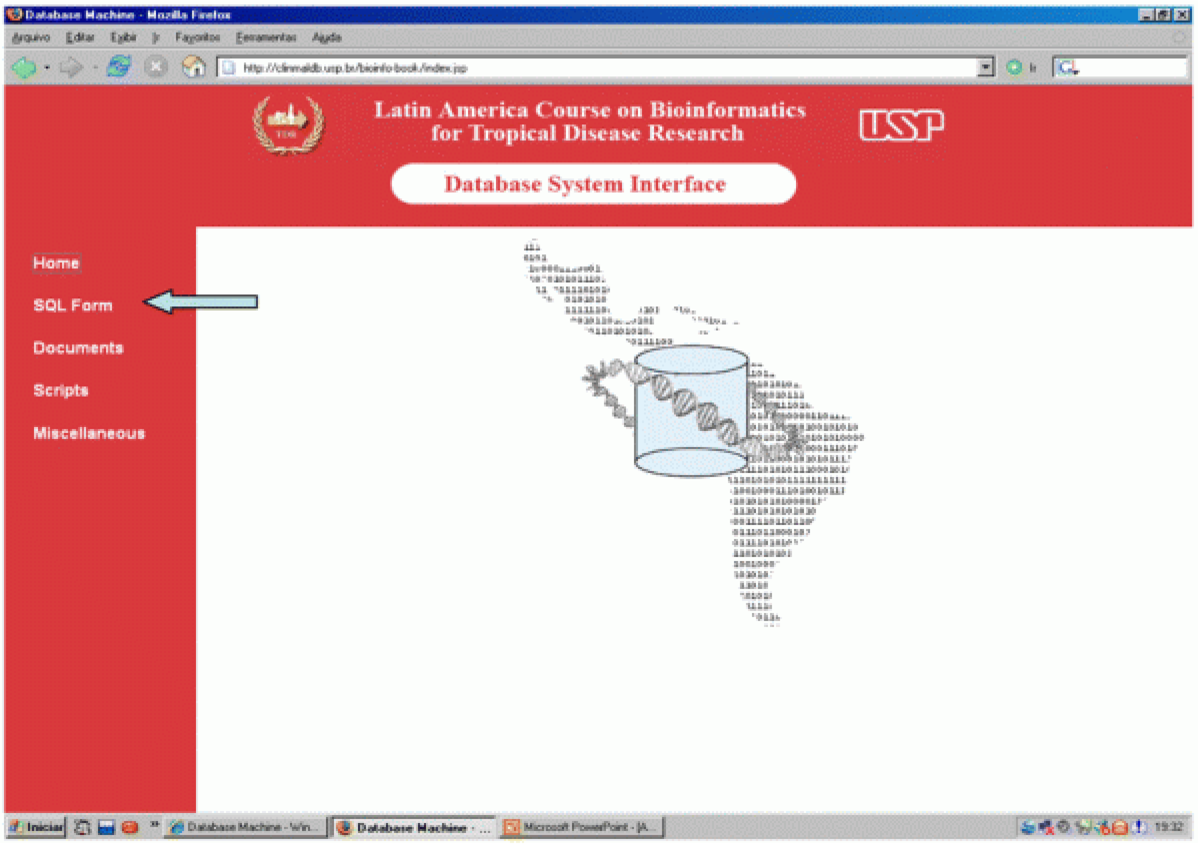
The basic type of the SELECT command is made up of three clauses: SELECT, FROM and WHERE:

SELECT <list of attributes>

FROM <list of tables>

WHERE <condition>

The condition is a expression that filters the data that will be retrieved by the query. To illustrate the use of the SELECT command, we will make use of the database of clinical data again, assuming that it has been populated using the database's Internet page (available at http://clinmaldb.usp.br/bioinfo-book):


Database Machine - Mozilla Firefox
Latin America Course on Bioinformatics for Tropical Disease Research
USP
Database System Interface
Home
SQL Form
Documents
Scripts
Miscellaneous

You should access the link "SQL Form", which will give you the query page, where you can run all queries described below through copy and paste mechanisms for each SQL expression and execute them using the "Execute" button. The first query is a command to show the existing entries (see section 4) in the relation PATIENT. It is described using the SQL expression2:

Query 1: **SELECT** * **FROM** PATIENT

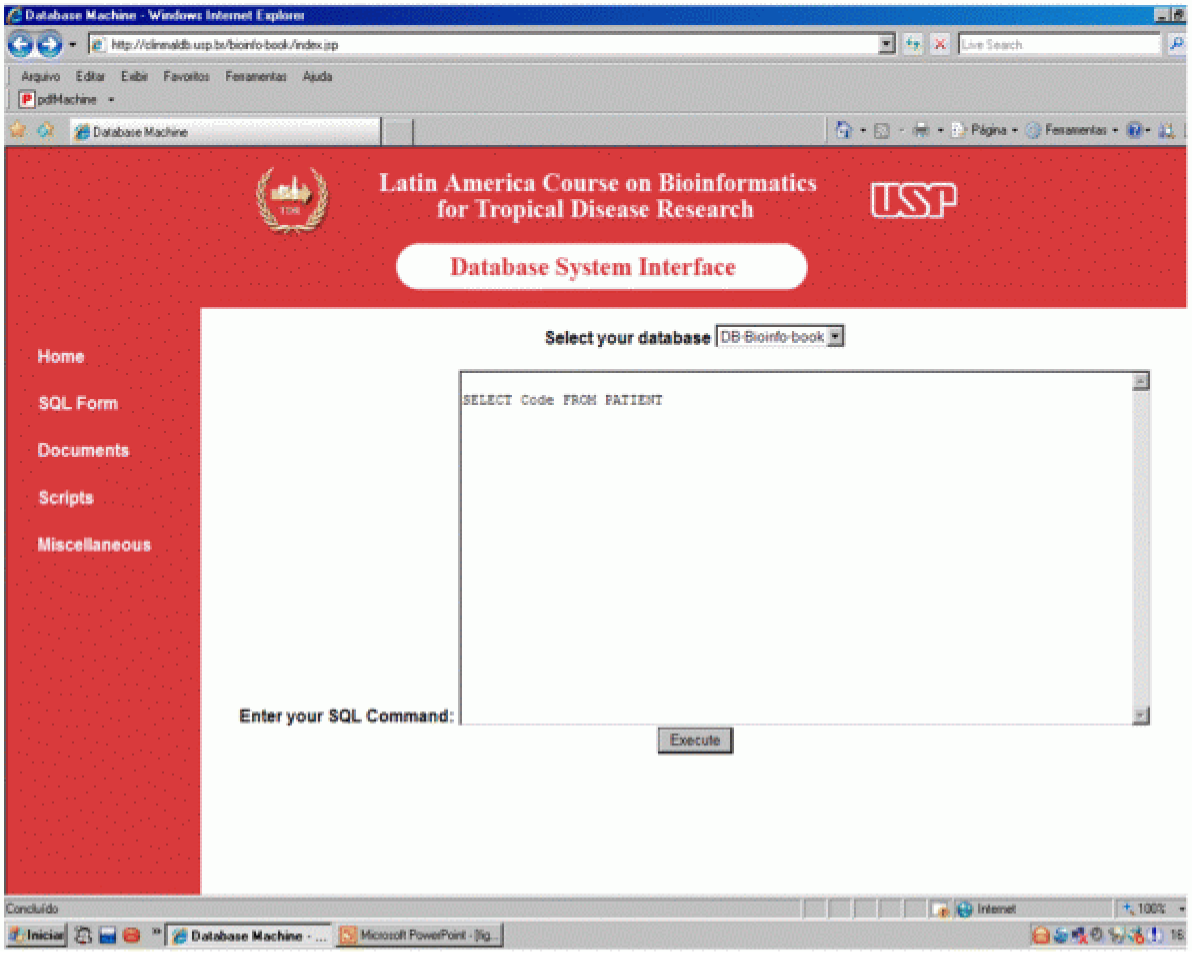
Executing this command in SQL Form Internet environment, it should return all the values of the relation PATIENT which were above. In case you are interested in getting only the values of the attribute PatientCode in the relation PATIENT, click the "New query" link at the bottom of the results and issue the query:

Query 2:

SELECT Code FROM PATIENT

To execute this command, type the text above in the SQL form page of the database:.

2 The asterisk indicates that the SELECT command should return the values of all the attributes of the relation PATIENT.

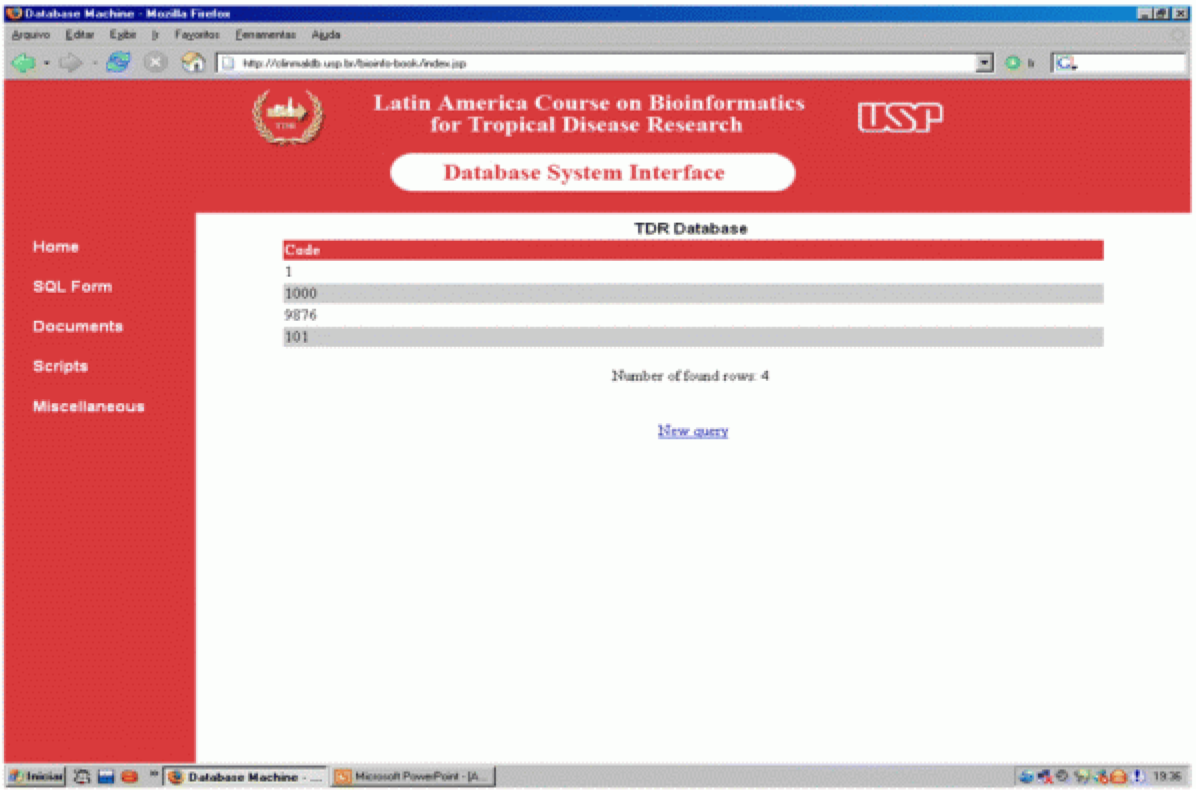

Latin America Course on Bioinformatics
for Tropical Disease Research
Database System Interface
Home
SQL Form
Documents
Scripts
Miscellaneous
Execute

Select your database DB-Bioinfo-book

Enter your SQL Command:

```
SELECT Code FROM PATIENT
```

After typing the text, click on the "Execute" button and you should get the response as shown below:


Latin America Course on Bioinformatics
for Tropical Disease Research
Database System Interface
Home
SQL Form
Documents
Scripts
Miscellaneous
New query

TDR Database

Code
1
1000
9876
101

Number of found rows: 4

In the examples given previously, we did not employ the clause WHERE. This clause is used to filter the information in a query. The conditions specified in a WHERE clause can be a disjunction or a conjunction of comparisons of attribute values. For example: imagine you are interested in obtaining the list of code numbers for patients of the female gender. This query would be formulated as:

Query 3:

SELECT Code FROM PATIENT WHERE Gender = 'M'

We can have more complex conditions after the WHERE clause. By using a conjunction (AND) of comparisons we can issue the following queries:

Query 4: Find from all patients, those that are female and whose birthday is before 1980-01-01.

SELECT Code

FROM PATIENT

WHERE Gender = 'M' and Birthday < '1980-01-01'

Query 5: Find all DNA sequence with GenomicRegion = Pol and size above 200 bp.

SELECT DNASequence

FROM SEQUENCE

WHERE Size > 200 and GenomicRegion = 'POL'

You can use the internet environment described above to issue each of these queries and compare the results with the previous one and with the database contents.

Queries do not need to be restricted to just one table. The JOIN operation is used for combining information of two or more tables. A "Join" is built using a special clause condition in WHERE. If we want to join the information belonging to PATIENT and SAMPLE tables, we need to define a join condition. Normally this is done through key and foreign key connections. In this case the FROM clause will mention two tables and the attributes will refer to which table they belong:

Query 6: Find the country of patient whose sample code is 15.

SELECT PATIENT.Country

FROM PATIENT, SAMPLE

WHERE SAMPLE.Code = 15 and

PATIENT.Code = SAMPLE.PatientFK

Another example, this time using two joins conditions. One for PATIENT and SAMPLE, and other for SAMPLE and SEQUENCE:

Query 7: Find the Gender of Patient, Source of Sample and GenomicRegion of Sequence whose size is longer than 800 bp.

SELECT Gender, Country, GenomicRegion

```
FROM PATIENT, SAMPLE, SEQUENCE
WHERE SEQUENCE.Size > 800
and
PATIENT.Code = SAMPLE.PatientFK
and
SAMPLE.Code = SEQUENCE.SampleFK
```

Finally we will show how to obtain the results in a user-specified order. The order of the results can either be numerical or alphabetical. In SQL, this is possible using the operator ORDER BY.

Query 8: Find the city and birthday of Patients that have samples collected in 2006-01-03, and show it ordered by Pacient's birthday.

```
SELECT PATIENT.City, PATIENT.Birthday
FROM PATIENT, SAMPLE
WHERE SAMPLE.DateOfCollect = '2006-01-03' and
PATIENT.Code = SAMPLE.PatientFK
ORDER BY PATIENT.Birthday
```

For a better understanding of the SQL language, re-execute these basic queries and try other queries using PATIENT, SAMPLE and SEQUENCE in the SQL Form environment.

4. Relational Data Model

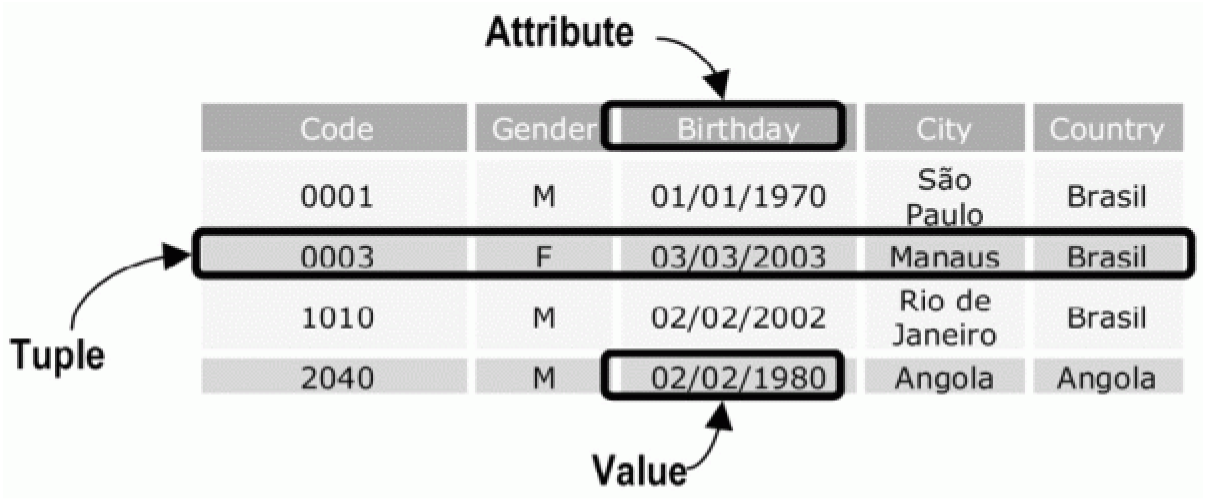
As we have seen above, SQL queries can be a powerful instrument for accessing the information contained in a database. SQL queries can only be used if we implement a Relational Database. To do this, we need to build the *Relational Data Model*. **Relational Data Model** is a logical data model used to develop logical database designs3.

The Relational Model represents data from a database as *relations*. The word *relation* is used to mean "a list or group of information" and not in the sense of "association or relationship".

Every relation can be understood as a table or as a simple registration file. However, keep in mind the following observation: during the logical design phase of the database, we recommend that you employ the word *relation* instead of *table*, since the word *table* is normally employed in the context of a specific Relational DBMS in the physical design phase.

An example illustrated in the next figure shows an extension of the relation **PATIENT**, with its **attributes** and **attribute values**.

3 The Database Management Systems (DBMS) that use the Relational Model are called Relational DBMS's.



Code	Gender	Birthday	City	Country
0001	M	01/01/1970	São Paulo	Brasil
0003	F	03/03/2003	Manaus	Brasil
1010	M	02/02/2002	Rio de Janeiro	Brasil
2040	M	02/02/1980	Angola	Angola

Each line of the relation is called **tuple** and each column is called an **attribute**. The **type of data** of the **values** that an **attribute** can assume is termed the **domain** of the attribute.

- It's important to emphasize that the Relational Model always takes into account that the attributes' values are indivisible, that is, they're **atomic**.

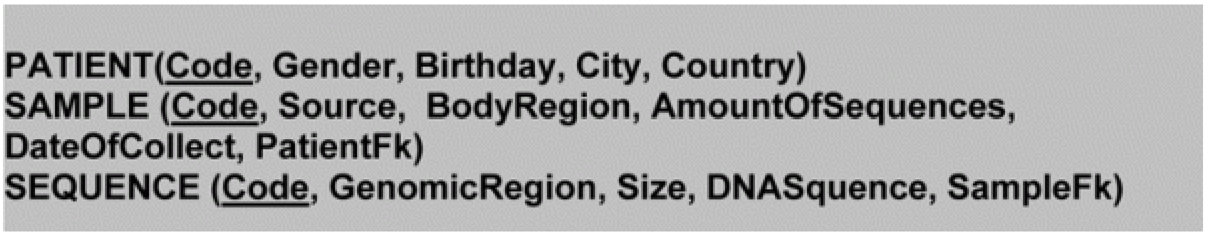
The Specification of a database using SQL

The Relational Data Model is also specified using the Data Definition Language (DDL), part of the SQL standard. In addition, as we will see below, we can use the SQL language to specify a series of integrity constraints on the database that will be maintained automatically by the DBMS.

Unfortunately, depending on the DBMS, the SQL language provided can be a mixture of ANSI SQL89, SQL92 and SQL99 standards. Furthermore, some developers of DBMS'S provide extensions that do not use the SQL language standard (such as an AUTOINCREMENT clause that does not exist in any standard ANSI SQL), which may make the portability of applications that use these extensions more difficult. For this reason, it is important for the designer to know how to evaluate the costs and the benefits of using non-standard SQL constructions in any given application in a database.

Before we start, we should talk about the concepts of *primary key* and *foreign key*. The primary key of a relational table uniquely identifies each record in the table. It can be a normal attribute that is guaranteed to be unique (such as Social Security Number in a table with no more than one record per person), A foreign key is an attribute that points to the primary key of another table. The purpose of the foreign key is to ensure referential integrity of the data. In other words, only values that are supposed to appear in the database are permitted.

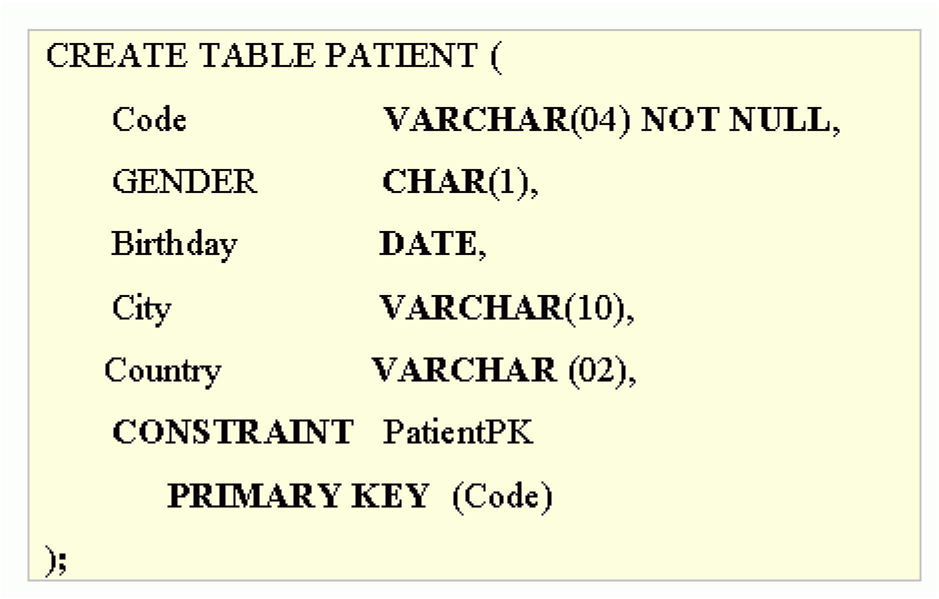
Let us consider the logical scheme of the relational database of clinical data, presented in the next box. For example: the Patient Code (marked with FK) of the relation SAMPLE is the foreign key that corresponds to the primary key Code in the relation PATIENT.



PATIENT(Code, Gender, Birthday, City, Country)
SAMPLE (Code, Source, BodyRegion, AmountOfSequences, DateOfCollect, PatientFk)
SEQUENCE (Code, GenomicRegion, Size, DNASquence, SampleFk)

Integrity constraints can be defined at the moment the relations in the relational database are created. For example: below, we can see the definition of the integrity constraints during the creation of the relation PATIENT by using the command: CREATE TABLE in the SQL language.

The command indicates the creation of the table PATIENT instead of the relation PATIENT. The SQL language is used in the database's physical design with the purpose of materializing relations in physical tables.



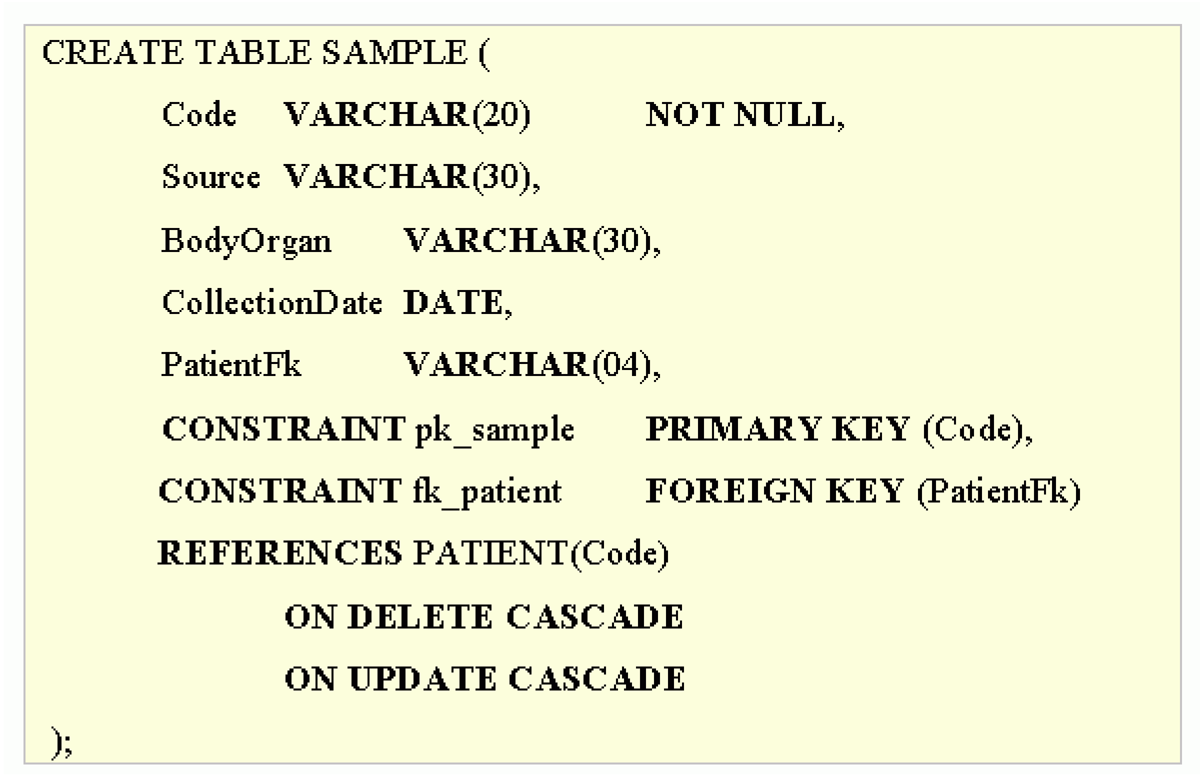
```
CREATE TABLE PATIENT (
      Code              VARCHAR(04) NOT NULL,
      GENDER            CHAR(1),
      Birthday          DATE,
      City              VARCHAR(10),
     Country            VARCHAR (02),
      CONSTRAINT   PatientPK
           PRIMARY KEY  (Code)
);
```

The command CREATE TABLE initially specifies the name of the relation and its set of attributes. Each attribute has a name, a type that specifies the domain of valid values and may have some attribute constraint such as NOT NULL.

In the SQL code above, the *entity integrity constraint* was ensured by indicating NOT NULL to the Code attribute. The *key integrity constraint* was ensured by the following clause: **CONSTRAINT** PatientPK PRIMARY KEY (Code).

The clause, called PatientPK, indicates that the attribute *Code* is a primary key; therefore, the key's integrity constraint will be ensured by the DBMS

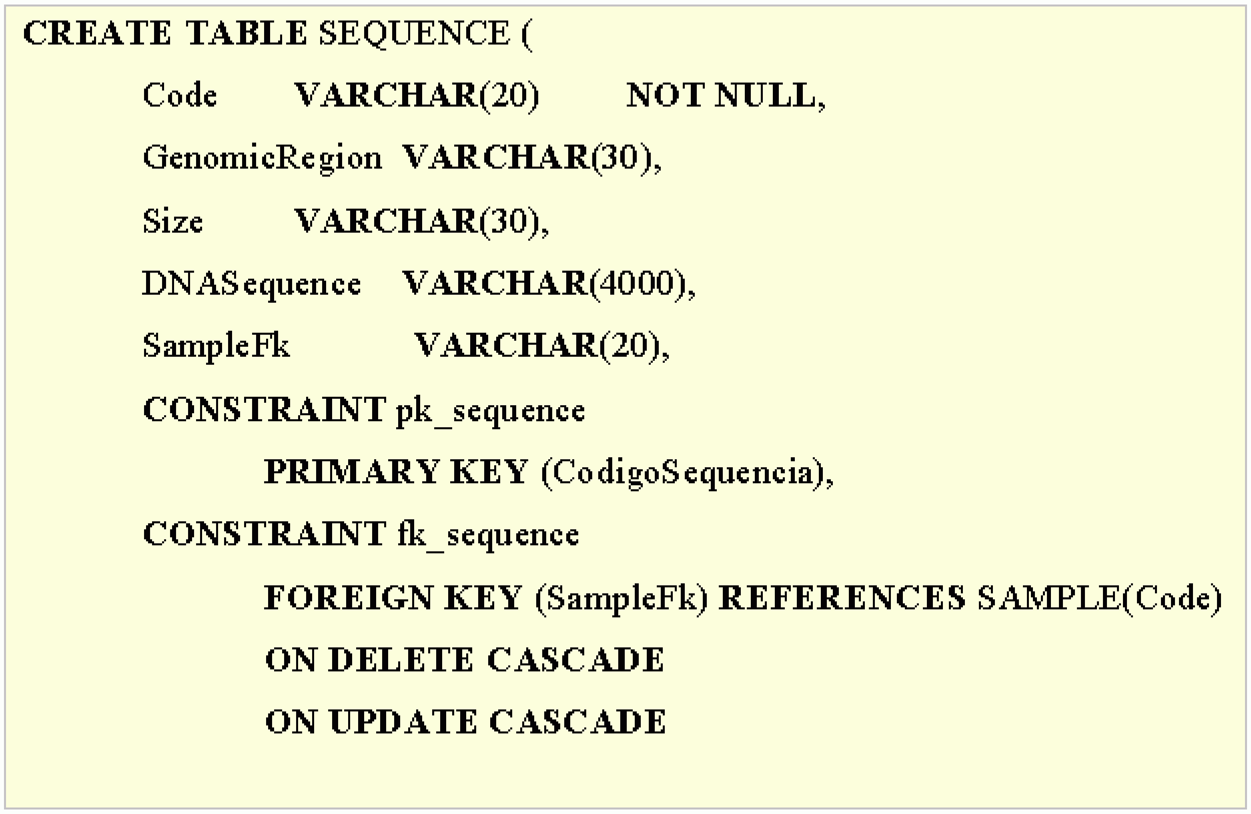
The referential integrity constraint is ensured by the clause FOREIGN KEY. For example: look at the code below, which shows the creation of the relation SAMPLE.



```
CREATE TABLE SAMPLE (
          Code    VARCHAR(20)          NOT NULL,
          Source  VARCHAR(30),
          BodyOrgan       VARCHAR(30),
          CollectionDate  DATE,
          PatientFk       VARCHAR(04),
          CONSTRAINT pk_sample      PRIMARY KEY (Code),
          CONSTRAINT fk_patient     FOREIGN KEY (PatientFk)
          REFERENCES PATIENT(Code)
                    ON DELETE CASCADE
                    ON UPDATE CASCADE
 );
```

The fk_patient constraint indicates that the attribute PatientFk is a foreign key that makes reference to the primary key PATIENT(Code) in the relation PATIENT. Moreover, the clause ON DELETE CASCADE, indicates that if a given Patient is removed from the database, the Sample tuples will be removed. The clause ON UPDATE CASCADE indicates that, when the attribute PatientCode of a Patient is changed, the attributes that made reference to the given patient will also be changes accordingly.

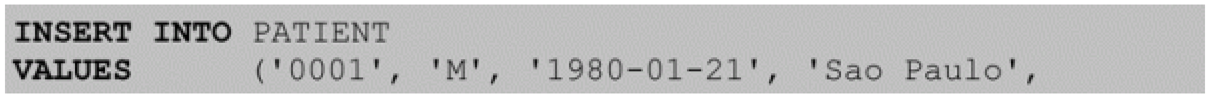
Similarly, we can define the relation for the sequence data in the following way:



```
CREATE TABLE SEQUENCE (
        Code        VARCHAR(20)      NOT NULL,
        GenomicRegion  VARCHAR(30),
        Size        VARCHAR(30),
        DNASequence   VARCHAR(4000),
        SampleFk       VARCHAR(20),
        CONSTRAINT pk_sequence
                PRIMARY KEY (CodigoSequencia),
        CONSTRAINT fk_sequence
                FOREIGN KEY (SampleFk) REFERENCES SAMPLE(Code)
                ON DELETE CASCADE
                ON UPDATE CASCADE
```

It's interesting to verify that the constraints imposed upon the relations work apropriately. In order to do this, we will present some commands in the SQL language that will allow you to make insertions and removals of tuples in a relation.

The box below shows the insertion of a new Patient by using the command INSERT. Notice that all the attribute values were filled out in the sequence in which the attributes were defined during the creation of the relation PATIENT. In this way, the first value, '0001', corresponds to the attribute PatientCode, the second value, 'M', corresponds to the attribute Gender, up until the last value, 'Brazil', which corresponds to the attribute value Country.



```
INSERT INTO PATIENT
VALUES      ('0001', 'M', '1980-01-21', 'Sao Paulo',
```

Many times, you don't have to fill out all the attribute values in a tuple. In these cases, the command INSERT allows you to specify the attributes whose values will be filled out.

There are variations in the INSERT command that can be seen in Elmasri and Navathe 2004. With these variations of the INSERT command, you can see that the DBMS guarantees the three integrity restrictions reviewed in this unity (key, entity and referential). Details regarding the creation and insert scripts of relations can be found menu option Scripts which is available on the web at http://clinmaldb.usp.br/bionfo-book.

5. Final Considerations

This Chapter was an introduction to the main aspects of the conceptual, logical and physical designs of a database. Section 1 was an introduction to the historical background of how Database Management Systems were developed. In Section 2, we discussed the ERM model for conceiving the conceptual data design. In Section 3, we showed examples of possible queries and in Section 4 we explained the **Relational Model**, which it is supported by Relational Database Manage Systems widely found in the market.

In this chapter, the process of mapping the conceptual scheme to the relational scheme was introduced. A simplified process was applied to generate the relational data scheme of the clinical database example. The SQL queries used were very basic constructions.

To construct more complex databases and queries, one needs to be familiar with Relational Algebra and Relational Calculus, and also requires a deeper knowledge of SQL. It is important to emphasize that this chapter is only an introduction to database design and the SQL languages. For a more in-depth understanding the user is referred to the *Further Readings* section below.

Further readings

1. Elmasri, R.; Navathe, S. B. Fundamentals of Database Systems. Pearson (Addison Wesley), 4th ed., 2004.
2. Korth, H.; Silberschatz. Database Systems. Third Edition. Makron Books, 1998.
3. Raghu Ramakrishnan e Johannes Gehrke, Database Management Systems, Second Edition, McGraw-Hill, 2000.
4. Teorey, T.J., Lighstone S., Nadeau, T. Database Modeling and Design, 4th. ed., Morgan Kaufmann Publishers, Inc, San Francisco, 2006.
5. Beaulieu A., Learning SQL ,O'Reilly Media, Inc., 2005.
6. Molinaro A., SQL Cookbook, O'Reilly Media, Inc., 2005
7. Taylor A. G., SQL For Dummies, For Dummies Publisher; 6 th ed., 2006