AUTOMATIC SORTING AND COMPARING OF HISTORICAL DATA LISTS

IRINA GAVRILĂ

In this article we try to describe the functional principles of a system for automatic data processing specialized in the comparison of multiple lists of names, recorded in historical documents at various times. This system has been used in a problem of social-history, in order to sort and intersect a number of lists containing historical information about the Romanian landlords in different moments of time: 1857, 1864, 1905, 1921.

Most times, the big volume of data does not allow for the manual comparison of such lists. The aim of the comparison is to establish how many and which individuals from a given list also appear in the other lists. In order to decide whether an individual from a list belongs also to another list, we may use his/her first and last name, or only the last name. The researcher cannot ignore the importance of the intermediary results of the comparison process.

For example, we could extract thousands of data items regarding the big landowners from "Anuarul General al României pe anul 1905". In the first phase of the research, these data items could automatically provide the lists of landowners for each county, as well as the alphabetically ordered list of the landowners with supplementary information, such as the village/town and county of each landowner. The lists could be separately generated for various social categories of individuals, and could contain supplementary information specific for the category. For example, for the boyards, the list could also contain the boyard's class, for officers – the military rank, for magistrates, prefects, and ministers – other information specific to the category. Once generated, these lists become valuable tools for other researchers as well.

This article will introduce the basic concepts of a system which could be applied in a variety of historical projects.

1. ALGORITHMS

The origin of the word "algorithm" is very interesting. Until 1957, this word was not found in the Webster's New World Dictionary. Instead, the dictionary contained

Rev. Roum. d'Hist., **XLIV**, *1–4*, p. 243–262, Bucarest, 2005

an old form, *algorism*, designating the process of arithmetic operations using the Arabic signs. After the Middle Ages, some linguists tried to explain the origin of the word algorism through the combination between the words *algiras* (painful) and *arithmos* (number). Others explained it as coming from the name of the king Algor of the Castile province.

Later, the mathematics historians explained the origin of the word algorism as coming from the name of a famous author of manuals, the Arab Abu Ja'far Mohamed Ibn Musa al Khovarizmi (825 A.D.), inhabitant of the city of Khovarizm, today Khiva. Gradually, the word algorism became algorithm. An old German mathematics dictionary defines the word algorithmus as follows: "under this name are known the four types of arithmetic computations, namely addition, subtraction, multiplication, and division".

By 1950, the word algorithm was frequently associated with the Euclidean algorithm, the process of finding the greatest common divisor of two integers, described in Euclid's Elements. Figure 1 presents a schema of the algorithm, where the initial values of a and b are the two integers, a rectangle indicate a computation, the diamond indicates a decision, and "a ← b" means replacing the value of a by the value of b, and "a mod b" means the remainder of the division of a to b.

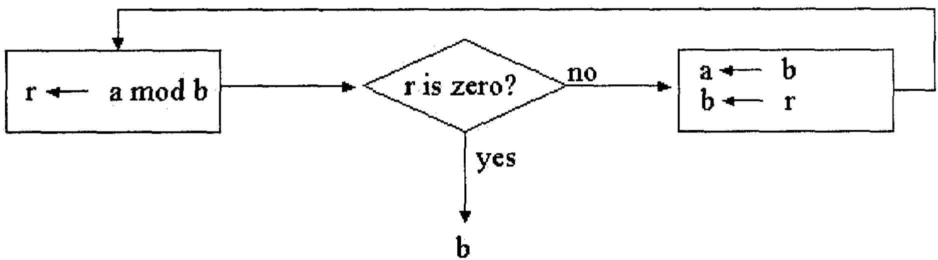

Fig. 1. – Euclid's algorithm.

There are five characteristics of any algorithm seen as a finite set of rules defining a sequence of operations that can be used to solve a specific type of problems:

a. *The finite character*. Any algorithm ends always after a finite number of steps. The number of steps can become arbitrarily big, but must be finite. It is important to note that the algorithm must end in a finite number of steps for any values of the input data. For example, Euclid's algorithm always (i.e., for any "correct" values of its input data – see below) terminates after a finite number of steps, which can be formally proved.
b. *Deterministic character*. Each step of an algorithm must be precisely defined, without ambiguities, and must produce the same result every time it's executed if the input data are the same.

c. *The input data.* An algorithm may have or not input data, i.e., the initial values given to the algorithm to operate on. The input data must belong to a specific set of objects. For example, the input data for Euclid's algorithm must be two integers from the set of positive integers.
d. *The output data.* An algorithm has output data, i.e., the answer provided to the specific problem. For example, Euclid's algorithm has as output data the greatest common divisor of the two positive integers passed as input data to the algorithm. It is important to note that the output data in some cases can be an error message, like "Wrong input data", or "The problem has no solution".
e. *Efficiency.* All operations executed within the algorithm must be fundamental enough, so that a person using paper and pen could be able to perform them correctly and in a finite time. For example, Euclid's algorithm involves only integer division, testing whether an integer is zero, and setting the value of a variable to the value of another variable.

In practice, when we solve a problem, we are looking not only for algorithms, but for good algorithms. A criterion to establish the "goodness" of an algorithm is the time necessary for its execution as function of its input. Usually, instead of the time (measured in seconds for example), we use the number of executions of each elementary step. Other criteria could be the simplicity and the elegance of the algorithm.

After this short description of the properties of algorithms, we will discuss a topic very frequently coming up in the programming theory and practice, namely the rearrangement of the elements of a list in ascending or descending order.

2. SORTING

Although the dictionary defines sorting as the process of separation and arrangement of objects by their class/type, the programming theory and practice use the term "sorting" in a much more special way, namely to designate the arrangement of objects in ascending or descending order. Obviously, that supposes that any two objects can be compared and it can be decided whether one of them is less, greater, or equal to the other.

Sorting has many applications, among them the following:

a. Grouping together all objects with the same value/identity belonging to a file. For example, suppose we have a file of 10,000 objects in arbitrary order, many of them with the same value, and we wish to rearrange the objects such that all objects with the same value appear in consecutive positions in the file. It's enough to sort the file's objects in ascending or descending order, and the problem is suddenly solved.

b. Finding all objects that are common to two or more files. The naive solution would be to start with the first object of the first file, and look for an object with the same value in the second file. Then start with the second object from the first file and look for an object with the same value from the beginning of the second file, etc. Note that for each object in the first file we start a search from the beginning of the second file. The solution becomes even more complex and inefficient when we have three or more files. By first sorting all files in the same – say ascending – order, finding the common objects is simply a matter of walking through all files only once.
c. Sorting as a searching aid. The simplest example is searching a word in a dictionary. Imagine how difficult would be to search for a word in a dictionary which is not alphabetically ordered. The search is much simpler and efficient when the dictionary is first sorted alphabetically.

Another example is the scientific compiler LARC created in 1960 by Computer Science Corporation. A compiler reads programs written in some sort of language and translates them in code understood by computers (machine code). LARC was an optimizing compiler for the FORTRAN language, and it used sophisticated sorting extensively, so that its algorithms processed portions of the source programs in a certain order that allowed for optimized, yet efficient translation (note that in the translation process, optimization means that the generated machine code is optimized, and that usually makes the compiler inefficient).

Sorting the objects of a file could be classified as: *internal sorting*, when all objects are stored in the computer's internal memory, or *external sorting*, when only the objects that are currently compared are stored in the computer's internal memory.

There are many sorting methods, some more efficient than others. In general, the minimum time needed to sort a file containing N objects is proportional to N log N. Here are a few sorting methods:

a. *Sorting by insertion*. The original file is not modified. A new file is created, whose objects will be sorted. Each object from the original file is inserted into the new file at its place.
b. *Sorting by interchange*. Starting with the first object in the original file, each object is compared with its neighbor at the right. If their order is not the expected one, they are interchanged. When we reach the end of the file, we start again from the beginning. We stop when no interchange is necessary.
c. *Sorting by selection*. Find the smallest object in the file and move it to the first place of the file. Then find the smallest object among the rest and move it to the second place of the file, etc.
d. *Sorting by counting*. Start with the first object in the file and count how many objects in the file are smaller than it. This number gives the object's final position in the sorted file.

As we have seen, sorting data files is useful because it groups together the objects with the same value, allows fast processing of multiple files sorted by the same criterion, and leads to efficient search algorithms. The rest of this section describes the basic principles of a system for sorting and automatic comparison of historical data lists.

3. CHARACTERISTICS OF A DATABASE MANAGEMENT SYSTEM

Today, the computer replaces tools traditionally used to accomplish everyday tasks. It replaced the typewriter in creating and editing of documents. It performs the mathematical operations in place of the electromechanical calculators. It replaced myriads of fiches, files, file cabinets as means of storing information. Compared to the old tools, the computer does much more, much faster, and with much more accuracy. There is an inconvenient: we do not have direct physical access to data, which can lead to permanent data loss when a computer system breaks down. But careful security and data back-up measures can prevent accidental data loss, so we can fully benefit from using the computer systems.

When storing important information in a computer system, we must respect the following four principles:

- storing data must be a simple and fast procedure, because it is supposed to be performed often;
- the media used to store data must be reliable, to prevent information loss;
- data retrieval must be a simple and fast procedure, regardless of the volume of data;
- there must be a simple way of separating and extracting only the information we need from the potentially huge volume of data stored.

The term of *database* has lost its initial meaning, that of any collection of objects. The stricter definition is that of a self-describing collection of integrated records. A record is a representation of some physical or conceptual object. For example, in a catagraphy we assign a record to each individual. Each record has multiple attributes, such as last name, first name, date of birth, city, county, etc. Individual names, cities, etc., are the data.

A database consists of both *data* and *metadata*. Metadata is the data that describes the data's structure within a database. If one knows how the data is arranged, then one can retrieve it. Because the database contains a description of its own structure, the database is self-describing. The database is integrated because it includes not only the data items, but also the relationships among data items.

The database stores metadata in an area called the *data dictionary*, which describes the tables, columns, indexes, and constraints. When there are no metadata,

like in the case of a flat file, the applications written to work with the flat files must contain the equivalent of the metadata as part of the application program.

Databases have different sizes, from simple collections of a few records to huge systems with millions of records. A personal database is designed by use by a single person on a single computer. Such a database usually has a simple structure and a relatively small size. A departmental database is used by the members of single department within an organization. This type of database is generally larger than a personal database and is necessarily more complex. Such a database must handle multiple users trying to access the same data at the same time. An enterprise database can be huge, and may model the critical information flow of an entire organization.

A *database management system* (DBMS) is a set of programs used to define, administer, and process databases and their associated applications. The database being managed is in essence a structure one builds to hold valuable data. A DBMS is the tool one uses to build that structure and operate on the data contained within the database.

There is a great variety of DBMS programs available on the market. Some run only on mainframe computers, some only on minicomputers, and some only on personal computers. There is strong trend for such products to work on multiple platforms or on networks that contain all three classes of machines.

Regardless of the size of the computer that hosts the database, the flow of information between database and user is always the same (Figure 2). The user communicates with the database through the DBMS. The DBMS masks the physical details of the database storage so that the application only has to concern itself with the logical characteristics of the data, not how the data is stored.

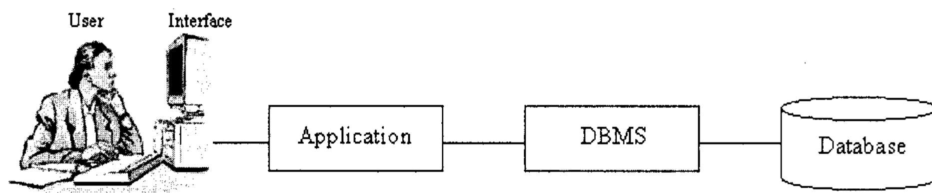

Fig. 2. – The user – database flow of information.

A *flat file* is simply a collection of data records, one after another, in a specified format, in effect, a list. Because the file doesn't store structural information (metadata), its overhead is minimal.

Each list of big landowners is an example of a flat file. Each field (last name, first name, city, county, etc.) has a fixed length (first name, for example, is always 20 characters long), and no structure separates one field from another. The person who created the database assigned field positions and lengths. Any program using

this file must know how each field was assigned, because that information is not contained in the database itself. Such low overhead means that operating on flat files can be very fast. On the minus side, however, application programs must include logic that manipulates the file's data at a very low level of complexity. The application must know exactly where and how the file stores its data.

Using a database instead of a flat file eliminates duplication of effort. Although database files themselves may have more overhead, the application programs can be easier to write, because the programmer doesn't need to know the physical details of where and how the data is stored.

Regardless if the database size, they are in general always structured according to one of three database models: relational, hierarchical, and network. The first databases to see wide use were large organizational databases, built according to either the hierarchical or the network model. Systems built according to the relational model followed several years later. The SQL language used to manipulate databases, applies only to the relational model and its descendant, the object-relational model.

We will concentrate now on the relational database model. Dr. E.F. Codd of IBM first formulated the relational database model in 1970, and this model started appearing in products about a decade later. Ironically, IBM did not deliver the first relational DBMS. That distinction went to a small start-up company, which named its product Oracle.

Relational databases have replaced earlier database types because relational databases have valuable attributes that distinguish them as superior. Probably the most important of these attributes is that relational databases enable you to change the database structure without making changes to applications that were based on the old structure. Suppose, for example, that you add one or more new columns to a database table. You don't need to change any previously written applications that will continue to process that table unless you alter one or more of the columns used by those applications. Of course, if you remove a column that an existing application references, you experience problems no matter what database model you follow.

Relational databases offer structural flexibility because their data reside in tables that are largely independent of each other. You can add, delete, or change data in a table without affecting the data in the other tables. A relational database is made up of one or more relations. Each relation has its own table.

A *relation* is a two-dimensional array of rows and columns, containing single-valued entries and no duplicate rows. Each cell in the array can have only one value and no two rows may be identical. As an example, the following is a fragment of a relation called "BIG LANDOWNERS 1905", with columns for a current number, last name, first name, city, and county. Each row in the relation contains a landowner attributes.

1	Ailenei	Grigore	Sarul Dornei	Suceava
2	Anechitei	Andrei Vasile	Sarul Dornei	Suceava
3	Anechitei	Gheorghe Vasile	Sarul Dornei	Suceava
4	Anechitei	Vasile Simion	Sarul Dornei	Suceava
5	Apopei	Grigore	Sarul Dornei	Suceava
6	Abaza	Elena	Vultureni	Tecuci
7	Acherman	Loebel	Caragele	Buzau

Columns in the array are self-consistent, in that a column has the same meaning in every row. The order in which the rows and columns appear in the array has no significance. As far as the DBMS is concerned, it doesn't matter which column is first, which is next, and which is last. The same is true of rows.

Every column in a database table embodies a single attribute of the table. The column meaning is the same for every row of the table. Each row in the table (also called a record) holds the data for a single landowner. The relations in a database model correspond to tables in a database based on the model.

Tables can contain many columns and rows. Sometimes all of that data interests us, and sometimes it doesn't. Only some columns of a table may interest use, or perhaps we want to see only rows that satisfy a certain condition. Some columns of one table and some other columns of a related table may interest us. To eliminate data that isn't relevant to our current needs, we can create a view. A *view* is subset of a database that an application can process. It may contain parts of one or more tables.

Views are sometimes called virtual tables. To the application or the user, views behave the same as tables. Vies, however, have no independent existence. They allow us to look at data, but are not part of the data.

Say, for example, that we work with a database that has a table "BIG LANDOWNERS" with the columns Last name, First name, City, County, Property, Surface. Assume that we want to see a screen that contains only the landowner names and the surface of his/her property. Creating from the "BIG LANDOWNERS" table a view that contains only those three columns enables us to view what we need without having to see all the unwanted data in the other columns. Figure 3 shows the derivation of the view. It should be noted that a view may derive its content from multiple tables, not just one.

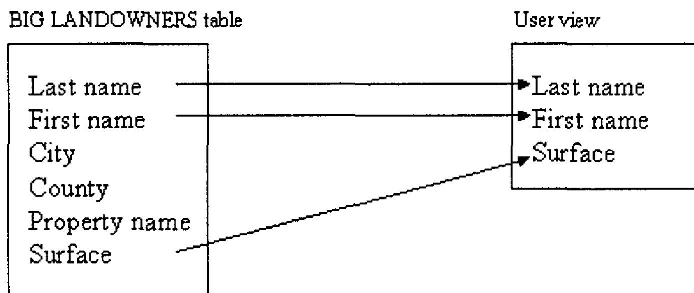

Fig. 3. – Deriving a view.

A database is more than a collection of tables. Additional structures, on several levels, help to maintain the data integrity. A database's schema provides an overall organization to the tables. The domain of a table column tells you what values you may store in the column. You can apply constraints to a database table to prevent anyone from storing invalid data in the table.

The structure of an entire database is its *schema*, or *conceptual view*. This structure is sometimes also called the complete logical view of the database. The schema is metadata and as such it is part of the database. The metadata itself is stored in tables that are just like the tables that store the regular data.

An attribute of a relation (that is a column of a table) can assume some finite number of values. The set of all such values is the domain of the attribute.

Constraints are an important, although often overlooked, component of a database. Constraints are rules that determine what values the table attributes can assume. As we said above, a column's domain is the set of all values that the column can contain. A constraint is a restriction on what a column may contain.

Database designers, like everyone else, are constantly looking to include the best features from different database models. This led to the hybrid object-relational model. Object-relational DBMSs extend the relational model to include support for object-oriented data modeling. Object-oriented features have been added to the international SQL standard, allowing relational DBMS vendors to transform their products into object-relational DBMSs, while retaining compatibility with the standard.

A database is a representation of a physical or conceptual structure. The accuracy of the representation depends on the level of detail of the database design. The amount of effort that you put into database design should depend on the type of information you want to get out of the database. Too much detail is a waste of effort, time, and hard drive space. Too little detail may render the database worthless. Usually you'll have to adjust the design (the level of detail) to meet changing real-world needs.

4. SQL FUNDAMENTALS

SQL is a flexible language that you can use in a variety of ways. The first thing to understand about SQL is that SQL is not a procedural language as are C and Java. To solve a problem in one of those procedural languages, you write a procedure that performs one specific operation after another until the task is complete. The procedure can be a linear sequence or may loop back on itself, but in either case the programmer specifies the order of execution.

SQL, on the other hand, is nonprocedural. To solve a problem using SQL, you simply tell SQL what you want, instead of telling the system how to get what you want. The DBMS decides the best way to get you what you request. However,

millions of programmers are accustomed to solving problems in a procedural manner. So, in recent years, there has been a lot of pressure to add some procedural language facilities, such as BEGIN blocks, IF statements, functions and procedures.

To illustrate what we mean by "telling the system what we want", suppose that we have a BIG LANDOWNERS table and we want to retrieve from that table the rows that correspond to landowners from the Prahova county. You can retrieve them by using the following query:

SELECT * from "BIG LANDOWNERS" WHERE County = Prahova;

In SQL, you don't have to specify how the information is retrieved. The database engine examines the database and decides for itself how to fulfill your request. You need only to specify what data you want to retrieve.

A *query* is a question you ask the database. If any if the data in the database satisfies the conditions of your query, SQL retrieves that data. You can extract information from a database in one of two ways:

- Make an ad hoc query from a computer console by just typing an SQL statement (query) and reading the results from the screen. This way is appropriate when we need that information right away, and probably we'll never need that information again.
- Execute a program that collects information from the database and then reports on the information, either on screen or in a printed report. Incorporating SQL queries in a program is a good way to run a complex query that you're likely to run again in the future.

SQL originated in one of IBM's research laboratories, as did relational database theory. In the early 1970s, as IBM researchers performed early development on relational DBMS, they created a data sub-language to operate on these systems. They named the prerelease version of this sub-language SEQUEL (Structured English QUEry Language). However, when it came time to formally release their query language as a product, they wanted to make sure that people understood that the released product was different from and superior to the prerelease DBMS.So thay named it SQL. By the time IBM introduced its SQL/DBMS in 1981, Relational Software Inc. (now Oracle Corporation) had already released its first relational DBMS. These early products immediately set the standard for a new class of DBMS. They incorporated variants of the SQL sub-language.

Soon a movement began to create an SQL standard to which everyone could adhere. In 1986, ANSI (American National Standards Institute) released a formal standard named SQL-86. ANSI updated that standard in 1989 and again in 1992. The most recent SQL standard is SQL-2003, updated in 2005.

The SQL command language consists of a limited number of commands that specifically relate to data handling. Some of these commands perform data definition

functions; some perform data manipulation functions; and others perform data control functions.

SQL is a data sub-language that works on a stand-alone system or on a multi-user system. SQL works particularly well on a client/server system. On such a system, users on multiple client machines that connect to a server machine can access a database that resides on the server to which they're connected. The application program on a client machine contains SQL data manipulation commands. The portion of the DBMS residing no the client sends these commands to the server across the communication channel that connects the client to the server. At the server, the server portion of the DBMS interprets and executes the SQL command and then sends the results back to the client across the communication channel. You can encode very complex operations into SQL at the client, and then decode and perform those operations at the server.

If you retrieve data by using SQL on a client/server system, only the data you want travels across the communication channel from the server to the client. The client/server architecture complements the characteristics of SQL to provide good performance at a moderate cost on small, medium, and large networks.

Unless it receives a request from a client, the server does nothing. It just stands around and waits. If multiple clients require service at the same time, however, the server needs to respond quickly. Servers generally differ from client machines in that they have large amounts of very fast disk storage. Servers are optimized for fast data access and retrieval. And because they must handle traffic coming in simultaneously from multiple client machines, servers need a fast processor, or even multiple processors.

The server is the part of a client/server system that holds the database. The server also holds the server portion of the DBMS. This part of the DBMS interprets commands coming in from the clients and translates these commands into operations in the database. The server software also formats the results of retrieval requests and sends the results back to the requesting client.

The client part of a client/server system consists of a hardware component and a software component. The hardware component is the client computer and its interface to the local area network. The client hardware may be very similar to the server hardware. The client software is the distinguishing component of the client. The client's primary job is to provide a user interface. As far as the user is concerned, the client machine is the computer, and the user interface is the application. The user may not even realize that the process involves a server. Aside from the user interface, the client also contains the application program and the client part of the DBMS. The application program performs the specific task you require, such as accounts receivable or order entry. The client part of the DBMS executes the application program's commands and exchanges data and SQL data manipulation commands with the server part of the DBMS.

Database operation on the Internet differs fundamentally from operation in a traditional client/server system. The difference is primarily on the client end. In a traditional client/server system, much of the functionality of the DBMS resides on the client machine. On an Internet-based database system, most or all of the DBMS resides on the server. The client may host nothing more than a Web browser, and maybe a browser extension (such as an ActiveX control). This shift of the system towards the server has several advantages: the client portion of the system is low cost or free (the browser); you have a standardized interface; the client is easy to maintain; you have a standardized client/server relationship; you have a common means of displaying multimedia data.

The main disadvantages involve security and data integrity: to protect information from unwanted access or tampering, both the Web server and the client browser must support strong encryption; browsers don't perform adequate data-entry validation checks; database tables residing on different servers may become desynchronized.

5. SQL APPLICATIONS TO SORTING AND COMPARING HISTORICAL DATABASES

This section shows how to use the SQL language to create database tables, enter data in the tables, update the data, and retrieve data from the tables. The following examples use the database management system Access, which is a component of the Microsoft Office suite.

a. CREATING TABLES

The Data Definition Language (DDL) is the part of SQL you use to create, change, or destroy the basic elements of a relational database. Basic elements include tables, views, schemas, catalogs.

You can create a table by using the SQL CREATE TABLE command. Within the command you have to specify the name and data type of each column. After you create a table, you can start loading it with data (which is a Data Manipulation Language function). You can change a table's structure by using the ALTER TABLE command. If a table becomes obsolete, you can eliminate it with the DROP command.

Let us create the table titled *1905* where each row contains the attributes of a big landowner registered in 1905: a unique identifier, the last name, the first name, the locality, and the county. These 5 attributes will be named 1905.Id, 1905. LastName, 1905.FirstName, 1905.Locality, 1905.County, where the prefix 1905 identifies the table. If there is no danger of confusion, the prefix 1905 can be dropped from the column names. We can create the table by using the SQL

command CREATE TABLE, or by using a graphical interface provided by the Access DBMS. Here we describe the first method.

Launch the DBMS Access and create a new, blank database called mydb.mdb anywhere on your hard disk. If the database is already created, double-click on its name in Windows Explorer in order to create the Access DBMS. In the window displayed in Figure 4 that appears in Access, select "Queries" from the left column, then select the option "Create query in Design view" and click on the "Open" menu.

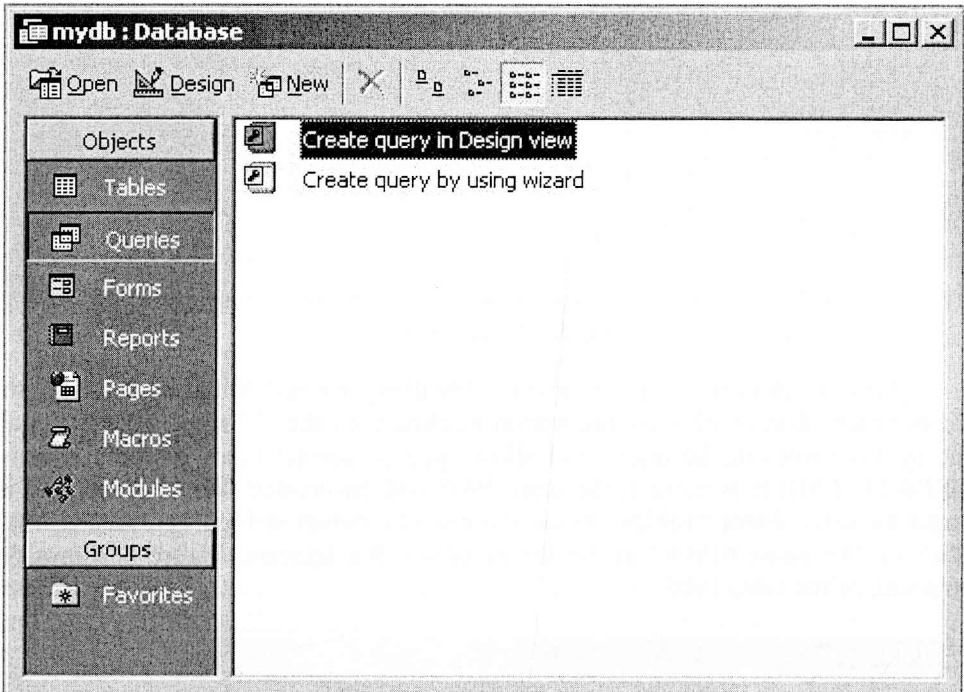

Fig. 4. – Selecting the option "Create query in Design view".

Close the window named "Show table", and select "SQL View" from the menu "View". The window called "Query1: Select Query" becomes an editing window where we can start writing queries (viz. Figure 5). Let's write the following SQL command:

```
CREATE TABLE 1905 (
Id INTEGER NOT NULL,
LastName CHAR(20),
FirstName CHAR(20),
Locality CHAR(20),
County CHAR(20));
```

The command may be written on a single line, or on multiple lines like in Figure 5.

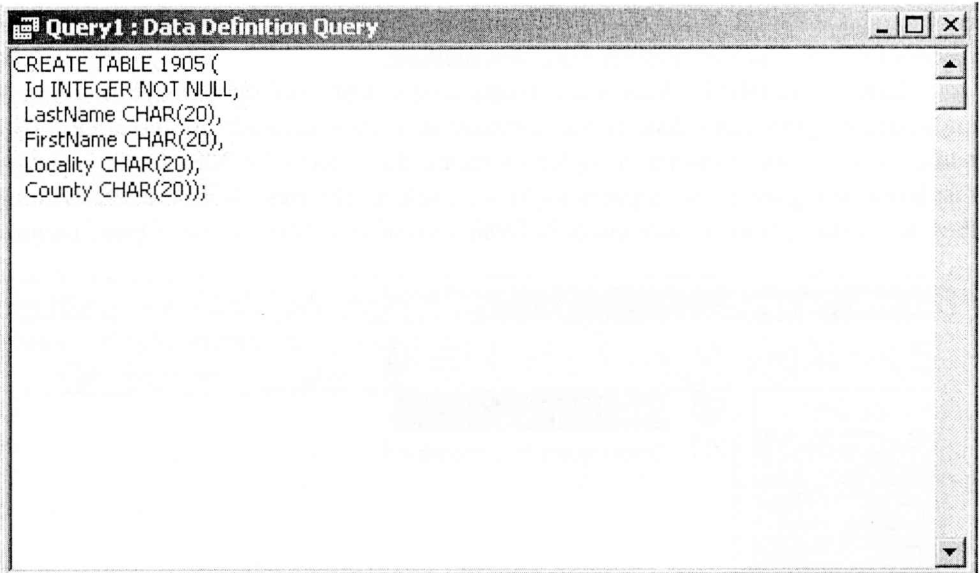

Fig. 5. – A query in Access.

Now we can ask Access to execute the query we just edited by selecting the menu Query/Run or clicking the button marked with the "!" sign. Access signals the syntax errors in the query and allows you to correct them. If the command CREATE TABLE is correct, the table 1905 will be created. We can examine its structure if we select "Tables" in the left column shown in Figure 4, then double-click on the name "1905" in the list of tables that appears. Figure 6 shows the structure of the table 1905.

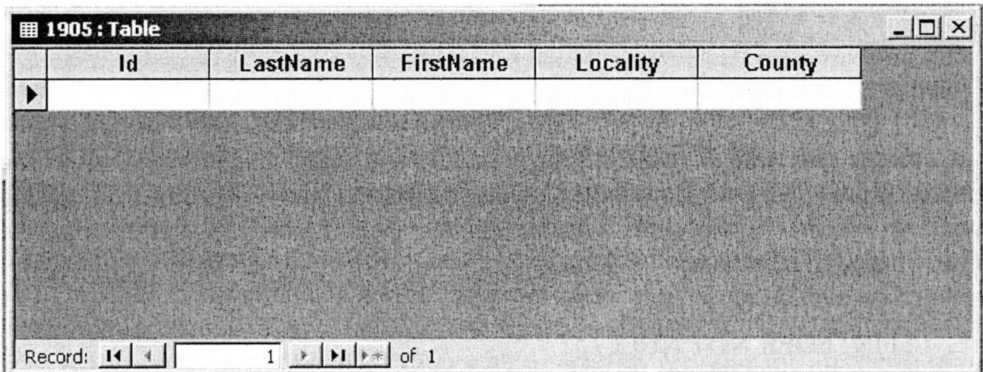

Fig. 6. – The structure of the table 1905.

We can obtain more information about the fields/attributes of the table 1905 if we select the menu View/Design view. Figure 7 shows the details about the field LastName.

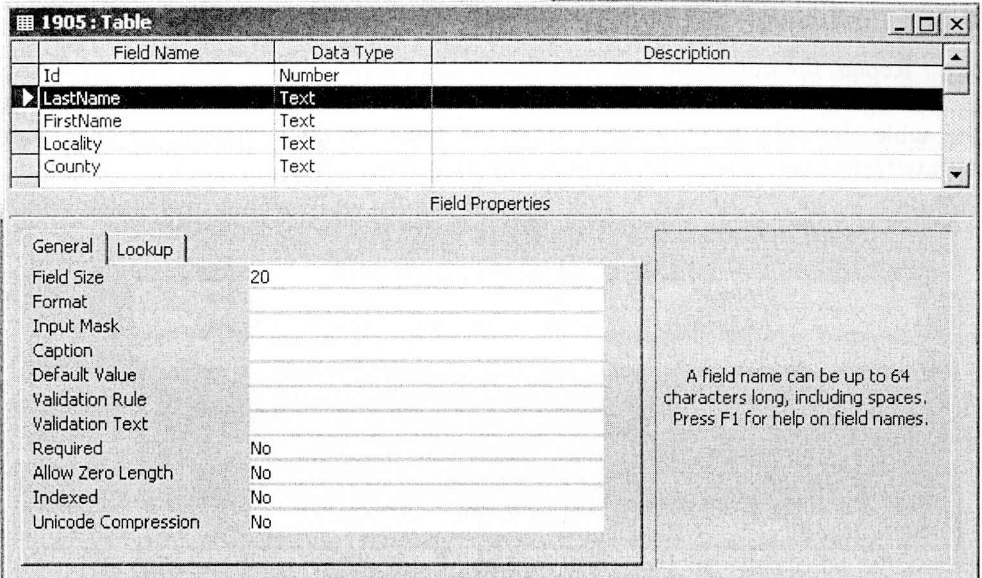

Fig. 7. – Structural details about the field LastName.

b. IDENTIFYING A PRIMARY KEY

Now we could fill out the table 1905 with data, but first we need to define something called a primary key for the new table. The primary key is a field or group of fields that uniquely identifies each row of the table by its value(s). In our case, Id is a good primary key. All other fields may have duplicate values: multiple landowners can have the same last or first name, locality, or county. However, the Id is unique. Designating a field or group of fields as a primary key may be performed through an SQL command or by using the graphical interface provided by Access. Here we present the SQL command:

ALTER TABLE 1905 ADD PRIMARY KEY (Id);

The ALTER TABLE command is entered in Access and executed in the same way as the CREATE TABLE command.

If we look again at the table structure, we notice a small key symbol that marks the field Id, which means that Id has become a primary key for the table 1905. From now on, if we try to enter two lines with the same value for the Id field, Access will signal an error.

c. LOADING DATA INTO THE TABLE

We can enter a row of data into the table by using the SQL command (a Data Manipulation Language command) INSERT INTO, namely:

INSERT INTO 1905 VALUES (1, "Ailenei", "Grigore", "Sarul Dornei", "Suceava");

Repeat the command but with the values for the second row, then third row, etc. Note the double quotes that delimit the alphanumeric values. We may examine the table, by selecting its name from the table list and clicking on the menu "View/Datasheet view". The result is shown in Figure 8.

1905 : Table

Id	LastName	FirstName	Locality	County
1	Alenei	Grigore	Sarul Dornei	Suceava
2	Anechitei	Andrei Vasile	Sarul Dornei	Suceava

Record: 2 of 2

Figure 8. Table 1905 with two records

d. LISTING THE CONTENT OF A TABLE

At any time we can list the content of a table using the SQL command SELECT. We can ask the DBMS to display all the fields, or just some selected fields. The following SQL command selects all fields:

SELECT * FROM 1905;

The following SQL command selects only the fields LastName and FirstName from each row:

SELECT LastName, FirstName from 1905;

The result is presented in Figure 9 (we assume the table 1905 had only two rows filled out).

Query2 : Select Query

LastName	FirstName
Alenei	Grigore
Anechitei	Andrei Vasile

Record: 1 of 2

Fig. 9. – A view with two fields selected.

e. RETRIEVING DATA FROM TABLES

The most frequent operation of data manipulation is extracting selected information from the database. The SQL language allows you to extract the content of a single record from a table with thousands of records; or all records that satisfy a combination of conditions; or all records from a table; or information from multiple tables linked between them, by using a single SELECT command. The simplest form of the SELECT command allows us to retrieve all information from a table:

SELECT * FROM [1905];

The star (*) specifies all fields from all records, of course from the table 1905, specified by the clause FROM [1905].

The SELECT command could be made much more complex, by adding clauses WHERE to the base command. The SELECT command with a WHERE clause has the following format:

SELECT column_list FROM table WHERE condition;

The column_list specifies the fields (i.e., columns) we wish to be retrieved in the result. The FROM clause specifies the table from which we want the record extracted. The condition specified in the WHERE clause acts like a filter for the records that will be retrieved, and can be very complex. For example, the following command displays only the last and first names of the records that have the locality value "Sarul Dornei" and the county value "Suceava":

SELECT LastName, FirstName FROM [1905] where Locality="Sarul Dornei" AND County = "Suceava";

f. OBTAINING THE LAST NAMES COMMON TO TWO TABLES

Assume that we define and load data to another table, 1921, with the same structure as the table 1905, and we want to find the last names that are common to the two tables. The UNION operator of the SQL language makes possible to extract information from two or more tables with exact the same structure, for example from 1905 and 1921. If the result would contain two or more identical records, a single record from them will be displayed, exactly as in the case of set union. We will select the last name from each table, with the condition that the name extracted from table 1905 has the same value as the name extracted from 1921, and we will use the operator UNION. Open a new query in Design view, as we did already once, and then select the menu Query/SQL Specific/Union. Now write the following command:

SELECT [1905.LastName] FROM [1905], [1921] WHERE [1905.LastName] = [1921.LastName] UNION SELECT [1921.LastName] FROM [1905], [1921] WHERE [1905.LastName] = [1921.LastName];

Access will display only the last names that are common to the two tables.

g. OBTAINING THE LAST AND FIRST NAMES OF THOSE RECORDS WITH LAST NAME COMMON TO TWO TABLES

It is enough to add the selection of the field FirstName at the previous command:

SELECT [1905.LastName], [1905.FirstName] FROM [1905], [1921] WHERE [1905.LastName] = [1921.LastName] UNION SELECT [1921.LastName], [1921.FirstName] FROM [1905], [1921] WHERE [1905.LastName] = [1921.LastName];

The result displayed by Access is ambiguous, in the sense that we don't know from which table is a resulting record (consisting of last and first names), unless we look for it in both tables. However, we can easily add the origin table to each record of the result with the following command:

SELECT [1905.LastName], [1905.FirstName], "1905" FROM [1905], [1921] WHERE [1905.LastName] = [1921.LastName] UNION SELECT [1921.LastName], [1921.FirstName], "1921" FROM [1905], [1921] WHERE [1905.LastName] = [1921.LastName];

Figure 10 shows the result.

Query2 : Union Query

1905.LastName	1905.FirstName	Expr1002
Gheorghe	Ion	1905
Gheorghe	Paul	1921
Gheorghe	Vasile	1921

Record: 1 of 3

Fig. 10. – Last name, first name, and origin table.

Note the names 1905.LastName, 1905.FirstName given to the resulting columns. They come from the fields selected from the first table, although the result contains records extracted from both tables. The good news is that we may rename the resulting columns, by using the clause AS in the SELECT command:

SELECT [1905.LastName] AS LastName, [1905.FirstName] AS FirstName, "1905" AS OrigTable FROM [1905], [1921] WHERE [1905.LastName] = [1921.LastName] UNION SELECT [1921.LastName], [1921.FirstName], "1921" FROM [1905], [1921] WHERE [1905.LastName] = [1921.LastName];

The resulting columns will be renamed LastName, FirstName, and OrigTable respectively.

h. OBTAINING THE RECORDS WITH THE SAME LAST NAME AND FIRST NAME IN TWO TABLES

It is enough to add another predicate that tests whether the first names are equal to the WHERE clause in our old SELECT command:

SELECT [1905.LastName] AS LastName, [1905.FirstName] AS FirstName, "1905" AS OrigTable FROM [1905], [1921] WHERE [1905.LastName] = [1921.LastName] AND [1905.FirstName] = [1921.FirstName] UNION SELECT [1921.LastName], [1921.FirstName], "1921" FROM [1905], [1921] WHERE [1905.LastName] = [1921.LastName] AND [1905.FirstName] = [1921.FirstName];

Creation and management of the historical databases are now an integral part of a new science, *the historical information science*. To historians, designing and managing databases, as well as mastering the powerful SQL language become more and more important in solving problems and testing historical hypothesis which involve a large quantity of information, impossible to handle by traditional methods.

GUIDE TO FURTHER READING

1. Alter G., M.P. Gutmann, *Casting Spells Database Concepts for Event-History Analysis*, in: "Historical Methods", vol. 33 (1999), no. 4, 165–176.
2. Boonstra O., M. Panhuysen, *From Source-oriented Databases to Event-history Data Files: A Twelve Step Action Plan for the Analysis of Individual and Household Histories*, in: "History and Computing", vol. 10 (1999), no. 1–3, 1–9.
3. Boonstra O., *Supply-side Historical Information Systems. The Use of Historical Databases in a Public Record Office*, in "Historical Social Research", no. 15, 1990, 66–71.
4. Bradley J., *Relational Database Design and the Reconstruction of the British Medical Profession: Constraints and Strategies*, in "History and Computing", vol. 6 (1994), no. 2, 71–84.
5. Breure L., *How to Live With Xbase: the Socrates Approach*, in: F. Bocchi, P. Denley, *Storia and Multimedia. Proceedings of the Seventh International Congress Association for History and Computing*, Bologna, Grafis Edizioni, 1994, 477–484.
6. Burnard L., *Relational Theory, SQL and Historical Practice*, in: C. Harvey, *History and Computing II*, Manchester, New York, Manchester University Press, 1989, 63–71.
7. Burnard L., *The Historian and the Database*, in: E, Mawdsley, N. Morgan, L. Richmond, Trainor R., *History and Computing III. Historians, Computers and Data. Applications in Research and Teaching*, Manchester, New York, Manchester University Press, 1990, 3–7.

8. Denley P., *Models, Sources and Users: Historical Database Design in the 1990'*, in: "History and Computing", vol. 6 (1994), no. 1, 33–43.
9. Greenhalgh M., *Databases for Art Historians: Problems and Possibilities*, in: P. Denley, D. Hopkin, History and Computing, Manchester, Manchester University Press, 1987, 156–167.
10. Greenstein D.I., *A Source-Oriented Approach to History and Computing: The Relational Database*, in "Historical Social Research", vol. 14 (1989), no. 51, 9–16.
11. Harvey C., J. Press, *Databases in Historical Research*, Wiltshire, Anthony Rowe, 1996.
12. Harvey C., J. Press, *Structured Query Language and Historical Computing*, in: "History and Computing", vol. 5 (1993), no. 3, 154–168.
13. McCrank L.J., *Historical Information Science. An Emerging Discipline*, Medford, New Jersey, 2002.
14. Welling G.M., *A Strategy for Intelligent Input Programs for Structured Data,* in: "History and Computing", vol. 5 (1993), no. 1, 35–41.