© Dmitriy Shironosov/ShutterStock, Inc.

LABORATORY 16

Databases

OBJECTIVE

- Write simple SQL queries using the Simple SQL app.

REFERENCES

Software needed:

1) Simple SQL app from the Lab Manual website (Simple SQL.jar)

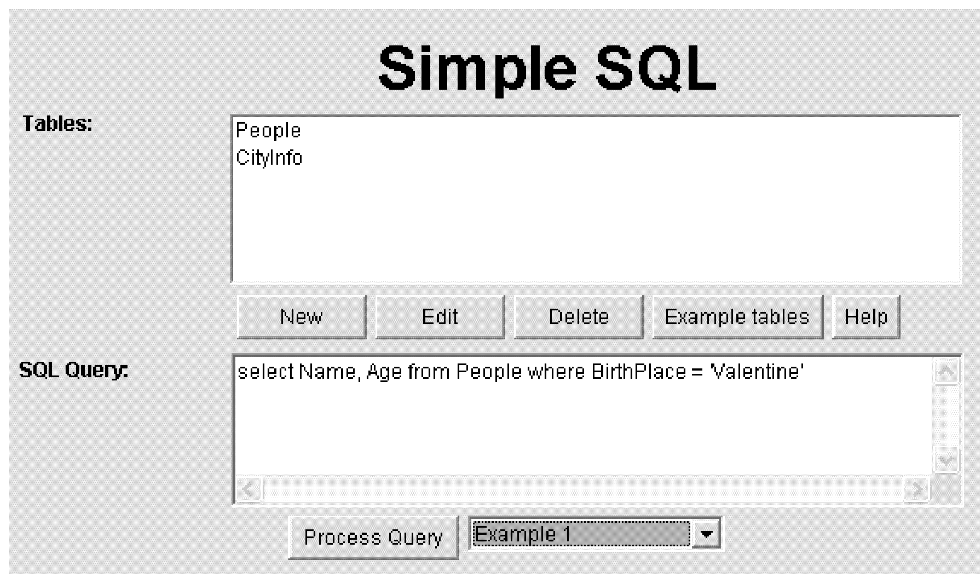
BACKGROUND

Database programs are less standard and generally much more expensive than spreadsheets. Microsoft Access is one of the most popular database programs. Depending upon which bundle of programs you have, you might have access to Access on your computer. This lab uses an app called "Simple SQL" to introduce relational databases and SQL (Structured Query Language) queries. Obviously, it is not as big and robust as Microsoft Access, but it has the virtue of being free and portable! If you have access to MySQL, Oracle, or other relational database products, you can experiment with more complicated SQL queries. Review these topics from your textbook, lecture notes, or online resources:

- Database management systems, the relational database model
- Structured Query Language (SQL), database design

ACTIVITY

Now that we've worked on spreadsheets, we'll experiment with another application that organizes vast amounts of information: databases. We will focus on SQL queries in a very simple relational database app. Start the Simple SQL app and click on the *Example tables* button. This generates two tables, *People* and *CityInfo*, and inserts them into the database. Next, choose *Example 1* from the *Process queries* pull-down list:



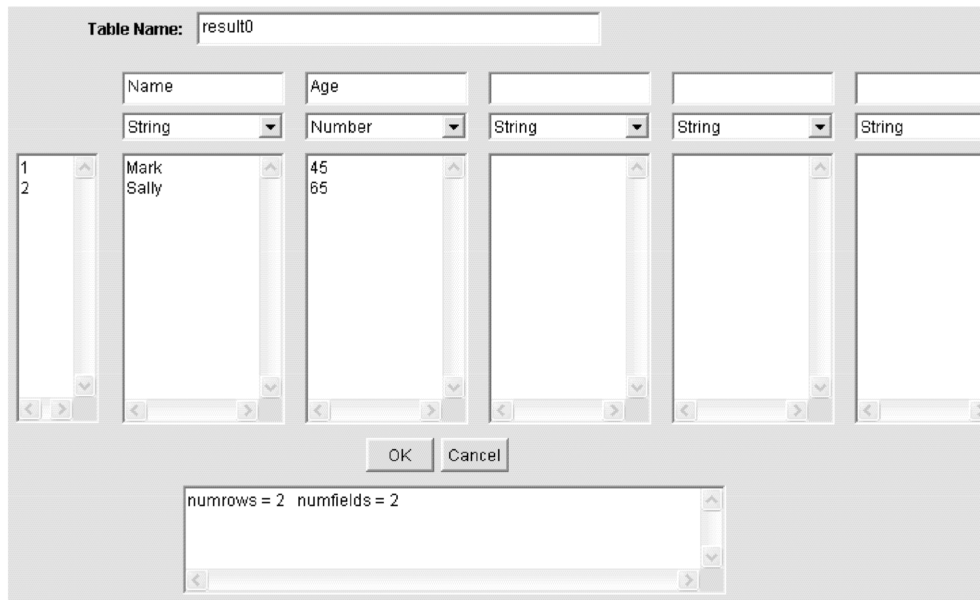
The tables in this database are listed in the upper text box labeled *Tables*. The SQL query appears in the lower box. You can type in your own queries, using *select* statements similar to those found in many textbooks and tutorials, or you can load the examples.

There are some major differences between real SQL and the SQL implemented by this app. For one thing, this app supports only a tiny subset of the full SQL language. Also, the app is case-sensitive, unlike real SQL, in which you can freely mix uppercase and lowercase. The Simple SQL app generally uses lowercase letters, but the names of the fields and the names of the tables, as well as the data inside apostrophes, often

contain uppercase letters. For example, the keyword *select* is all lowercase, but *Name* and *Age* are the names of two fields from a table named *People*.

Click on *Process Query,* and the app attempts to execute the SQL command in the lower text area. If it is successful, it makes a new table with a name like *result0*, *result1*, etc. These names appear in the *Tables* list in the upper box. You will see a new table, *result0*, added to the *Tables* list.

To view a table, click once on its name and then press the *Edit* button. Do that now with the *result0* table. A new window appears:



Each field gets its own column, with a name and a type. Only two types are allowed: *String* and *Number*. The rows of the table appear across the tops of the columns, and to the left is a list of row numbers. At the bottom is an informational box.

This edit window allows only five fields to be displayed, even though tables in Simple SQL can contain up to eight fields (which is necessary when joining, which we'll talk about in a moment). There are modification commands in SQL you can read about, such as *insert*, *update*, and *delete*. Such commands are essential if a program is changing the table, such as a banking program written in COBOL or a medical statistics program written in C++ would do.

However, Simple SQL supports only two commands: *select* and *join*. To change anything in a table, such as the table name, field names, field types, or data in the fields, you must do it in the edit window, then click the *OK* button to save your changes. To add a new field, simply pick a blank column and type in information. Make sure to name your new column of data by typing a name in the blank text field at the top of the column. If Simple SQL does not find a name there, your new field will not be recognized.

To copy a table, select it from the *Tables* list and click the *Edit* button. Rename the table in the edit window and save it by clicking *OK*. To delete a table, click once on its name in the main window and then click the *Delete* button.

When editing, you must be careful if you decide to delete a row. If you delete one row from a particular column, you must delete the same row from the other columns. Otherwise, Simple SQL will line up the rows as they appear in the columns, and your data may get "twisted." Also, note that in some circumstances, Simple SQL pads out missing rows in some columns with 0. In addition, don't edit the row numbers in the leftmost column. Simple SQL ignores whatever you do there and refreshes the numbers when you redisplay the table.

Next, let's *join* for a while. No, this is not carpentry or some collective form of meditation. *Join* is one of those mathematical operations of SQL, and its basis is the Cartesian product. Select *Example* 4 from the pull-down box and click *Process Query*. Look at the resulting table:

Table Name: result1

	People.Name	People.Age	People.BirthPlace	CityInfo.City	CityInfo.Famo
	String	Number	String	String	String
1	Mark	45	Valentine	Valentine	cattle
2	Mark	45	Valentine	Greeley	cattle
3	Mark	45	Valentine	San Francisco	Rice-a-Roni
4	Sally	65	Valentine	Valentine	cattle
5	Sally	65	Valentine	Greeley	cattle
6	Sally	65	Valentine	San Francisco	Rice-a-Roni
7	Kathy	36	Ainsworth	Valentine	cattle
8	Kathy	36	Ainsworth	Greeley	cattle
9	Kathy	36	Ainsworth	San Francisco	Rice-a-Roni
10	Doran	40	San Francisco	Valentine	cattle
11	Doran	40	San Francisco	Greeley	cattle
12	Doran	40	San Francisco	San Francisco	Rice-a-Roni

numrows = 15 numfields = 5

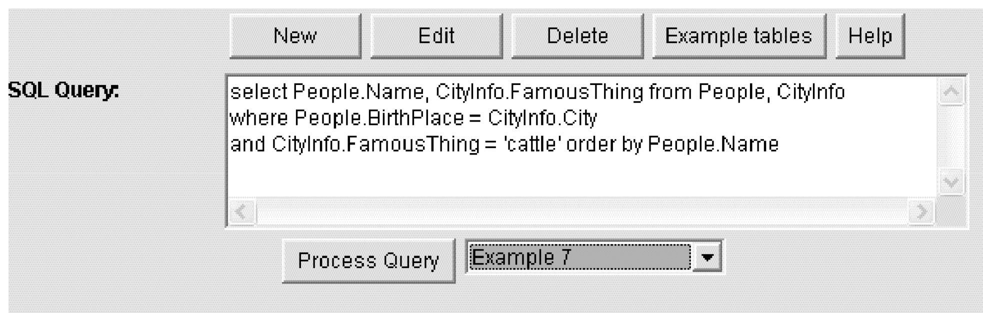
The statement

```
People join CityInfo
```

is a valid SQL statement that takes the Cartesian product. This means that all the fields in both the *People* and the *CityInfo* tables are in the new result, and the rows follow this pattern. For each and every row in *People*, append each row of *CityInfo*. Since there are five rows in *People* and three rows in *CityInfo*, there will be 15 rows (5×3) in the result. Look at the first three rows of the result: *Mark*, *45*, *Valentine*. Notice that this is the same as the first row of *People*. But the first three rows of *CityInfo* are appended to the end, forming three new, unique rows. This is the Cartesian product and the basis of *join*.

Why do tables join? To make room for a large family dinner? No, sorry—tables join so that the information in them can be cross-correlated. In a moment, we will correlate information from both *People* and *CityInfo*. In order to do that, all of the information needs to be temporarily stored in a Cartesian product.

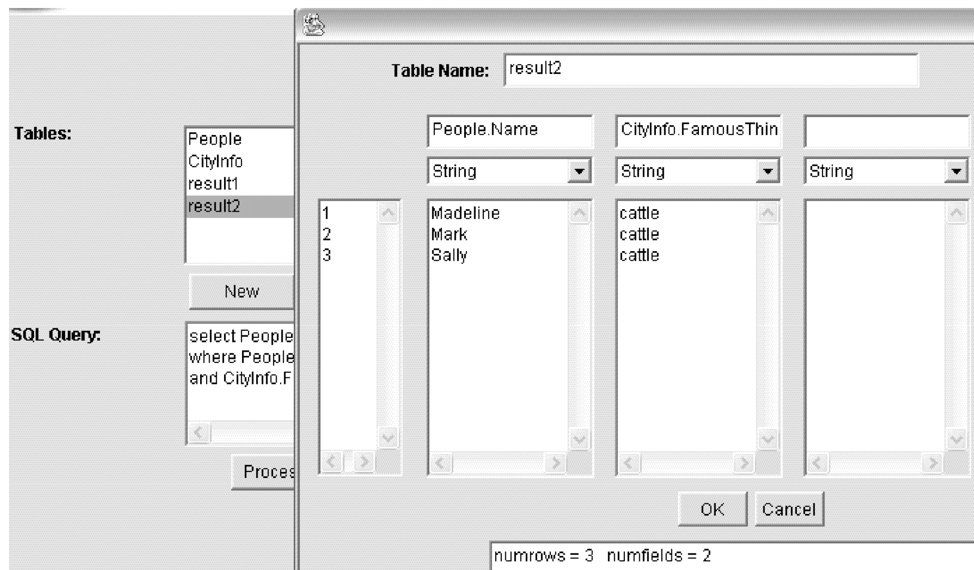
Select *Example 7* from the pull-down menu and take a look at the lengthy SQL query that appears.



Let's figure out what the query will do before we try it. It says that we will select two fields from two tables: *People* and *CityInfo*. (Keywords are helpful in deciphering computer statements like this. Look for *select*, *from*, *where*, *and*, and *order by*.) The names of the fields in a joined table are prefixed with the names of the tables from which they came in order to differentiate them, since it is common for tables to use the same field names. You can change the field names by using the edit window.

We aren't going to keep all the rows in the Cartesian product table, though—only those that fit the criteria we specify. We want rows where the *BirthPlace* field equals the *CityInfo* field, and at the same time those rows where the *FamousThing* is *cattle*. (Of course, these examples, which are quite simplistic and use conjured-up data, are for demonstration purposes only. Valentine, Nebraska, and Greeley, Colorado, are both great places with a lot more going for them than just ranching, as the author knows!)

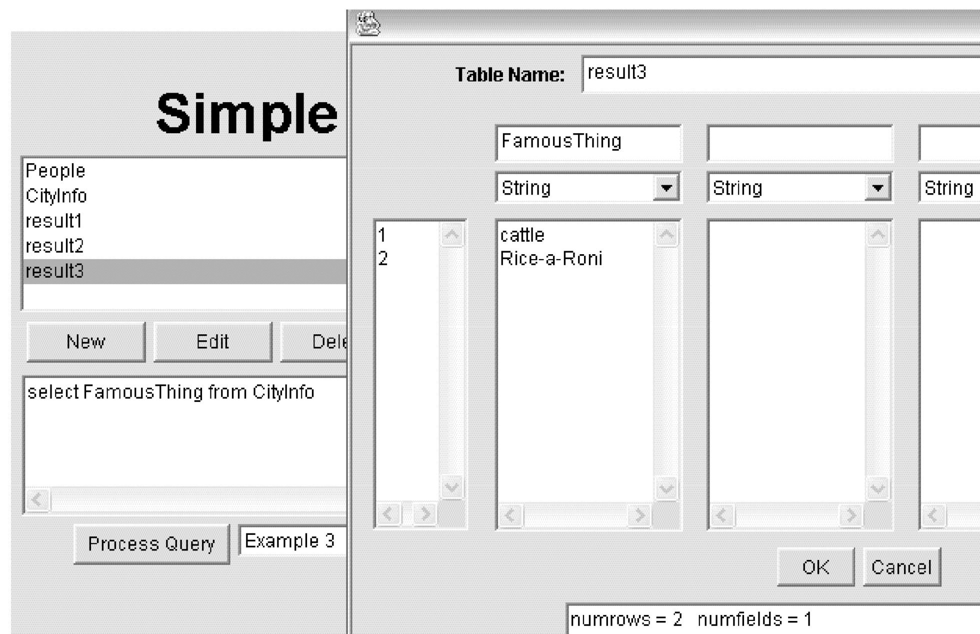
Notice the *and* keyword that joins the two conditions. You can also join conditions with *or*. Simple SQL also permits the *like* operator for very simple string patterns.



Lastly, the *order by* clause sorts the new table by the indicated field. No changes are made to the existing tables. Only the new one is affected by your query. Let's run the query and see what we get.

Only the cities Valentine and Greeley have "cattle" listed as their *FamousThing*. Mark and Sally were both born in Valentine, and Madeline was born in Greeley, so this table lists these three people, after sorting by their names.

Let us try one last thing with relational databases. Select *Example 3* from the pull-down menu and click on *Process Query*.



On the face of it, this query is a simple projection of the *CityInfo* table. A projection keeps only the named columns and throws away the rest. It keeps all the rows, however, whereas the `select...where` query throws away some of the rows.

Wait a minute! If you view the original *CityInfo* table (by selecting it and clicking the *Edit* button), you see there were three rows—but in the projection there are only two. What happened? Relational databases typically suppress duplicate rows. That is, they do not permit two rows to be exactly identical in every field. Notice that in the original *CityInfo* table, no two rows were identical. Even though the word "cattle" appears in two rows, the city names were different. When you delete a column, some rows may end up being identical, so databases remove the duplicates.

TIP

Simple SQL can load and save your tables (from the edit window). This makes it much easier to save your work and resume it later.

EXERCISE 1

Name________________________________ Date____________________

Section______________________________

1) Start the Simple SQL app.
2) Load the two example tables, *People* and *CityInfo*, by clicking on the *Example tables* button.
3) Write an SQL query that selects people whose age is greater than 25. Include all fields by specifying `select *`.
4) Now print out only the names of these people. (This is a *select* statement that looks like *Example 3*.)
5) Edit the result so the window pops up, displaying the result table, and take a screenshot.

EXERCISE 2

Name____________________________ Date____________________

Section____________________________

1) Start the Simple SQL app.
2) Load the two example tables: *People* and *CityInfo*.
3) Edit the *CityInfo* table and add a new field called *Population*. Its type is *Number*. Fill in the following information: San Francisco has about 800,000 people, Valentine 2,000, and Greeley 60,000.
4) Write an SQL query that prints out the cities where cattle is the famous thing. We don't want to see any other field except the city name.
5) Take a screenshot of the resulting table.
6) Now write an SQL query that prints out the famous thing of big cities, where "big" means the population is over 100,000.
7) Take a screenshot of the resulting table.

EXERCISE 3

Name____________________________________ Date______________________

Section___________________________________

1) Keep using the Simple SQL app. You will need the modified *CityInfo* table, which has the population information from Exercise 2.
2) Write an SQL query that prints out the names of the people in the *People* table alongside the population of the city in which they were born. (For help in writing this, look at Example 7 in the app.) Your result may be sorted or unsorted.
3) Take a screenshot of the main window so that your SQL query appears.

EXERCISE 4

Name____ Date________________________

Section_______________________________________

1) Start the Simple SQL app and load the two example tables: *People* and *CityInfo*.
2) Edit the *CityInfo* table and add a row for Ainsworth. Its famous thing is corn. (You do not need to type in a row number, since the app will fill this in when you save it.)
3) Add a new field called *State*. Its type is *String*. Fill in the following information: Valentine and Ainsworth are in Nebraska; Greeley is in Colorado, and San Francisco is in California.
4) Take a screenshot. Save your changes by clicking on the *OK* button.
5) Now we will make an entirely new table. Back at the main Simple SQL window, click on *New*. An edit window will pop open with an empty table named *newtable*. Change the name to *StateInfo*.
6) Set up three fields:

Name:	Name of the state
Capital:	Name of the capital city
Population:	Total population of the state
Bird:	The state bird

   Can you guess the proper types of each of these?

7) Type the right data into each field. You could find the data on www.wikipedia.com, but here are the details for your convenience:

California	Sacramento	33,871,000	California Quail
Nebraska	Lincoln	1,711,000	Meadowlark
Colorado	Denver	4,301,000	Lark Bunting

   (The app will permit you to type commas into your numbers, which is very humane. Many computer applications and programming languages are more persnickety.)

8) Take a snapshot of the table and click *OK* to save your changes to *StateInfo*.
9) Now we are going to link three tables at once, which is definitely not for the faint of heart. We do this in order to answer the following question:

   What are the names of the state birds for each of the people in our People table?

   See if you can answer this "by hand" first without writing SQL. As you struggle to find the people's birthplaces, and the states that those birthplaces are in, and the state birds of the states that those birthplaces are in—whew! Gets complicated! This is why computers are still not able to understand human language—there's so much to know and so much logic to use when you chain facts together.

10) Now comes the computer part: Write an SQL query that answers this question. Once you get the SQL query correct, take a snapshot of your screen. (Hints are provided in step 11.)

11) There are several ways to tackle this. You might first join *People* and *CityInfo*, which is Example 4, and then join the resulting table with *StateInfo*. However, our app permits only five columns per table, so you will have to project out unnecessary columns, like the people's ages.

12) A better way is to write one SQL statement, like Example 7, that implicitly joins the tables. Unfortunately, Simple SQL requires that no more than two tables appear in a single select statement, a restriction that grown-up databases like Oracle and MySQL do not share. But you can write two SQL statements to do the joins. You will have to write one SQL query statement and process that, take a snapshot, and then write a second SQL query statement and process that, taking a second snapshot.

DELIVERABLES

Turn in your screenshot showing your program after it finishes running and your screenshot showing the edit window with your program in it.

Since you are running Simple SQL as a standalone Java application, save your tables and then print them out. Use a word processor or a text application (like Notepad) to print out the file. Your instructor may ask you to hand in the file electronically, too. Consult the instructor for details on how to do this.

DEEPER INVESTIGATION

Databases get a bad rap as being boring, sort of like cost accounting for business majors. However, there are a lot of fascinating issues, and it can be great fun writing tricky queries to get the computer to do your bidding, not unlike programming.

One thing that many databases require you to identify is a key, which is one of the fields, or a combination of the fields, that uniquely identifies the rows of a table. Can you find keys in both the *People* and *CityInfo* tables? What kind of keys would be good to use in a large database of the general population? Which fields would not be keys, because the information is duplicated?

Lastly, let's think about how human language interacts with SQL. Interestingly, SQL was originally named SEQUEL, for "Structured English Query Language," because its inventors at IBM wanted people to think that they could almost talk to the computer in a natural way.

There are several things that make translating ordinary language into query language difficult: ambiguity of words, changes in syntax, and different vocabulary. Think of several English words that are obvious synonyms. Then try writing one of the Example queries in more-or-less normal English, something you could give to a brother or sister who doesn't know SQL, and that would tell them what to do with the data. Lastly, take an English description of some data you would like and see if you can cast that into an SQL query. There are some programs that attempt to do this, but they are far from complete or foolproof.

Many spreadsheets act as databases. For example, Excel allows you to make simple queries involving selection. A deeper question is: Why do we need separate applications if the boundaries are so blurry that spreadsheets end up acting like databases, and databases incorporate spreadsheet functionality?