W4111_F21_MIDTERM

November 5, 2021

COMS W4111-002, Fall 21: Take Home Midterm

Chandan Suri CS4090

1 Overview

1.1 Instructions

Due Date: Sunday, October 31, 2021 at 11:59pm You have one week to complete the take home portion of the midterm. All of the work must be your own, you may not work in groups or teams. You may use outside sources so long as you cite them and provide links.

Points will be taken off for any answers that are extremely verbose. Try to stay between 2-3 sentences for definitions and 5 sentences for longer questions.

You may post **privately** on Ed or attend OH for clairification questions. TAs will not be providing hints.

1.2 Environment Setup

- **Note:** You will need to change the MySQL userID and password in some of the cells below to match your configuration.

[2]:
```
%load_ext sql
```

[3]:
```
%sql mysql+pymysql://root:dbuserdbuser@localhost/lahmansbaseballdb
```

[19]:
```
from sqlalchemy import create_engine
```

[20]:
```
sql_engine = create_engine("mysql+pymysql://root:dbuserdbuser@localhost")
```

[21]:
```
import pandas as pd
```

2 Written Questions

2.1 W1

Provide a short (two or three sentence) definition/description of the following terms. Provide an example from the Lahman's Baseball DB for each.

Notes: - Lahman's Baseball DB uses auto-increment ID columns for primary keys. If ignoring the ID column makes answering the question easier, you can assume that the column and current primary keys are not defined. - Some of these concepts do not have a single, precise, agreed definition. You may find slight differences in your research. Focus on the concept and grading will be flexible.

1. Super Key
2. Candidate Key
3. Primary Key
4. Alternate Key
5. Unique Key
6. Natural Key
7. Surrogate Key
8. Substitute Key
9. Foreign Key
10. External Key

Answer: 1. Super Key: Super Key is a single key or group of multiple keys that can uniquely identify tuples in a table. It can contain multiple attributes that might not be able to independently identify tuples in a table, but when grouped with certain keys, they can identify tuples uniquely. Furthermore, some attributes in the key might be unnecessary in order to uniquely identify tuples. Example: (playerID, yearID, stint, POS, teamID) in 'fielding' table is a Super Key as we can uniquely identify rows with this and actually without teamID, we can also uniquely identify the rows. 2. Candidate Key: This is a single key or group of multiple keys that uniquely identify rows in a table. A Candidate key is a subset of Super Keys and is devoid of any unnecessary attributes that are not important for uniquely identifying tuples. The value for Candidate key is unique and non-null for all the tuples.Moreover, every table has to have at least one Candidate Key. But, there can be more than one Candidate Key too. Example: (schoolID) and (name_full, city) in 'schools' table are Candidate keys as we can unqiuely identify rows with it. 3. Primary Key: It is the Candidate key selected by the database administrator to uniquely identify tuples in a table. Out of all the Candidate Keys possible for a table, there will be only one key that will be used to retrieve unique tuples from the table. This Candidate key is called the Primary key for that table. Primary key can constitute a single attribute or a group of attributes. Furthermore, Primary Key should be unique and should contain non-null attributes. Lastly, there can be only one Primary key assigned for a table. Example: (schoolID) in 'schools' table is a primary key. 4. Alternate Key: Alternate Keys are those candidate keys that are not the primary key. As there can be only one Primary key per table, rest of the Candidate keys are known as the Alternate Keys. They can also uniquely identify tuples in a table, but the database administrator chose another one as the Primary key. Example: (name_full, city) in 'schools' table is an alternate key (unqiuely identify with this) as the PK for this is 'schoolID' (another Candidate). 5. Unique Key: This is a unique identifier for the tuples of a table when primary key is not present. Multiple unique keys can be present in a table, and can contain NULL values too. They can also be used as foreign key for another table. Furthermore, selection using unique key creates a non-clustered index. Example: (yearID, teamID, inseason) in 'managers' table is a unique key. 6. Natural Key: A natural key is a column or set of columns that already exist in the table (they are attributes of the entity within the data model) and uniquely identify a record in a table. Since these columns are attributes of the entity, they obviously have significant business meaning pertaining to the use case. Example: 'playerID' in 'people' table is a natural key and is significant to the business logic that contains this table for all the players in the database. 7. Surrogate Key: This is a system generated value

with no business meaning pertaining to the use case at hand. It is also used to uniquely identify a record in the table. They key itself could be made up of multiple columns. For instance, simply a column with some sequential integers could be deemed as a surrogate key. Example: 'ID' column in 'halloffame' table is a surrogate key which is used to uniquely identify but has not pertaining business logic and is auto-incremented (automatically generated for each record). 8. Substitute Key: This is a single, small attribute that has at least some descriptive value (like abbreviations). These are also created for the convenience of a database designer. However, they can still be a nuisance for the database users. Unlike surrogate keys, the values of this key are widely used and are understood by the people. (Like Airport codes!) Example: 'lgID' in 'leagues' table is a substitute key as these are abbreviations for different leagues. 9. Foreign Key: It is an attribute which is a primary key in it's parent table, but is included as an attribute in the host table. It generates a relationship between the parent and the host table. It also makes it easier to update the database when required and keeps a check on the associated primary key values and thus, the relationship between the tables. Example: 'playerID' in 'appearances' table is a foreign key that references 'playerID' from 'people' table. 10. External Key: Sometimes, we might find that an entity already has an attribute that appears to be a surrogate or a substitute key, but has been defined by someone else - usually a standards setting organization or a government agency. We call this kind of attribute as an external key. That means that in the external organization's database, there is a candidate key for the same, whether or not we have access to it or include the values in our own database. Like other descriptive attributes, an external key might or might not become a part of the candidate key in our database. Example: 'votedBy' column in 'halloffame' table is an external key as those abbreviations could be in an external database and would mean something externally (to the lahmans DB). Source Credits:

https://www.analyticsvidhya.com/blog/2020/07/difference-between-sql-keys-primary-key-super-key-candidate-key-foreign-key/

https://www.tutorialspoint.com/primary-key-vs-unique-key

https://www.mssqltips.com/sqlservertip/5431/surrogate-key-vs-natural-key-differences-and-when-to-use-in-sql-server/

https://web.csulb.edu/colleges/coe/cecs/dbdesign/dbdesign.php?page=keys.php

Prof. Fergussons' Slides

3 W2

- Define the concept of *immutable* column and key.
- Why do some sources recommend that a primary key should be immutable?
- How would to implement immutability for a primary key in a table?

Answer Immutable Column and Key: Immutable column is a column in a table from which the data cannot be deleted or modified after it has been added once. This kind of restriction on updation or deletion of some part of data is possible in some databases but not all. Immutable key is a key (or a primary key) column/s that cannot be modified or deleted once added. This is mostly applicable to the primary keys as in some cases, primary key shouldn't be updated or modified as it is used to uniquely identify a record in the table and doing so can change that mapping which might not be useful (could be even detrimental) in some of the cases according to the business logic. Why a

primary key should be immutable according to some sources: A primary key is used to uniquely identify a record/tuple in the table. Sometimes, a business logic wouldn't allow the primary keys to be immutable as it might lose it's purpose and could hinder with the business operations. For example, for a company which stores employee information, employee ID could be the primary key, which when changed, could change a lot of other settings of the user according to the user's role in the company. It could also hinder some of the business operations! Thus, when it's needed, the primary can and should be made immutable, both to support the business logic for which it is being used and also to prevent accidental change of that primary key which could hinder the business operations. Also, from performance point of view:

You will need to update all foreign keys that reference the updated key. A single update can lead to the update of potentially lots of tables/rows.

If the foreign keys are unindexed you will have to maintain a lock on the children table to ensure integrity.

If your foreign keys are indexed (as they should be), the update will lead to the update of the index (delete+insert in the index structure), this is generally more expensive than the actual update of the base table.

In ORGANIZATION INDEX tables (in other RDBMS, see clustered primary key), the rows are physically sorted by the primary key. A logical update will result in a physical delete+insert (more expensive).

```
<br>
<br>
```

Implementation for immutability of the primary key in a table: This could be done in many ways:

One way would be to use a trigger in order to rollback the value in the primary key whenever someone updates anything in the primary key column/s.

While table creation, add 'ON UPDATE RESTRICT' on the primary key constraint in the DDL statement. According to me, this would be a much better way than the first one to create an immutable primary key.

```
<br><br>
Source Credits:
<br>
<ol>
    <li>https://dba.stackexchange.com/questions/8360/making-a-column-immutable-in-mysql</li>
</ol>
```

4 W3

Views are a powerful concept in relational database management systems. List and briefly explain 3 benefits of/reasons for creating a view.

Answer The benefits of creating a view:

The views offer a added layer of security. If we want our application to not access the base tables directly, we could use a view to do so. If you want your application not to access the base tables, you can create a view that refers to the Table you want to use. Suppose your application is using

the customer table. You do not want to show the customer's SSN details then you can create a view that populates the customer's general information (Name, Address, contact details) grant access to the View only.

The views also offer consistency. Suppose for a use case, you only want to show a subset of data and want to compute a column based on the other columns. One way would be to do that by creating a new table with all the data and calculate based on the columns in the data. But this is not a consistent way to do so. As when you update the base table in the future, you need to do the recomputations for the table you created for the base table as that doesn't change automatically. This could lead to stale data and inconsistencies if not done. However, with a view, that's not the case! As a view is not physically stored, it always does the computations on the go from the base table whenever one tries to access the view. Thus, it always takes into consideration the current state of the table or tables from which the view is created. This is one of the most significant advantages of creating a view.

The view also helps to simplify the complex business logic written in some complex SQL Queries. Instead of executing the same complex query multiple times, you can create a view from it. This view can be referenced simply by using a simple SELECT query. Also, as the views are not physically stored, the business logic needed can be achieved without filling up the memory with this temporary data that might be needed as some parts of the application that is in turn driven by the business logic.

```
<br>
Source Credits:<br>
<ol>
    <li>https://www.sqlshack.com/learn-mysql-the-basics-of-mysql-views/</li>
    <li>Prof. Fergussons' Slides</li>
</ol>
```

5 W4

Briefly explain the concepts of *procedural* language and *declarative* language. SQL is a declarative language. What are some advantages of a declarative language over a procedural language?

Answer Procedural Language: In this language, you define the whole process and provides the steps on "how" to achieve the goal or the outcome needed. The whole definition of process that needs to be done, needs to be specified along with all the steps in the procedure. Basically, we describe tha algorithm step-by-step, at various degrees of abstraction. Examples include: C, Java, Python, etc. Declarative Language: In this, we just set the command or order, and let it be on the system how to complete the order. Basically, we just tell the system what do we need (outcomes) without blatantly specifying all the steps needed in order to achieve that outcome. Basically, we get the outcome via a "black box" while we just specify to the "black box", what we need as an outcome. Examples include: SQL, Scala, etc. Advantages of a Declarative Language over a procedural language:

Short and Efficient Code: This is one of the biggest strengths of Declarative languages that they offer a way to do things such that the problems can be solved more briefly and succinctly than procedural or imperative languages.

Easy optimization as implementation is controlled by an algorithm.

Maintenance possible independent of application development: Because implementation is clearly delineated from the system using an algorithm, maintenance can be performed independently of application development. Interruptions of day-to-day operations are reduced to a minimum.

Can be implemented using methods not yet known at the time of programming.

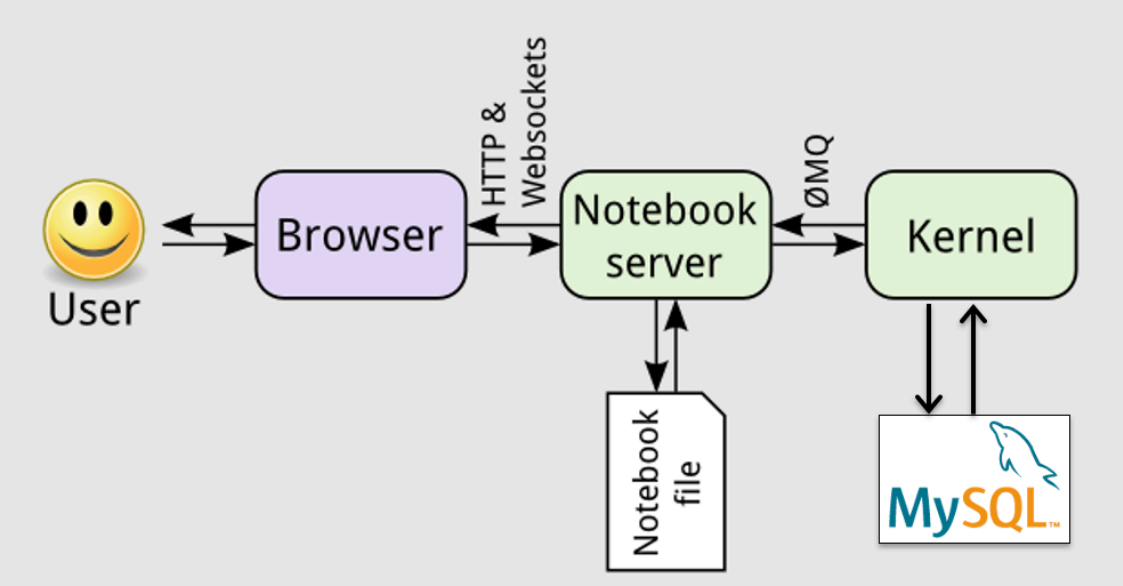
```
<br>
Source Credits:<br>
<ol>
    <li>https://stackoverflow.com/questions/1619834/what-is-the-difference-between-declarative
    <li>https://theburningmonk.com/2010/01/imperative-vs-declarative-languages/</li>
    <li>https://www.ionos.com/digitalguide/websites/web-development/declarative-programming/</l
</ol>
```

6 W5

The following diagram is a simple representation of the architecture of a Jupyter notebook using MySQL. Is this a two-tier architecture or a three-tier architecture? Explain your answer briefly.

[28]:
```
from IPython.display import Image
Image('./jupyter-architecture.png')
```

[28]:



Answer The representation shown above is clearly a three-tier architecture for the following reasons and/or capabilities:

As we can see above, the Client-Application tier has been divided into two. In other words, it's a modular client-server architecture that consists of a presentation (client - Browser) tier and an application tier (Notebook server & Notebook File) and a Data Tier (MySQL that interacts with the application tier through a kernel).

Presentation Tier: This is the topmost layer of this application. This interface layer has the job of translating the tasks and information to something the user can understand. It also takes in a user request, send it to the application layer (in this case the Notebook server) and results are sent back to the user. Here the presentation layer is the Browser that acts as an interface between the user and the application tier here.

Application Tier: This layers essentially contains the business logic and operations for performing tasks like processing commands, validations, logical evaluations, decision making, and database interactions. Here this tier consists of the Notebook server that has some notebook file/s that contains the logic and interacts with the presentation layer through websockets and with the data layer through the kernel.

Data Tier: This is basically the backend layer. It is similar to the database server on a two-tier architecture. This consists of the MySQL that interacts with the application layer via the kernel.

All this precisely tells us that the representation show above is basically a three-tier architecture rather than a two-tier architecture because the Client Application layer has been divided into 2 separate tiers, namely the Presentation and the Application Tiers.

```
<br>
Source Credits:<br>
<ol>
    <li>https://medium.com/@fleviankanaiza/two-tier-three-tier-architecture-8b02536d3482</li>
</ol>
```

7 W6

What is the difference between the database schema and the database instance? Use the following conventions of the relational model for documenting a relation schema to answer the question if: 1. *country_code* is the country code for the phone number, e.g. +1 2. *main_number* is the main number, e.g. 212-555-1212. 3. *extension* is the extension, e.g. x1002 4. *phone_type* is an enum, e.g. *work, home, mobile, others.*

$$PhoneNumber(\underline{country\_code}, \underline{main\_number}, \underline{extension}, phone\_type) \tag{1}$$

Answer The main difference between the database schema and the database instance is that the schema is the structural view of the database, while the instance is the actual data stored in the database at a particular point in time. Basically, the schema tells us about what all the type of data can be stored in the database and the relationship between different tables or relations in the database. It doesn't show us the actual data stored in the database. However, an instance would include the actual data stored according to the structure that is shown in the schema. For instance, according to the relational model shown above, the database schema would include a table named "Phone Number". In that table, the columns would be defined by the column names as "Country_Code", "Main_Number", "Extension", and "Phone_Type". The types defined for those columns would be "VARCHAR(5)" (allowing for longer country codes), "VARCHAR(12)" (allowing to include the dashes with the numerals), "VARCHAR(5)", and "ENUM(Work, Home, Mobile, Others)" respectively. The Primary Key constraint will be added on the columns "Country_Code", "Main_Number", and "Extension". All these details are a part of the database schema. On the other hand, a database instance would include a tuple in that database as "+1", "212-555-1212",

"x1002", and "Work". This represents one of the rows in that database at a point in time. This is the actual data stored in the database and thus, is the database instance. Source Credits: https://pediaa.com/what-is-the-difference-between-schema-and-instance/

8 W7

Briefly explain the differences between:

- Database stored procedure
- Database function
- Database trigger

Answer

Execution: Database Stored Procedure: We can execute the stored procedure whenever required using the statement such as "CALL procedure_name(columns)". This can also be done as a part of another procedure. Database Function: We can't execute a function separatelt as function is not in a pre-compiled form. This is generally called inside a SQL query or another function. Database Trigger: Trigger can be executed automatically designated to a specified action on a table like update, delete or insert.

Calling: Database Stored Procedure: Stored procedures can't be called from a function because functions can be called from a select statement. But you can call a stored procedure from a trigger. Also, the stored procedure can be called from another procedure or embedded SQL using the call statement. Database Function: Functions can be called from a function, a stored procedure, trigger or inside SQL queries too. Database Trigger: Triggers can't be called from stored procedure or a function. As they are delegated with an event like updation, deletion or insertion, it can't be called by itself as it's called automatically when an event occurs.

Parameters: Database Stored Procedure: These can accept any type of parameter. They keywords in and out are the parameters used with all the attributes of a function, that are expected to have values assigned to them and parameters whose values are set in the procedure in order to give some results (attribute with the out keyword). Database Function: These can also accept any kind of parameter but in and out keywords are not used with the parameters passed to the function. Simply the attributes are passed to the function without any keywords like in and out because those aren't needed for a function. Database Trigger: We can't pass any parameter to a trigger. Although for referencing the attributes before and after an update, "referencing old row as" and "referencing new row as" can be used.

Returns: Database Stored Procedure: The values aren't returned for the procedures. If any output from the procedure is needed, then that variable is declared and passed as an out parameter to the procedure. After the procedure is executed, the out parameter holds the values that were needed out of the procedure execution. Database Function: Functions must return a value. Values can be returned from a function by simply using the return statement inside the function. We also need the type of the return value to be in the function declaration. Database Trigger: Triggers does not return any value on execution. Rather it is used to do something on case of an event occurs to which the trigger has been delegated. You can do some operations on the rows involved in any kind of updation like insert, delete or update when that particular event occurs for which the trigger was delegated or associated.


```
Source Credits:<br>
<ol>
    <li>https://www.c-sharpcorner.com/blogs/about-store-proc-function-trigger-in-brif</li>
    <li>https://stackoverflow.com/questions/3744209/mysql-stored-procedure-vs-function-which-w
    <li>https://www.tutorialspoint.com/difference-between-stored-procedure-and-triggers-in-sql
    <li>Prof. Fergusson's Slides</li>
</ol>
```

9 W8

Briefly explain:

- Natural join
- Equi-join
- Theta join
- Self-join

Answer

Natural Join: A natural join is a type of join operation that creates an implicit join by combining tables based on columns with the same name and data type. It does not utilize any of the comparison operators. For this join, there should be at least one common attribute between the two relations that needs to be joined. Basically, it implicitly performs selection forming equality on those attributes which appear in both the relation and eliminates the duplicate attributes. The syntax is as: SELECT [column_names|*] FROM table_name_1 NATURAL JOIN table_name_2;

Equi-Join: This is a basic join with a WHERE clause that contains a condition specifying that the value in one column in the first table must be equal to the value of a corresponding column in the second table. The syntax is as: SELECT [column_names|*] FROM table_name_1, table_name_2 WHERE table_name_1.col_1 = table_name_2.col_2;

Theta Join: This allows us to join two tables based on a condition represented by theta. This works for all comparison operators (not just equal to). The tables are joined based on a condition stating that one column in the first table is greater than, less than, etc the column in the second table. The syntax is as (example): SELECT [column_names|*] FROM table_name_1, table_name_2 WHERE table_name_1.col_1 >= table_name_2.col_2;

Self-Join: This is a regular join, such that the table is joined with itself according to a condition (can be multiple conditions too). For example, the following SQL statement matches customers that are from the same city: SELECT A.CustomerName AS CustomerName1, B.CustomerName AS CustomerName2, A.City FROM Customers A, Customers B WHERE A.CustomerID != B.CustomerID AND A.City = B.City ORDER BY A.City;

```
<br><br>
Source Credits:<br>
<ol>
    <li>https://www.javatpoint.com/mysql-natural-join</li>
    <li>https://www.dummies.com/programming/sql/how-to-use-an-equi-join-in-sql/</li>
    <li>https://www.guru99.com/joins-sql-left-right.html</li>
    <li>https://www.w3schools.com/sql/sql_join_self.asp</li>
</ol>
```

10 W9

Briefly explain the difference between a *unique (key) constraint* and a *primary key constraint?*

Answer The main differences are:

You can create multiple unique constraints in a table. On the other hand, you can only have one primary key constraint in a table.

Unique Key constraint allows for one NULL value (such that at least it maintains the uniqueness of the values in a column or a group of columns under that unique key constraint), but the PRIMARY KEY constraint does not allow any NULL value. In fact, the column/s under the primary key constraint are non-nullable.

Primary key constraint creates clustered index automatically but the unique key constraint does not.

Source Credits:
https://stackoverflow.com/questions/1308593/what-is-the-difference-between-a-primary-key-and-a

11 W10

Briefly explan *domain constraint* and give an example.

Answer It is an attribute that specifies all the possible values that the attribute can hold like integer, character, date, etc. It defines the domain or the set of values for an attribute and ensures that the value taken by an attribute must be atomic from it's domain. There are two types of constraints that come under this category:

Not Null: This restricts a column to not accept NULL values which means that it only restricts a field to contain a value which means that you cannot insert a new record or update a record without adding a value to the field. For instance, in an employee database, every employee must have a id and name associated with them. CREATE TABLE employee (employee_id VARCHAR(25) NOT NULL, employee_name VARCHAR(50) NOT NULL, salary BIGINT);

Check: This defines a condition that each row for the attribute/s must satisfy which means that it restricts the value of a column between ranges or in some set of possible values. It is just like a condition or filter checking before saving data into a column. It ensures that when a tuple is inserted inside a relation, it must satisfy the predicate given in the check clause. Suppose, we need to check if an id is greater than zero and that the name is not an empty string. Create TABLE employee (employee_id VARCHAR(25) NOT NULL CHECK(employee_id > 0), employee_name varchar(30) NOT NULL CHECK(employee_name != ''), salary BIGINT);

Source Credits:
https://www.geeksforgeeks.org/domain-constraints-in-dbms/

12 Entity Relationship Model

Question

- This question tests transforming a high-level description of a data model into a more concrete logical ER diagram. You will produce a logical ER-diagram using Lucidchart, or a similar tool. You should use Crow's Foot notation and conventions we have used in lectures.
- The data model is a simple representation of a university.
- The model has the following entity types.
  - School:
    - School code, e.g. "SEAS," "GSAS," "LAW," … …
    - School name, e.g. "School of Engineering and Applied Science."
  - Department:
    - Department code, e.g. "COMS," "MATH," "ECON," … …
    - Department name, e.g. "Department of Computer Science."
  - Faculty:
    - UNI
    - last_name
    - first_name
    - email
    - title, e.g. "Professor," "Adjunct Professor," … …
  - Student:
    - UNI
    - last_name
    - first_name
    - email
  - Course:
    - Course number is a composite key, e.g. "COMSW4111" is
      - Dept. code "COMS"
      - Faculty code "W"
      - Course number "4111"
    - Course title
    - Course description
  - Section:
    - Call number
    - Course number
    - Year
    - Semester
    - Section
- A Faculty has complex states and relationships.
  - A Faculty can have a role relative to a Department, e.g. Chair.
  - Roles are a small set of possible values.
  - Roles change over time. The data model must support the ability to handle current roles and previous roles.
  - A Faculty can have a role in a department at most once.
- A Student has a relationship to Section.
  - The possible roles are `(Enrolled, Waitlist, Dropped, TA)`
  - The student has a current role, and there can be only one current row.

  - The data model must support the ability to handle roles changing over time and retaining information about prior roles.

- A Faculty *may* teach a Section. All sections have exactly one Faculty.
- The relationship between Department and School is many-to-many. Each Department is in at least one school and each school has at least one department.

Notes: 1. There is no single correct answer to this question. You will have to make some design decisions and assumptions. You should document your decisions and assumptions. 2. You do not have to worry about **isA** relationships. 3. You do not have to document or worry about attribute types. 4. The ER diagram must be implementable in the relational/SQL model.

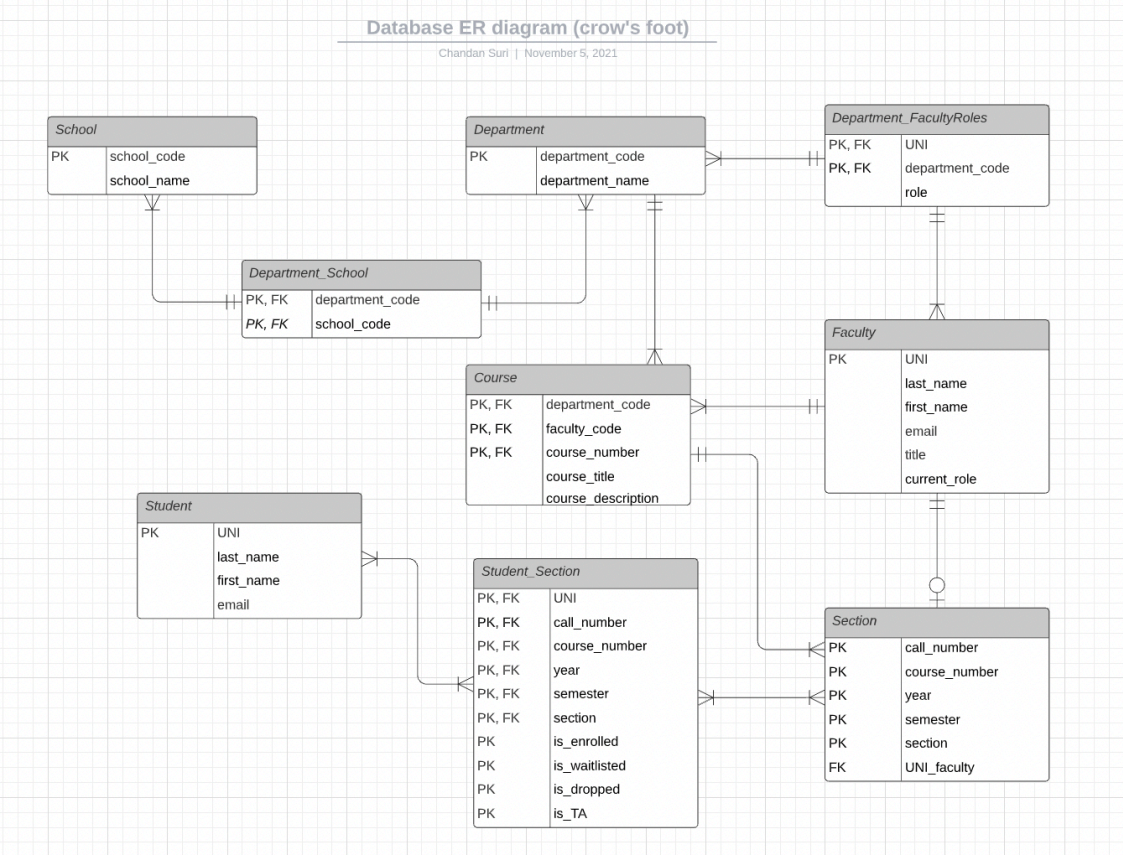
Answer

Assumptions, Decsions and Notes:

1. Call Number in Section: I am assuming that the call number won't necessarily change every year or semester. Also, if those are randoly generated. At some point in time, it could be the same as some other course in the past. So, I am taking all the attrbutes in section to be the Primary Key.
2. As we had to take multiple roles into consideration for Student and Section relationship, I drew an associativity between them to store all the roles in the past. (Many to Many relationship)
3. As there should have been a separate row for the current role in the Student-Section associativity, I have divided the roles into multiple columns that will tell us the current role between a student and a section as well as the history clearly. I am also taking an assumption there that these possible roles are static and generally won't change for the database. (from schema perspective)
4. Assumption on Student-Section: Additional assumption for relationship between these is the many-to-many relationship between these. A student can be in multiple sections at a point in time and there can be multiple students in a section. Also, there will be at least one student in a section and a student would have selected at least one section.
5. Relationship between Department and Course: A department can have multiple courses but a course will only be a part of some specific department.
6. Relationship between Course and Faculty: A faculty can have multiple courses running at a time for which they are the instructors but a course will only have one faculty assigned to it.
7. Relationship between Course and Section: A Course can have multiple sections under it but a section will only be for a specific course.

[1]:
```
from IPython.display import Image
Image('er_diagram_univ_model.png')
```

[1]:



13 Relational Algebra

13.1 R1

Use the RelaX Calculator and the Silberschatz - UniversityDB for this question.

Two time slots X and Y obviously overlap if: 1. They are not the same time slot, i.e. do not have the same *time_slot_id*. 2. They have at least one lecture on *the same day,* the start hour for X is before the start hour for Y, and the end hour for X is after the start hour for Y. 3. To make the question easier, you do not need to consider minutes in computing overlap but must show minutes in the result.

Write the relational algebra expression that identifies obviously overlapping time slots, and only lists overlapping pairs of time slots once.

Your output must match the answer below.

[29]:
```
from IPython.display import Image
Image('./relational_result.png')
```

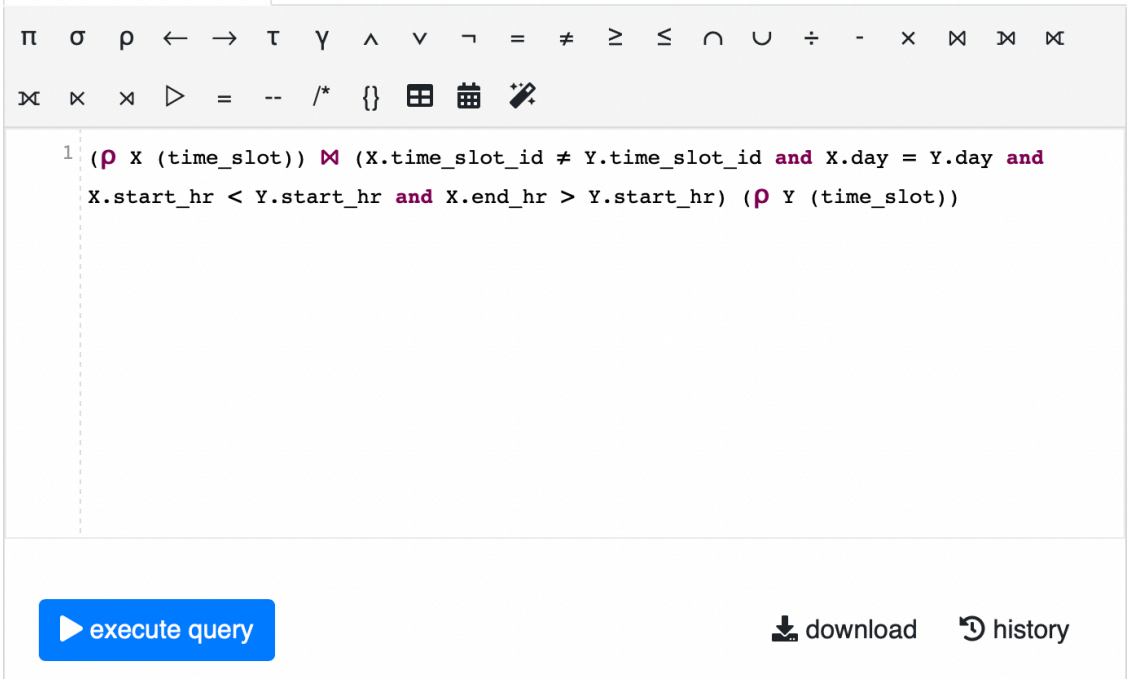
[29]:

X.time_slot_id	X.day	X.start_hr	X.start_min	X.end_hr	X.end_min	Y.time_slot_id	Y.day	Y.start_hr	Y.start_min	Y.end_hr	Y.end_min
'H'	'W'	10	0	12	30	'C'	'W'	11	0	11	50

Answer

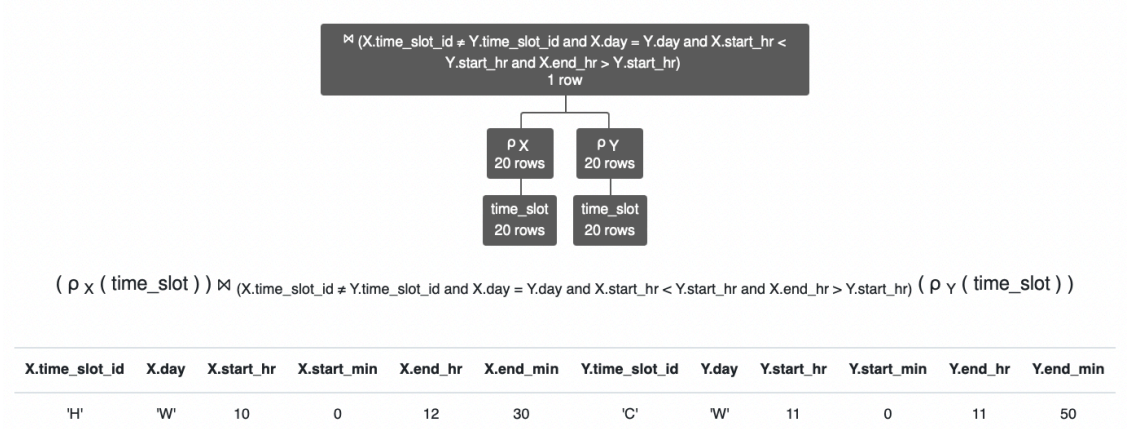
[3]:
```
from IPython.display import Image
Image('rel_query_1.png')
```

[3]:



[4]:
```
Image('rel_query_1_solution.png')
```

[4]:



X.time_slot_id	X.day	X.start_hr	X.start_min	X.end_hr	X.end_min	Y.time_slot_id	Y.day	Y.start_hr	Y.start_min	Y.end_hr	Y.end_min
'H'	'W'	10	0	12	30	'C'	'W'	11	0	11	50

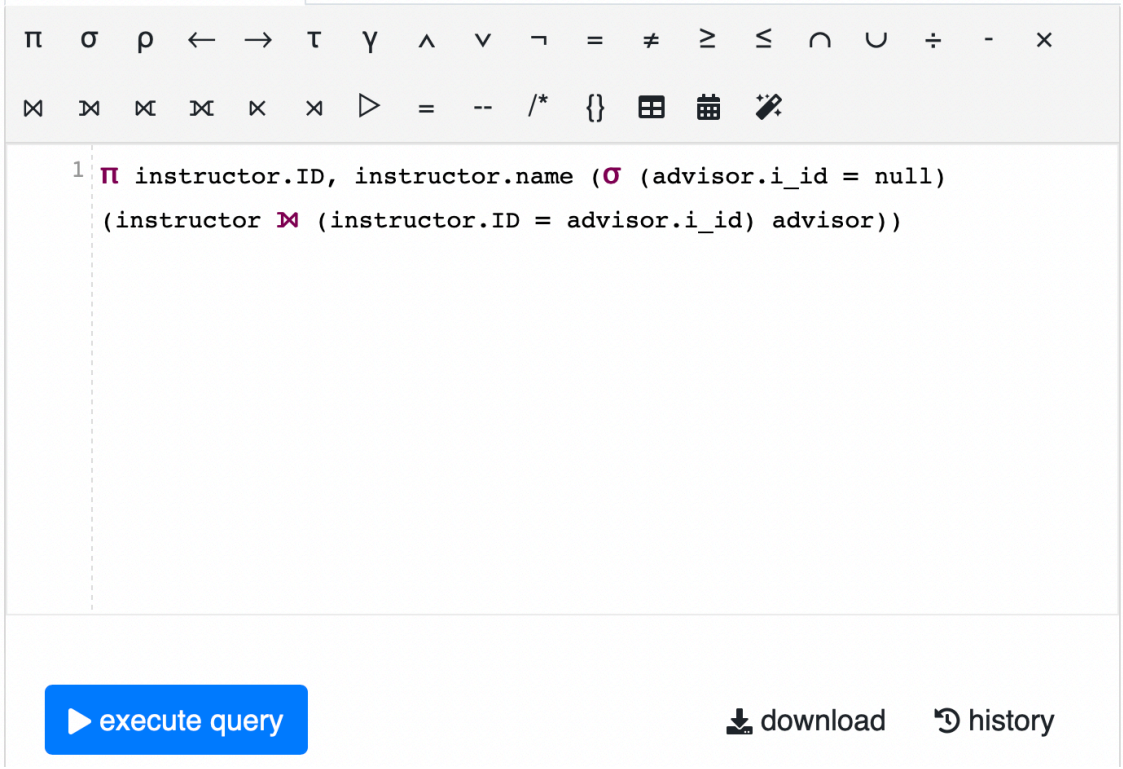
13.2 R2

1. You **may not** use the subtraction operator - to write this query.
2. Produce a relation that:
   - Has column names `instructor_ID, instructor_name.`
   - Contains the `ID` and `name` of instructors who do not advise any students.

Answer

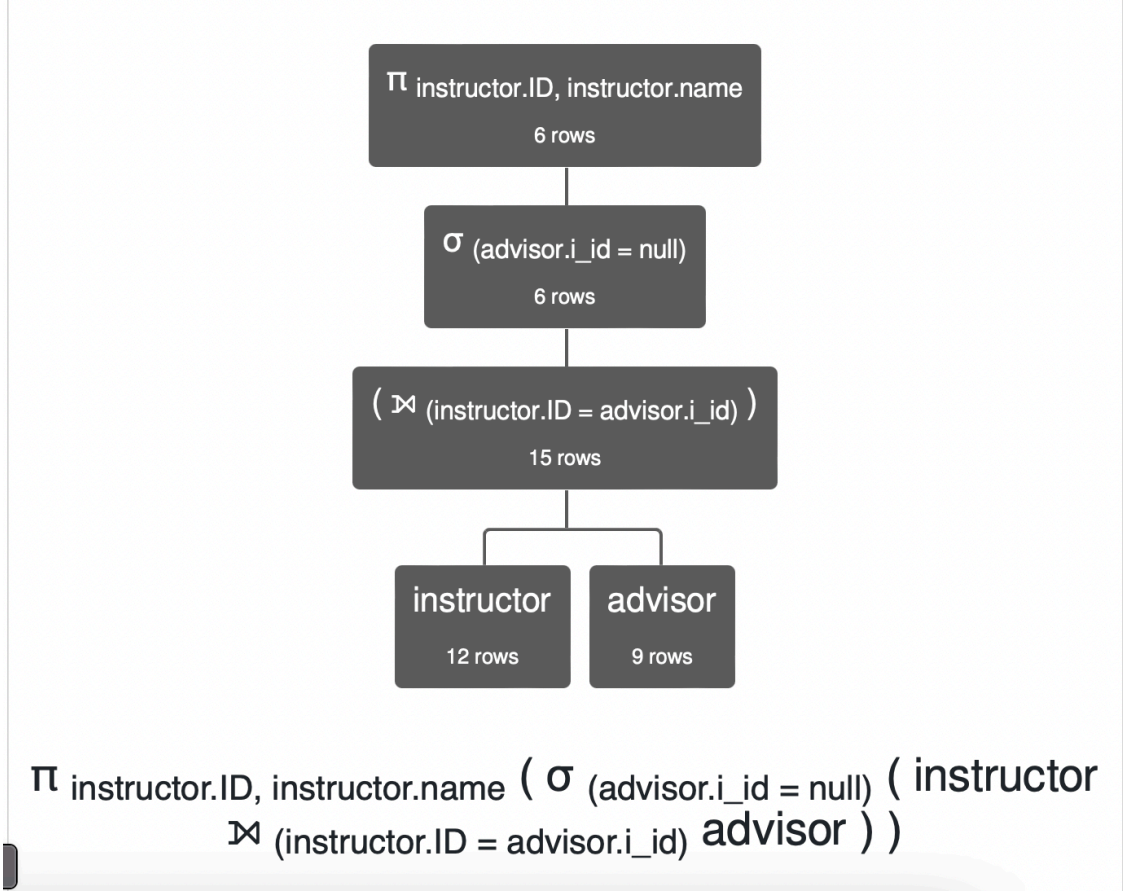
[10]:
```
from IPython.display import Image
Image('rel_query_2.png')
```

[10]:



[11]:
```
Image('rel_query_2_solution_graph.png')
```

[11]:



[12]:
```
Image('rel_query_2_solution_data.png')
```
[12]:

instructor.ID	instructor.name
12121	'Wu'
15151	'Mozart'
32343	'El Said'
33456	'Gold'
58583	'Califieri'
83821	'Brandt'

14 SQL Schema and DDL

14.0.1 Objective

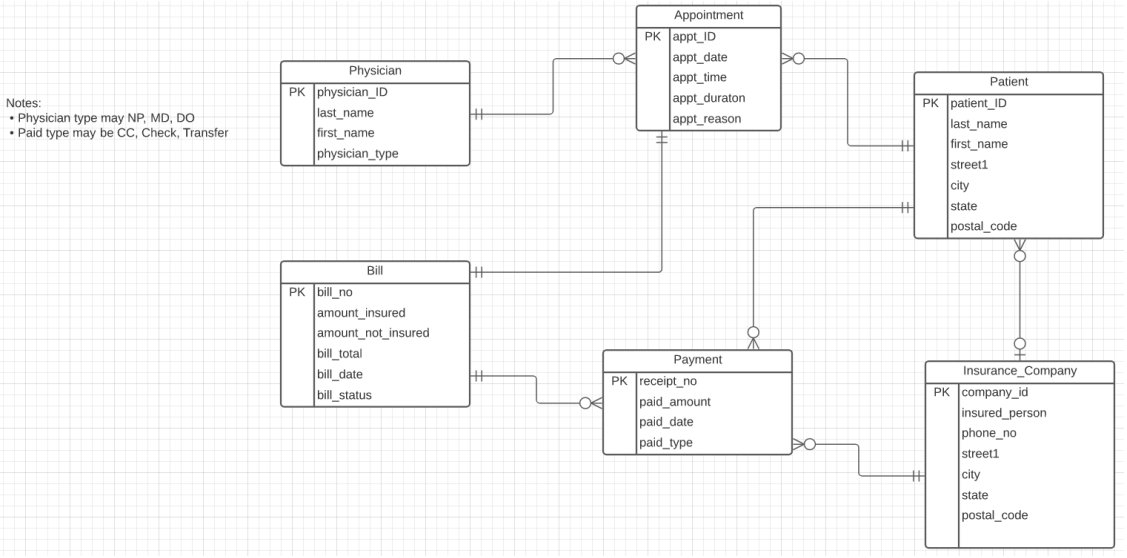
- You have a logical datamodel ER-diagram (see below).
- You need to use DDL to define a schema that realizes the model.
- Logical models are not specific enough for direct implementation. This means that:
  - You will have to assign concrete types to columns, and choose things like `GENERATED`, `DEFAULT`, etc.
  - You may have to decompose a table into two tables, or extract common attributes from multiple tables into a single, referenced table.

- Implementing the relationships may require adding columns and foreign keys, associative entities, etc.
- You may have to make other design and implementation choices. This means that there is no single correct answer.

14.0.2 ER Diagram

[30]:
```
from IPython.display import Image
Image('./er-to-sql-3.png')
```

[30]:



Answer

Design Decisions, Notes, etc.

1. For IDs (physician_ID, company_ID, patient_ID, bill_no, appt_ID, receipt_no) for all the table, I have selected VARCHAR datatype of length 25. 2. Taken the defaults for all the enums as (physician_type -> 'NP', bill_status -> 'NOT PAID', paid_type -> 'Check'). 3. Amount in the Bills can be of decimal type with a precision of 4. 4. Bill status for the Bills is an ENUM with 4 possible values and 'NOT PAID' as the default value. 5. Patient table is linked to the Insurance_Company table by company_ID (Foreign Key) 6. Appointment table is linked to the Physician table by physician_ID (Foreign Key) 7. Appointment table is linked to the Bill table by bill_no (Foreign Key) 8. Appointment table is linked to the Patient table by patient_ID (Foreign Key) 9. Payment table is linked to the Bill table by bill_no (Foreign Key) 10. Payment table is linked to the Patient table by patient_ID (Foreign Key) 11. Payment table is linked to the Insurance_Company table by company_ID (Foreign Key)

DDL

- Execute your DDL in the cell below. You may use DataGrip or other tools to help build the schema.
- You can copy and paste the `SQL CREATE TABLE` below, but you MUST execute the statements.

[36]:
```
%%sql
CREATE TABLE IF NOT EXISTS Physician
(
    physician_ID VARCHAR(25)  NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    first_name VARCHAR(50) NOT NULL,
    physician_type ENUM('NP', 'MD', 'DO') DEFAULT 'NP',

    CONSTRAINT PK_Phy_ID PRIMARY KEY (physician_ID)
);
```

```
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/lahmansbaseballdb
0 rows affected.
```

[36]:
```
[]
```

[37]:
```
%%sql
CREATE TABLE IF NOT EXISTS Insurance_Company
(
        company_ID VARCHAR(25) NOT NULL,
        insured_person VARCHAR(255) NOT NULL,
        phone_no INT(10) NOT NULL,
        street1 VARCHAR(100),
        city VARCHAR(100),
        state VARCHAR(100),
        postal_code INT(7),

        CONSTRAINT PK_Company_ID PRIMARY KEY (company_ID)
);
```

```
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/lahmansbaseballdb
0 rows affected.
```

[37]:
```
[]
```

[38]:
```
%%sql
CREATE TABLE IF NOT EXISTS Patient
(
    patient_ID VARCHAR(25) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    first_name VARCHAR(50) NOT NULL,
    street1 VARCHAR(100),
    city VARCHAR(100),
    state VARCHAR(100),
    postal_code INT(7),
    ins_company_ID VARCHAR(25),
```

```
    CONSTRAINT PK_Patient_ID PRIMARY KEY (patient_ID),
    CONSTRAINT Patient_FK_Company_ID FOREIGN KEY(ins_company_ID) REFERENCES␣
 ↪Insurance_Company(company_ID)
);
```

```
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/lahmansbaseballdb
0 rows affected.
```

[38]: []

[39]:
```
%%sql
CREATE TABLE IF NOT EXISTS Bill
(
        bill_no varchar(25) NOT NULL,
        amount_insured DECIMAL(20, 4),
        amount_not_insured DECIMAL(20, 4) NOT NULL,
        bill_total DECIMAL(25, 4) NOT NULL,
        bill_date DATETIME NOT NULL,
        bill_status ENUM('PAID_FULL', 'PAID_PARTIAL', 'PROCESSING', 'NOT PAID')␣
 ↪DEFAULT 'NOT PAID',

        CONSTRAINT PK_Bill_No PRIMARY KEY (bill_no)
);
```

```
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/lahmansbaseballdb
0 rows affected.
```

[39]: []

[40]:
```
%%sql
CREATE TABLE IF NOT EXISTS Appointment
(
    appt_ID VARCHAR(25) NOT NULL,
    appt_date DATE NOT NULL,
    appt_time TIME NOT NULL,
    appt_duration DECIMAL(4, 2) NOT NULL,
    appt_reason VARCHAR(255),
    physician_ID VARCHAR(255) NOT NULL,
    patient_ID VARCHAR(255) NOT NULL,
    bill_no VARCHAR(255) NOT NULL,

    CONSTRAINT PK_Appt_ID PRIMARY KEY (appt_ID),
    CONSTRAINT Appointment_FK_Physician FOREIGN KEY (physician_ID) REFERENCES␣
 ↪Physician (physician_ID),
```

```
    CONSTRAINT Appointment_FK_Patient FOREIGN KEY (patient_ID) REFERENCES␣
 ↪Patient (patient_ID),
    CONSTRAINT Appointment_FK_Bill FOREIGN KEY (bill_no) REFERENCES Bill␣
 ↪(bill_no)
);
```

```
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/lahmansbaseballdb
0 rows affected.
```

[40]: []

[41]:
```
%%sql
CREATE TABLE IF NOT EXISTS Payment
(
        receipt_no varchar(25) NOT NULL,
        paid_amount DECIMAL(25, 4) NOT NULL,
        paid_date DATETIME NOT NULL,
        paid_type  ENUM('CC', 'Check', 'Transfer') DEFAULT 'Check',
        bill_no  VARCHAR(25) NOT NULL,
        patient_ID VARCHAR(25) NOT NULL,
        ins_company_ID VARCHAR(25),

        CONSTRAINT PK_Receipt_No PRIMARY KEY (receipt_no),
        CONSTRAINT Payment_FK_Bill_No FOREIGN KEY(bill_no) REFERENCES␣
 ↪Bill(bill_no),
        CONSTRAINT Payment_FK_Patient_ID FOREIGN KEY(patient_ID) REFERENCES␣
 ↪Patient(patient_ID),
        CONSTRAINT Payment_FK_Company_ID FOREIGN KEY(ins_company_ID) REFERENCES␣
 ↪Insurance_Company(company_ID)
);
```

```
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/lahmansbaseballdb
0 rows affected.
```

[41]: []

14.1 Complex SQL

14.1.1 Birth Countries and Death Countries

Question

- In `lahmansbaseballdb.people` there is information about people's `birthCountry` and `deathCountry`.
- There are countries in which at least person was born but in which no person has died.
- Write a query that produces a table of the form:

  - birthCountry
  - no_of_births, which is the total number of births in the country

- The table contains all rows in which there with births but no deaths.

Answer

[22]:
```
%%sql
SELECT birthCountry, COUNT(playerID) AS no_of_births
FROM people
WHERE birthCountry NOT IN (
    SELECT deathCountry FROM people
    WHERE deathCountry IS NOT NULL
    )
GROUP BY birthCountry
ORDER BY no_of_births DESC;
```

```
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/lahmansbaseballdb
35 rows affected.
```

[22]:
```
[('Germany', 45),
 ('Colombia', 24),
 ('South Korea', 23),
 ('Curacao', 15),
 ('Nicaragua', 15),
 ('Russia', 9),
 ('Italy', 7),
 ('Czech Republic', 6),
 ('Aruba', 5),
 ('Brazil', 5),
 ('Poland', 5),
 ('Sweden', 4),
 ('Spain', 4),
 ('Jamaica', 4),
 ('Norway', 3),
 ('Honduras', 2),
 ('Guam', 2),
 ('South Africa', 2),
 ('Saudi Arabia', 2),
 ('Afghanistan', 1),
 ('Greece', 1),
 ('Hong Kong', 1),
 ('Viet Nam', 1),
 ('Denmark', 1),
 ('Switzerland', 1),
 ('Singapore', 1),
 ('Belgium', 1),
 ('Peru', 1),
```

```
('Belize', 1),
('Indonesia', 1),
('Finland', 1),
('Lithuania', 1),
('Slovakia', 1),
('Portugal', 1),
('Latvia', 1)]
```

14.1.2 Best Baseball Players

Question

- This question uses `lahmansbaseballdb.batting`, `lahmansbaseballdb.pitching` and `lahmansbaseballdb.people`.
- There query computes performance metrics:
  - Batting:
    - On-base percentage: OBP is (sum(h) + sum(BB))/(sum(ab) + sum(BB))
    - Slugging percentage: SLG is

      ```
      (
      (sum(h) - sum(`1b`) - sum(`2b`) - sum(`3b`) - sum(hr)) +
      2*sum(`2b`) + 3*sum(`3b`) + 4*hr
      )/sum(ab)
      ```
    - On-base percentage plus slugging: OPS is is `(obp + slg)`.
  - Pitching:
    - total_wins is `sum(w)`.
    - total_loses is `sum(l)`.
    - win_percentage is `sum(w)/(sum(w) + sum(l))`.
- Professor Ferguson has two criteria for someone being a great baseball player.
  - Batting:
    - Total number of `ab >= 1000`.
    - OPS: Career `OPS >= .000`
  - Pitching:
    - `(sum(w) + sum(l)) >= 200`.
    - `win_percentage >= 0.70)` or `sum(w) >= 300.`
- This is one of the rare cases where Prof. Ferguson will provide the answer. So, please produce the table below. Some notes:
  - `great_because` is either `Pitcher` or `Batter` based on whether the player matched the batting or pitching criteria.
  - The values from `batting` are `None` if the player did not qualify based on batting.
  - The values from `pitching` are `None` if the player did not qualify on pitching.

Answer

[24]:
```
%%sql
SELECT (SUM(h) + SUM(BB))/(SUM(ab) + SUM(BB)) AS OBP,
       ((SUM(h) - SUM(`2b`) - SUM(`3b`) - SUM(hr))
        + 2 * SUM(`2b`) + 3 * SUM(`3b`)
        + 4 * SUM(hr))/SUM(ab) AS SLG ,
       ((SUM(h) - SUM(`2b`) - SUM(`3b`) - SUM(hr))
        + 2 * SUM(`2b`) + 3 * SUM(`3b`)
        + 4 * SUM(hr))/SUM(ab) +
       (SUM(h) + SUM(BB))/(SUM(ab) + SUM(BB)) AS OPS
FROM batting
GROUP BY playerID
HAVING SUM(AB) >= 1000 and OPS > 1.000;
```

```
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/lahmansbaseballdb
7 rows affected.
```

[24]:
```
[(Decimal('0.4428'), Decimal('0.6069'), Decimal('1.0497')),
 (Decimal('0.4275'), Decimal('0.6093'), Decimal('1.0368')),
 (Decimal('0.4447'), Decimal('0.6324'), Decimal('1.0772')),
 (Decimal('0.4103'), Decimal('0.6050'), Decimal('1.0153')),
 (Decimal('0.4308'), Decimal('0.5765'), Decimal('1.0073')),
 (Decimal('0.4718'), Decimal('0.6898'), Decimal('1.1616')),
 (Decimal('0.4806'), Decimal('0.6338'), Decimal('1.1144'))]
```

[26]:
```
%%sql
SELECT SUM(pitch.W) AS total_wins, SUM(pitch.L) AS total_loses,
       SUM(pitch.W)/(SUM(pitch.W) + SUM(pitch.L)) AS win_percentage
FROM pitching pitch
GROUP BY pitch.playerID
HAVING (SUM(pitch.W) + SUM(pitch.L)) >= 200
AND (win_percentage >= 0.70 OR total_wins >= 300)
```

```
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/lahmansbaseballdb
25 rows affected.
```

[26]:
```
[(Decimal('373'), Decimal('208'), Decimal('0.6420')),
 (Decimal('329'), Decimal('244'), Decimal('0.5742')),
 (Decimal('328'), Decimal('178'), Decimal('0.6482')),
 (Decimal('354'), Decimal('184'), Decimal('0.6580')),
 (Decimal('365'), Decimal('310'), Decimal('0.5407')),
 (Decimal('305'), Decimal('203'), Decimal('0.6004')),
 (Decimal('300'), Decimal('141'), Decimal('0.6803')),
 (Decimal('303'), Decimal('166'), Decimal('0.6461')),
 (Decimal('417'), Decimal('279'), Decimal('0.5991')),
 (Decimal('342'), Decimal('225'), Decimal('0.6032')),
 (Decimal('355'), Decimal('227'), Decimal('0.6100')),
```

```
 (Decimal('373'), Decimal('188'), Decimal('0.6649')),
 (Decimal('362'), Decimal('208'), Decimal('0.6351')),
 (Decimal('318'), Decimal('274'), Decimal('0.5372')),
 (Decimal('314'), Decimal('265'), Decimal('0.5423')),
 (Decimal('326'), Decimal('194'), Decimal('0.6269')),
 (Decimal('310'), Decimal('194'), Decimal('0.6151')),
 (Decimal('324'), Decimal('292'), Decimal('0.5260')),
 (Decimal('311'), Decimal('205'), Decimal('0.6027')),
 (Decimal('363'), Decimal('245'), Decimal('0.5970')),
 (Decimal('252'), Decimal('65'), Decimal('0.7950')),
 (Decimal('324'), Decimal('256'), Decimal('0.5586')),
 (Decimal('307'), Decimal('210'), Decimal('0.5938')),
 (Decimal('300'), Decimal('244'), Decimal('0.5515')),
 (Decimal('511'), Decimal('315'), Decimal('0.6186'))]
```

[30]:
```
%%sql
DROP VIEW IF EXISTS batting_player_view
```

```
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/lahmansbaseballdb
0 rows affected.
```

[30]: []

[31]:
```
%%sql
CREATE VIEW batting_player_view AS
SELECT peeps.*,
       (SUM(bat.h) + SUM(bat.BB))/(SUM(bat.ab) + SUM(bat.BB)) AS OBP,
       ((SUM(bat.h) - SUM(bat.`2b`) - SUM(bat.`3b`) - SUM(bat.hr)) +
        2 * SUM(bat.`2b`) + 3 * SUM(bat.`3b`) + 4 * SUM(bat.hr))/SUM(bat.ab) AS␣
 ↪SLG ,
       ((SUM(bat.h) - SUM(bat.`2b`) - SUM(bat.`3b`) - SUM(bat.hr)) +
        2 * SUM(bat.`2b`) + 3 * SUM(bat.`3b`) + 4 * SUM(bat.hr))/SUM(bat.ab) +
       (SUM(bat.h) + SUM(bat.BB))/(SUM(bat.ab) + SUM(bat.BB)) AS OPS,
        'Batter' AS great_because
FROM batting bat CROSS JOIN people peeps ON peeps.playerID = bat.playerID
GROUP BY peeps.playerID
HAVING SUM(AB) >= 1000 and OPS > 1.000
```

```
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/lahmansbaseballdb
0 rows affected.
```

[31]: []

[32]:
```
%%sql
DROP VIEW IF EXISTS pitching_player_view;
```

```
   mysql+pymysql://root:***@localhost
```

```
 * mysql+pymysql://root:***@localhost/lahmansbaseballdb
0 rows affected.
```

[32]:
```
[]
```

[33]:
```
%%sql
CREATE VIEW pitching_player_view AS
SELECT peeps.*,
       SUM(pitch.W) AS total_wins, SUM(pitch.L) AS total_loses,
       SUM(pitch.W)/(SUM(pitch.W) + SUM(pitch.L)) AS win_percentage,
        'Pitcher' AS great_because
FROM pitching pitch LEFT OUTER JOIN people peeps ON pitch.playerID = peeps.
 ↪playerID
GROUP BY pitch.playerID
HAVING (SUM(pitch.W) + SUM(pitch.L)) >= 200
AND (win_percentage >= 0.70 OR total_wins >= 300)
```

```
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/lahmansbaseballdb
0 rows affected.
```

[33]:
```
[]
```

[34]:
```
%%sql
SELECT bpv.playerID, bpv.nameLast, bpv.nameFirst,
       bpv.great_because, bpv.debut_date, bpv.finalgame_date,
       bpv.OBP, bpv.SLG, bpv.OPS, ppv.total_wins,
    ppv.total_loses, ppv.win_percentage
FROM batting_player_view bpv
LEFT OUTER  JOIN  pitching_player_view ppv ON bpv.playerID = ppv.playerID
UNION
SELECT ppv.playerID, ppv.nameLast, ppv.nameFirst,
       ppv.great_because, ppv.debut_date, ppv.finalgame_date,
       bpv.OBP, bpv.SLG, bpv.OPS, ppv.total_wins,
        ppv.total_loses, ppv.win_percentage
FROM batting_player_view bpv
RIGHT OUTER  JOIN  pitching_player_view ppv ON bpv.playerID = ppv.playerID;
```

```
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/lahmansbaseballdb
32 rows affected.
```

[34]:
```
[('bondsba01', 'Bonds', 'Barry', 'Batter', datetime.date(1986, 5, 30),
datetime.date(2007, 9, 26), Decimal('0.4428'), Decimal('0.6069'),
Decimal('1.0497'), None, None, None),
 ('foxxji01', 'Foxx', 'Jimmie', 'Batter', datetime.date(1925, 5, 1),
datetime.date(1945, 9, 23), Decimal('0.4275'), Decimal('0.6093'),
Decimal('1.0368'), None, None, None),
 ('gehrilo01', 'Gehrig', 'Lou', 'Batter', datetime.date(1923, 6, 15),
```

```
datetime.date(1939, 4, 30), Decimal('0.4447'), Decimal('0.6324'),
Decimal('1.0772'), None, None, None),
 ('greenha01', 'Greenberg', 'Hank', 'Batter', datetime.date(1930, 9, 14),
datetime.date(1947, 9, 18), Decimal('0.4103'), Decimal('0.6050'),
Decimal('1.0153'), None, None, None),
 ('hornsro01', 'Hornsby', 'Rogers', 'Batter', datetime.date(1915, 9, 10),
datetime.date(1937, 7, 20), Decimal('0.4308'), Decimal('0.5765'),
Decimal('1.0073'), None, None, None),
 ('ruthba01', 'Ruth', 'Babe', 'Batter', datetime.date(1914, 7, 11),
datetime.date(1935, 5, 30), Decimal('0.4718'), Decimal('0.6898'),
Decimal('1.1616'), None, None, None),
 ('willite01', 'Williams', 'Ted', 'Batter', datetime.date(1939, 4, 20),
datetime.date(1960, 9, 28), Decimal('0.4806'), Decimal('0.6338'),
Decimal('1.1144'), None, None, None),
 ('spaldal01', 'Spalding', 'Al', 'Pitcher', datetime.date(1871, 5, 5),
datetime.date(1878, 8, 31), None, None, None, Decimal('252'), Decimal('65'),
Decimal('0.7950')),
 ('galvipu01', 'Galvin', 'Pud', 'Pitcher', datetime.date(1875, 5, 22),
datetime.date(1892, 8, 2), None, None, None, Decimal('365'), Decimal('310'),
Decimal('0.5407')),
 ('keefeti01', 'Keefe', 'Tim', 'Pitcher', datetime.date(1880, 8, 6),
datetime.date(1893, 8, 15), None, None, None, Decimal('342'), Decimal('225'),
Decimal('0.6032')),
 ('welchmi01', 'Welch', 'Mickey', 'Pitcher', datetime.date(1880, 5, 1),
datetime.date(1892, 5, 17), None, None, None, Decimal('307'), Decimal('210'),
Decimal('0.5938')),
 ('radboch01', 'Radbourn', 'Old Hoss', 'Pitcher', datetime.date(1880, 5, 5),
datetime.date(1891, 8, 11), None, None, None, Decimal('310'), Decimal('194'),
Decimal('0.6151')),
 ('clarkjo01', 'Clarkson', 'John', 'Pitcher', datetime.date(1882, 5, 2),
datetime.date(1894, 7, 12), None, None, None, Decimal('328'), Decimal('178'),
Decimal('0.6482')),
 ('nichoki01', 'Nichols', 'Kid', 'Pitcher', datetime.date(1890, 4, 23),
datetime.date(1906, 5, 18), None, None, None, Decimal('362'), Decimal('208'),
Decimal('0.6351')),
 ('youngcy01', 'Young', 'Cy', 'Pitcher', datetime.date(1890, 8, 6),
datetime.date(1911, 10, 6), None, None, None, Decimal('511'), Decimal('315'),
Decimal('0.6186')),
 ('mathech01', 'Mathewson', 'Christy', 'Pitcher', datetime.date(1900, 7, 17),
datetime.date(1916, 9, 4), None, None, None, Decimal('373'), Decimal('188'),
Decimal('0.6649')),
 ('planked01', 'Plank', 'Eddie', 'Pitcher', datetime.date(1901, 5, 13),
datetime.date(1917, 8, 6), None, None, None, Decimal('326'), Decimal('194'),
Decimal('0.6269')),
 ('johnswa01', 'Johnson', 'Walter', 'Pitcher', datetime.date(1907, 8, 2),
datetime.date(1927, 9, 30), None, None, None, Decimal('417'), Decimal('279'),
Decimal('0.5991')),
```

```
 ('alexape01', 'Alexander', 'Pete', 'Pitcher', datetime.date(1911, 4, 15),
datetime.date(1930, 5, 28), None, None, None, Decimal('373'), Decimal('208'),
Decimal('0.6420')),
 ('grovele01', 'Grove', 'Lefty', 'Pitcher', datetime.date(1925, 4, 14),
datetime.date(1941, 9, 28), None, None, None, Decimal('300'), Decimal('141'),
Decimal('0.6803')),
 ('wynnea01', 'Wynn', 'Early', 'Pitcher', datetime.date(1939, 9, 13),
datetime.date(1963, 9, 13), None, None, None, Decimal('300'), Decimal('244'),
Decimal('0.5515')),
 ('spahnwa01', 'Spahn', 'Warren', 'Pitcher', datetime.date(1942, 4, 19),
datetime.date(1965, 10, 1), None, None, None, Decimal('363'), Decimal('245'),
Decimal('0.5970')),
 ('perryga01', 'Perry', 'Gaylord', 'Pitcher', datetime.date(1962, 4, 14),
datetime.date(1983, 9, 21), None, None, None, Decimal('314'), Decimal('265'),
Decimal('0.5423')),
 ('niekrph01', 'Niekro', 'Phil', 'Pitcher', datetime.date(1964, 4, 15),
datetime.date(1987, 9, 27), None, None, None, Decimal('318'), Decimal('274'),
Decimal('0.5372')),
 ('carltst01', 'Carlton', 'Steve', 'Pitcher', datetime.date(1965, 4, 12),
datetime.date(1988, 4, 23), None, None, None, Decimal('329'), Decimal('244'),
Decimal('0.5742')),
 ('ryanno01', 'Ryan', 'Nolan', 'Pitcher', datetime.date(1966, 9, 11),
datetime.date(1993, 9, 22), None, None, None, Decimal('324'), Decimal('292'),
Decimal('0.5260')),
 ('suttodo01', 'Sutton', 'Don', 'Pitcher', datetime.date(1966, 4, 14),
datetime.date(1988, 8, 9), None, None, None, Decimal('324'), Decimal('256'),
Decimal('0.5586')),
 ('seaveto01', 'Seaver', 'Tom', 'Pitcher', datetime.date(1967, 4, 13),
datetime.date(1986, 9, 19), None, None, None, Decimal('311'), Decimal('205'),
Decimal('0.6027')),
 ('clemero02', 'Clemens', 'Roger', 'Pitcher', datetime.date(1984, 5, 15),
datetime.date(2007, 9, 16), None, None, None, Decimal('354'), Decimal('184'),
Decimal('0.6580')),
 ('maddugr01', 'Maddux', 'Greg', 'Pitcher', datetime.date(1986, 9, 3),
datetime.date(2008, 9, 27), None, None, None, Decimal('355'), Decimal('227'),
Decimal('0.6100')),
 ('glavito02', 'Glavine', 'Tom', 'Pitcher', datetime.date(1987, 8, 17),
datetime.date(2008, 8, 14), None, None, None, Decimal('305'), Decimal('203'),
Decimal('0.6004')),
 ('johnsra05', 'Johnson', 'Randy', 'Pitcher', datetime.date(1988, 9, 15),
datetime.date(2009, 10, 4), None, None, None, Decimal('303'), Decimal('166'),
Decimal('0.6461'))]
```

14.1.3 Putting Together DDL, DML, Functions, Triggers

Question

- Use the database that comes with the textbook for this question.

  - Create a new database `db_book_midterm`.
  - Copy the data and table definitions for `Student` and `Instructor`
  - You may have to remove some constraints from the copied data/definition to make it work.
- Base tables:
  - `Student` has the form `Student(ID, name, dept_name, total_cred)`.
  - `Instructor` has the form `Instructor(ID, name, dept_name, salary)`.
- There is a *logical* base type `Person`. The logical `isA` model is:

Logical Model

- `role` is either `S` or `F` based on whether the `Person` is a `Student` or `Instructor`.
- Implement a *two table* solution to realize `Person`. This means define `Person` as a view.
- You do not need to worry about generating the primary key ID. Your implementation MUST, however, enforce the rule that the ID is immutable.
- You must also create a *stored procedure* `create_person`. The template for the implementation is:

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `create_person`(
    in person_name varchar(32),
    in dept_name varchar(32),
    in total_cred decimal(3,0),
    in salary decimal(8,2),
    out ID varchar(5)
    )
BEGIN

    declare bad_person boolean;
    declare new_id varchar(12);

    set bad_person = false;
    set new_id = '00000';


    /*
        The logic of the stored procedure is the following:
            - The request is invalid and sets bad_person to true if:
                - Any of person_name, dept_name is NULL.
                - Either total_cred is NULL and salary is NOT NULL, or salary is NULL and total
            - The procedure must compute a new, unique ID. The approach is to find the manimum
                ID value over Student and Instructor. Add 1 to the value to produce the new, u
            - The procedure then adds the information to Student or Instructor based on whethe
                total_cred is NULL or salary is NULL.
    */
```

```
    /* <YOUR CODE GOES HERE> */

    if bad_person is true then
        SIGNAL SQLSTATE '50001'
            SET MESSAGE_TEXT = 'Invalid Person information input';
    end if;

    /* NOTE: ID is the out parameter. You must set ID to the new, unique ID */

END
```

Answer

Create view statement

```
CREATE DATABASE db_book_midterm;

CREATE TABLE IF NOT EXISTS db_book_midterm.Student
    LIKE F21MidTerm.student;
INSERT db_book_midterm.Student SELECT * FROM F21MidTerm.student;

CREATE TABLE IF NOT EXISTS db_book_midterm.Instructor
    LIKE F21MidTerm.instructor;
INSERT db_book_midterm.Instructor SELECT * FROM F21MidTerm.instructor;

CREATE VIEW Person AS SELECT * FROM
(SELECT id, name, dept_name, 'F' role FROM db_book_midterm.Instructor AS ins
UNION
SELECT id, name, dept_name, 'S' role FROM db_book_midterm.Student AS stu) AS person;
```

Create procedure statement

```
DROP PROCEDURE IF EXISTS create_person;
CREATE DEFINER=`root`@`localhost` PROCEDURE `create_person`(
    in person_name varchar(32),
    in dept_name varchar(32),
    in total_cred decimal(3,0),
    in salary decimal(8,2),
    out ID varchar(5)
    )
BEGIN

    declare bad_person boolean;
    declare new_id varchar(12);
    declare max_id varchar(12);

    set bad_person = false;
    set new_id = '00000';

    /*
```

```
        The logic of the stored procedure is the following:
            - The request is invalid and sets bad_person to true if:
                - Any of person_name, dept_name is NULL.
                - Either total_cred is NOT NULL and salary is NOT NULL, or salary is NULL and
            - The procedure must compute a new, unique ID. The approach is to find the manimum
                ID value over Student and Instructor. Add 1 to the value to produce the new, u
            - The procedure then adds the information to Student or Instructor based on whethe
                total_cred is NULL or salary is NULL.
    */

    /* <YOUR CODE GOES HERE> */

    if person_name is null or dept_name is null or
       (total_cred is not null and salary is not null) or
       (salary is null and total_cred is null) then
        SET bad_person = true;
    end if;

    SELECT MAX(CAST(db_book_midterm.person.id AS UNSIGNED)) INTO max_id
    FROM db_book_midterm.person;
    SET new_id = max_id + 1;

    if bad_person is true then
        SIGNAL SQLSTATE '50001'
            SET MESSAGE_TEXT = 'Invalid Person information input';
    end if;

    /* NOTE: ID is the out parameter. You must set ID to the new, unique ID */
    SET ID = new_id;

    if total_cred is null and salary is not null then
        INSERT INTO db_book_midterm.Instructor
        VALUES (new_id, person_name, dept_name, salary);
    elseif total_cred is not null and salary is null then
        INSERT INTO db_book_midterm.Student
        VALUES (new_id, person_name, dept_name, tot_cred);
    end if;
END;
```

Tests

Run the SQL in the following cells to test your solution.

[5]:
```
%sql mysql+pymysql://root:dbuserdbuser@localhost/db_book_midterm
```

- Test 1: Show the view.

[6]:
```
%sql select * from Person;
```

```
 * mysql+pymysql://root:***@localhost/db_book_midterm
```

```
   mysql+pymysql://root:***@localhost/lahmansbaseballdb
25 rows affected.
```

[6]:
```
[('10101', 'Srinivasan', 'Comp. Sci.', 'F'),
 ('12121', 'Wu', 'Finance', 'F'),
 ('15151', 'Mozart', 'Music', 'F'),
 ('22222', 'Einstein', 'Physics', 'F'),
 ('32343', 'El Said', 'History', 'F'),
 ('33456', 'Gold', 'Physics', 'F'),
 ('45565', 'Katz', 'Comp. Sci.', 'F'),
 ('58583', 'Califieri', 'History', 'F'),
 ('76543', 'Singh', 'Finance', 'F'),
 ('76766', 'Crick', 'Biology', 'F'),
 ('83821', 'Brandt', 'Comp. Sci.', 'F'),
 ('98345', 'Kim', 'Elec. Eng.', 'F'),
 ('00128', 'Zhang', 'Comp. Sci.', 'S'),
 ('12345', 'Shankar', 'Comp. Sci.', 'S'),
 ('19991', 'Brandt', 'History', 'S'),
 ('23121', 'Chavez', 'Finance', 'S'),
 ('44553', 'Peltier', 'Physics', 'S'),
 ('45678', 'Levy', 'Physics', 'S'),
 ('54321', 'Williams', 'Comp. Sci.', 'S'),
 ('55739', 'Sanchez', 'Music', 'S'),
 ('70557', 'Snow', 'Physics', 'S'),
 ('76543', 'Brown', 'Comp. Sci.', 'S'),
 ('76653', 'Aoi', 'Elec. Eng.', 'S'),
 ('98765', 'Bourikas', 'Elec. Eng.', 'S'),
 ('98988', 'Tanaka', 'Biology', 'S')]
```

Create an Instructor and Student

[7]:
```
%sql CALL create_person('Ferguson', 'Comp. Sci.',  NULL, 30000.00, @prof_id);
```

```
 * mysql+pymysql://root:***@localhost/db_book_midterm
   mysql+pymysql://root:***@localhost/lahmansbaseballdb
1 rows affected.
```

[7]:
```
[]
```

[8]:
```
%sql SELECT @prof_id;
```

```
 * mysql+pymysql://root:***@localhost/db_book_midterm
   mysql+pymysql://root:***@localhost/lahmansbaseballdb
1 rows affected.
```

[8]:
```
[('98989',)]
```

[9]:
```
%sql CALL create_person('Ferguson', 'Comp. Sci.',  100.0, NULL, @student_id);
```

```
 * mysql+pymysql://root:***@localhost/db_book_midterm
```

```
   mysql+pymysql://root:***@localhost/lahmansbaseballdb
1 rows affected.
```

[9]:
```
[]
```

[10]:
```
%sql SELECT @student_id;
```

```
 * mysql+pymysql://root:***@localhost/db_book_midterm
   mysql+pymysql://root:***@localhost/lahmansbaseballdb
1 rows affected.
```

[10]:
```
[('98990',)]
```

- Try an error

[11]:
```
try:
    %sql CALL create_person('Ferguson', 'Comp. Sci.',  100.0, 30000,␣
 ↪@student_id);
except Exception as e:
    print(e)
res = %sql select @student_id;
```

```
 * mysql+pymysql://root:***@localhost/db_book_midterm
   mysql+pymysql://root:***@localhost/lahmansbaseballdb
(pymysql.err.OperationalError) (1644, 'Invalid Person information input')
[SQL: CALL create_person('Ferguson', 'Comp. Sci.' , 100.0, 30000, @student_id);]
(Background on this error at: https://sqlalche.me/e/14/e3q8)
 * mysql+pymysql://root:***@localhost/db_book_midterm
   mysql+pymysql://root:***@localhost/lahmansbaseballdb
1 rows affected.
```

Include DDL that Show Enforcing Immutable ID

```
CREATE TRIGGER db_book_midterm.id_update_check_st BEFORE UPDATE
ON db_book_midterm.Student
FOR EACH ROW
IF NEW.id != OLD.id THEN
SIGNAL SQLSTATE '50002' SET MESSAGE_TEXT = 'Id cannot be updated.';
END IF;

CREATE TRIGGER db_book_midterm.id_update_check_ins BEFORE UPDATE
ON db_book_midterm.Instructor
FOR EACH ROW
IF NEW.id != OLD.id THEN
SIGNAL SQLSTATE '50002' SET MESSAGE_TEXT = 'Id cannot be updated.';
END IF;
```

Write a test that shows you implemented immutable IDs

[12]:
```
%sql UPDATE db_book_midterm.Student SET ID = '11111' WHERE ID = '00128';
```

```
 * mysql+pymysql://root:***@localhost/db_book_midterm
   mysql+pymysql://root:***@localhost/lahmansbaseballdb
(pymysql.err.OperationalError) (1644, 'Id cannot be updated.')
[SQL: UPDATE db_book_midterm.Student SET ID = '11111' WHERE ID = '00128' ;]
(Background on this error at: https://sqlalche.me/e/14/e3q8)
```

[13]:
```
%sql UPDATE db_book_midterm.Instructor SET ID = '00011' WHERE ID = '10101';
```

```
 * mysql+pymysql://root:***@localhost/db_book_midterm
   mysql+pymysql://root:***@localhost/lahmansbaseballdb
(pymysql.err.OperationalError) (1644, 'Id cannot be updated.')
[SQL: UPDATE db_book_midterm.Instructor SET ID = '00011' WHERE ID = '10101' ;]
(Background on this error at: https://sqlalche.me/e/14/e3q8)
```

15 Data and Schema Cleanup

15.1 Part 1 – Countries and Cities

- There is a file `worldcities` in the same folder as this notebook.
- In the following code cell, use Pandas to:
  - Read the CSV file into a Data Frame.
  - Convert the Data Frame to contain only the following columns:
    - city
    - city_ascii
    - lat
    - lng
    - country
    - iso2
    - iso2
    - id
  - Write the data to the table `worldcities` in the schema `F21W4111Midterm`.
- Use the SQL after the code cell to display part of your new table.

Answer

[18]:
```
import pandas as pd
world_cities_df = pd.read_csv('./worldcities.csv')
```

[19]:
```
world_cities_df = world_cities_df[["city", "city_ascii", "lat", "lng",␣
↪"country", "iso2", "iso3", "id"]]
```

[20]:
```
import mysql.connector
from sqlalchemy import create_engine

mydb_connection = mysql.connector.connect(host="localhost", user="root",␣
↪password="dbuserdbuser")
mycursor = mydb_connection.cursor()
```

```
mycursor.execute("CREATE DATABASE F21W4111Midterm")

db_engine = create_engine("mysql+pymysql://root:dbuserdbuser@localhost/
 ↪F21W4111Midterm")
world_cities_df.to_sql('worldcities', con = db_engine, if_exists='replace')
```

[21]:
```
%sql mysql+pymysql://root:dbuserdbuser@localhost/F21W4111Midterm
```

- Display data.

[22]:
```
%%sql
SELECT * FROM F21W4111Midterm.worldcities
ORDER BY city
LIMIT 30;
```

```
 * mysql+pymysql://root:***@localhost/F21W4111Midterm
   mysql+pymysql://root:***@localhost/db_book_midterm
   mysql+pymysql://root:***@localhost/lahmansbaseballdb
30 rows affected.
```

[22]:
```
[(19249, '‘Adrā', '`Adra', 33.6, 36.515, 'Syria', 'SY', 'SYR', 1760640037),
 (9815, '‘Ajlūn', '`Ajlun', 32.3325, 35.7517, 'Jordan', 'JO', 'JOR',
1400775371),
 (2469, '‘Ajmān', '`Ajman', 25.3994, 55.4797, 'United Arab Emirates', 'AE',
'ARE', 1784337875),
 (13032, '‘Akko', '`Akko', 32.9261, 35.0839, 'Israel', 'IL', 'ISR', 1376781950),
 (23726, '‘Alavīcheh', '`Alavicheh', 33.0528, 51.0825, 'Iran', 'IR', 'IRN',
1364605877),
 (9612, '‘Amrān', '`Amran', 15.6594, 43.9439, 'Yemen', 'YE', 'YEM', 1887433410),
 (13142, '‘Āmūdā', '`Amuda', 37.1042, 40.93, 'Syria', 'SY', 'SYR', 1760247135),
 (26026, '‘Anadān', '`Anadan', 36.2936, 37.0444, 'Syria', 'SY', 'SYR',
1760993442),
 (37808, '‘Assāl al Ward', '`Assal al Ward', 33.8658, 36.4133, 'Syria', 'SY',
'SYR', 1760181042),
 (6512, '‘Ataq', '`Ataq', 14.55, 46.8, 'Yemen', 'YE', 'YEM', 1887172893),
 (35329, '‘Ayn ‘Īsá', '`Ayn `Isa', 36.3858, 38.8472, 'Syria', 'SY', 'SYR',
1760078370),
 (4900, '‘Ibrī', '`Ibri', 23.2254, 56.517, 'Oman', 'OM', 'OMN', 1512077267),
 (22092, '’Aïn Abessa', "'Ain Abessa", 36.3, 5.295, 'Algeria', 'DZ', 'DZA',
1012074116),
 (13597, '’Aïn Arnat', "'Ain Arnat", 36.1833, 5.3167, 'Algeria', 'DZ', 'DZA',
1012453452),
 (13002, '’Aïn Azel', "'Ain Azel", 35.8433, 5.5219, 'Algeria', 'DZ', 'DZA',
1012746080),
 (19677, '’Aïn el Hammam', "'Ain el Hammam", 36.5647, 4.3061, 'Algeria', 'DZ',
'DZA', 1012595495),
 (28994, '’Aïn Leuh', "'Ain Leuh", 33.2833, -5.3833, 'Morocco', 'MA', 'MAR',
1504668626),
```

```
 (26882, ''Aïn Roua', "'Ain Roua", 36.3344, 5.1806, 'Algeria', 'DZ', 'DZA',
1012529757),
 (20104, ''Ali Ben Sliman', "'Ali Ben Sliman", 31.9053, -7.2144, 'Morocco',
'MA', 'MAR', 1504127885),
 (22485, ''Ayn Bni Mathar', "'Ayn Bni Mathar", 34.0889, -2.0247, 'Morocco',
'MA', 'MAR', 1504845272),
 (3842, ''s-Hertogenbosch', "'s-Hertogenbosch", 51.6833, 5.3167, 'Netherlands',
'NL', 'NLD', 1528012333),
 (1746, 'A Coruña', 'A Coruna', 43.3713, -8.4188, 'Spain', 'ES', 'ESP',
1724417375),
 (2400, 'Aachen', 'Aachen', 50.7762, 6.0838, 'Germany', 'DE', 'DEU',
1276805572),
 (29829, 'Aadorf', 'Aadorf', 47.4939, 8.8975, 'Switzerland', 'CH', 'CHE',
1756022542),
 (4077, 'Aalborg', 'Aalborg', 57.0337, 9.9166, 'Denmark', 'DK', 'DNK',
1208789278),
 (11447, 'Aalen', 'Aalen', 48.8372, 10.0936, 'Germany', 'DE', 'DEU',
1276757787),
 (15609, 'Aalsmeer', 'Aalsmeer', 52.2639, 4.7625, 'Netherlands', 'NL', 'NLD',
1528899853),
 (10664, 'Aalst', 'Aalst', 50.9333, 4.0333, 'Belgium', 'BE', 'BEL', 1056695813),
 (17071, 'Aalten', 'Aalten', 51.925, 6.5808, 'Netherlands', 'NL', 'NLD',
1528326020),
 (20206, 'Äänekoski', 'Aanekoski', 62.6042, 25.7264, 'Finland', 'FI', 'FIN',
1246710490)]
```

15.2 P2 – Modify World City Data

- Having multiple rows that repeat `country, iso2` and `iso3` is a poor design.
- Create two new tables:
  - `countries` that contains `country, iso2` and `iso3`.
  - `cities` that contains only the remaining fields.
  - Pick either `iso2` or `iso3` to define a foreign key between the tables.
- Add primary keys, unique keys, select column data types, etc. to define a better schema for the two tables.
- Show you SQL statements for creating and modifying the tables below.
- **Note:** A small number of the ISO2 and ISO3 codes are incorrect and will prevent creating keys. You must correct this data.
- Show you DDL below.

Answer

```
DROP TABLE IF EXISTS  F21W4111Midterm.cities;
DROP TABLE IF EXISTS F21W4111Midterm.countries;
```

```
CREATE TABLE F21W4111Midterm.cities AS
    (
        SELECT city, city_ascii, lat, lng, id , iso2
        FROM F21W4111Midterm.worldcities
    );

ALTER TABLE F21W4111Midterm.cities
    MODIFY city VARCHAR(50) NOT NULL;
ALTER TABLE F21W4111Midterm.cities
    MODIFY city_ascii VARCHAR(50) NOT NULL;
ALTER TABLE F21W4111Midterm.cities
    MODIFY lat DOUBLE;
ALTER TABLE F21W4111Midterm.cities
    MODIFY lng DOUBLE;
ALTER TABLE F21W4111Midterm.cities
    MODIFY id BIGINT UNIQUE;
UPDATE F21W4111Midterm.cities
SET iso2 = 'NA' WHERE iso2 IS NULL;
ALTER TABLE F21W4111Midterm.cities
    MODIFY iso2 char(2) NOT NULL;

ALTER TABLE F21W4111Midterm.cities
    ADD CONSTRAINT PK_cities_id
        PRIMARY KEY (id);


CREATE TABLE F21W4111Midterm.countries AS
    (
        SELECT DISTINCT country, iso2, iso3
        FROM F21W4111Midterm.worldcities
    );

ALTER TABLE F21W4111Midterm.countries
    MODIFY country VARCHAR(50) NOT NULL;
ALTER TABLE F21W4111Midterm.countries
    MODIFY iso2 CHAR(2) UNIQUE;
ALTER TABLE F21W4111Midterm.countries
    MODIFY iso3 CHAR(3);

UPDATE F21W4111Midterm.countries
SET iso2 = 'NA' WHERE iso2 IS NULL;

UPDATE F21W4111Midterm.countries
SET iso3 = "ARM"
WHERE country = 'Armenia';
```

```
ALTER TABLE F21W4111Midterm.countries
    ADD CONSTRAINT PK_countries_iso2
        PRIMARY KEY (iso2);

ALTER TABLE F21W4111Midterm.cities
    ADD CONSTRAINT cities_FK_countries_iso2
        FOREIGN KEY (iso2) REFERENCES countries (iso2);
```

15.3 P3 – An Easy Question

- An interesting question. Is there a better SQL type for latitude and longitude than `DOUBLE`? If you think there is a better type, what would it be? (You do not need to perform any conversions)

Answer We could use the DECIMAL datatype. As the latitude values range between -90 and 90 while the longitude values range between -180 and 180, we can fix the digits needed (before the decimal point) for both to be 2 and 3 respectively. Also, according to the values in the database, the precision needed for these values (after the decimal point) would be 4 digits. Thus, we could use DECIMAL(6, 4) for latitude and DECIMAL(7, 4) for longitude. This would be a better type as it better restricts the domain of the data possible in those fields.

15.4 FInal Create Table Statements

- Use the DataGrip tool to generate final `CREATE TABLE` statements below. You do not need to execute the statements.

Answer

```
create table if not exists countries
(
    country varchar(50) not null,
    iso2    char(2)     not null,
    iso3    char(3)     null,
    constraint iso2
        unique (iso2)
);

alter table countries
    add primary key (iso2);

create table if not exists cities
(
    city       varchar(50) not null,
    city_ascii varchar(50) not null,
    lat        double      null,
    lng        double      null,
    id         bigint      not null,
    iso2       char(2)     not null,
    constraint id
```

```
        unique (id),
    constraint cities_FK_countries_iso2
        foreign key (iso2) references countries (iso2)
);

alter table cities
    add primary key (id);

create table if not exists worldcities
(
    `index`    bigint null,
    city       text   null,
    city_ascii text   null,
    lat        double null,
    lng        double null,
    country    text   null,
    iso2       text   null,
    iso3       text   null,
    id         bigint null
);

create index ix_worldcities_index
    on worldcities (`index`);
```

15.5 Fixing People Table

Create a Copy People

- Create a table `F21Midterm.people_modified` that has the same schema and data as `lahmansbaseballdb.people`.
- SQL:

Answer

```
CREATE TABLE IF NOT EXISTS F21W4111Midterm.people_modified
    LIKE lahmansbaseballdb.people;

INSERT F21W4111Midterm.people_modified
SELECT * FROM lahmansbaseballdb.people;
```

Fixing `birthCountry`

- The query below indicates that some `birthCountry` entries in `people` do not map to a know country.

[23]:
```
%%sql
select distinct birthCountry, count(*) as count  from people_modified where
        birthCountry not in (select country from countries)
    group by birthCountry;
```

```
 * mysql+pymysql://root:***@localhost/F21W4111Midterm
   mysql+pymysql://root:***@localhost/db_book_midterm
   mysql+pymysql://root:***@localhost/lahmansbaseballdb
10 rows affected.
```

[23]:
```
[('USA', 17254),
 ('D.R.', 761),
 ('CAN', 255),
 ('P.R.', 268),
 ('Bahamas', 6),
 ('South Korea', 23),
 ('Czech Republic', 6),
 ('V.I.', 14),
 ('Viet Nam', 1),
 ('At Sea', 1)]
```

- My proposed corrections for `birthCountry` are:

birthCountry	ISO3	ISO2	Correct Country Name
Bahamas	BHS	BS	Bahamas, The
CAN	CAN	CA	Canada
Czech Republic	CZE	CZ	Czechia
South Korea	KOR	KR	Korea, South
USA	USA	US	United States
Viet Nam	VNM	VN	Vietnam
D.R.	DOM	DO	Dominican Republic
P.R.	PRI	PR	Puerto Rico
V.I.	USA	US	United States
At Sea	NULL	NULL	NULL

- Correct `people_modified`, making the following changes:
  1. Add a column `birthCountryISO3`
  2. Correct the entries for `birthCountry`.
  3. Populate the values for `birthCountryISO3`
  4. Set up a foreign key relationship from `people_modified` to `countries`.
  5. Drop the column `birthCountry`.

Answer

- Show your SQL statements for altering the table below.

```sql
ALTER TABLE F21W4111Midterm.people_modified
ADD COLUMN birthCountryISO3 VARCHAR(350);

UPDATE F21W4111Midterm.people_modified
SET birthCountry = 'Bahamas, The'
WHERE birthCountry = 'Bahamas';
```

```
UPDATE F21W4111Midterm.people_modified
SET birthCountry = 'Canada'
WHERE birthCountry = 'CAN';

UPDATE F21W4111Midterm.people_modified
SET birthCountry = 'Czechia'
WHERE birthCountry = 'Czech Republic';

UPDATE F21W4111Midterm.people_modified
SET birthCountry = 'Korea, South'
WHERE birthCountry = 'South Korea';

UPDATE F21W4111Midterm.people_modified
SET birthCountry = 'United States'
WHERE birthCountry = 'USA';

UPDATE F21W4111Midterm.people_modified
SET birthCountry = 'Vietnam'
WHERE birthCountry = 'Viet Nam';

UPDATE F21W4111Midterm.people_modified
SET birthCountry = 'Dominican Republic'
WHERE birthCountry = 'D.R.';

UPDATE F21W4111Midterm.people_modified
SET birthCountry = 'Puerto Rico'
WHERE birthCountry = 'P.R.';

UPDATE F21W4111Midterm.people_modified
SET birthCountry = 'United States'
WHERE birthCountry = 'V.I.';

UPDATE F21W4111Midterm.people_modified
SET birthCountry = NULL
WHERE birthCountry = 'At Sea';

UPDATE F21W4111Midterm.people_modified AS peeps_mod
INNER JOIN F21W4111Midterm.countries AS c
ON peeps_mod.birthCountry = c.country
SET peeps_mod.birthCountryISO3 = c.iso3;

CREATE UNIQUE INDEX uidx_countries_iso3 ON countries (iso3);
ALTER TABLE people_modified
    ADD CONSTRAINT people_modified_fk_countries_iso3
        FOREIGN KEY (birthCountryISO3) REFERENCES countries (iso3);

ALTER TABLE F21W4111Midterm.people_modified DROP birthCountry;
```

- Run a couple of queries to show correctly modified table.

[24]:
```
%%sql
SELECT * FROM F21W4111Midterm.countries
LIMIT 10
```

```
 * mysql+pymysql://root:***@localhost/F21W4111Midterm
   mysql+pymysql://root:***@localhost/db_book_midterm
   mysql+pymysql://root:***@localhost/lahmansbaseballdb
10 rows affected.
```

[24]:
```
[('Andorra', 'AD', 'AND'),
 ('United Arab Emirates', 'AE', 'ARE'),
 ('Afghanistan', 'AF', 'AFG'),
 ('Antigua And Barbuda', 'AG', 'ATG'),
 ('Anguilla', 'AI', 'AIA'),
 ('Albania', 'AL', 'ALB'),
 ('Armenia', 'AM', 'ARM'),
 ('Angola', 'AO', 'AGO'),
 ('Argentina', 'AR', 'ARG'),
 ('American Samoa', 'AS', 'ASM')]
```

[25]:
```
%%sql
SELECT * FROM F21W4111Midterm.cities
LIMIT 10
```

```
 * mysql+pymysql://root:***@localhost/F21W4111Midterm
   mysql+pymysql://root:***@localhost/db_book_midterm
   mysql+pymysql://root:***@localhost/lahmansbaseballdb
10 rows affected.
```

[25]:
```
[('Kandahār', 'Kandahar', 31.6078, 65.7053, 1004003059, 'AF'),
 ('Qalāt', 'Qalat', 32.1061, 66.9069, 1004016690, 'AF'),
 ('Sar-e Pul', 'Sar-e Pul', 36.2214, 65.9278, 1004047427, 'AF'),
 ('Pul-e Khumrī', 'Pul-e Khumri', 35.95, 68.7, 1004123527, 'AF'),
 ('Ma mūd-e Rāqī', 'Mahmud-e Raqi', 35.0167, 69.3333, 1004151943, 'AF'),
 ('Ghaznī', 'Ghazni', 33.5492, 68.4233, 1004167490, 'AF'),
 ('Pul-e ʻAlam', 'Pul-e `Alam', 33.9953, 69.0227, 1004180853, 'AF'),
 ('Kunduz', 'Kunduz', 36.728, 68.8725, 1004227517, 'AF'),
 ('Herāt', 'Herat', 34.3738, 62.1792, 1004237782, 'AF'),
 ('Asadābād', 'Asadabad', 34.8742, 71.1528, 1004251962, 'AF')]
```

[26]:
```
%%sql
SELECT * from F21W4111Midterm.people_modified
LIMIT 10
```

```
 * mysql+pymysql://root:***@localhost/F21W4111Midterm
   mysql+pymysql://root:***@localhost/db_book_midterm
   mysql+pymysql://root:***@localhost/lahmansbaseballdb
```

```
10 rows affected.
```

[26]:
```
[('aardsda01', 1981, 12, 27, 'CO', 'Denver', None, None, None, None, None, None,
'David', 'Aardsma', 'David Allan', 215, 75, 'R', 'R', '2004-04-06',
'2015-08-23', 'aardd001', 'aardsda01', datetime.date(1981, 12, 27),
datetime.date(2004, 4, 6), datetime.date(2015, 8, 23), None, 'USA'),
 ('aaronha01', 1934, 2, 5, 'AL', 'Mobile', None, None, None, None, None, None,
'Hank', 'Aaron', 'Henry Louis', 180, 72, 'R', 'R', '1954-04-13', '1976-10-03',
'aaroh101', 'aaronha01', datetime.date(1934, 2, 5), datetime.date(1954, 4, 13),
datetime.date(1976, 10, 3), None, 'USA'),
 ('aaronto01', 1939, 8, 5, 'AL', 'Mobile', 1984, 8, 16, 'USA', 'GA', 'Atlanta',
'Tommie', 'Aaron', 'Tommie Lee', 190, 75, 'R', 'R', '1962-04-10', '1971-09-26',
'aarot101', 'aaronto01', datetime.date(1939, 8, 5), datetime.date(1962, 4, 10),
datetime.date(1971, 9, 26), datetime.date(1984, 8, 16), 'USA'),
 ('aasedo01', 1954, 9, 8, 'CA', 'Orange', None, None, None, None, None, None,
'Don', 'Aase', 'Donald William', 190, 75, 'R', 'R', '1977-07-26', '1990-10-03',
'aased001', 'aasedo01', datetime.date(1954, 9, 8), datetime.date(1977, 7, 26),
datetime.date(1990, 10, 3), None, 'USA'),
 ('abadan01', 1972, 8, 25, 'FL', 'Palm Beach', None, None, None, None, None,
None, 'Andy', 'Abad', 'Fausto Andres', 184, 73, 'L', 'L', '2001-09-10',
'2006-04-13', 'abada001', 'abadan01', datetime.date(1972, 8, 25),
datetime.date(2001, 9, 10), datetime.date(2006, 4, 13), None, 'USA'),
 ('abadfe01', 1985, 12, 17, 'La Romana', 'La Romana', None, None, None, None,
None, None, 'Fernando', 'Abad', 'Fernando Antonio', 220, 73, 'L', 'L',
'2010-07-28', '2019-09-28', 'abadf001', 'abadfe01', datetime.date(1985, 12, 17),
datetime.date(2010, 7, 28), datetime.date(2019, 9, 28), None, 'DOM'),
 ('abadijo01', 1850, 11, 4, 'PA', 'Philadelphia', 1905, 5, 17, 'USA', 'NJ',
'Pemberton', 'John', 'Abadie', 'John W.', 192, 72, 'R', 'R', '1875-04-26',
'1875-06-10', 'abadj101', 'abadijo01', datetime.date(1850, 11, 4),
datetime.date(1875, 4, 26), datetime.date(1875, 6, 10), datetime.date(1905, 5,
17), 'USA'),
 ('abbated01', 1877, 4, 15, 'PA', 'Latrobe', 1957, 1, 6, 'USA', 'FL', 'Fort
Lauderdale', 'Ed', 'Abbaticchio', 'Edward James', 170, 71, 'R', 'R',
'1897-09-04', '1910-09-15', 'abbae101', 'abbated01', datetime.date(1877, 4, 15),
datetime.date(1897, 9, 4), datetime.date(1910, 9, 15), datetime.date(1957, 1,
6), 'USA'),
 ('abbeybe01', 1869, 11, 11, 'VT', 'Essex', 1962, 6, 11, 'USA', 'VT',
'Colchester', 'Bert', 'Abbey', 'Bert Wood', 175, 71, 'R', 'R', '1892-06-14',
'1896-09-23', 'abbeb101', 'abbeybe01', datetime.date(1869, 11, 11),
datetime.date(1892, 6, 14), datetime.date(1896, 9, 23), datetime.date(1962, 6,
11), 'USA'),
 ('abbeych01', 1866, 10, 14, 'NE', 'Falls City', 1926, 4, 27, 'USA', 'CA', 'San
Francisco', 'Charlie', 'Abbey', 'Charles S.', 169, 68, 'L', 'L', '1893-08-16',
'1897-08-19', 'abbec101', 'abbeych01', datetime.date(1866, 10, 14),
datetime.date(1893, 8, 16), datetime.date(1897, 8, 19), datetime.date(1926, 4,
27), 'USA')]
```

[27]:
```
%%sql
SELECT birthCountry FROM F21W4111Midterm.people_modified
LIMIT 10
```

```
 * mysql+pymysql://root:***@localhost/F21W4111Midterm
   mysql+pymysql://root:***@localhost/db_book_midterm
   mysql+pymysql://root:***@localhost/lahmansbaseballdb
(pymysql.err.OperationalError) (1054, "Unknown column 'birthCountry' in 'field
list'")
[SQL: SELECT birthCountry FROM F21W4111Midterm.people_modified
LIMIT 10]
(Background on this error at: https://sqlalche.me/e/14/e3q8)
```