Lab 4

SQL and Relational Databases

Lab Objective: *Understand concepts of a relational database and the fundamentals of the SQL language via SQLite.*

When working with large amounts of data, it is important to be able to quickly find and retrieve interesting information. Fortunately, there is a way to handle such massive amounts of data in a reasonably efficient way: a database. A database is simply a structured repository of data, and it allows us to store and retrieve information very quickly. It is managed by a *database management system*, or DBMS. The DBMS is software that allows users to interact directly with the database.

Relational Databases

A *relational database* is a paradigm for organizing data inside of a database. In this paradigm, the data is broken down into tuples of information. These tuples are then grouped into tables, or *relations*, each of which is simply a set of tuples. Each table has a *schema* that defines the attributes of the tuples within the table. If we fix an order to the attributes in the schema, we can think of each attribute as a column of the table, and each tuple as a row of the table. See Figure 4.1 for an illustration of these ideas.

As an example, suppose we have demographic data for a large number of individuals. If we are interested in the gender and age of the individuals, we might make a table with schema (Name, Gender, Age). This table would consist of several 3-tuples, such as (Jane Doe, F, 20). Alternatively, we can view this table as having three columns and as many rows as there are individuals within our data set. We might also create a table with schema (Name, Employment Status, Income, Education).

In the relational paradigm, there must be at least one attribute in each schema that can act as a *primary key*. This can uniquely identify each tuple of the table. It is common to use an ID number or other such unique information for the primary key. In our example above, the "Name" attribute acted as a primary key. However, this attribute only works as a primary key provided no two individuals within the data set have the same name.

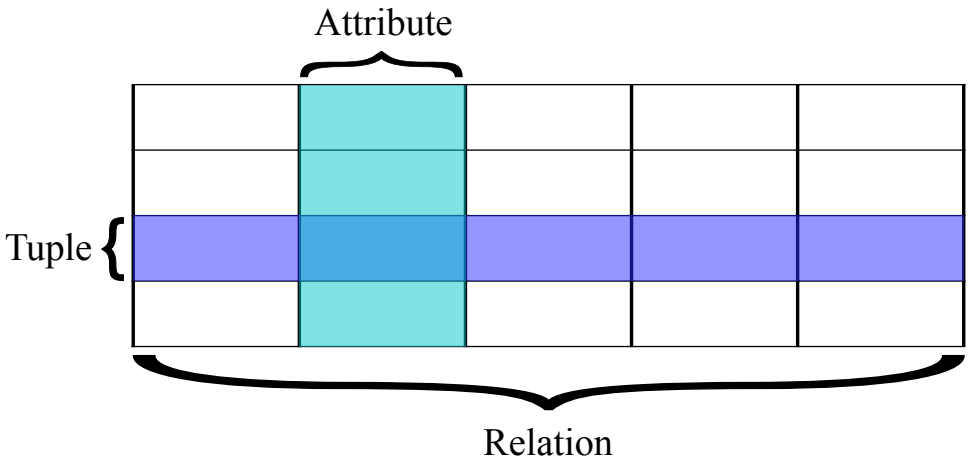

Figure 4.1: Elements of a relation.

One important feature of a database is the *transaction*, which is a conceptual protocol for interacting with the database. Most relational databases are transactional databases. The best way to conceptualize this is imagine that your database is like a bank. Your connection to the database is analogous to the bank teller. When you make one or more deposits and withdrawals, you are making a transaction. A database transaction should have certain properties to protect the integrity of the data. These properties are described in detail in `en.wikipedia.org/wiki/ACID`

Introduction to SQL

Most common DBMSs use a variant of the SQL language to interact with the database. SQL is an acronym for *Structured Query Language*, and may be pronounced like the word "sequel" or by saying the letters "s", "q", and "l" separately. While SQL is not generally portable across different DBMSs, we will focus on the parts of SQL that are relatively common. In particular, we will base our discussion on the SQLite database management system, a very popular DBMS.

SQL consists of blocks of code called statements. Each statement is made up of clauses which may or may not require predicates. Predicates specify conditions that can limit the effect of a clause.

> **Note**
>
> SQL commands are often written in all caps to help distinguish them from the other parts of the query. It is only a matter of style. SQLite, along with most other database managers, is case insensitive. In Python's SQL interface, the semicolon is also not needed. However, most other database systems will require it, so it's a good idea to conform in Python.

Let's look at an example SQL statement:

Keyword	Syntax
CREATE TABLE	CREATE TABLE <table> (<col1> <type>, <col2> <type>, ...);
DROP TABLE	DROP TABLE <table>;
CREATE INDEX	CREATE INDEX <name> ON <table> (<col>);
DROP INDEX	DROP INDEX <name>;

Table 4.1: The SQL Schema commands

Keyword	Syntax
INSERT INTO	INSERT INTO <table> <attributes> VALUES (<value1>, <value2>, ...);
UPDATE	UPDATE <table> SET (<col1>=<val1>, <col2>=<val2>, ...) WHERE <condition>;
DELETE	DELETE FROM <table> WHERE <condition>;
SELECT	SELECT <attributes> FROM <table> WHERE <condition>;

Table 4.2: The SQL Data Manipulation commands

```
SELECT * FROM table WHERE id=3+1 AND name='Bob';
```

This statement includes a SELECT clause and a WHERE clause. The WHERE clause contains two predicates: `id=3+1` and `name='Bob'`. These two predicates limit the effect of the SELECT clause because any resulting tuples in the table must satisfy both conditions. This entire statement is classified as a query since it does not modify the database in any way.

SQL has several classes of statements. The two main classes we will cover in this lab are schema (Table 4.1) and data manipulation (Table 4.2). We will give you a simplified description of each command and its syntax. You are encouraged to look up the full syntax outside of this lab.

SQL in Python

Python has built-in support for SQLite databases using the standard library. Let's open a database called `test1`.

```
import sqlite3 as sql
db = sql.connect("test1")
```

The `connect()` function is used to connect to a database. If it does not already exist, then a new database will be created using the string passed as the argument for the name. The new database was created as a file in the current working directory.

Ending the SQL Session

Once we are finished performing SQL statements and interacting with the database, we need to commit our changes and safely close the connection to the database. This can be done by calling methods on the database connection object.

```
db.commit()     #save changes made in the transaction
db.close()      #safely close the database
```

Method	Description
`execute`	Execute a single SQL statement
`executemany`	Execute a single SQL statement over a sequence
`executescript`	Execute a SQL script (multiple SQL commands)
`close`	Closes the cursor object

Table 4.3: Cursor object methods

Python Type	SQLite Type
`None`	`NULL`
`int`	`INTEGER`
`long`	`INTEGER`
`float`	`REAL`
`str`	`TEXT`
`buffer`	`BLOB`

Table 4.4: Python and SQLite types mapping

A database connection is automatically closed in Python when the connection object is garbage-collected. However, it is nice to be safe and explicit in closing a database connection using the `close()` method.

Cursor

To execute SQL commands, we need to get a cursor object from the database.

```
cur = db.cursor()
```

The cursor object has several useful methods (Table 4.3). Through the cursor, we will execute all of our SQL commands.

Before creating a table, we need to understand how SQLite stores information in a database. SQLite uses five native data types (a simplified system from other SQL database managers). Table 4.4, gives a mapping between Python and SQLite native types.

Creating and Dropping Tables

Let's create a table.

```
cur.execute('CREATE TABLE StudentInformation (StudentID INTEGER NOT NULL, Name ↩
    TEXT, MajorCode INTEGER);')
```

This will create the empty table in Table 4.5.

StudentID	Name	MajorCode

Table 4.5: StudentInformation

The arguments in parentheses are the column names followed by the data type that entries in that column will be, and together these form the schema of the table. The `INTEGER` data type in SQLite is a 1, 2, 3, 4, 6, or 8 byte integer depending on the value. The `NOT NULL` command is a *constraint* on the StudentID column. It requires that all records in the table have a student ID.

> **Note**
>
> SQLite does not enforce types on columns. Just like Python, SQLite is dynamically typed. However, most other database systems strictly enforce column types. It is a good idea to conform to the column types specified in the schema.

Note that each command we execute returns the same cursor object. This object is equipped with a method that allows us to look at any results of the previous command. The result is formally known as the *result set*. If you use `cur.fetchall()`, you will see an empty list. That is because the create table command does not return a result set.

Now we want to build a relation between students, the courses they've had, and their grades in those courses.

```
cur.execute('CREATE TABLE StudentGrades (StudentID INT NOT NULL, CourseID INT, ↩
    Grade TEXT);')
```

Problem 1. In this problem, you will create two new tables in the database "sql1". The first table will be called MajorInfo and have a column called MajorID and MajorName. MajorID is an integer and MajorName is a string.

The second table will be called CourseInfo and have columns called CourseID and CourseName, also integers and strings, respectively.

Hint: In order to view information about the columns of the table, run the following command:

```
cur.execute("PRAGMA table_info('table_name')")
for info in cur:
    print info
```

For each column, this command will output (ID, name, type, notnull, default value, primary key). Also, don't forget to commit and close your database.

We can also destroy tables using the `DROP TABLE` command.

```
cur.execute("CREATE TABLE test_table (id INT, name TEXT);")
```

We can delete the table by dropping it.

```
cur.execute("DROP TABLE test_table;")
```

StudentID	Name	MajorCode
55	John Smith	2

Table 4.6: StudentInformation

If a table doesn't exist, an exception will be raised. We can tell the database to drop the table only if it really exists by using `DROP TABLE IF EXISTS test_table;`.

Inserting and Removing Data

Let's insert some data into our new tables. We can add rows to tables using the `INSERT INTO` command.

```
cur.execute("INSERT INTO StudentInformation VALUES(55, 'John Smith', 2);")
```

After running this statement, we will have the table in Table 4.6.

Note that SQLite will assume that values match sequentially with the schema of the table. We can also specify the schema of the table to use in the mapping of the values.

```
cur.execute("INSERT INTO StudentInformation(MajorCode, Name, StudentID)  VALUES↩
    (55, 'John Smith', 2);")
```

This will map the value 55 to MajorCode and the value 2 to StudentID. This may be useful sometimes.

It can quickly become tedious to insert large amounts of data into a table, one row at a time. We can automate the process somewhat by using the `executemany` method of the cursor object. To insert several rows into a table using a single command, we can do the following:

```
cur.executemany("INSERT INTO StudentInformation VALUES (?, ?, ?, ?);", rows)
```

In the code above, we assume that `rows` is a Python list of tuples, each tuple containing the data for one row.

We may remove rows from a table using the `DELETE FROM` command.

```
cur.execute("DELETE FROM StudentInformation WHERE MajorCode=55;")
```

Achtung!

Never use Python's string operations to construct a SQL query. It is extremely insecure and is an easy target for a well known type of database called a SQL injection attack.

Parameter substitution can be used to construct dynamic queries. In the simplest way, it involves using a '?' character whenever you want to use a value and providing a sequence of values as a second argument to `execute()`.

```
statement = "INSERT INTO StudentInformation VALUES(?, ?, ?, ?);"
```

```
values = (55, 'John Smith', 372897382, 2)
cur.execute(statement, values)
```

Problem 2. The ICD is a large collection of codes used to classify any diagnosis that a doctor would make. When someone goes to the hospital or doctors office, their visit will be recorded using these codes. Insurance companies, the government, and researchers find this data useful. The data file provided to you, `icd9.csv`, has simulated health histories for one million persons. Each line has columns for identification number, gender, and age, followed by ICD-9 codes of various quantities. Note that the codes for each individual are written in a single string, each code separated by semicolons. Create a new database with a single table to store all the simulated data. Call the database "sql2" and the table "ICD." Your table should have four columns, one each for id number, gender, age, and codes.

Because of the volume of data, it is highly recommended you use the `executemany()` method of the cursor. It will be about twice as fast as using an `execute()` for each line of the CSV file. Recall the `csv` package in Python. To read a CSV file into a list of tuples, where each tuple consists of the delimited values of a particular line in the file, one can use the following code as a guideline:

```
import csv
with open('filename', 'rb') as csvfile:
    rows = [row for row in csv.reader(csvfile, delimiter=',')]
```

Hint: Don't forget to commit and close your database.

Problem 3. Create the following tables in the same database you created in Problem 1 ("sql1"). You may do so however you think is best.

StudentID	Name	MajorCode
401767594	Michelle Fernandez	1
678665086	Gilbert Chapman	1
553725811	Roberta Cook	2
886308195	Rene Cross	3
103066521	Cameron Kim	4
821568627	Mercedes Hall	3
206208438	Kristopher Tran	2
341324754	Cassandra Holland	1
262019426	Alfonso Phelps	3
622665098	Sammy Burke	2

Table 4.7: StudentInformation

ID	Name
1	Math
2	Science
3	Writing
4	Art

Table 4.8: MajorInfo

StudentID	ClassID	Grade
401767594	4	C
401767594	3	B-
678665086	4	A+
678665086	3	A+
553725811	2	C
678665086	1	B
886308195	1	A
103066521	2	C
103066521	3	C-
821568627	4	D
821568627	2	A+
821568627	1	B
206208438	2	A
206208438	1	C+
341324754	2	D-
341324754	1	A-
103066521	4	A
262019426	2	B
262019426	3	C
622665098	1	A
622665098	2	A-

Table 4.9: StudentGrades

ClassID	Name
1	Calculus
2	English
3	Pottery
4	History

Table 4.10: CourseInfo

Updating Rows of a Table

We can modify records in a table by using the `UPDATE` command.

```
cur.execute("UPDATE StudentInformation SET MajorCode=2, StudentID=55, Name='↩
    Jonathan Smith' WHERE StudentID=2;")
```

NOTE

When updating a table, having a sufficient `WHERE` clause is essential. Any record that matches the criteria will be modified. If we omitted the `WHERE` clause, every record in the table would be set to the values given in the example.

Adding Columns to a Table

After you have created a table, you can add more columns to the table by using the `ALTER TABLE` command. For example, if we wanted to add the column "age" into our students table. We run this command.

```
cur.execute("ALTER TABLE StudentInformation ADD COLUMN age INTEGER;")
```

Selecting Data From Tables

The process of retrieving data from a table in a database is accomplished by the `SELECT` statement. The `SELECT` statement can be thought of as a very high level set description. For example, to view the contents of an entire table, we simply need to unconditionally select its contents.

```
SELECT * FROM students;
```

This is equivalent to the following set (where x is a row).

$$\{x : x \in \text{classes}\}$$

We can also select specific columns.

```
SELECT StudentID, Name FROM students;
```

Or we can impose conditions on the selected rows.

```
SELECT StudentID, Name FROM students WHERE MajorCode=1;
```

This query results in the following table (Table 4.11) where the contents are all the students that are math majors.

You can also further refine which rows you select by using the `AND` or `OR` commands. These commands will connect expressions. For example, to get the math and science majors. One could run the command.

```
SELECT StudentID, Name FROM students WHERE MajorCode=1 OR MajorCode=2;
```

Select statements return a *result set*. This is an iterable object. Each row in the object is represented as a tuple of values.

```
cur.execute('SELECT StudentID, Name FROM students WHERE MajorCode=1;')
for student in cur:
    print student
```

We can also use the fetch methods of the returned cursor to extract rows from the result set (Table 4.12).

StudentID	Name
401767594	Michelle Fernandez
678665086	Gilbert Chapman
341324754	Cassandra Holland

Table 4.11: Selected students who are math majors.

Method	Description
`fetchone()`	Return a single row from the result set
`fetchmany(n)`	Return the next n rows from the result set
`fetchall()`	Return the entire result set

Table 4.12: Fetch methods of a cursor.

Problem 4. From the ICD9 table you created in Problem 2, how many men between the ages of 25 and 35 are there? How many women between those same ages? Return your answers as a tuple.

When an Error Occurs

It is important to be able to recover from errors gracefully, especially when working a database. Data integrity in a database is often a critical need. When an error occurs, we need to undo the changes that triggered the error. Fortunately, `sqlite3` reports a variety of errors. These errors and when they are raised is explained in PEP249 (`http://legacy.python.org/dev/peps/pep-0249/`).

Error The base class for errors thrown by `sqlite3`. All other errors inherit from this class. Catching this error will catch any error raised.

InterfaceError Raised when there is a problem with the interface to the database rather than the database itself.

DatabaseError Raised when there is an error with the database itself.

DataError Subclass of DatabaseError. Raised when there are errors in the processed data (division by zero, value out of range, etc.).

OperationalError Subclass of DatabaseError. Raised for errors related to the database that are not the fault of the programmer. For example, an unexpected disconnect, failure to process a transaction, a memory allocation error during a transaction, etc.

IntegrityError Subclass of DatabaseError. Raised when the relational integrity of the database is compromised.

InternalError Subclass of DatabaseError. Raised when there is an internal error such as an invalid cursor, out-of-sync transaction, etc.

ProgrammingError Subclass of DatabaseError. Raised for programming errors.

NotSupportedError Subclass of DatabaseError. Raised when a method is called that is not supported by the database.

The way to gracefully recover from errors is to catch them and handle them accordingly. For example, if any error occurs, with the interface or the database, we immediately rollback the transaction. If no error occurs, commit. We could use if-statements or we could use a try-except block.

```
try:
    <code>
    db.commit()
except sql.Error:
    db.rollback()
```

Note that rolling back is not needed if we are just performing queries. If we don't change any of the data in the database, there is no need to roll anything back. However, even with queries, there is the potential for errors. You must design your code to handle these errors gracefully.