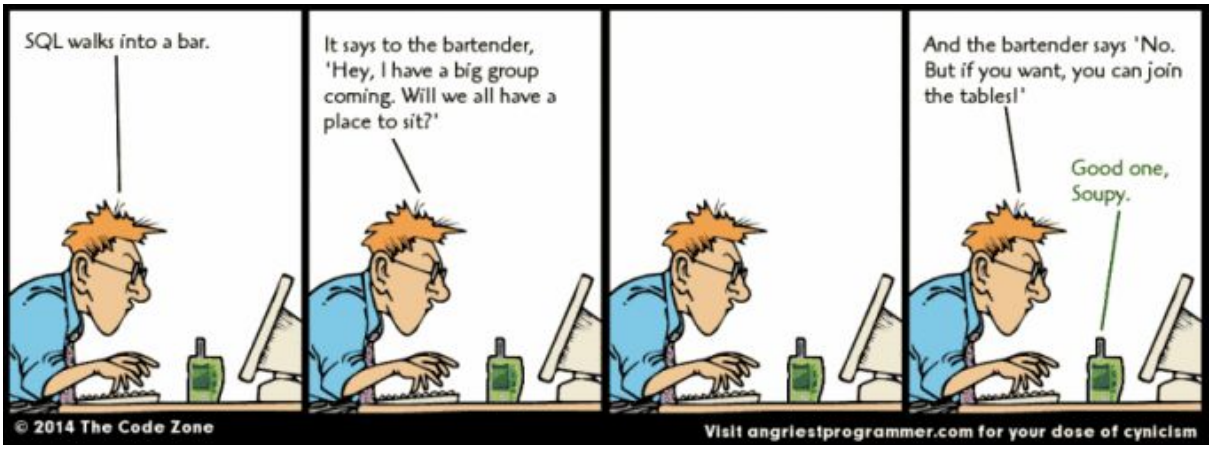
Outlab - 7
Relational Databases

Please refer to the general instructions and submission guidelines at the end of this document before submitting.



In this lab you are going to play with relational databases. A **Database** is a software **system** which is responsible for **data management**. Now, we know that data can be stored and managed in a multitude of ways. Relational databases approach it using a basic construct called **Table**. A Table is just what it sounds like, it has several columns, a header row and member rows. The concept itself is very intuitive.

In Relational Databases data is stored in the form of tables. Some basic operations for any table are

1. Create table
2. Insert Row(s)
3. Update Row(s)
4. Delete Row(s)
5. Select Row(s) etc..

To formalize the operations on tables a language called **SQL** (pronounced as SEQUEL) was introduced. Any operation that you want to perform on tables/database can be written in SQL. That SQL statement is written called a **Query**. A good reference for writing SQL queries can be found here.

A simple SQL query can have the form

```
SELECT column1, column2, ...
FROM table_name;
```

One important thing to note is that unlike programming languages like C/C++, Python, Java etc... there is no single standard for SQL. In that the syntax of the same SQL query might change with the actual relational database implementation you use. Some of the rdb implementations are MySQL, Postgres, sqlite. Today we will be using sqlite, so please stick to the SQL syntax that works for **sqlite**.

Once a query is written, it is passed to the database for execution. There are a couple of ways to do that. One might directly type it into an interactive db session or use a third party language to talk to the database and pass the query. Here we are going to use **Python3** as a mediator between us and sqlite.

Question 1: IPL Forever ! [60 points]

You have been provided with files **`player.csv`**, **`match.csv`**, **`player_match.csv`**, **`team.csv`** and **`ball_by_ball.csv`**.

- **`player.csv`** represents a table containing player id, name, dob and other skills
- **`match.csv`** contains match id, team1, team2 and other columns
- **`team.csv`** contains team_id and team_name
- **`player_match.csv`** contains playermatch_key, match_id, player_id and other columns (mapping between player and match tables)
- **`ball_by_ball.csv`** contains match_id, innings_no, over_id, ball_id and other columns.

All these files can be assumed to be present in the same directory as your scripts.
A schema for the same is provided in the **`IPL_schema.pdf`**. For each of the following tasks you will need to write multiple SQL statements.

Task 1 [2 points]

Create a python script **`create_tables.py`**, which when executed

1. creates an SQLite DB file named **`ipl.db`** and
2. creates 5 tables named
    a. **TEAM** with table structure (i.e. column names and types) as in **`team.csv`**
    b. **PLAYER** for **`player.csv`**
    c. **MATCH** for **`match.csv`**
    d. **PLAYER_MATCH** for **`player_match.csv`**
    e. **BALL_BY_BALL** for **`ball_by_ball.csv`**

You can assume that all numbers are of type INT, dob (date of birth) is of type TIMESTAMP and rest all are of the type TEXT. We will use the **`ipl.db`** created in this task for all further tasks in this lab.

In this question you need to add constraints like primary key and foreign key in the SQL query while creating the tables. **DO NOT** proceed ahead without doing this. In order to add constraints you need to know about these terms in database terminology

Attributes, Relation schema and Instance, Keys (super key, candidate key, primary key, foreign key), referencing and referenced

Task 2 [5 points]

Create a python script **insert.py**, which when executed populates the tables i.e.

1. Inserts all rows of **team.csv** in **TEAM** table
2. Insert all rows of **match.csv** in **MATCH** table
3. Similarly, insert all rows of **player.csv** in **PLAYER** table
4. Similarly, insert all rows of **player_match.csv** in **PLAYER_MATCH** table
5. And insert all rows of **ball_by_ball.csv** in **BALL_BY_BALL** table

You may want to confirm that the inserts are indeed successful by reading and printing the data in tables.

Grading will be done for this task by inspecting the state of ipl.db after running create_tables.py & insert.py

Getting bored with the same old stuff from inlab. Something interesting for you in the store.

Task 3 (Chinnaswamy or Wankhede ?) [5 points]

Create a python script **average_runs.py**, which when executed finds, for each match venue, the average number of runs scored per match (total of both teams) in the stadium. You can get the runs scored from the BALL_BY_BALL table. Output venue name and average runs per match, in descending order of average runs per match. Output format is as follows (note that this is the not the final output)

```
Punjab Cricket Association IS Bindra Stadium
Mohali,129.66666666666666
Maharashtra Cricket Association Stadium,120.0
Rajiv Gandhi International Stadium Uppal,118.5
```

Grading will be done for this task by matching the output of average_runs.py after running create_tables.py & insert.py

Task 4 (Orange and Purple Cap) [8 points]

Create a python script **max_runs.py** which when executed finds the player_id, name and total runs scored by a player in the tournament and sort them in descending order based on total runs. **Extra runs don't count towards the batsmen's total.** Print only **first 20 rows i.e. top 20 run scorers only.**

Similarly create a python script **max_wickets.py** which when executed finds the player_id, name and total wickets taken by a player in the tournament and sort them in descending order based on total wickets. Print only **first 20 rows i.e. top 20 wicket takers only.**

Note : The striker attribute in the ball_by_ball relation is the player who scored the runs.

The sample output format for max_runs.py is as follows (output will be similar for max_wickets.py, but the last column will contain total wickets instead of total runs) (this is not final answer)

```
139,LA Pomersbach,217

338,MC Juneja,197

396,KS Williamson,195
```

If two (or more) players have the same score, print them in lexicographic order of their name (ascending)
Grading will be done for this task by matching the output of max_runs.py/max_wickets.py after running create_tables.py & insert.py

Task 5 (HITMAN!) [10 points]

Create a python script **hitman.py** which when executed finds the player id, player name, the number of times the player has got 6 runs in a ball, the number of balls faced, and the fraction of 6s ordered in the descending order of the fraction of 6s hit by the player.

This can be used to find the player who is a most frequent six hitter. (Surprised to see the player top in the list ? :P)

Note:

1. The striker attribute in the ball_by_ball relation is the player who scored the runs.
2. Int divided by int gives an int, so make sure to multiply by 1.0 before division.

The output format is as follows : (not final answer)

```
57,RG Sharma,8,75,0.10666666666666667

221,KA Pollard,5,47,0.10638297872340426
```

```
46,RV Uthappa,4,38,0.10526315789473684
```

Grading will be done for this task by matching the output of hitman.py after running create_tables.py & insert.py
If two (or more) players have the same fraction, print them in lexicographic order of their name (ascending)

Task 6 (Managing the points table) [10 points]

Create a python script **make_points_table.py** which makes a new table named **POINTS_TABLE** which has first two columns similar to the current **TEAM** table (Note that in POINTS_TABLE you need not add any foreign key relationship and you can just copy TEAM table) but adds two more columns to this table namely points and nrr.

1. The points column is of type int and should be initialized with value 0.
2. The nrr column is of type decimal and should also be initialized with value 0.

Now using the **MATCH table** update this points table keeping in mind the following rules:

1. The team that wins a match gets 2 points and the losing team gets 0 points.
2. **If the match is a tie, ignore the match_winner and award 1 point to each team. nrr doesn't change in this case.**
3. If a match gets abandoned (given by NULL match winner in match.csv) then each team gets 1 point.
4. Net run rate is calculated as follows (this is not used in actual practice):

   If won by runs: NRR = Margin of Runs/20

   If won by wickets: NRR = Wickets Left/10

   Now subtract this NRR value from the losing team and add this NRR value to the winning team.

   Note that a win from null margin is equivalent to 0 margin of runs and 0 wickets left

The final output should print the POINTS_TABLE in sorted order based on points and then on basis of nrr in descending orders for both. The POINTS_TABLE can be stored without worrying about sorting it.

The output should be as follows (not final answer):

1,Kolkata Knight Riders,14,0.25

2,Mumbai Indians,12,0.11

And so on for all teams

Grading will be done for this task by inspecting the state of ipl.db after running create_tables.py & insert.py & make_points_table.py

Task 7 [10 points]

Now we introduce a bit of automation to python's SQLite API. Like C/C++ you can insert values into SQL statements instead of hard-coding them.

Create a python script **prep_stmt.py** which inserts a row into some table in the IPL database. The script accepts arguments from the command-line which are as follows.

The first argument is a number from 1 to 5 which corresponds to which table to insert.

1 --> team , 2 --> player , 3 --> match , 4 --> player_match , 5 --> ball_by_ball

The next n arguments correspond to the attributes of a row belonging to a particular table.

Ex : If you need to insert (14, "XXX") into the table "TEAM", then

1st argument - 1

2nd argument - 14

3rd argument - XXX

Take arguments via terminal as follows: python prep_stmt.py "1st arg" "2nd arg" "3rd arg". Enclose all arguments within quotes

Note that **you have to use prepared statements for this task.**

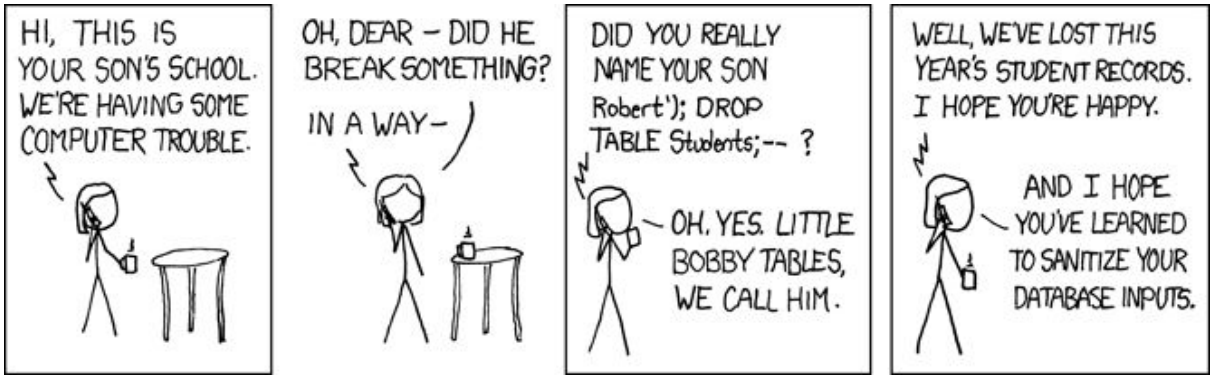
Grading will be done for this task by inspecting the state of ipl.db after running create_tables.py & insert.py & prep_stmt.py.

Task 8 (Answer a few questions) [10 points]

Give the answers to these questions as True or False, in a file called **True_or_False.txt** Each line should be an answer, and nothing else. So line x has the answer to question x. If you don't know the answer, guess, there's no negative marks. But have 10 lines in your file.

1. Primary key uniquely identifies a record in the table and it can't accept null values.
2. Foreign key is a field in the table that is the primary key in another table.
3. We can have only one Primary key but more than one foreign key(s) in a table.
4. (team_id) is a primary key in the TEAM table.
5. There are no foreign keys in the TEAM table
6. team1 is a foreign key in the MATCH table and it references the TEAM table.
7. player_id is a foreign key in the PLAYER_MATCH table and it references the PLAYER table.
8. There are a total of 3 foreign keys in the PLAYER_MATCH table.
9. (match_id, innings_no, over_id,ball_id) together form a primary key for BALL_BY_BALL table
10. There are a total of four foreign keys in the BALL_BY_BALL table.

SQL Injection & Prepared Statements (for Question 2)



When the user input is not validated, users can give undesired data as arguments and perform operations on the database in ways that are not authorized. This is where **prepared statements** kick in to prevent these attacks.

For example, consider the following well known example, named "users".

ID	name	Salary
1	a	s1
2	b	s2
3	c	s3
4	d	s4

Suppose that each user is allowed to check their salary by providing their name at the prompt.

The script most likely contains something like this.

```
statement = "SELECT * FROM users WHERE name = '" + name + "';"
```

The "name" in the above statement comes from the user.
Everything's fine as long as the user inputs one of **a, b, c** or **d** as the response.

Suppose the user inputs the following string -

```
' OR '1'='1
```

The resulting query is

```
SELECT * FROM users WHERE name = ‘’ OR ‘1’ = ‘1’;
```

which is essentially

```
SELECT * FROM users;
```

meaning the user has access to the salaries of all users in the table/database.

Exploiting this vulnerability is known as **SQL injection**

There are ways to let the computer know that the input is to be treated specially. To be precise, the input's only purpose is to give value to the row attribute (**name** in this case) and not to manipulate the behavior of the original SQL statement.

Read up about techniques on how to prevent SQL injection.

Question 2: To inject or not to inject [30 points]

Task 1: Create and insert (5 points)

This task is a bit different from the create and insert of the previous question, and hence one is at no loss if one starts this without having done question 1.
You're given 2 files - "University Schema" and "smallRelationsInsertFile.sql".
Your task is to create **create_and_insert.py** that reads these files one after the other, create tables, and insert the data in a db named **univ.db.** You don't need to think of what instructions to write to make the tables or insert data, you simply need to run the sql instructions in these two files. If any throws an error (for e.g. if a table that does not exist is dropped), ignore it. At the end of running these two files, the tables from University Schema should be generated in univ.db with the data from smallRelationsInsertFile.sql. You can assume these file names will be the same (and hardcode them)

Task 2: Querying (10 + 15 points)

This task has a twist. Create **query.py** that takes 4 **command line arguments.**
The 1st argument is an integer: 0 or 1. 0 means your code should be vulnerable to sql injection, 1 means it shouldn't be.
The 2nd argument is the table that is being queried. You don't need to check if the table given is in the schema, it will be. Has no spaces.
The 3rd argument is the column on which the constraint will be on. Again, this will be there in the table that is input. Has no spaces
The 4th argument will be the value on which the constraint is made. This will be put in double quotes, and may contain spaces. This may also be an injection attack.

Your output from this file should be **every row that is fetched from the select query.** (This will **usually be one where injection is disabled [unless condition matches multiple rows],** and may be many for injection vulnerable code). In addition to this, if the 4th argument is such an injection that it can lead to selecting all rows, all rows **should not be selected** when 1st argument is 1, and **should be selected** when 1st argument is 0. Assume use of single quotes in the 4th argument to inject like given in the above reading, and allow multiple commands to be executed (The idea of this exercise is to know exactly what can go wrong in a query injection)
Print in the same syntax as question 1 comma separated, no added space around commas. multiple rows should be allowed, just to reiterate.

```
python query.py 0 classroom building "' OR '1' = '1"
```

[should select everything in classroom because of the injection]

```
python query.py 1 student name "' OR '1' = '1"
```

[should not give in to the injection, should not print anything, since that named student is not present]

query.py will be run after running **create_and_insert.py.**

Example input output:
in: python3 query.py 0 student name "' OR '1' = '1"
out (actual answers):
00128,Zhang,Comp. Sci.,102
12345,Shankar,Comp. Sci.,32
19991,Brandt,History,80
23121,Chavez,Finance,110
44553,Peltier,Physics,56
45678,Levy,Physics,46
54321,Williams,Comp. Sci.,54
55739,Sanchez,Music,38
70557,Snow,Physics,0
76543,Brown,Comp. Sci.,58
76653,Aoi,Elec. Eng.,60
98765,Bourikas,Elec. Eng.,98
98988,Tanaka,Biology,120

in: python3 query.py 0 student name "' OR '1' = '1' ; DROP TABLE student ; --"
out:

[output is empty, the drop doesn't work because of how sqlite3 works on python. the select doesn't work either because your query fails. make sure you catch the exception thrown. this ofc won't work if 1st argument is 1 either.]

in: python3 query.py 1 time_slot day "M"
out:
A,M,8,0,8,50
B,M,9,0,9,50
C,M,11,0,11,50
D,M,13,0,13,50
G,M,16,0,16,5

in: python3 query.py 1 time_slot day "' OR '1' = '1"
out:

[empty output again, coz not-vulnerable is on]

Submission Instructions

After creating your directory, package it into a tarball **<roll-no1>-<roll-no2>-outlab7.tar.gz** in ascending order. Submit once only per team from the moodle account of the smallest roll number.

We will untar your submissions using

$ tar xvf <roll-no1>-<roll-no2>-outlab7.tar.gz

Make sure that when the above is executed, the resulting <team_name>-outlab7/ directory has the correct directory structure.

The directory structure should be as follows (nothing more nothing less, order not enforced, obviously).

<roll-no1>-<roll-no2>-outlab7/

```
├── references.txt
├── ipl_q1
│     ├── create_tables.py
│     ├── True_or_False.txt
│     ├── insert.py
│     ├── average_runs.py
│     ├── max_wickets.py
│     ├── max_runs.py
│     ├── hitman.py
│     ├── make_points_table.py
│     ├── ~~win_chance.py~~
│     ├── prep_stmt.py
│     └── ~~sql_injection.py~~
└── univ_q2
      ├── create_and_insert.py
      └── query.py
```