Assignment #2

CS 348 - Winter 2025

Due : 11:59 p.m., Fri, Feb 14, 2025

Appeal Deadline: One week after being returned

Submission Instruction

This assignment consists of 2 parts. Part I will be submitted through Crowdmark. See the website for more detailed instructions. In particular, do not forget to submit one file per question to make the lives of TAs easier. Part 2 is a programming question and requires submission of files through Marmoset.
For Part 1: You don not have to type the questions if you are using Latex. Just specifying the question number is good enough. You do not need to keep the same font style. Here is a draft/empty latex template you may download and use: `https://www.overleaf.com/read/pxmgghkqksvg#39ea01`. Handwritten solutions and scanned to pdf are also acceptable.

Part I

This part consists of 3 questions covering topics on SQL querying, SQL constraints, and functional dependencies.
*Important Note on SQL Questions: Unless otherwise specified in a particular question, you can only use the SQL clauses that appear in the *lectures* (including slides). For example, do not use clauses like COALESCE or IFNULL. Our goal is *not* to make the lives of students who are experienced in SQL harder. Limiting to what we covered in lectures forces you to stay closer to the core relational algebraic way of programming, which is a learning goal of this assignment. Do not worry that using a clause that has been used infrequently in the lectures might lead to TAs taking off marks unfairly and do not double check with the teaching staff whether you can use a clause or not. The rule is simple: if a clause was used in the lectures (including slides), you can. Otherwise you can't.*

Question 1.

[10 marks in total] Suppose we are given a database of flights, `Flights(src, dst, cost)` that records each flight of an airline from a city src to a city dst and the cost (there can be multiple flights from src to dst). You can assume `src` and `dst` columns are strings and `cost` is an integer. We will assume that the graph we would get if we modeled each record in this table as an edge is a directed acyclic graph (DAG). Suppose we are interested in computing the cheapest flights from source "Waterloo" to every other reachable destination from Waterloo. One approach is to compute a recursively defined relation `AllPaths(src, dst, cost)` using WITH RECURSIVE and then select the minimum cost flight where `src` is "Waterloo". This code would look as follows:

```
WITH RECURSIVE
    AllPaths AS (
        (SELECT * FROM Flights)
        UNION
        (SELECT AllPaths.src, Flights.dst, AllPaths.cost + Flights.cost as cost
         FROM AllPaths, Flights
         WHERE AllPaths.dst = Flights.src)
         )
SELECT dst, min(cost) FROM AllPaths WHERE src = "Waterloo" GROUP BY dst
```

Although correct, as the fixed point computation executes, this query can generate very large intermediate relations for `AllPaths`. For example, on the following input graph, with 100 nodes and $100k$ edges, it may generate intermediate relations with size in the order of $\Theta(k^{100})$, because the number of paths from v_1 to v_{100} is in that order (assuming the cost of these paths are all different from each other).
Write a much more efficient recursive query that will compute shortest paths from Waterloo. Specifically, given the relational representation of a DAG G with n nodes and m edges as input, your query: (i) should

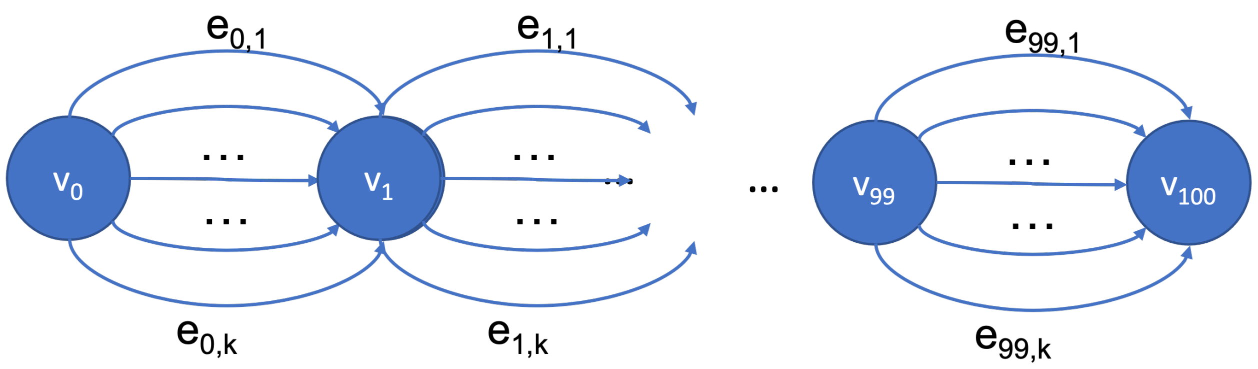


Figure 1: Example DAG with very large number of paths.

not generate an intermediate relation with size more than n during the fixed point computation for any of the recursive relations that it computes (i.e., for any of the iterations of the fixed point computation); and (ii) should not run more than $d+1$ iterations, where d is the depth of G. Queries that do not meet these bounds but improve over the above query will get partial marks. Your query may be non-monotone and use SQL clauses from the lectures that implement set operations other than the core relational algebra operations, as long as it converges. Explain why your query will not generate intermediate results larger than n.
[Note: Follow the definition of WITH RECURSIVE tables from the lecture. Recall that in the lecture we defined the queries definition temporary recursive tables to be arbitrary queries. That is, in the temporary table definition "WITH RECURSIVE Tmp AS Q", Q is an arbitrary query that can reference Tmp and Tmp is defined as the fixed point of Q seeded with $\text{Tmp}_0 = \emptyset$. Also, treat this question as a "pen and paper" question. Most SQL systems in practice will likely given parsing errors on your queries because of various reasons, such as not allowing aggregation functions.]
[Hint: Consider a query that at iteration i computes the shortest paths from Waterloo to each (reachable) destination with at most i edges.]

Possible solution : We will need to apply 2 optimizations to meet the goal of keeping the intermediate relations bounded by size n: (i) we need to only compute paths from Waterloo as part of the intermediate relations; (ii) we need to only compute shortest paths from Waterloo to all other nodes in the recursive query instead of all paths from Waterloo. The query is shown below. `CPsWaterloo` below stands for **c**heapest **p**aths from Waterloo. The invariant of $\texttt{CPsWaterloo}_i$ for $i = 1, ..., d$ is that it contains the cheapest paths to each reachable destination with at most i edges in it. Since G is a DAG, assuming this invariant is true, the fixed point computation can run at most d iterations and will converge (because the shortest paths will converge). Note that the query is not monotone w.r.t. `CPsWaterloo`. It is possible during the fixed point computation, when moving from iteration i to $i+1$ that if we find a cheaper flight to a location L, then the previous cheapest path can be deleted. But since the depth of the graph is d, iteration $d+1$ cannot improve the shortest paths.

Let us focus on the inner query and explain why the invariant is true: The inner query has the standard $Q_B \cup Q_R$ template for recursive queries and at iteration i of the fixed point computation, the query computes the union of Q_B: cheapest 1-edge paths from Waterloo to each destination (i.e., that is reachable by 1 edge) and Q_R computes the cheapest paths that have between 2 and i-edges. So taking the union of these with a following group by and aggregate on `dst` values gives the shortest paths to each reachable destination with at most i edges. Note that the invariant holds at iteration i=1 and by the above argument will remain to be true at all later iterations. You did not need to argue the correctness at this detail. These details are provided to better explain the correctness of the query. (Markers did not take points off for not arguing for correctness.)

Finally note that by construction, the query cannot generate more than n tuples at any iteration because we are computing a group by on one of the columns, which contain different locations (nodes) and we are assuming that there are n of them.

```
WITH RECURSIVE
    CPsWaterloo AS
    (
        SELECT dst, min(cost) AS minCost
        FROM (
            (SELECT dst, min(cost) as minCost
             FROM Flights
             WHERE src = "Waterloo"
             GROUP BY dst)
            UNION
            (SELECT F.dst, min(C.minCost + F.cost) as minCost
            FROM CPsWaterloo C, Flights F
            WHERE C.dst = F.src
            GROUP BY F.dst)
            )
        GROUP BY dst
    )
SELECT dst, minCost FROM CPsWaterloo
```

Question 2.

[16 marks in total] Consider a database schema for a bookstore described as follows:

Author(aid, name, contact_info)
Book(ISBN, name, genre, price, copies, pid)
  FK: pid references to Publisher
Writes(aid, ISBN)
  FK: aid references to Author
  FK: ISBN references to Book
Publisher(pid, name)
Customer(cid, name)
Sale(sid, date, total_price, cid)
  FK: cid references to Customer
SaleDetails(ISBN, sid, quantity)
  FK: ISBN references to Book
  FK: sid references to Sale

Note that the quantity attribute in SaleDetails indicates the number of copies of a book sold in a particular sale. Based on the given schema and its constraints, answer the following questions.

(a) (4 points) Answer the two related questions.

1. Write a DDL query (CREATE TABLE) to create the `SaleDetails` table incorporating the specified constraints. Assume ISBN is of type string and SID and quantity are of type integer.

   Possible solution :
   ```
   CREATE TABLE salesDetails
       (ISBN varchar(ANY_NUM) or char(ANY_NUM)),
        sid INT, quantity INT,
        PRIMARY KEY (ISBN, sid),
        FOREIGN KEY (ISBN) REFERENCES Book(ISBN),
        FOREIGN KEY (sid) REFERENCES Sale(sid))
   ```

2. After creating the table using the above command, the bookstore decides to add a constraint on the number of books purchased in each sale. The bookstore wants to allow only those sale entries that have a minimum quantity of 1 book and a maximum quantity of 100 books. Write an SQL query to add this constraint on the `SaleDetails` table.

   Possible solution :
   ```
   ALTER TABLE salesDetails
       ADD CONSTRAINT quanity_check CHECK(quantity > 0 AND quantity <= 100)
   ```

(b) (6 marks) Write a SQL query to generate a 'master' relation consisting of the *aids, author names, ISBNs, book names* and *book copies* listing all the books written by authors where a book has at least 1 copy. The output of your query should also contain: (i) author details who have not written any books or written books that all have 0 copies (with null book details); and (ii) books with at least 1 copy whose author details are absent in the `Author` table (with null author details)."

Possible solution :
```
WITH tempTb AS (
    SELECT aid, books.isbn, name as bname, copies
    FROM writes RIGHT OUTER JOIN books ON
        writes.isbn = books.isbn
    WHERE copies > 0)
SELECT authors.aid, name, isbn, bname, copies
FROM tempTb FULL OUTER JOIN authors ON
tempTb.aid = authors.aid
```

(c) (6 marks) For each publisher that has sold at least two of its (distinct) books, select the name and the total quantity of the highest sold book. If a publisher has two (or more) books with the same highest sales quantity, either of the books can be selected.

Possible solution :
```
WITH temp AS (
    SELECT name, qsum, pid
    FROM (SELECT isbn, sum(quantity) AS qsum
            FROM saledetails
            GROUP BY isbn) NATURAL JOIN books)
SELECT name, maxQSum
FROM temp, (SELECT pid, MAX(qsum) as maxQSum
```

```
                    FROM temp
                    GROUP BY pid
                    HAVING COUNT(*) > 1) as pidMax
WHERE temp.pid = pidMax.pid AND qsum = maxQSum
```

Possible solution : – Alternate solution

```
WITH temp AS (
    SELECT name, qsum, pid
    FROM (SELECT isbn, sum(quantity) AS qsum
              FROM saledetails
              GROUP BY isbn) NATURAL JOIN books)
SELECT name, qsum
FROM temp t
WHERE qsum IN (SELECT max(qsum)
                    FROM temp
                    WHERE pid = t.pid
                    GROUP BY pid
                    HAVING count(*) > 1)
```

Question 3.

[7 marks in total]

(a) (3 marks) Argue why every binary relation R, i.e., one with only two attributes R(A, B) is in BCNF.

[Although we were not explicit in class about this, when defining fds, you can assume that in any fd $\mathcal{X} \longrightarrow \mathcal{Y}$, $\mathcal{X}$ (and $\mathcal{Y}$) is not the empty set (for your own interest, think about what the natural definition of a fd with the empty set on the left would mean).]

Possible solution : There are two non-trivial possible fds that can exist in R. $A \longrightarrow B$, and $B \longrightarrow A$, but in both cases, this would imply that the set of attributes on the left is a key.

(b) (4 marks) Given a set of attributes $\mathcal{A}$ and a set of fds F, the closure $\mathcal{A}^+$ of $\mathcal{A}$ is the set of all attributes B such that every relation R that satisfies F satisfies $\mathcal{A} \longrightarrow B$. Given F and $\mathcal{A}$, the following is the standard algorithm for computing $\mathcal{A}^+$:

```
AttributeClosure($\mathcal{A}$, $F$)
result: $\mathcal{A}$
repeat
   foreach fd $\mathcal{X} \longrightarrow Y \in F$:
      if $\mathcal{X} \subseteq result$ then result:  result $\cup$ $Y$
until result does not change
```

Note that the algorithm considers the fds in F on the foreach line and not F^+ (for your own sake, you can try to prove why the algorithm is correct simply by considering the fds in F). This standard algorithm has several applications, such as finding if a set of attributes is a key.

Given the following fds on a relation with schema $R(A, B, C, D, E)$:

- fd$_1$: $E \longrightarrow AB$
- fd$_2$: $BD \longrightarrow C$
- fd$_3$: $A \longrightarrow D$
- fd$_4$: $C \longrightarrow E$

(i) Compute the closure of A.

(ii) Compute the closure of E.

Show your derivation for both (i) and (ii). Argue in one or two sentences which one of A or E can be declared as a primary key on R and why?

Possible solution : (i) Closure of A: applying fd3 we can add D to *result* but no further fds can be applied. So the closure of A is AD.
(ii) Closure of E: first apply fd1 to add AB to *result*. Then we can apply fd3 to add D. *result* at this point is $ABDE$. We can then apply fd2 to add C, so result becomes $ABCDE$ and we don't need to search for more fds to apply because we see that E implies all attributes.
Since E implies, i.e., determines, all attributes, by definition it is a key.

Question 4.

[16 marks in total] Given the relation $R = ABCDEF$, with the set of dependencies:

$$F = \{AB \to CE,\ ACD \to B,\ BE \to CF,\ AC \to DE,\ BF \to A\},$$

answer each of the following questions. **Note**: When solving each task you may use the information already obtained from the tasks that preceded it.

a. (4 points) List all candidate keys for R (as groups of attributes separated by comma, e.g. A, BC).

b. (4 points) Derive a minimal cover for F. List the set of dependencies obtained in each step of the minimal cover algorithm.

c. (4 points) Determine if a decomposition $R_1 = ABC$, $R_2 = ACD$, $R_3 = ACEF$ preserves the dependencies $AB \to E$ and $BE \to F$. Explain your answer.

d. (4 points) Given $R_2 = ABCDE$ and $F_2 = \{A \to B,\ BC \to E,\ ED \to A\}$, derive a BCNF lossless-join decomposition of R_2. Show each step of the algorithm.

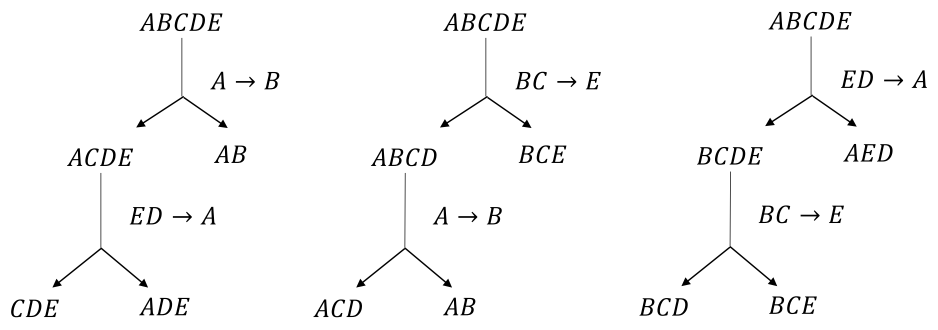
Possible solution :

a. AB, AC, BE, BF. One way to identify this is to come up with attribute closures of each attribute and see what minimal combinations are needed to come up with candidate keys.

b. Solution is not unique. One possible solution
Step-1: $F_1 = \{AB \to C,\ AB \to E,\ ACD \to B,\ BE \to C,\ BE \to F,\ AC \to D,\ AC \to E,\ BF \to A\}$
Step-2: $F_1 = \{AB \to C,\ AB \to E,\ AC \to B,\ BE \to C,\ BE \to F,\ AC \to D,\ AC \to E,\ BF \to A\}$
Step-3: $F_1 = \{AB \to E,\ AC \to B,\ BE \to C,\ BE \to F,\ AC \to D,\ BF \to A\}$

c. $AB \to E$ is preserved. Because from R_1 we can conclude $AB \to AC$, and from R_3 we can conclude $AC \to E$. By transitivity, $AB \to E$ is preserved. $BE \to F$ is not preserved. Because $BE \cap R_1 = B$, $BE \cap R_2 = \emptyset$, $BE \cap R_3 = E$, and $B^+ = B$, $E^+ = E$, therefore, we cannot conclude $BE \to F$ from current decomposition without joining tables.

d. Muliple decomposition choices. Depending on the sequence of applying FDs to decomposition. We show some sample solutions below:



Part II

This part consists of one programming question. We have provided several starter files which you can find on Learn. These files are under A2/EnrollmentAppData.zip file.

Submission

You need to submit your assignment via `https://marmoset.student.cs.uwaterloo.ca/`.

Marmoset will automatically test and mark your code. **IMPORTANT: Your submission must be a single zipped folder which includes only your `EnrollmentApp.java` file**. This means that when you unzip the folder you should see the `EnrollmentApp.java` file. Ensure that the file is named exactly `EnrollmentApp.java`.

Each question has only 1 public test and many private tests. You will only see the results of the public test (whether or not you passed or failed the public test).

IMPORTANT: the public tests do not cover every aspect of the question - they are intended only for you to check that your submission went through correctly and that you do not have any obvious problems. Passing a public test does not guarantee that you pass the private tests.

To receive full marks your code must pass all of the tests. Therefore, it is important that you test your code yourself.

To pass a test the output returned by your query must match the expected output. The row/tuple order however matters if the question asks for a specific order (otherwise it does not matter).

You can re-submit until the deadline and your best submission will be used for the final grade. Do not be alarmed if marmoset is slow, marmoset can often take a rather long time to run and grade your submission.

NOTE: If you fail a public test you will be able to see your output compared to the expected output. The output that you see on marmoset does not include the inputs to the program so you might see something like a prompt for input that is immediately followed by the program output. This formatting is just an artifact of marmoset and can be ignored.

Setup

For this question, you will be using the DuckDB JDBC API. You can find the documentation for the DuckDB JDBC API at: `https://duckdb.org/docs/api/java.html`. This documentation includes examples and descriptions of the various API calls.

Similar to Assignment 1, you can use DuckDB locally on your own machine or you can use the university's servers located at: linux.student.cs.uwaterloo.ca. You can remotely connect to these machines via ssh. If you are connecting from off campus you will need to have already setup ssh keys on the server. You can find detailed information about these servers here:
`https://uwaterloo.ca/computer-science-computing-facility/teaching-hosts`.

It is important to note that we cannot guarantee the uptime of the school's machines so we encourage you to install DuckDB on your own machine.

If you choose to use your own local machine you will need to ensure that you have a valid Java installation. Ensure that you have downloaded and installed the Java Development Kit. You can download the Oracle JDK here: `https://www.oracle.com/java/technologies/downloads/`. The following assumes that you have already installed the Java Development Kit.

The starter files under A2/EnrollmentAppData.zip include some sample test data, the DuckDB JDBC API jar file, SQL scripts for creating and populating the necessary tables (i.e., we provide the schema definitions) and a starter `EnrollmentApp.java`. This question has 3 parts which correspond to 3 empty methods in the `EnrollmentApp.java` file. Your task is to edit the three empty methods in the `EnrollmentApp.java` file to solve the problems described in Question 4.

IMPORTANT: do **NOT** change any of the existing code in the `EnrollmentApp.java` - you should only be adding code in the empty methods.

IMPORTANT: for the commands described below to work properly, all of the starter files must be in the same directory. So, make sure all of the starter files are in the same directory.

Note that the name of the persistent DuckDB database is encoded in the starter `EnrollmentApp.java` file. Specifically, the name of your persistent DB is `cs348`. This is relevant since you should setup the proper schema and populate the tables with test data before you work on the methods you need to complete for the assignment. When testing your code, make sure you setup the database schema (create the tables) BEFORE you run your java file. If you do not setup setup the database schema before running your Java file, you will receive runtime errors whenever you attempt to access a table.

We have provided a `makeTables.sql` file which you can use to setup the tables. To do this, you must execute the following command: `$ duckdb cs348 < makeTables.sql`. We have also provided some sample data and a `populateTables.sql` script to add the data to the database. To add the sample data execute the following command: `$ duckdb cs348 < populateTables.sql`. The sample test data is intended to help you get started quickly but it is not guaranteed to test all possible scenarios. It is up to you to sufficiently test your code - this will likely involve adding additional test data.

Note that the schema for this assignment is slightly different to the one from Assignment 1. You should review the `CREATE TABLE` statements in the `makeTables.sql` file and ensure that you understand the schema.

To run the java file execute the following command:

```
$ java -cp duckdb_jdbc-1.1.3.jar EnrollmentApp.java
```

This ensures that the DuckDB JDBC API jar is included in the classpath. You should test that running this command works without errors. Running the starter `EnrollmentApp.java` file will display a menu-style input prompt to stdout. The menu options correspond to each part of Question 4.

You are now ready to fill in the empty methods in `EnrollmentApp.java` according the questions below.

Question 5.

[18 marks in total]

Consider the incomplete EnrollmentApp.java class structure given with this assignment, where questions (a) to (c) require writing some code necessary to achieve some task described below. When you implement the following tasks, pay close attention that at the end of your program the database connection is closed as well as all the statements and result sets that were used.

(a) **[6 marks]** Print to `std` a list of classes that a student has been enrolled in with an ID number supplied as an integer argument to the command line. Print one class per row and the section number the given student is enrolled in next to each class (separated by tab "`\t`"). Sort the output on class name. Write your code inside the getClassByStudentID function.

E.g. getClassByStudentID(112348546), you print:

```
Database Systems 1
Operating System Design 1
```

Note: *You can assume a student cannot be enrolled in classes with the same name but different section.*

(1) **Errors** If the given ID is invalid (i.e., does not exist for student), return an error message "[Error] Invalid ID".

(2) **Warnings** If the ID is valid but the corresponding student is not enrolled in any class, return a warning saying "[Warning] No enrollment".

(3) **Time-conflicting class** If there are time-conflicting (same meeting time) classes, first print "[Warning] Conflicting classes", then print class name and meeting time (separated by tab "`\t`") for conflicting classes only. Sort the output first on meeting time then on class name.

E.g. getClassByStudentID(301221823), you print:

```
American Political Parties 1
Multivariate Analysis 1
Perception 1
Social Cognition 1
[Warning] Conflicting classes
American Political Parties TuTh 2-3:15
Multivariate Analysis TuTh 2-3:15
```

(b) **[6 marks]** Given a student ID number and a list of class names separated with comma as arguments on the command line, print to `std` in alphabet order the list of student names who have been enrolled in all the given classes as the given student has (including the given student him/herself);

E.g., searchCommonClassmate(112348546, "Database Systems,Operating System Design"), you print:

```
Ana Lopez
Christopher Garcia
Joseph Thompson
Lisa Walker
```

Write your code inside the searchCommonClassmate function.

Note 1: Note that on the command line, the classes are given without any quotation mark but your searchCommonClassmate function will be called with a single string argument that contains all classes separated with comma.

Note 2 (Empty class list): *If no class names are given and the class list is empty, then you should interpret this as the class name list containing all the classes the given student is enrolled in. That is, empty class name is a shortcut to specify all classes the given student is in.*

(1) **Errors**
  - Check if provided student ID is valid, i.e., exists in the database. If not, print the error message “[Error] Invalid ID”.
  - Check if provided class names are valid, i.e. exists in the database. If not, print the error message “[Error] Invalid class”.
  - If a student ID is valid and the class list is not empty but the corrsponding student in not enrolled in one (or more) of the classes in the list, return an error saying “[Error] Invalid enrollment”.

(2) **Warnings**
  - If a student ID is valid and but an empty class list is given, but the corresponding student is not enrolled in any class, return a warning saying “[Warning] No enrollment”. Recall that an empty class list is a shortcut to represent all classes the student is enrolled in.

(3) **Multi-section** If a class has multiple sections, only students from the same section are considered as classmates;

(c) **[6 marks]** Given a course name as a string on the command line, print to `std` the majors of enrolled students and the count of students from each major, respectively. Result should be sorted by major name. One major per row;

E.g., getClassStatis(“Database System”), you print:

```
Computer Engineering 1
Computer Science 4
```

(1) **Errors** Check if provided class name is valid (exists in the database). If not, print the error message “[Error] Invalid class”;

(2) **Multi-section** If the input class has multiple sections, only print the aggregated major distribution from all sections;

(3) **Undecided major** If some student’s major is not decided yet, i.e., the `major` column in `Student` table is `NULL`, such students can be printed under a major “TBD”. The major “TBD” should be sorted with regular majors.