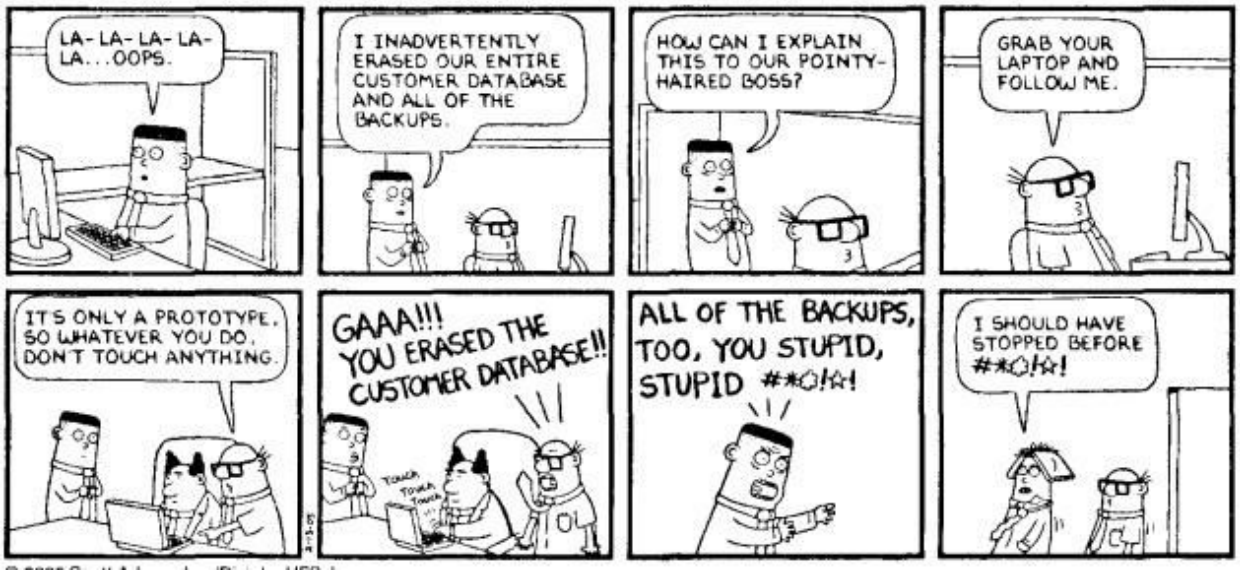
Outlab 10 : Relational Databases

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 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 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 a 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 the sqlite.

IPL Forever ! [100 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.

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 [8 points]

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

1. creates a 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

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

Task 2 [10 points]

Write **true** or **false** (not True, TRUE, true. etc...) for each of the following questions, one answer per line in that order in **trueorfalse.txt**

a. Primary key uniquely identify a record in the table and it can't accept null values.
b. Foreign key is a field in the table that is primary key in another table.
c. We can have only one Primary key but more than one foreign keys in a table.
d. (team_id) is a primary key in the TEAM table.
e. There are no foreign keys in the TEAM table
f. team1 is a foreign key in MATCH table and it references TEAM table.
g. player_id is a foreign key in PLAYER_MATCH table and it references PLAYER table.
h. There are a total of 3 foreign keys in PLAYER_MATCH table.
i. (match_id, innings_no, over_id,ball_id) together form a primary key for BALL_BY_BALL table
j. There are a total of four foreign keys in BALL_BY_BALL table.

Task 3 [7 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 4 [10 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 5 [20 points]

Create a python script **average_balls.py** which when executed finds the player_id, name and average balls per match faced by the player. Print only the **rows with a sparse rank <= 10** (you may get more than 10 in case of ties).

Note that a batsman faced a ball if there is an entry in ball_by_ball with that player as the striker.

The sample output format is as follows (this is not final answer)

```
139,LA Pomersbach,17
```

```
338,MC Juneja,17

396,KS Williamson,15
```

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

Task 6 [20 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

Task 6 [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

Give **one argument per line** so that space separated strings are not an issue while reading.

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

SQL Injection & Prepared Statements

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** kicks 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 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.

Task 7 [15 points]

This is similar to the previous task, but this time you delete rows instead of inserting them.

Create a python script **sql_injection.py** whose arguments are as follows.

1st argument - A number from 1 to 3 (from the above task)

2nd argument - 0 or 1

0 --- your code must be vulnerable to sql injection

1 --- your code must not be vulnerable to sql injection

1st argument	Table	3rd argument (value of)
1	team	team_name
2	player	player_name
3	match	match_id

i.e., when 1st argument is 2, then you must delete the row(s) in the player table with player_name equal to 3rd argument.

For the 1st argument, forget values **4** and **5** for simplicity.

For example, you must be able to delete all the rows in a table if the 2nd argument = 0 using some set of arguments but not when it is 1.

Give **one argument per line** for this task as well.

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

General Instructions

- Make sure you know what you write, you might be asked to explain your code at a later point in time
- The submission will be graded automatically, so stick to the naming conventions strictly
- Do not submit your **ipl.db**
- The deadline for this lab is **Monday, 15th October, 23:55.**

Submission Instructions

After creating your directory, package it into a tarball **`<rollno1>-<rollno2>-<rollno3>-outlab10.tar.gz`** in ascending order. Submit once only per team from the moodle account of smallest roll number.
The directory structure should be as follows (nothing more nothing less)

```
<rollno1>-<rollno2>-<rollno3>-outlab10
      ├── create_tables.py
      ├── true or false.txt
      ├── insert.py
      ├── average_runs.py
      ├── average_balls.py
      ├── hitman.py
      ├── prep_stmt.py
      └── sql_injection.py
```