Relational Data

Chances are, you’re going to be processing data stored in any more structured form than simple flat files, it’s going to be in some kind of database. And although schemaless/NoSQL/non-relational databases are popular for some applications, in a large number of cases you’re going to be dealing with data in a standard relational database (and this format honestly makes sense for the vast majority of use cases that require a database).

Overview

The basic unit in any relational data is the notion of a “relation” or “table”.

Person table

ID	Last Name	First Name	Role
1	Kolter	Zico	Instructor
2	Xi	Edgar	TA
3	Lee	Mark	TA
4	Mani	Shouvik	TA
5	Gates	Bill	Student
6	Musk	Elon	Student

Relations

Where relational data becomes interesting is when we have multiple tables and explicit relationships between them.

For example, storing role as a string, that could have any value, is not an ideal approach. A better alternative is to create a separate “Role” table, that lists the allowable roles for the course.

Role table

ID	Name
1	Instructor
2	TA
3	Student

Using this table, we can replace the “Role” attribute in our Person table with a “Role ID” attribute that points to the ID of the respective role for each person:

ID	Last Name	First Name	RoleId
1	Kolter	Zico	1
2	Xi	Edgar	2
3	Lee	Mark	2

ID	Last Name	First Name	RoleId
4	Mani	Shouvik	2
5	Gates	Bill	3
6	Musk	Elon	3

Primary and Foreign keys

It's still possible for the "Role ID" attribute to contain some number, e.g. 4, that doesn't have a corresponding entry in the Role table. This brings us to the concepts of keys and constraints.

In the above examples, "ID" attribute serves as what is called a primary key. A primary key is a unique identifier for each row in the table. It is common to have a single column (like we do here as the "ID" column) serve as the primary key, but that is not required; the primary key can consist of multiple columns so long as they are unique in every row. Every relation (table) in the database must have exactly one primary key.

A foreign key is an attribute that "points" to the primary key of another table. Thus, in the above example, the "Role ID" attribute in the Person table is a foreign key, pointing to the primary key of the Role table. And the foreign key constraint enforces the fact that the foreign key must point to a valid primary key in the relevant table.

The foreign key constraint help enforce consistency of the database, and also forces us to be careful when we delete elements. For example, if we delete a row from Role, we must also delete all the rows from Person that point to that primary key, or the foreign key constraint would be violated.

Indexes

Indexes are created to "quickly" look up rows by some of their attributes. For example, suppose we wanted to find all the people in our Person table with the last name of "Gates". Naively, there would be no way to do this except search over the entire table to see every column that matched this last name. Instead of doing this, we can build an index on the "Last Name" attribute to provide an efficient means for retreiving tuples based upon last name. To see how this works conceptually, consider a slightly more explicit form of the Person table, where we explicitly denote the location, on disk or in memory, where each tuple occurs (preusming here that each row takes exactly 100 bytes)

You can think of an index like a table with just the indexed attribute and the location field (location in the original table), but sorted by the indexed attribute. So for instance, an index on the Last Name attribute would take the form.

Entity relationships

The nature of inter-table relationships via primary and foreign keys actually leads to a number of different possible entity relationships (i.e., relationships between a row in one table and a row in another). Some of the common types are:

- One-to-one
- One-to-zero/one

- One-to-many
- Many-to-many

Pandas

There are a number of Python libraries that handle relational data, typically written as interfaces to several differen relational database management systems (software such as PostgreSQL, MySQL, or a variety of others). [Note: one aside is that software like PostreSQL and MySQL is most correctly refered to as a relation database managment system (RDBMS), not a database. The database is the actual tables and records specifying an actual collection of data.]

In this class, though, we'll mainly interact with relational data through two libraries: Pandas and SQLite. These are especially simple libraries as far as real databases go: Pandas is decidedly not a real relational database system (although it provides functions that mirror some functionality of them), whereas SQLite is a "real" RDBMS, but an extremely simple one without the standard client/server architecture of virtually any real production database. Nonetheless, for many data science problems they will suffice, and so we focus on them here.

We have already briefly seen Panda when we discussed data collection, and it ends up being one of the most useful Python libraries for data science. As we mentioned above (but we're going to repeat this fact many times), Pandas is not a relational database library, but instead a "data frame" library. You can think of a data frame as being essentially like a 2D array, except that entires in the data frame can be any type of Python object (and have mixed types within the array), and the rows/columns can have "labels" instead of just integer indices like in a standard array.

Let's see how to first create a data frame in Pandas that mirrors our Person table above (we'll leave out the "Role ID" column just to keep things simple).

```python
import pandas as pd

df = pd.DataFrame([(1, 'Kolter', 'Zico'),
                   (2, 'Xi', 'Edgar'),
                   (3, 'Lee', 'Mark'),
                   (4, 'Mani', 'Shouvik'),
                   (5, 'Gates', 'Bill'),
                   (6, 'Musk', 'Elon')],
                  columns=["id", "last_name", "first_name"])
df
```

```
   id last_name first_name
0   1    Kolter       Zico
1   2        Xi      Edgar
2   3       Lee       Mark
3   4      Mani    Shouvik
4   5     Gates       Bill
5   6      Musk       Elon
```

"Index" for Pandas actually means something more like "primary key" in a database table (though with the exception that it is possible to have duplicate entries).

That is, an index (if done right, without duplicate indices) is a identifier for each row in the database. We can set the index to one of the existing columns using the `.set_index()` call.

But you need to be very careful about one thing here. By default, most Pandas operations, like `.set_index()` and many others, and not done in place. That is, while the df.set_index("id") call above returns a copy of the df dataframe with the index set to the id column (remember that Jupyter notebook displays the return value of the last line in a cell), the original df object is actually unchanged here.

If we want to actually change the df object itself, you need to use the `inplace=True` flag for these functions (or assign the original object to the result of a function, but this isn't as clean).

```
print(df.set_index("id"))

# df

df.set_index("id", inplace=True)
df
```

```
   last_name first_name
id
1     Kolter       Zico
2         Xi      Edgar
3        Lee       Mark
4       Mani    Shouvik
5      Gates       Bill
6       Musk       Elon
```

You can access individual elements using the `.loc[row, column]` notation, where row denotes the index you are searching for and column denotes the column name.

```
df.loc[1, "last_name"]
```

```
'Kolter'
```

If we want to access all last names, (or all elements in a particular row), we use the : wildcard. For example

```
df.loc[:, "last_name"]
```

```
id
1    Kilter
2        Xi
3       Lee
4      Mani
5     Gates
6      Musk
7     Moore
Name: last_name, dtype: object
```

We can pass a list of desired columns, to get a DataFRame return objet:

```
df.loc[:, ["last_name"]]
```

```
   last_name
id
1     Kilter
2         Xi
3        Lee
4       Mani
5      Gates
6       Musk
7      Moore
```

We can do a similar thing with row indexes.

```
df.loc[[1,2],:]
```

```
   last_name first_name
id
1     Kilter       Zico
2         Xi      Edgar
```

We can additionally use `.loc` to change the content of existing entries:

```
df.loc[1,"last_name"] = "Kilter"
df
```

```
   last_name first_name
id
1     Kilter       Zico
2         Xi      Edgar
3        Lee       Mark
4       Mani    Shouvik
5      Gates       Bill
6       Musk       Elon
```

We can even add additional rows/columns that don't exist.

```
df.loc[7,:] = ('Moore', 'Andrew')
df
```

```
   last_name first_name
id
1     Kilter       Zico
2         Xi      Edgar
3        Lee       Mark
4       Mani    Shouvik
5      Gates       Bill
```

```
6       Musk        Elon
7      Moore      Andrew
```

Finally, remember that `.loc` always indexes based upon the “index” (i.e., effectively primary key) of the data frame along with the column name. If you want to instead access based upon positional index (i.e., using 0-indexed counters for both the rows and columns), you can use the `.iloc` property

```
df.iloc[4,1]
```

```
'Bill'
```

SQLite

Unlike Pandas, SQLite actually is a full-featured database, but unlike most production databases, it does not use a client/server model. Databases are instead stored direclty on disk and accessed just via the library. This has the advantage of being very simple, with no server to configure and run, but for large applications it is typically insufficient: because files are not very good at concurrent access (that is, many different processes/threads cannot simultaneously read and write from a single file), the system is not ideal for very large databases where multiple threads need to be constant readings from and writing to the database. Note that SQLite does have some limited forms of concurrency in this respect, but nothing sophisticated when compared to larger scale databases. If you do want to investigate a more “production strength” client/server database, I’d highly recommend looking into PostreSQL.

SQLite, as the name suggests, uses the SQL (structured query language) language for interacting with the database; note both “Sequel” and “Ess Queue Ell” are acceptable prononciations of SQL, but I personally learned it as “Sequel”, so would be completely unable to do anything else.

Interacting with SQLite (or any other SQL-based database) from Python is not ideal, because you typically use Python code to generate SQL expressions as strings, then execute them, which is not the most beautiful coding paradigm. For simple databases, though, it usually suffices to get the job done.

Let’s look at how to create a simple database with the “Person” and “Grades” tables that we had considered earlier.

```
import sqlite3
conn = sqlite3.connect("data_base.db")
cursor = conn.cursor()

### when you are done, call conn.close()
```

This code imports the library, creates a connection to the “database.db” file (it will create it if it does not already exist), and then creates a “cursor” into the database. The notion of cursor is common to a lot of database libraries, but essentially a cursor is an object that allows us to interact with the database. If we want to create the Person and Grades tables we saw above (to keep things simple, and later to illustrate joins, we’ll use the first version of the Grages table, with no associative table), we would use the following syntax.

```
cursor.execute("""
CREATE TABLE person (
    id INTEGER PRIMARY KEY,
    last_name TEXT,
    first_name TEXT
);""")

cursor.execute("""
CREATE TABLE grades (
    person_id INTEGER PRIMARY KEY,
    hw1_grade INTEGER,
    hw2_grade INTEGER
);""")
```

```
<sqlite3.Cursor at 0x22c43c45a40>
```

Let’s insert some data into these tables.

```
cursor.execute("INSERT INTO person VALUES (1, 'Kolter', 'Zico');")
cursor.execute("INSERT INTO person VALUES (2, 'Xi', 'Edgar');")
cursor.execute("INSERT INTO person VALUES (3, 'Lee', 'Mark');")
cursor.execute("INSERT INTO person VALUES (4, 'Mani', 'Shouvik');")
cursor.execute("INSERT INTO person VALUES (5, 'Gates', 'Bill');")
cursor.execute("INSERT INTO person VALUES (6, 'Musk', 'Elon');")

cursor.execute("INSERT INTO grades VALUES (5, 85, 95);")
cursor.execute("INSERT INTO grades VALUES (6, 80, 60);")
cursor.execute("INSERT INTO grades VALUES (100, 100, 100);")
```

```
<sqlite3.Cursor at 0x22c43c45a40>
```

If we want to see what has been added to the database, we can do this the “SQLite Python” way, which involves running a query and then iterating over the rows in a result returned by a `cursor.execute()` result, as so:

```
for row in cursor.execute("SELECT * FROM person;"):
    print(row)
```

```
(1, 'Kolter', 'Zico')
(2, 'Xi', 'Edgar')
(3, 'Lee', 'Mark')
(4, 'Mani', 'Shouvik')
(5, 'Gates', 'Bill')
(6, 'Musk', 'Elon')
```

Alternatively, it can be handy to dump the results of a query directly into a Pandas DataFrame. Fortunately, Pandas provides a nice call for doing this, the `pd.read_sql_query()` function, with takes the database connection and an optional argument to set the index of the Pandas dataframe to be one of the columns.

```
pd.read_sql_query("SELECT * from person;", conn, index_col="id")

   last_name first_name
id
1     Kolter       Zico
2         Xi      Edgar
3        Lee       Mark
4       Mani    Shouvik
5      Gates       Bill
6       Musk       Elon
```

The SELECT statement is probably the SQL command you'll use most in data science: it is used to query data from th database.

```
pd.read_sql_query("SELECT id,last_name FROM person WHERE id > 2;",
conn, index_col="id")

   last_name
id
3        Lee
4       Mani
5      Gates
6       Musk
```

Lastly, we can also delete values from tables using the DELETE FROM SQL command, using a similar WHERE clause as in the SELECT command.

```
cursor.execute("INSERT INTO person VALUES (7, 'Moore', 'Andrew');")
pd.read_sql_query("SELECT * from person;", conn, index_col="id")

cursor.execute("DELETE FROM person where id = 7;")
pd.read_sql_query("SELECT * from person;", conn, index_col="id")

   last_name first_name
id
1     Kolter       Zico
2         Xi      Edgar
3        Lee       Mark
4       Mani    Shouvik
5      Gates       Bill
6       Musk       Elon
```

Joins

Briefly, join operations multiple multiple tables into a single relation, matching between attributes in the two tables. There are four types of joins, though only the first two are used much in practice:

- Inner

- Left
- Right
- Outer

You join two tables on columns from each table, where these columns specify how to match the rows between the two columns. This should become more clear with a few examples. In the examples that follow, we're going to consider our Person and Grades tables, that we just created above, and we're join to join the tables on the `person.id` and `grades.person_id` columns.

Innner Joins:

If you don't know what type of join you want, you probably want an inner join. This does the "obvious" thing, of only returning those rows where the two columns in each table have matching values, and it appends the rows together for each of these matching rows.

In Pandas, you should do joins with the `.merge()` command: there is an alternative `.join()` command, but this always assumes you want to join on the index column for one of the data frames, and not the index frame for another, and overall is just a special case of `.merge()`.

```
df_person = pd.read_sql_query("SELECT * FROM person", conn)
df_grades = pd.read_sql_query("SELECT * FROM grades", conn)
df_person.merge(df_grades, how="inner", left_on = "id",
right_on="person_id")

   id last_name first_name  person_id  hw1_grade  hw2_grade
0   5     Gates       Bill          5         85         95
1   6      Musk       Elon          6         80         60

pd.read_sql_query("SELECT * FROM person, grades WHERE person.id =
grades.person_id" , conn)
```

Left joins

Whereas an inner join only kept those rows with corresponding entires in both tables, a left join will keep all the items in the left table, and add in the attribution from the right table (filling with NaNs if no match exists in the right table). Any row value that occurs in the right table but not the left table is discarded.

```
df_person = pd.read_sql_query("SELECT * FROM person", conn)
df_grades = pd.read_sql_query("SELECT * FROM grades", conn)
df_person.merge(df_grades, how="left", left_on = "id",
right_on="person_id")

   id last_name first_name  person_id  hw1_grade  hw2_grade
0   1    Kolter       Zico        NaN        NaN        NaN
1   2        Xi      Edgar        NaN        NaN        NaN
2   3       Lee       Mark        NaN        NaN        NaN
3   4      Mani    Shouvik        NaN        NaN        NaN
```

```
4   5      Gates         Bill          5.0         85.0         95.0
5   6       Musk         Elon          6.0         80.0         60.0
```

```
pd.read_sql_query("SELECT * FROM person LEFT JOIN grades ON person.id
= grades.person_id" , conn)
```

Right joins

A right join does what you might expect, the converse of the left join, where all the rows in the right matrix are kept. While SQLite has no syntax for right joins (you can achieve the same results by simply reversing the order of the two tables and doing a left join), Pandas does have built-in syntax for the right join

```
df_person.merge(df_grades, how="right", left_on = "id",
right_on="person_id")
```

```
    id last_name first_name  person_id  hw1_grade  hw2_grade
0  5.0     Gates       Bill          5         85         95
1  6.0      Musk       Elon          6         80         60
2  NaN       NaN        NaN        100        100        100
```

Outer joins

Finally, outer joins (also called a cross product) do what you may expect, and keep all rows that occur in either table, so essentially take the union of the left and right joins.

```
df_person.merge(df_grades, how="outer", left_on = "id",
right_on="person_id")
```

```
    id last_name first_name  person_id  hw1_grade  hw2_grade
0  1.0    Kolter       Zico        NaN        NaN        NaN
1  2.0        Xi      Edgar        NaN        NaN        NaN
2  3.0       Lee       Mark        NaN        NaN        NaN
3  4.0      Mani    Shouvik        NaN        NaN        NaN
4  5.0     Gates       Bill        5.0       85.0       95.0
5  6.0      Musk       Elon        6.0       80.0       60.0
6  NaN       NaN        NaN      100.0      100.0      100.0
```

joins