CS 605.641 Principles of Database Systems
Summer 2025 Final Examination
Dates: 8/3/2025 – 8/6/2025
No discussion, collaboration, consultation, sharing, and Gen AI

Name: Nicolas Escudero

Please answer all questions

Answer the following five database-related questions: (50 points; 10 points each problem)

1. **Explain the major challenges you may encounter when you implement an EER with a superclass/subclass relationship. (3 points)**

When considering the implementation of a subclass/superclass relationship, one of the main challenges that designers will often face is the issue of complexity. In many instances, designers must choose whether or not it is necessary to have the subclass/superclass relation. If a subclass was defined but contains very few attributes that are local to that subclass, as well as has no specific relationships to other entities within the subclass, then it would be wiser for this subclass to simply merge into its superclass, instead of keeping this as a separate relationship. Keeping the subclasses would create redundancy issues, and make the overall design more cluttered. The same concept applies to subclasses of a specialization/generalization type; if it lacks specificity, it would be better to merge into the superclass as a type attribute rather than an entirely new subclass.

Another issue with implementation of subclass and superclass relationships in general is translating these relationships into an appropriate database tool. Many current relational tools, such as MySQL and PostGreSQL, may not currently have full functionalities of EERD designs. Although there are plenty of ways to implement work-arounds to mimic he characteristics of subclasses and superclasses, this may be more difficult and time consuming, depending on the complexity of the relationship. Because of this, designers should consider if having the subclass/superclass relation is necessary for representing the database in an efficient manner.

Explain 2NF remedy what type of problems in a non-normalized relation. (2 points)

A relation is considered to be in 2NF form from two determining factors:

- The relation is already in 1NF, or the characteristic that all underlying domains contain single-valued, atomic values. No sets of values are defined in any of the columns, and no repeating groups are found in the table.
- The relation's non-key attributes are fully functionally dependent on all parts of the table's primary key. No partial dependencies should exist in this table.

Partial dependencies are common for tables that contain a composite key, or a table that contains two attributes that are both needed to uniquely define a tuple. Relations that have partial dependencies may make the database perform less efficiently, depending on the use case. With 2NF normalization, certain normalization strategies can help remedy the redundancy that partial dependencies could introduce and make querying more efficient. One method to make a relation into 2NF is to make a new table that contains the partiallydependent attribute and whichever partial composite of the primary key that the attribute depended on. This way, the new table will have the attribute depend on only that primary key, thereby eliminating the partial dependency.

Explain 3NF remedy what type of problems in a non-normalized relation. (2 points)

A relation is considered to be in 3NF based off of two criteria:

- The relation is already at least in 2NF, as previously described.
- The relation contains no transitive dependencies. These are dependencies amongst attributes where a non-key attribute functionally depends on another non-key attribute that then depends on a primary key, rather than directly depending on a primary key.

Relations that are in 3NF must eliminate these transitive dependencies to ensure unproblematic functional dependencies throughout the relation, as well as keep the database more organized for easier querying. A method for normalizing a relation into 3NF includes splitting the table in two separate relations, each of which contain one of the non-primary key attributes that were in the original transitive dependency. One of the new tables will have a non-primary key attribute be the new primary key of that table, while the other new table will now have the other non-key attribute functionally dependent on the primary key.

Explain the relationship from an original relation and a new relation in 1NF, 2NF, and 3NF process respectively. (3 points)

For a 1NF relationship:

- An original relationship in this case is unnormalized, likely containing non-atomic values and columns that are multi-valued. A 1NF relation would separate these multi-valued columns in the original relationship into single-valued columns, and each instance of a tuple would now have only one value in each column. Here's an example:

ORIGINAL

Student_no	**Student_name**	**Course_id**	**Course_name**
1	Tyler	1,2,3	Physics, Music, Biology
2	Frank	1,2	Physics, Music

FIRST_NORM

Student_no	**Student_name**	**Course_id**	**Course_name**
1	Tyler	1	Physics
1	Tyler	2	Music
1	Tyler	3	Biology
2	Frank	1	Physics
2	Frank	2	Music

CS 605.641 Principles of Database Systems
Summer 2025 Final Examination
Dates: 8/3/2025 – 8/6/2025
No discussion, collaboration, consultation, sharing, and Gen AI

Name: Nicolas Escudero

For a 2NF relationship:

- The relationships contain non-key attributes and composite primary-key attributes, and a functional dependence exists between the two. The original relationship would contain partial dependencies between the non-key and one part of the composite primary-key; that is, not all composite parts of the primary-key are needed to uniquely identify the non-key attribute. A relationship that is in 2NF would separate these partial dependencies by creating a new relationship table, where the non-key attribute and its partial dependency on the primary-key part are migrated to the new table. This makes each table have its attributes depend on its primary key in its entirety. Here's an example:

ORIGINAL

Student_no	**Course_id**	**Student_name**	**Course_name**
1	2	Tyler	Music
1	3	Tyler	Biology
2	1	Frank	Physics

Student_no and Course_id are composite key. Course_name attribute depends only on Course-id, and Student_name only deepends on Student_no, breaking 2NF.

Instead, we should have this:

SEC_FORM_S

Student_no	**Student_name**
1	Tyler
1	Tyler
2	Frank

SEC_FORM_C

Course_id	**Course_name**
2	Music
3	Biology
1	Physics

Now each table has an attribute that depends on the entirety of a primary key.

For a 3NF relationship:

- The original relationship contains non-key attributes that are functionally dependent on other non-key attributes, and not on its primary key. These are called transitive dependencies, and they may cause redundancies in querying. The 3NF relationship table removes these non-key to non-key dependencies to ensure that all functional dependencies come from only the primary key of the table in its entirety. Here's an example of this:

ORIGINAL

Student_no	**Student_name**	**Lab_id**	**Lab_name**
1	Tyler	3	MicroBiology
2	Frank	4	Bioinformatics

Here, Lab_id is Functionally dependent to Student_no, and Lab_name is functionally dependent on Lab_id. This is not allowed in 3NF, since Lab_name depends on Lab_id for uniqueness, not Student_no.

In 3NF, it would look like this:

THIRD_FORM_S

Student_no	**Student_name**	**Lab_id**
1	Tyler	3
2	Frank	4

THIRD_FORM_L

Lab_id	**Lab_name**
3	MicroBiology
4	Bioinformatics

2. **In an ERD, there are two tables, EMPLOYEE and TASK, with a many-to-many relationship. A business rule requires that each employee must work on three (minimum) to six (maximum) tasks as one of the database business requirements. Please explain how to enforce minimum and maximum participation constraints of this business requirement at the backend (at the DB level). (3 points)**

Constraints on an ERD can be integrated during the DB level of design and implementation. Minimum and maximum participation constraints can be implemented in the form of triggers that sets the range of acceptable tasks that one instance of employee can have at a time. To demonstrate, consider the tables EMPLOYEE with primary key **Essn** and a TASKS with a primary key **Task_no.**

- Within the query, the user can define a trigger that counts how many task instances are already inserted for one employee. These can be in the form of a BEFORE INSERT trigger on the Tasks.
- The main part of the trigger will have specifications for the count. This uses SELECT, COUNT INTO on the Task_no attribute of the TASKS table, and a WHERE clause to relate the Tasks_no count to the associated Essn in the EMPLOYEE table.

- Specify the trigger's constraints using IF-THEN statement. Define it where the count obtained from the task is between 3 <= count <= 6. If it meets this criteria, insertion can be performed. If not, an error will generate and the task is stopped.

In this sense, every new TASK insertion will initiate the trigger. Within the trigger, a count on the Task_no is performed, and the result of the count is retrieved and related to the Essn that each Task_no is related to. If a count is outside the acceptable range of 3-6 for the Essn that the INSERT is trying to relate to, then it will not perform the insertion.

Explain any issues when you have enforced the above business requirement in the production database. (2 points)

One issue that came up when implementing this trigger is originally using a CHECK constraint instead of an IF-THEN statement within the trigger. In this case, the user should avoid using a CHECK constraint, as these constraints are useful only on a single row scenario. For this, we need to check the constraint across multiple tables, TASK and EMPLOYEE, so CHECK cannot be used in this case.

Another issue that was addressed was efficiency. If we used the trigger to check the task_count for every Essn, then the program would have to check and count the task across every unique Essn in the database. If the database has a large table of Essn's, such as if a department has a large number of employees, then counting the tasks assigned for each Essn may slow down the query.

Consider the following relation

STUDENT_OFFERING_GRADE

Student_id	Offering_id	Grade

The business requirement is to track a student taking a particular Offering_id and its grade. The PK is (Student_id, Offering_id, Grade). Please explain any design problems. (2 points)

First, it is important to note that this database having multiple primary keys is completely valid. Each tuple instance can be uniquely defined by these three attributes with no violation to uniqueness. For example, the same student can take a different offering, and the same student can also have the same Grade for different offerings:

Student_id	Offering_id	Grade
001	003	A
001	004	A
002	003	A

These are all unique instances of the composite primary key, and all valid tuples.

Given this, the design is not necessarily practical for the case that the business wants to track. It is simply tracking Students, which Offering they are taking, and what Grade they got in the Offering. It is leaving the rest of the assumptions very open-ended, such as:

- What if the same Student wants to retake the same Offering, and manages to obtain the same Grade as the last time they took it? This would make an invalid entry.
- Does each Offering go based off the same Grading scale? What if an Offering were Pass/No Pass, and would this be represented in the same Grade column?
- If a Student cannot finish an Offering, what Grade would they get since the Grade column is a PK and must be defined/NOT NULL?
- What are the relationships between these columns? Typically, an Offering can obtain any Grade to it, while one Grade must be associated with one and only one Offering_id (M:1).

Given these criteria, it would make more sense for the Grade column to be represented as not a PK, but more as a non-key attribute that is functionally dependent on the two other PKs. In this way, the Grade attribute can be specified as NULL if needed, and this would work for the event that a Student cannot take the same Offering twice.

If it was specified that the Student can retake the same Offering, then it would be better to keep the Offering column as its own non-key attribute as well. This way, both Offering_id and Grade can depend on the primary key of Student_id, and addresses much of the open-ended assumptions for greater flexibility.

In terms of normalization, having only the Student_id as a primary key also allows the relationship to be in 3NF, a highly normalized form that has no partial dependencies or transitive dependencies.

CS 605.641 Principles of Database Systems
Summer 2025 Final Examination
Dates: 8/3/2025 – 8/6/2025
No discussion, collaboration, consultation, sharing, and Gen AI

Name: Nicolas Escudero

Give a relation R = {A, B, C, D, E, F, G, H, I, J} where A and B are joined PK.
R has the following Functional dependencies (FDs):
{A, B} → {J};
{A} → {H, I};
{B} → {G};
{G} → {E, F};
{H} → {C, D}

Use the functional dependency rules to identify and correct two incomplete FDs and explain your reason(s). You do NOT need to perform normalization. (3 points)

Functional dependency rules for a primary key, whether joined or a single column, states that the primary key functionally determines all other non-key attributes of the table. In this case, the joined primary key {A,B} must therefore functionally determine all non-key attributes, and all non-key attributes must be functionally dependent on all parts of the primary key. This rule alone is violated in two of the relations above: {A} -> {H,I} and {B} -> {G}.

For {A} -> {H,I} , this breaks the PK constraint since the subset {H, I} only depends on one of the composite primary keys, {A}, when it should depend on both A and B. It should instead look like: {A, B} -> {H, I}

For {B} -> {G}, this breaks the same PK constraint. It should instead look like this: {A, B} -> {G}.

3. **Consider the following relation:**
SUPPLIER_PART_PROJECT

Supplier #	**Part#**	**Project#**	**Quantity**

The business requirement is to track the quantity of a particular part from one particular supplier for one project. Based on the given primary key (Supplier#, Part#, Project#), please specify the relation is in (1NF, 2NF, 3NF, BCNF, 4NF, 5NF) compliant (multiple choices). Please explain your rationale (5 points).

Checking each normal form in order:

- First Normal Form: This is defined as a relation that contains no non-atomic values, and no repeating columns. The first three columns (Supply, Part, Project) are definitely atomic, as these are a part of the primary key and must have one unique value tied to each column per instance. The Quantity column, or the lone non-key attribute, in this case must be atomic, since this value can only be determined by one combination of primary keys per one instance. In other words, Quantity cannot have multiple values associated to it for one instance.

- Second Normal Form: This is defined as a relation that contains no partial dependencies, and that all non-key attributes must be fully dependent on all parts of a primary key. In this case, since Quantity is the only non-key attribute, this attribute can only exist if all parts of the primary key are defined. Quantity therefore depends on all parts of this primary key, obeying 2NF.

- Third Normal Form: This relation is defined as having no transitive dependencies within it. Since this relation contains only one non-key attribute, there are no non-key to non-key dependencies within the relation.

- BCNF: This relation is defined as having each non-key attribute as having to be dependent on a superkey. Since the three parts of the primary key re the only other attributes in this relation, these attributes are defined as the minimal superkey. Quantity therefore depends on an instance of a minimal superkey for every instance, obeying BCNF.

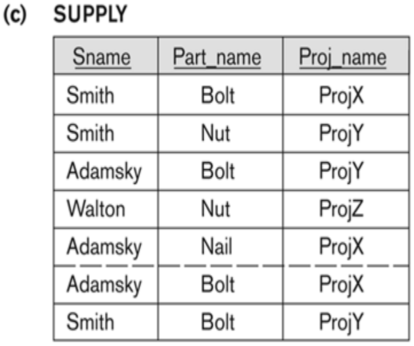
- Fourth Normal Form: These relations are defined as not having multivalued dependencies, or two independent functional dependencies from a primary key to two unrelated non-key attributes. Since Quantity is the only non-key attribute that functionally depends on the set of primary keys, this obeys 4NF.

- Fifth Normal Form: This is defined as the ability to have no nontrivial join dependencies. A join dependency is a constraint which states that, for a given relation R, this relation is equal to the join of its sub-relations R1, R2 … Rn. It shows that if R exhibits such behavior, then R can in fact be decomposed further into smaller sub-relations of itself. If no join dependency exists, then R is in 5NF form. If a join dependency does exist, then R can still be in 5NF if the join dependency is trivial, or if one of the sub-relations of R {R1, R2, R3} is equal to R and can't be further decomposed. For this relation, R can be decomposed into

CS 605.641 Principles of Database Systems
Summer 2025 Final Examination
Dates: 8/3/2025 – 8/6/2025
No discussion, collaboration, consultation, sharing, and Gen AI

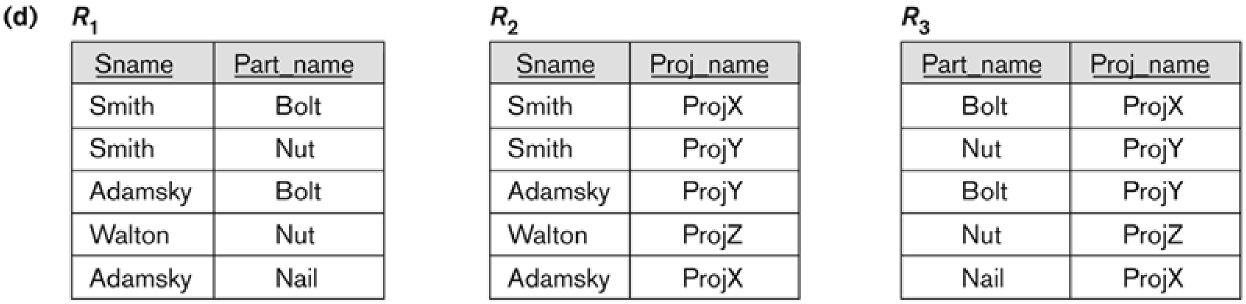
Name: Nicolas Escudero

One reason to perform normalization is to remove redundancy. Please explain why the relation SUPPLY below is normalized to 5NF, but the total size of relations R1, R2, and R3 below increases, not decreases. (5 points)



(c) SUPPLY

Sname	Part_name	Proj_name
Smith	Bolt	ProjX
Smith	Nut	ProjY
Adamsky	Bolt	ProjY
Walton	Nut	ProjZ
Adamsky	Nail	ProjX
Adamsky	Bolt	ProjX
Smith	Bolt	ProjY



(d) R_1

Sname	Part_name
Smith	Bolt
Smith	Nut
Adamsky	Bolt
Walton	Nut
Adamsky	Nail

R_2

Sname	Proj_name
Smith	ProjX
Smith	ProjY
Adamsky	ProjY
Walton	ProjZ
Adamsky	ProjX

R_3

Part_name	Proj_name
Bolt	ProjX
Nut	ProjY
Bolt	ProjY
Nut	ProjZ
Nail	ProjX

Despite the overall size of the sub-relations being larger than their original form, these tables are successful in removing redundancies. This is because of the nature of normalizing a relation into 5NF – to determine a relationship to be in 5NF, one must check for join dependencies by deconstructing the original relation into projections of the relationship. For this specific example, since a constraint defines these columns as all depending on each other in some form (supplier supplies parts, project uses parts, supplier supplies to project), these can be split into three separate relationship tables R1, R2, and R3, in which some of these tables have similar columns with one another. Doing so, however, puts the three new tables in 5NF, as these now can no longer be deconstructed further without resulting in a trivial relation.

The relation cannot be proven to be in 5NF without the ability to deconstruct into their appropriate projections, which requires at least two of the key attributes to initiate the JOIN clause.

CS 605.641 Principles of Database Systems
Summer 2025 Final Examination
Dates: 8/3/2025 – 8/6/2025
No discussion, collaboration, consultation, sharing, and Gen AI
Name: Nicolas Escudero

4. **Do you create an ordered file whose records are based on a primary key such as customer name and use a primary index on the attribute? Explain your reasons. (5 points)**

Ordered files are a type of file organization that provides structure whenever a User needs to access, insert/delete, or manipulate a database record. Files are physically ordered based on a criterion that is specified by the user, otherwise known as an ordering field. This ordering field can indeed be a primary key, and there are plenty of benefits for using a primary key with the associated primary index.

Ordering files allows for much more efficient searches of a specific record. Given the ordering field, the database can simply search for a record based on only criteria of the ordering field, and this search is already in an ordered format from the ordering field. This is much more beneficial when compared to unsorted files, which search for an entire file block by block until it finds whichever block the User is looking for. Search actions, as well as deletions, within an unsorted file prove to be much less efficient in these manners, hence why ordered files are typically preferred for efficiency.

A primary index on the ordered file is necessary when performing searches on an ordered file. The index is specified on the ordering key field of an ordered file, which then allows for physical ordering of the file based on the primary key attribute. Putting an index search on a primary key allows the user to define uniqueness in their search, which helps retrieve specific results much faster. Therefore, it would be useful to create an ordered file of the records of interest, and insert a primary index on the customer name primary key. Then, we can perform a binary search to easily sift through the ordered list, allowing us to find our record on interest without having to traverse through the entire database.

What is primary index? What is clustering index? What is secondary index? (3 points)

A primary index is an ordered sorting index that is based on ordering a record by its primary key. Doing an index on the primary key allows the search to specify uniqueness, as each value of a primary key has to be unique. Primary indexes, in terms of structure, contain a fixed length of two fields: one that contains the ordering key field (primary key) and one that defines a pointer to a file's block address.

A clustering index is similar in structure to the primary index; however, it is applied mainly for ordering files where the ordering field is a non-key attribute. This makes it so that indexing values does not maintain uniqueness, as values can be repeated for non-key attributes. This may be useful for efficiency in searches that need multiples of the same values.

A secondary index can be added to an ordered file where a primary index is already defined. These allow a user to expand the indexing criteria that is separate from the primary method They can be created on other parts of a candidate key of the relation, or it can be placed on a non-key attribute with repeating values like a clustering index.

Describe how to map an EER model into relation model for a specialization whose subclasses are overlapping. (2 points)

For EER diagrams containing overlapping, non-disjointed subclasses, two methods can be proposed to accurately exhibit the subclass relations into a relational model.

1. The user can inherit a multi-table method. First, relation tables are created for each subclass (and a relation is created for a superclass too, if needed). The attributes are added to each relation, where the primary key of each subclass is inherited from the primary key of the superclass.
2. If multi-table methods do not work or become cluttered, the user can inherit a single relation method and base the distinction between subclasses by variable type. One relation schema is created that contains the attributes of the overlapping subclasses. Then, for each attribute, a type is specified in Boolean form. This is used to specify the placement of incoming tuples; if a tuple belongs to a certain subclass, the Boolean value will state true or false if the tuple fits the subclass type. It allows the Boolean to direct where the tuples go within the different subclasses.

5. **Database requirements include many business rules. What are they? How can we implement them? (6 Points)**

Business rules are designer-specified procedures or constraints that allow the database to be protected from inadvertent actions, ensuring accuracy and organization of the database. Business rules are a valuable tool for maintaining data integrity, as the user can freely determine the proper business rules and tailor them to how the database should be protected. Once implemented, a database can check whenever this constraint is violated, and if violated, signal itself to perform the corrective action that was specified by the user, typically in the form of denying the violation action from being performed.

In implementation, the business rules can be written by the use of If/Then statements that specify certain criteria that must be met. For instance, if the user wanted to specify a rule that a new entry into attribute "Quantity" must be greater than 2, then they can write an IF/THEN statement for this: IF Quantity < 2, THEN output "ERROR: invalid entry." Furthermore, for a business rule to be continuously checked whenever a specific action is performed on the database, a user can wrap their business rules in a trigger. Triggers are tools that allow the database to pause when a specific data manipulation is performed. Some common triggers are BEFORE/AFTER INSERT, which pauses to check for any specified actions before/after an insert command is executed. If the user were to wrap the previous business rule into a

BEFORE INSERT trigger, and then attempt to insert a new Quantity entry of 1, the following would happen:

1. User specifies an INSERT into table, where Quantity attribute = 1, then runs the code.
2. Database reads the INSERT command. Because of the BEFORE INSERT trigger specified by the user, the database is signaled to pause BEFORE performing the insert, and check what is specified in the trigger.
3. The trigger contains the IF/ELSE statement where Quantity cannot be <2. The database sees this, compares it to the new value that the User wants to insert, and sees that Quantity contains an invalid entry of 1.
4. Database outputs the "ERROR: invalid entry" output as stated by the trigger. It then cancels the INSERT command due to the violation.

In relational database, how do you enforce data quality, integrity rules using database schema during database implementation? (4 points)

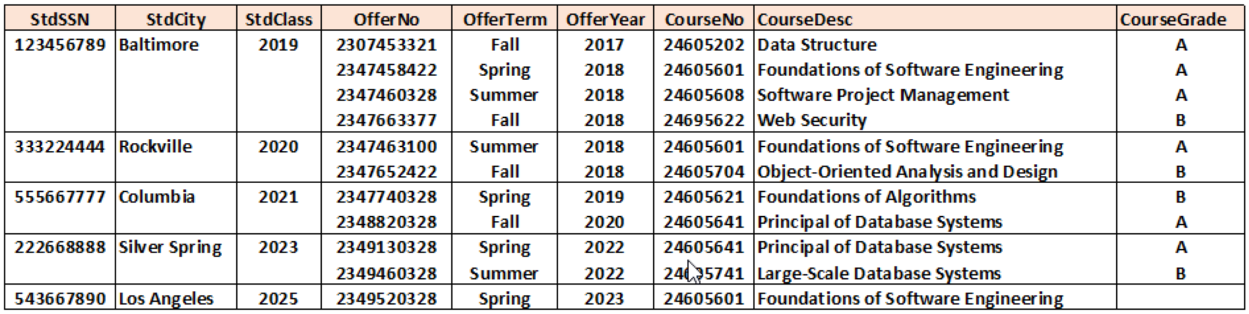
There are plenty of methods that designers can use to ensure proper data quality and security. General, enforcing data quality across the database involves the creation of several User-specified and SQL standard constraints. To increase data security, users can specify new constraints in the form of Triggers, Stored Procedures, and Stored Functions. As described earlier, Triggers are extremely useful in adding extra layers of protective measures to a database and can ensure data integrity by enforcing rules on a per-action basis. Stored procedures and functions are persistently defined and kept within a database until they are needed. Unlike triggers that check based on a specific execution, stored procedures can be called up at any point in time by using the CALL statement. Both triggers and stored procedures can be defined at any point in SQL implementation.

Another method of ensuring data quality is through role assignments on specific users. Accessibility of the database, including the ability to perform certain manipulations such as INSERT/DELETE, or UPDATE on a certain table, can be tailored to certain job roles. For example, a User can assign a DELETE role that is specific to only the Manager of the database. This would restrict the ability to DELETE on the database by any other user, thus ensuring no inadvertent deletions can occur unless specified by the Manager.

Lastly, data quality is further enforced by using integrity rules that are standard for SQL databases. Integrity rules are essentially "laws" that are inherent to a database – any violation of these laws and constraint will signal the database to negate further action. One example is Referential Integrity, which defines the use of Primary Keys and Foreign Keys to ensure that references that are passed between tables are proper and intentional. A value that is passed to a child entity via a foreign key must have been present in the primary key value of the parent table. Another integrity rule is domain integrity, which specifies that new values that are inserted into a data table are valid entries. These can be inherently specified in the database, such as a NULL/NOT NULL definition, or they can be user defined through the use of

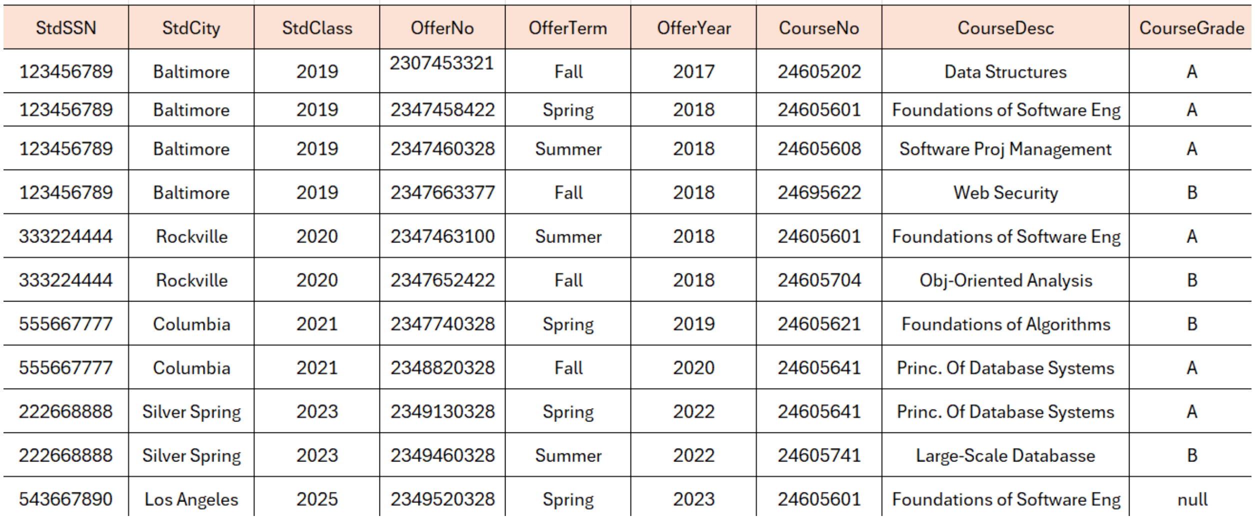
CHECK statements, such as CHECK a new value to ensure it is greater than two. All these integrity rules can be written out during implementation by using the CONSTRAINT clause, then specify the type of constraint after (FOREIGN KEY, CHECK, etc.).

6. **Normalization: (Total: 20 points)**
6.a Describe Functional Dependencies (FDs) of the unnormalized University Database table (see below). Use the example below to demonstrate the process and the functional dependencies between attributes. Only partial sample data is displayed below. (15 points)



StdSSN	StdCity	StdClass	OfferNo	OfferTerm	OfferYear	CourseNo	CourseDesc	CourseGrade
123456789	Baltimore	2019	2307453321	Fall	2017	24605202	Data Structure	A
			2347458422	Spring	2018	24605601	Foundations of Software Engineering	A
			2347460328	Summer	2018	24605608	Software Project Management	A
			2347663377	Fall	2018	24695622	Web Security	B
333224444	Rockville	2020	2347463100	Summer	2018	24605601	Foundations of Software Engineering	A
			2347652422	Fall	2018	24605704	Object-Oriented Analysis and Design	B
555667777	Columbia	2021	2347740328	Spring	2019	24605621	Foundations of Algorithms	B
			2348820328	Fall	2020	24605641	Principal of Database Systems	A
222668888	Silver Spring	2023	2349130328	Spring	2022	24605641	Principal of Database Systems	A
			2349460328	Summer	2022	24605741	Large-Scale Database Systems	B
543667890	Los Angeles	2025	2349520328	Spring	2023	24605601	Foundations of Software Engineering	

It would be more helpful to first break the table down into 1NF. This would get rid of the multiple values in some of the rows:



StdSSN	StdCity	StdClass	OfferNo	OfferTerm	OfferYear	CourseNo	CourseDesc	CourseGrade
123456789	Baltimore	2019	2307453321	Fall	2017	24605202	Data Structures	A
123456789	Baltimore	2019	2347458422	Spring	2018	24605601	Foundations of Software Eng	A
123456789	Baltimore	2019	2347460328	Summer	2018	24605608	Software Proj Management	A
123456789	Baltimore	2019	2347663377	Fall	2018	24695622	Web Security	B
333224444	Rockville	2020	2347463100	Summer	2018	24605601	Foundations of Software Eng	A
333224444	Rockville	2020	2347652422	Fall	2018	24605704	Obj-Oriented Analysis	B
555667777	Columbia	2021	2347740328	Spring	2019	24605621	Foundations of Algorithms	B
555667777	Columbia	2021	2348820328	Fall	2020	24605641	Princ. Of Database Systems	A
222668888	Silver Spring	2023	2349130328	Spring	2022	24605641	Princ. Of Database Systems	A
222668888	Silver Spring	2023	2349460328	Summer	2022	24605741	Large-Scale Databasse	B
543667890	Los Angeles	2025	2349520328	Spring	2023	24605601	Foundations of Software Eng	null

Now we can analyze the functional dependencies better:

FUNCTIONAL DEPENDENCIES:

CS 605.641 Principles of Database Systems
Summer 2025 Final Examination
Dates: 8/3/2025 – 8/6/2025
No discussion, collaboration, consultation, sharing, and Gen AI

Name: Nicolas Escudero

First thing to notice is that there is only one attribute where every value is unique for every tuple. This is the OffeNo attribute, and this can be set as the primary key of the table. All other attributes and combinations of attributes are functionally dependent on the primary key, therefore:

{OfferNo} -> {StdSSN, StdCity, StdClass, OfferTerm, OfferYear, CourseNo, CourseDesc, CourseGrade} or any subset of this ({StdSSN, OfferYear}, {StdCity, StdClass}, etc.)

No other functional dependencies exist among the "Offer" columns. Next, we have the "Course" columns:

{CourseNo} -> {CourseDesc}
{CourseDesc} -> {CourseNo}

These two attributes are functionally dependent to each other, as every unique instance of one of the attributes gives the same value on the other attribute (Ex: 24605601 – Foundations of Software Eng.).

It is also worth noting that, by ONLY using the provided data, the functional dependency {CourseNo, CourseDesc} -> {CourseGrade} is true for the table. However, since it is stated that the data is not complete, and assuming that a CourseGrade can easily be a different letter grade for any class, this is more than likely false in a real scenario.

For the "Student" columns, we see that every row of {StdSSN, StdCity, StdClass} gives a unique tuple per instance. In a case where we are using ONLY the current values in the table, means that every "Student" column is functionally dependent on each other. However, in a real world scenario, it is more than likely that two students can live in the same city, or be able to graduate within the same class. For the sake of real-world design, we will stick with only one true FD:

{StdSSN} -> {StdCity, StdClass}

Another functional dependency exists across the groups of attributes:

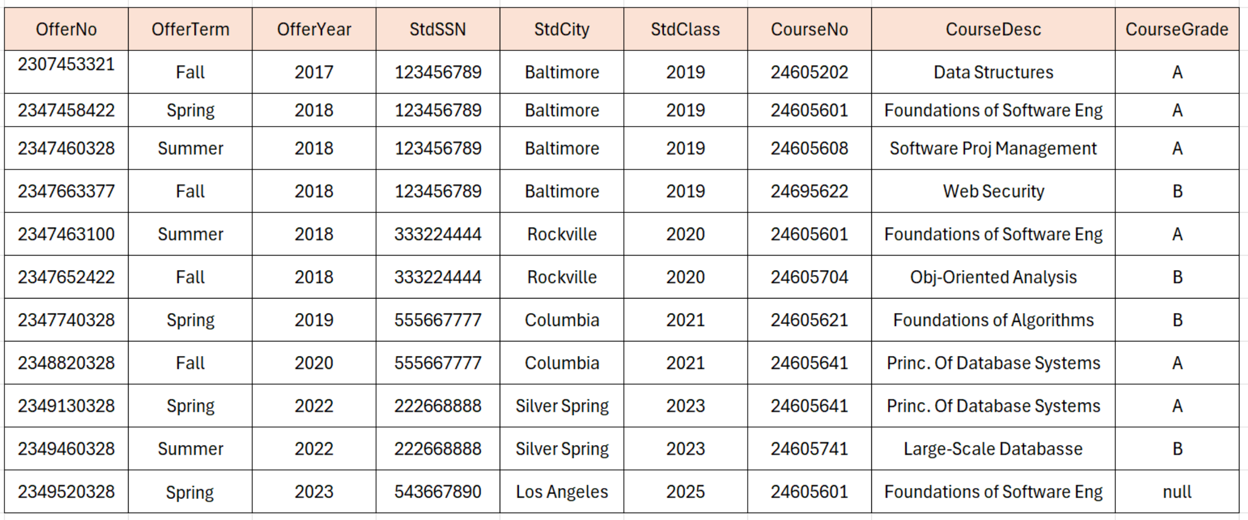
{StdSSN, CourseNo} -> {CourseDesc, CourseGrade}

And since CourseNo and CourseDesc are also functionally dependent with each other, these are also interchangeable:

{StdSSN, StdClass, CourseDesc} -> {CourseNo, CourseGrade}
{StdSSN, StdCity, StdClass, CourseNo, CourseDesc} -> {CourseGrade}

- **Based on your FDs, what is the First Normal Form (1NF) and what are the steps for converting to the 1NF? (4 points)**

First Normal Form states that each entry must be atomic, and every instance's domain cannot have multiple values in a column, or contain no repeated values. To convert the original table into 1NF, we would need to separate those tuples that contain multiple values in them and place each value into a separate row. This was shown earlier, and will be shown below with the most unique key + associated attributes now moved to the front:



OfferNo	OfferTerm	OfferYear	StdSSN	StdCity	StdClass	CourseNo	CourseDesc	CourseGrade
2307453321	Fall	2017	123456789	Baltimore	2019	24605202	Data Structures	A
2347458422	Spring	2018	123456789	Baltimore	2019	24605601	Foundations of Software Eng	A
2347460328	Summer	2018	123456789	Baltimore	2019	24605608	Software Proj Management	A
2347663377	Fall	2018	123456789	Baltimore	2019	24695622	Web Security	B
2347463100	Summer	2018	333224444	Rockville	2020	24605601	Foundations of Software Eng	A
2347652422	Fall	2018	333224444	Rockville	2020	24605704	Obj-Oriented Analysis	B
2347740328	Spring	2019	555667777	Columbia	2021	24605621	Foundations of Algorithms	B
2348820328	Fall	2020	555667777	Columbia	2021	24605641	Princ. Of Database Systems	A
2349130328	Spring	2022	222668888	Silver Spring	2023	24605641	Princ. Of Database Systems	A
2349460328	Summer	2022	222668888	Silver Spring	2023	24605741	Large-Scale Databasse	B
2349520328	Spring	2023	543667890	Los Angeles	2025	24605601	Foundations of Software Eng	null

- **What is the Second Normal Form (2NF) and what are the steps for converting to the 2NF? (4 points)**

Second Normal Form happens under two general circumstances:

- The relation is already in 1NF, or the characteristic that all underlying domains contain single-valued, atomic values. No sets of values are defined in any of the columns, and no repeating groups are found in the table.
- The relation's non-key attributes are fully functionally dependent on all parts of the table's primary key. No partial dependencies should exist in this table.

For this, considering that OfferNo is the only purely unique attribute, this can act as our primary key. We know that the other "Offer" attributes are fully functionally dependent on OfferNo. However, attributes from either the "Course" or the "Student" groups are NOT fully functionally dependent on the OfferNo primary key, as dependencies also exist in their own respective groupings:

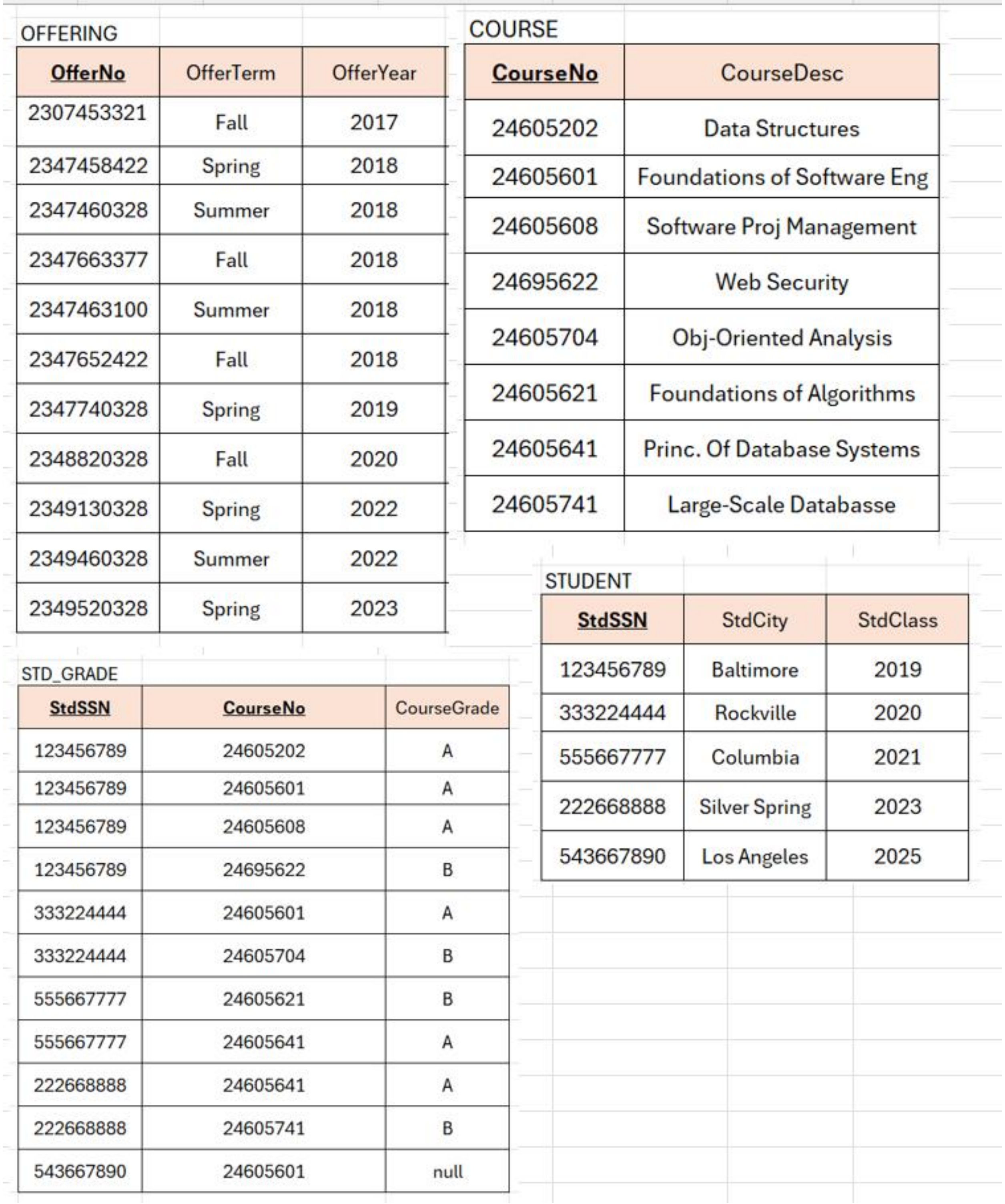
{CourseNo} -> {CourseDesc}
{CourseDesc} -> {CourseNo}

{StdSSN} -> {StdCity, StdClass}
{StdSSN, CourseDesc} -> {CourseNo, CourseGrade}
{StdSSN, CourseNo, CourseDesc} -> {CourseGrade}

This is an example of a partial dependency and this breaks 2NF. To fix this, we would have to create new tables that each have its own primary key to fully functionally depend on. Note that, since [StdSSN, CourseNo} is a valid unique key, this together can be used for one of the tables to define {CourseGrade}:

CS 605.641 Principles of Database Systems
Summer 2025 Final Examination
Dates: 8/3/2025 – 8/6/2025
No discussion, collaboration, consultation, sharing, and Gen AI

Name: Nicolas Escudero



OFFERING

OfferNo	OfferTerm	OfferYear
2307453321	Fall	2017
2347458422	Spring	2018
2347460328	Summer	2018
2347663377	Fall	2018
2347463100	Summer	2018
2347652422	Fall	2018
2347740328	Spring	2019
2348820328	Fall	2020
2349130328	Spring	2022
2349460328	Summer	2022
2349520328	Spring	2023

COURSE

CourseNo	CourseDesc
24605202	Data Structures
24605601	Foundations of Software Eng
24605608	Software Proj Management
24695622	Web Security
24605704	Obj-Oriented Analysis
24605621	Foundations of Algorithms
24605641	Princ. Of Database Systems
24605741	Large-Scale Databasse

STUDENT

StdSSN	StdCity	StdClass
123456789	Baltimore	2019
333224444	Rockville	2020
555667777	Columbia	2021
222668888	Silver Spring	2023
543667890	Los Angeles	2025

STD_GRADE

StdSSN	**CourseNo**	CourseGrade
123456789	24605202	A
123456789	24605601	A
123456789	24605608	A
123456789	24695622	B
333224444	24605601	A
333224444	24605704	B
555667777	24605621	B
555667777	24605641	A
222668888	24605641	A
222668888	24605741	B
543667890	24605601	null

In this sense, all non-key attributes are fully functionally dependent on the primary key of its table, including the composite key for STD_GRADE. It also regulates the functional dependencies so that only one functional dependency exists between the non-key attribute and its associated tables:

{OfferNo} -> {OfferTerm, OfferYear}
{CourseNo, StdSSN} -> {CourseGrade}
{CourseNo} -> {CourseDesc}
{StdSSN} -> {StdCity, StdClass}

Keep in mind that if we ONLY used the presented data, the STUDENT table could present some redundancy, since they technically do functionally depend on each other. For example, if only {StdSSN} was a primary key, but the dependency between {StdCity} -> {StdClass} still exists, then this could break 2NF as it does not depend on the primary key alone. However, since we chose to keep StdCity and StdClass as dependent on StdSSN only, this does not apply.

- **What is the Third Normal Form (3NF) and what are the steps for converting to 3NF? (5 points)**

Third Normal Form is a further normalization step that is based off of two criteria:

- The relation is already at least in 2NF, as previously described.
- The relation contains no transitive dependencies. These are dependencies amongst attributes where a non-key attribute functionally depends on another non-key attribute that then depends on a primary key, rather than directly depending on a primary key.

For our tables, there should be no currently existing transitive dependencies in the OFFERING, COURSE, and STD_GRADE tables. Normally, if there was a dependency that existed, then we would split that table into two separate tables once more, each containing the primary key attribute and one of the non-key attribute instances. For example, if OfferYear was transitively dependent on OfferNo through an FD with OfferTerm, then the table would split in two. One table would have {OfferNo} -> {OfferTerm}, and the other table would have {OfferNo} -> {OfferYear}. This however is not the case for this example.

Again, to ensure that STUDENT does not violate 3NF, the introduction of similar instances such as a student from "Baltimore" or a graduate of the Class "2020" would make these attributes not functionally dependent on each other. Therefore, these non-key attributes only rely on its primary key, StdSSN, with no transitive dependencies existing. If we used ONLY the values within the table, we could easily fix the non-key dependencies by making all the attributes of the STUDENT table a composite primary key, which is a valid move since all instances are unique.

- **Does the sample data affect your normalization process? Explain your answer. (2 points)**

Yes; as previously described throughout this problem, the fact that the STUDENT table originally had all unique representations of {StdSSN, StdCity, StdClass}, this made it valid to say that this can be a unique key, a functional determinant of other factors in the table, and more, if we went strictly by the table. Another example of this confusion is the CourseGrade column, where in some instances such as CourseNo: 24605601, CourseDesc: Fouundations of Software Eng, instances of these tuples actually all contained a CourseGrade "A". Of course, in a real-world scenario, a student can obtain any grade for a course, which is why this was assumed not to be a true functional dependency, but just a result of the lack of data.

6.b Consider the following relation (table) StudentInfo with the following schema {StudentId, Student Name, Class, Class Id} and sample data. The relation has a PK, StudentId. A student can take one or more classes. Is StudentInfo a legitimate relation (table)? If not, how do you fix this relation so that it can be legitimate? Please show your relations with the sample data and explain your rationale. (5 points)

StudentInfo

StudentId	**StudentName**	**Class**	**ClassId**
1	John	Math	101
1	John	Science	102
1	John	Math V	403
2	Jill	Math	101

The StudentInfo table does have some flaws in its design that makes it not a legitimate relation. This can be seen in the primary key – for a table that has a primary key, this primary key should be 1) unique for each instance, and 2) functionally determine all the other attributes. For example, if I were to ask for the attributes of StudentId "1", then I should be able to return a unique set of attributes that is associate with "1". This is not the case for this table, since we can see that for every row with a StudentId "1", a different value is defined in each row of the Class and ClassId columns. This means that Class and ClassId are NOT functionally dependent on StudentId, and StudentId does not functionally determine what attributes are given for Class and ClassID. It does not obey like how a primary key should be, which means that StudentId cannot be the primary key.

Instead, we can determine the best possible primary key by looking at the possible functional dependencies:

FUNCTIONAL DEPENDENCIES:

- **{ClassId} -> {Class}**
  {Class} -> {ClassId}
  - Valid, as every unique instance of ClassId has an associated unique Class value.

- **{StudentId} -> {StudentName}**
  {StudentName} -> {StudentId}
    - Valid, as every unique instance of StudentId has an associated unique StudentName to it.
- **{ClassId, StudentId} -> {StudentName, Class}**
- **{StudentName, Class} -> {ClassId, StudentId}**
    - Valid, now that ClassId and StudentId are put together, every instance of this combination is unique

Now, we can design a table that properly obeys these functional dependencies, and assign primary keys in valid spots. There are plenty of ways to make a valid relation – for this example, I chose to do a single table relation that revolves around the FD: {ClassId, StudentId} -> {StudentName, Class}. This makes the two ID attributes a candidate key, the ClassId as the primary key, and everything else functionally dependent on the combination of these two:

StudentInfo

ClassId	**StudentId**	StudentName	Class
101	1	John	Math
102	1	John	Science
403	1	John	Math V
101	2	Jill	Math

This is a legitimate relationship that does not break any functional dependency rules. It is not exactly the most normalized, however. The multiple table method makes this relation more normalized:

CLASS

ClassId	Class
101	Math
102	Science
403	Math V

STUDENT

StudentId	StudentName
1	John
2	Jill

STUDENT_INFO

ClassId	**StudentId**	Class
101	1	Math
102	1	Science
403	1	Math V
101	2	Math

CS 605.641 Principles of Database Systems
Summer 2025 Final Examination
Dates: 8/3/2025 – 8/6/2025
No discussion, collaboration, consultation, sharing, and Gen AI

Name: Nicolas Escudero

Here, we have three tables that separate each dependency. If set up in this way, the relationship is much more normalized (at least 3NF).

7. Create an Airline database schema based on the ERD in the textbook (also see below). You need to implement your DDL statements. Your DDL statements should clearly specify the Primary Key, Foreign Key, Unique, NOT NULL and Check constraints.

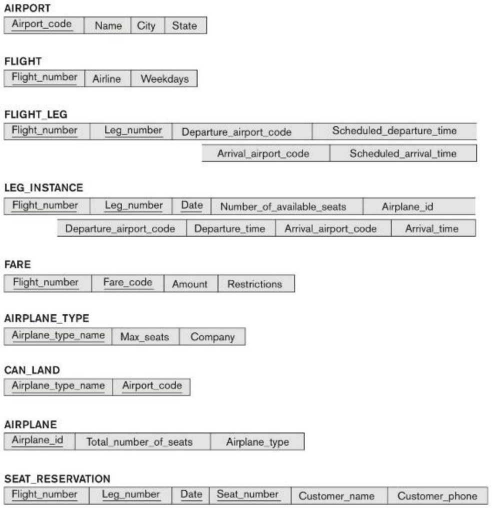

AIRPORT
Airport_code | Name | City | State
FLIGHT
Flight_number | Airline | Weekdays
FLIGHT_LEG
Flight_number | Leg_number | Departure_airport_code | Scheduled_departure_time | Arrival_airport_code | Scheduled_arrival_time
LEG_INSTANCE
Flight_number | Leg_number | Date | Number_of_available_seats | Airplane_id | Departure_airport_code | Departure_time | Arrival_airport_code | Arrival_time
FARE
Flight_number | Fare_code | Amount | Restrictions
AIRPLANE_TYPE
Airplane_type_name | Max_seats | Company
CAN_LAND
Airplane_type_name | Airport_code
AIRPLANE
Airplane_id | Total_number_of_seats | Airplane_type
SEAT_RESERVATION
Flight_number | Leg_number | Date | Seat_number | Customer_name | Customer_phone

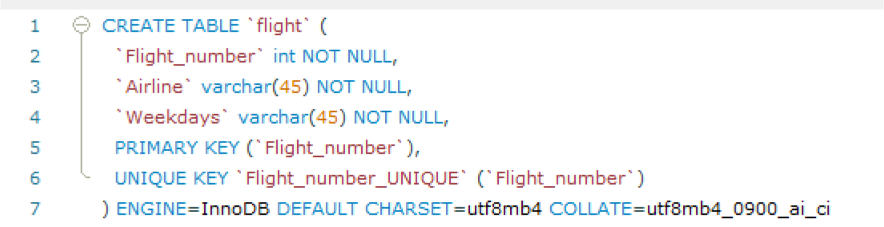
Figure 5.8 The AIRLINE relational database schema.

Part 1: You need to implement the following business rules that are declaratively specified in your schema or be implemented through triggers. If your RDBMS supports triggers, you may implement your trigger solutions. If your RDBMS does not support triggers, you need to show your trigger scripts (no need to implement them) (15 points):

NOTE: FOR DDL AND ATTRIBUTE DEFINITIONS, PLEASE SEE ATTACHED FILES "Final Escudero,Nicolas DDL.pdf" AND "Final Escudero, Nicolas SQL.sql".

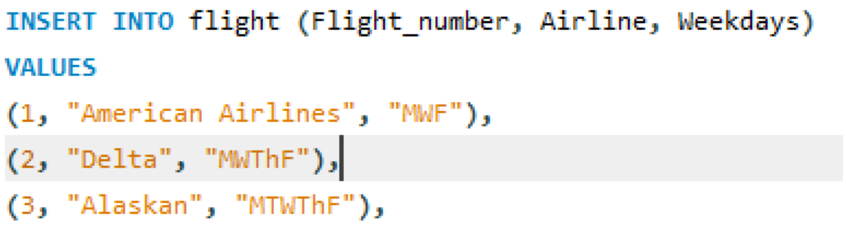
(a) **The Number in the FLIGHT must be unique.**

This was implemented as a PRIMARY KEY for the attribute 'Flight_number.' For extra security, I also added the same attribute as a UNIQUE KEY. Here is a screenshot of the constraint working for this case:

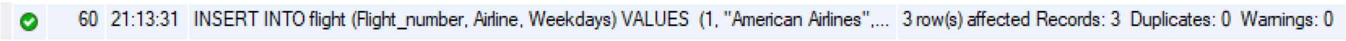


```
CREATE TABLE `flight` (
  `Flight_number` int NOT NULL,
  `Airline` varchar(45) NOT NULL,
  `Weekdays` varchar(45) NOT NULL,
  PRIMARY KEY (`Flight_number`),
  UNIQUE KEY `Flight_number_UNIQUE` (`Flight_number`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

Next, I performed an INSERT into the FLIGHT table, with unique numbers, and the resulting output:

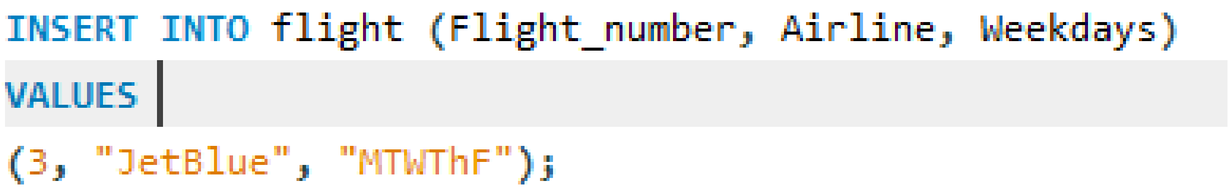


```
INSERT INTO flight (Flight_number, Airline, Weekdays)
VALUES
(1, "American Airlines", "MWF"),
(2, "Delta", "MWThF"),
(3, "Alaskan", "MTWThF"),
```

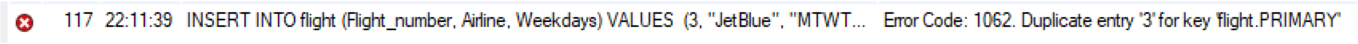


```
60  21:13:31  INSERT INTO flight (Flight_number, Airline, Weekdays) VALUES  (1, "American Airlines",...  3 row(s) affected Records: 3  Duplicates: 0  Warnings: 0
```

Now if we try to insert a new entry with the same flight number “3”:



```
INSERT INTO flight (Flight_number, Airline, Weekdays)
VALUES
(3, "JetBlue", "MTWThF");
```

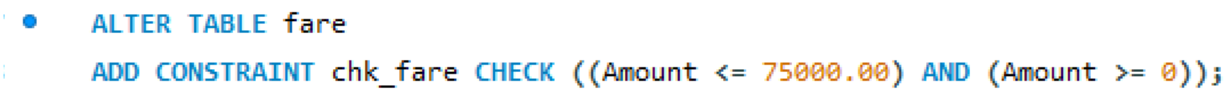


```
117  22:11:39  INSERT INTO flight (Flight_number, Airline, Weekdays) VALUES  (3, "JetBlue", "MTWT...   Error Code: 1062. Duplicate entry '3' for key 'flight.PRIMARY'
```

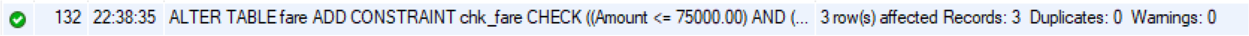
This shows an error, with a message saying there was a duplicate entry.

(b) The fare amount (Amount) is in a range of ($0 – $75,000).

We define in our query a new constraint using the CHECK clause:



```
ALTER TABLE fare
ADD CONSTRAINT chk_fare CHECK ((Amount <= 75000.00) AND (Amount >= 0));
```

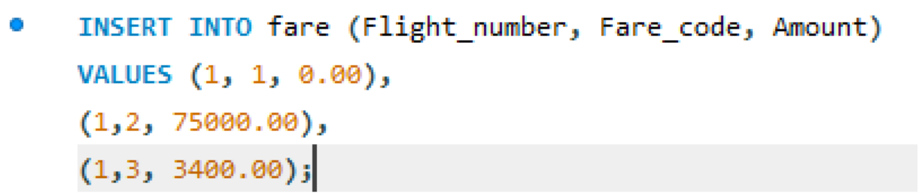


```
132  22:38:35  ALTER TABLE fare ADD CONSTRAINT chk_fare CHECK ((Amount <= 75000.00) AND (...  3 row(s) affected Records: 3  Duplicates: 0  Warnings: 0
```

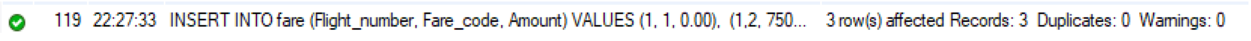
CS 605.641 Principles of Database Systems
Summer 2025 Final Examination
Dates: 8/3/2025 – 8/6/2025
No discussion, collaboration, consultation, sharing, and Gen AI

Name: Nicolas Escudero

Inserting valid values into the FARE table, testing the two extreme values:

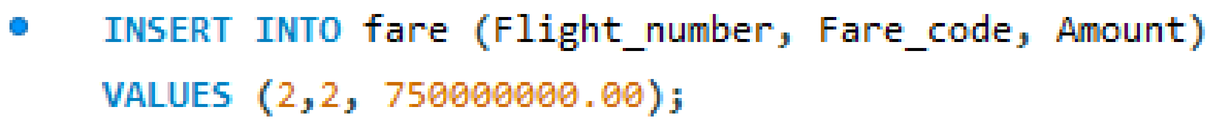


```
INSERT INTO fare (Flight_number, Fare_code, Amount)
VALUES (1, 1, 0.00),
(1,2, 75000.00),
(1,3, 3400.00);
```

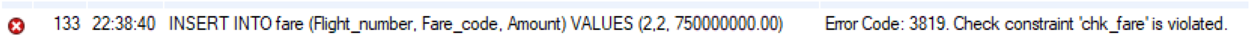


```
119  22:27:33  INSERT INTO fare (Flight_number, Fare_code, Amount) VALUES (1, 1, 0.00),  (1,2, 750...   3 row(s) affected Records: 3  Duplicates: 0  Warnings: 0
```

Now, inserting a value that is well out of range of the CHECK constraint:



```
INSERT INTO fare (Flight_number, Fare_code, Amount)
VALUES (2,2, 750000000.00);
```

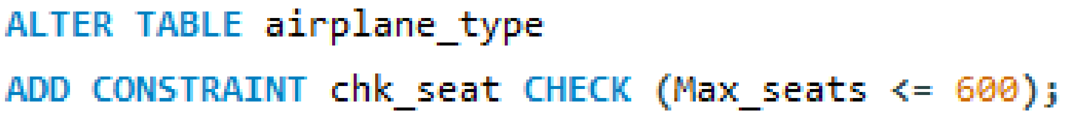


```
133  22:38:40  INSERT INTO fare (Flight_number, Fare_code, Amount) VALUES (2,2, 750000000.00)   Error Code: 3819. Check constraint 'chk_fare' is violated.
```

The error message shows that check constraint was violated, showing successful implementation.

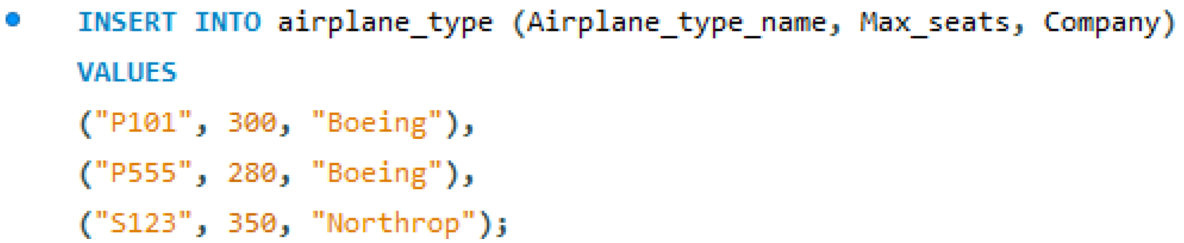
(c) <u>**The maximum seats (Max_seats) for any airplane type cannot exceed 600.**</u>

Just like in (b), we add a CHECK constraint that specifies Max_seats:

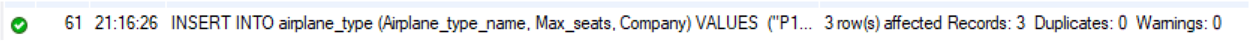


```
ALTER TABLE airplane_type
ADD CONSTRAINT chk_seat CHECK (Max_seats <= 600);
```

Inserting valid values, seat numbers are below 600:

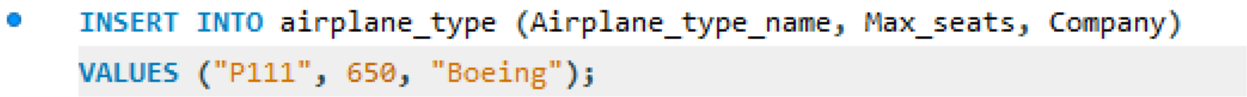


```
INSERT INTO airplane_type (Airplane_type_name, Max_seats, Company)
VALUES
("P101", 300, "Boeing"),
("P555", 280, "Boeing"),
("S123", 350, "Northrop");
```

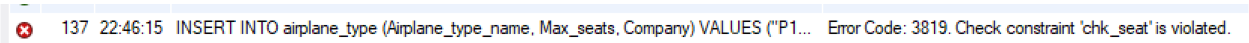


```
61  21:16:26  INSERT INTO airplane_type (Airplane_type_name, Max_seats, Company) VALUES  ("P1...  3 row(s) affected Records: 3  Duplicates: 0  Warnings: 0
```

Inserting not valid value, where seat number is greater than 600:



```
INSERT INTO airplane_type (Airplane_type_name, Max_seats, Company)
VALUES ("P111", 650, "Boeing");
```

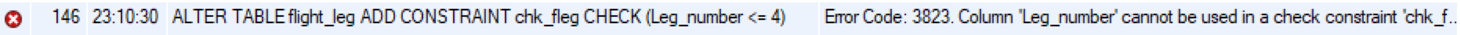


```
137  22:46:15  INSERT INTO airplane_type (Airplane_type_name, Max_seats, Company) VALUES ("P1...  Error Code: 3819. Check constraint 'chk_seat' is violated.
```

The error message shows that check constraint was violated, showing successful implementation.

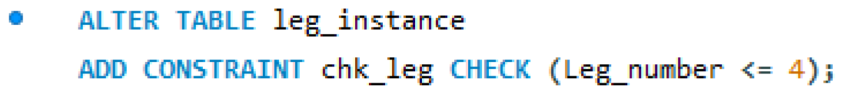
(d) <u>The maximum number of flight legs (Leg_number in the FLIGHT_LEG) cannot exceed 4.</u>

Now we add a CHECK constraint for specifying that the maximum number of flight legs cannot exceed 4. For this constraint, it must be noted that FLIGHT_LEG contains a foreign key that refers to the LEG_NUMBER table. Because of this, we cannot add a CHECK constraint to the FLIGHT_LEG table directly, as this results in an error:



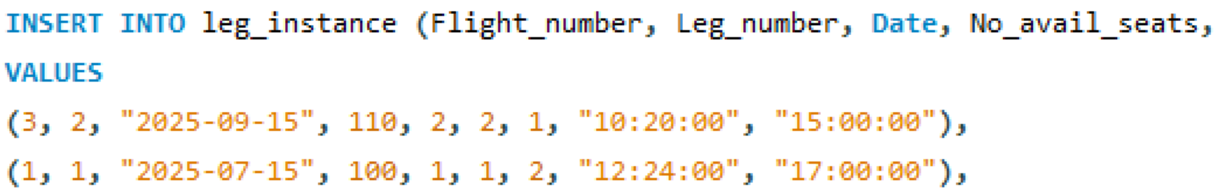
```
146  23:10:30  ALTER TABLE flight_leg ADD CONSTRAINT chk_fleg CHECK (Leg_number <= 4)   Error Code: 3823. Column 'Leg_number' cannot be used in a check constraint 'chk_f..
```

Because of this error, we have to apply the constraint at the source, in this case the Parent table LEG_INSTANCE

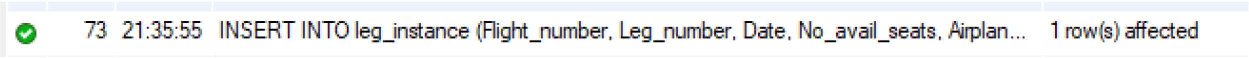


```
ALTER TABLE leg_instance
ADD CONSTRAINT chk_leg CHECK (Leg_number <= 4);
```

Inserting valid tuples:

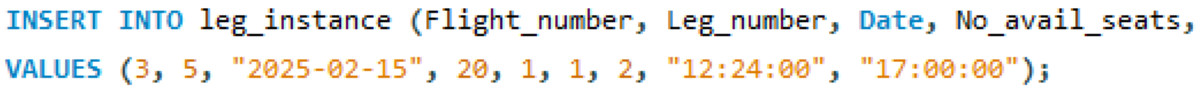


```
INSERT INTO leg_instance (Flight_number, Leg_number, Date, No_avail_seats,
VALUES
(3, 2, "2025-09-15", 110, 2, 2, 1, "10:20:00", "15:00:00"),
(1, 1, "2025-07-15", 100, 1, 1, 2, "12:24:00", "17:00:00"),
```

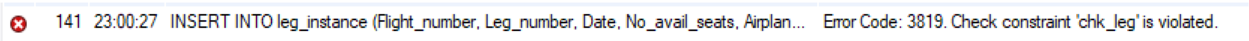


```
73  21:35:55  INSERT INTO leg_instance (Flight_number, Leg_number, Date, No_avail_seats, Airplan...  1 row(s) affected
```

Inserting a non-valid leg_number into the INSERT generates an error:

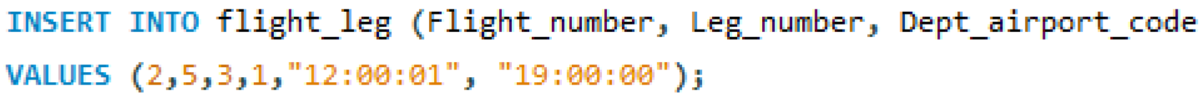


```
INSERT INTO leg_instance (Flight_number, Leg_number, Date, No_avail_seats,
VALUES (3, 5, "2025-02-15", 20, 1, 1, 2, "12:24:00", "17:00:00");
```

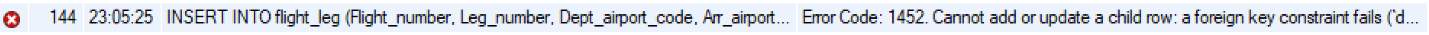


```
141  23:00:27  INSERT INTO leg_instance (Flight_number, Leg_number, Date, No_avail_seats, Airplan...  Error Code: 3819. Check constraint 'chk_leg' is violated.
```

To further show that the flight_leg is an FK to the leg_number and cannot output a value > 4, here's an example of if we try to INSERT directly into flight_leg:



```
INSERT INTO flight_leg (Flight_number, Leg_number, Dept_airport_code
VALUES (2,5,3,1,"12:00:01", "19:00:00");
```



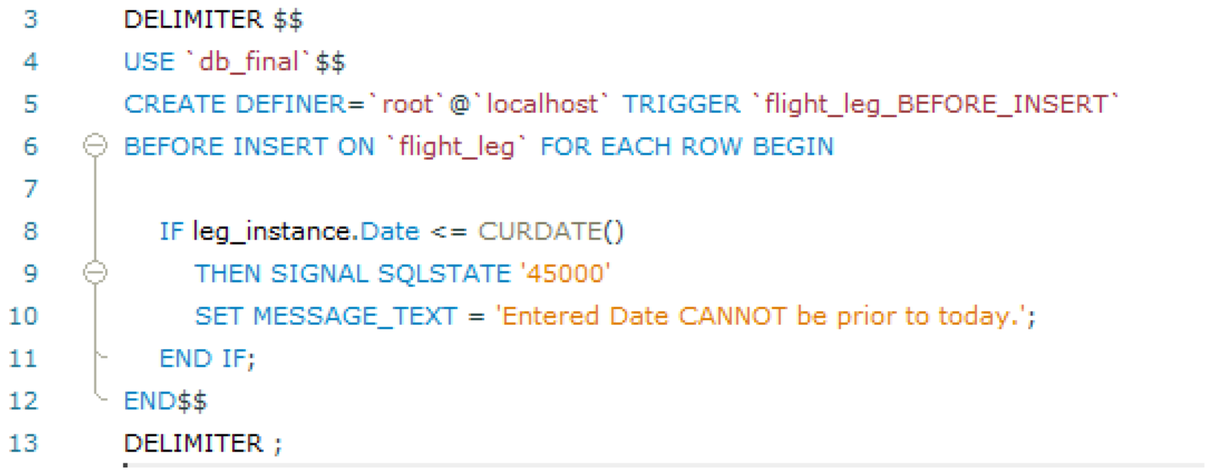
```
144  23:05:25  INSERT INTO flight_leg (Flight_number, Leg_number, Dept_airport_code, Arr_airport...  Error Code: 1452. Cannot add or update a child row: a foreign key constraint fails ('d...
```

(e) **<u>For any instance of a flight leg, the date of (Date in the LEG_INSTANCE) must be either current date or a future date.</u>**

For this constraint, every new instance of a FLIGHT_LEG must come from a LEG_INSTANCE where the Date attribute is current or in the future. We can implement this constraint through the use of a Trigger, specifically a BEFORE INSERT trigger that checks for user specified clauses before an insert command can be executed.

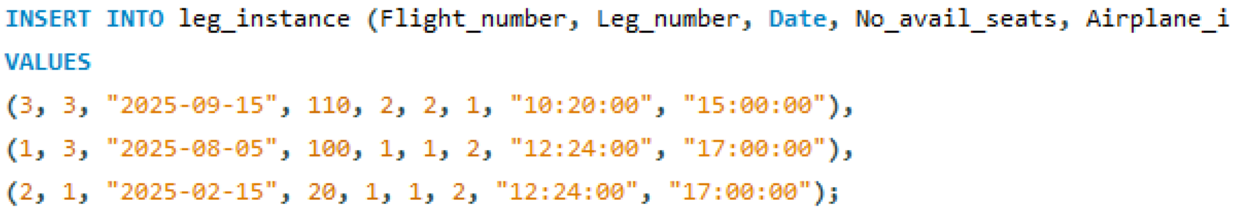
CS 605.641 Principles of Database Systems
Summer 2025 Final Examination
Dates: 8/3/2025 – 8/6/2025
No discussion, collaboration, consultation, sharing, and Gen AI

Name: Nicolas Escudero



```
DELIMITER $$
USE `db_final`$$
CREATE DEFINER=`root`@`localhost` TRIGGER `flight_leg_BEFORE_INSERT`
BEFORE INSERT ON `flight_leg` FOR EACH ROW BEGIN

    IF leg_instance.Date <= CURDATE()
        THEN SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Entered Date CANNOT be prior to today.';
    END IF;
END$$
DELIMITER ;
```



```
INSERT INTO leg_instance (Flight_number, Leg_number, Date, No_avail_seats, Airplane_i
VALUES
(3, 3, "2025-09-15", 110, 2, 2, 1, "10:20:00", "15:00:00"),
(1, 3, "2025-08-05", 100, 1, 1, 2, "12:24:00", "17:00:00"),
(2, 1, "2025-02-15", 20, 1, 1, 2, "12:24:00", "17:00:00");
```

CS 605.641 Principles of Database Systems
Summer 2025 Final Examination
Dates: 8/3/2025 – 8/6/2025
No discussion, collaboration, consultation, sharing, and Gen AI

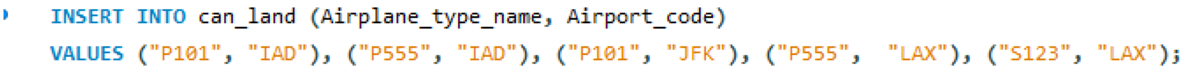
Name: Nicolas Escudero

Part 2: You need to implement your DML statements with sample test data to retrieve the following information (15 points):

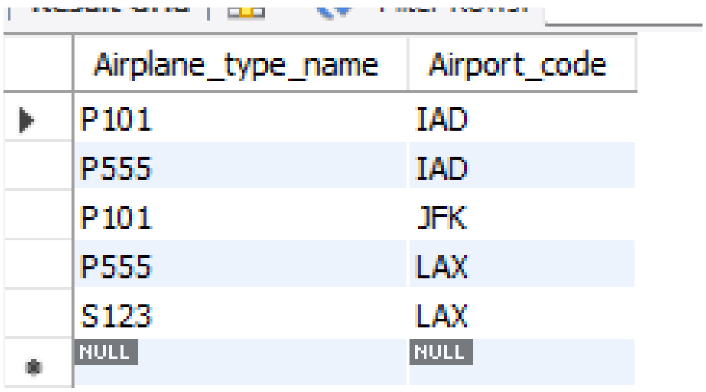
(a) Write a query statement to create a list of aircraft types that can land in the airport at Washington Dulles International Airport (Airport_code is 'IAD').

Some DML's with the test data, and the SELECT outputs for each table used in the query:

CAN_LAND

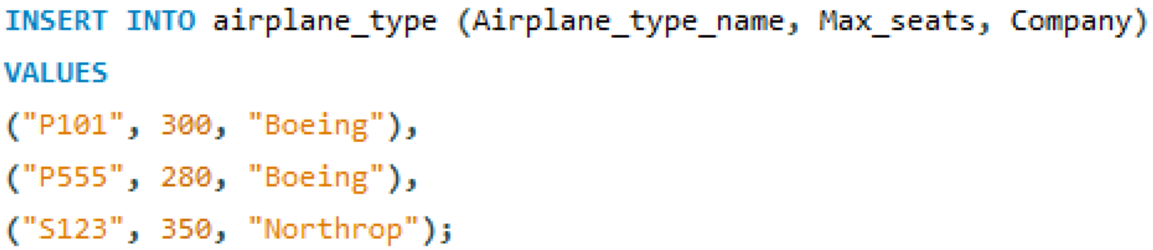


```
INSERT INTO can_land (Airplane_type_name, Airport_code)
VALUES ("P101", "IAD"), ("P555", "IAD"), ("P101", "JFK"), ("P555",  "LAX"), ("S123", "LAX");
```



Airplane_type_name	Airport_code
P101	IAD
P555	IAD
P101	JFK
P555	LAX
S123	LAX
NULL	NULL

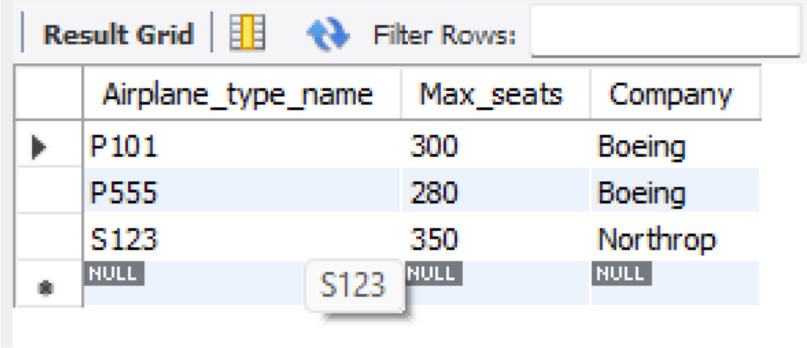
AIRPLANE_TYPE



```
INSERT INTO airplane_type (Airplane_type_name, Max_seats, Company)
VALUES
("P101", 300, "Boeing"),
("P555", 280, "Boeing"),
("S123", 350, "Northrop");
```

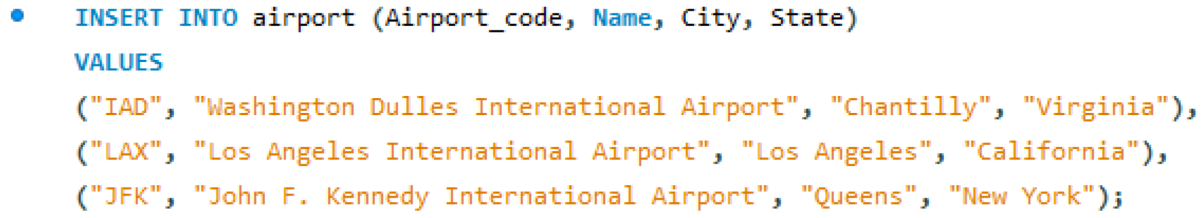
CS 605.641 Principles of Database Systems
Summer 2025 Final Examination
Dates: 8/3/2025 – 8/6/2025
No discussion, collaboration, consultation, sharing, and Gen AI

Name: Nicolas Escudero

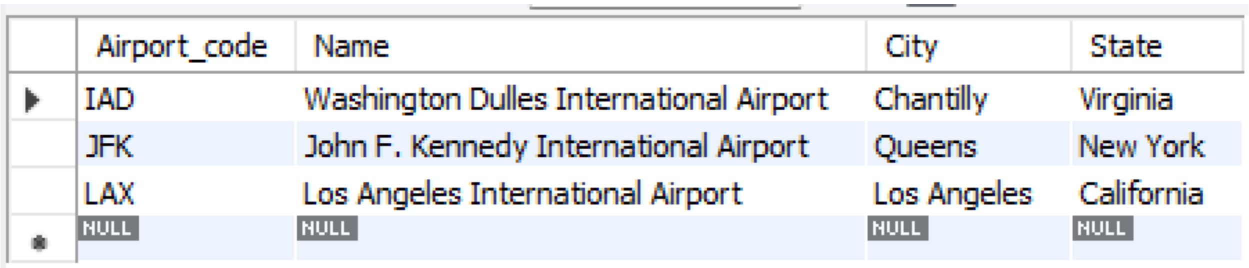


	Airplane_type_name	Max_seats	Company
▶	P101	300	Boeing
	P555	280	Boeing
	S123	350	Northrop
✱	NULL	NULL	NULL

AIRPORT

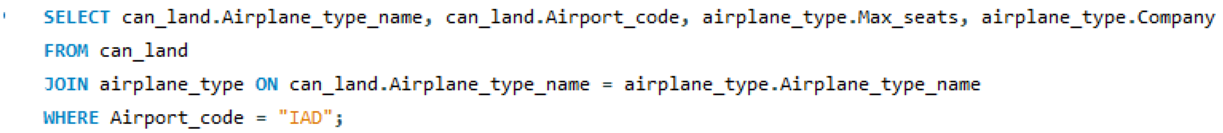


```
INSERT INTO airport (Airport_code, Name, City, State)
VALUES
("IAD", "Washington Dulles International Airport", "Chantilly", "Virginia"),
("LAX", "Los Angeles International Airport", "Los Angeles", "California"),
("JFK", "John F. Kennedy International Airport", "Queens", "New York");
```

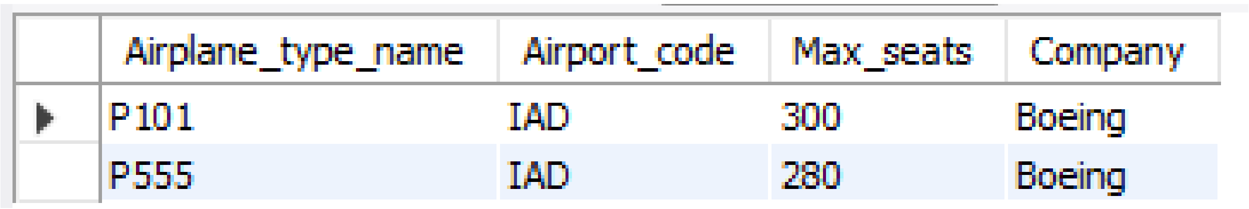


	Airport_code	Name	City	State
▶	IAD	Washington Dulles International Airport	Chantilly	Virginia
	JFK	John F. Kennedy International Airport	Queens	New York
	LAX	Los Angeles International Airport	Los Angeles	California
✱	NULL	NULL	NULL	NULL

Next is the query statement, using a SELECT on CAN_LAND, a JOIN to merge AIRPLANE_TYPE and CAN_LAND together, and a WHERE clause to specify for only the planes that can land in "IAD." Selection obtained was for the Airplane type names, the code to confirm the right landing destination, as well as the rest of the attributes for AIRPLANE_TYPE:



```
SELECT can_land.Airplane_type_name, can_land.Airport_code, airplane_type.Max_seats, airplane_type.Company
FROM can_land
JOIN airplane_type ON can_land.Airplane_type_name = airplane_type.Airplane_type_name
WHERE Airport_code = "IAD";
```

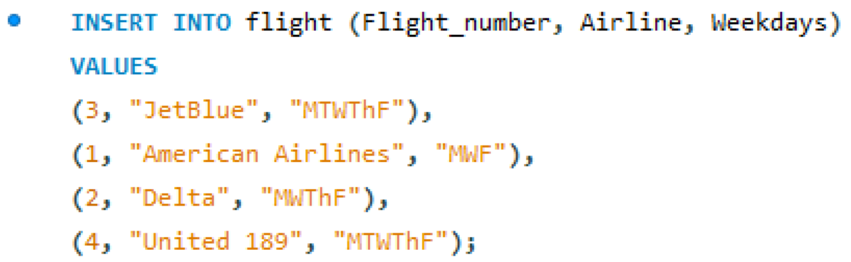


	Airplane_type_name	Airport_code	Max_seats	Company
▶	P101	IAD	300	Boeing
	P555	IAD	280	Boeing

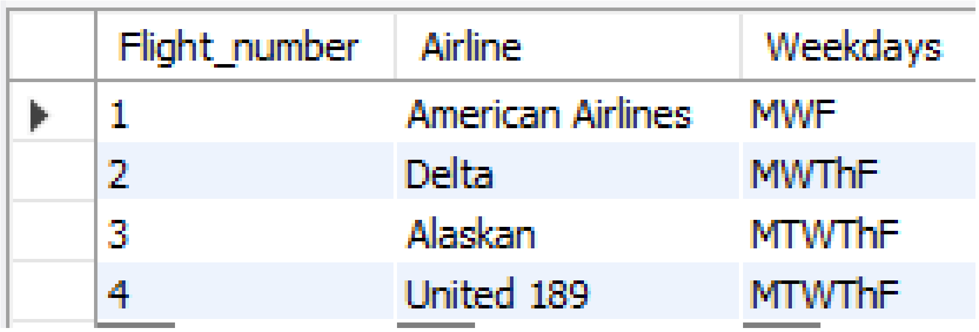
(b) Write a query statement to list all fare information for flight 'United 189'.

Some DML's to insert new data into the involved tables:

FLIGHT

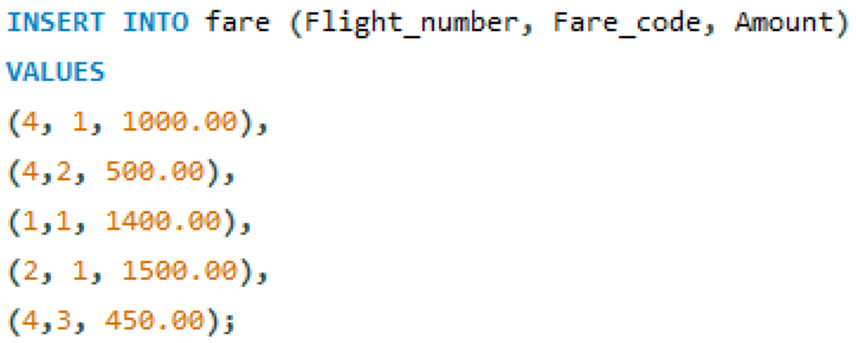


```
INSERT INTO flight (Flight_number, Airline, Weekdays)
VALUES
(3, "JetBlue", "MTWThF"),
(1, "American Airlines", "MWF"),
(2, "Delta", "MWThF"),
(4, "United 189", "MTWThF");
```

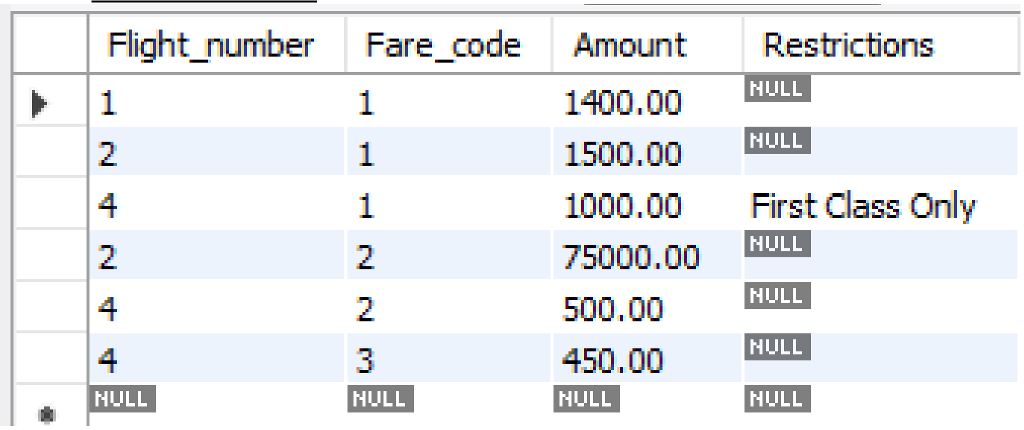


	Flight_number	Airline	Weekdays
▶	1	American Airlines	MWF
	2	Delta	MWThF
	3	Alaskan	MTWThF
	4	United 189	MTWThF

FARE

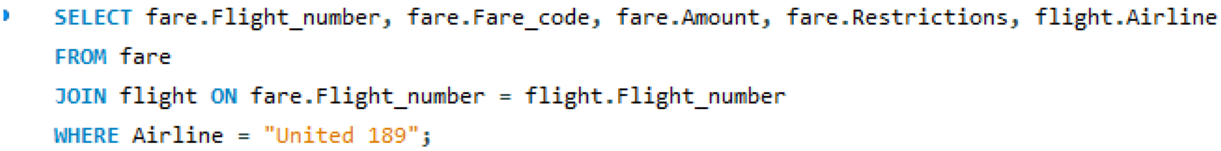


```
INSERT INTO fare (Flight_number, Fare_code, Amount)
VALUES
(4, 1, 1000.00),
(4,2, 500.00),
(1,1, 1400.00),
(2, 1, 1500.00),
(4,3, 450.00);
```

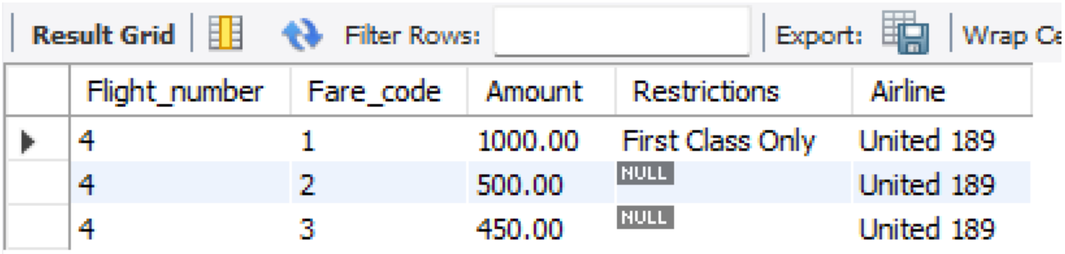


Flight_number	Fare_code	Amount	Restrictions
1	1	1400.00	NULL
2	1	1500.00	NULL
4	1	1000.00	First Class Only
2	2	75000.00	NULL
4	2	500.00	NULL
4	3	450.00	NULL
NULL	NULL	NULL	NULL

Now we use SELECT, JOIN, and WHERE clauses to get the appropriate data. I chose to SELECT for Flight_number, Fare_code, Amount, and Restrictions for the FARE table, obtaining all fare information. I also chose to SELECT for Airline from the FLIGHT table, to confirm that all returned flights are United 189. I performed a JOIN on the two tables on matching Flight_number, and then specified the WHERE clause to only give instances of "United 189":



```
SELECT fare.Flight_number, fare.Fare_code, fare.Amount, fare.Restrictions, flight.Airline
FROM fare
JOIN flight ON fare.Flight_number = flight.Flight_number
WHERE Airline = "United 189";
```



Flight_number	Fare_code	Amount	Restrictions	Airline
4	1	1000.00	First Class Only	United 189
4	2	500.00	NULL	United 189
4	3	450.00	NULL	United 189

(c) **Write a query statement to create a list of direct flights (including scheduled departure time and arrival time) starting from Baltimore Washington International Airport (Airport_code is 'BWI') and terminating at San Francisco International Airport (Airport_code is 'SFO') which have more than two seats available on 8/16/2025. In addition, write another**

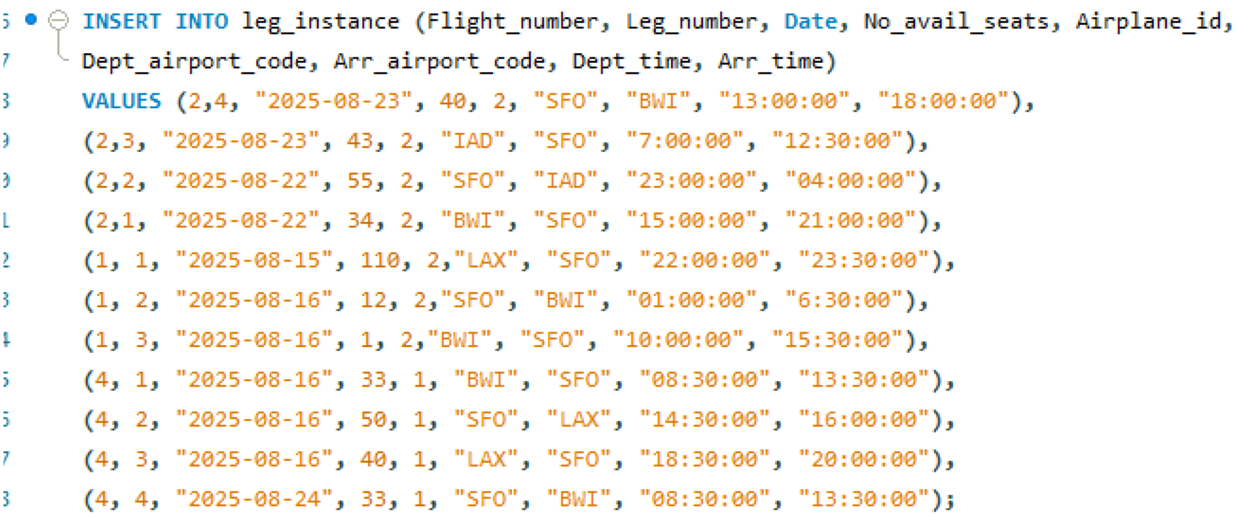
CS 605.641 Principles of Database Systems
Summer 2025 Final Examination
Dates: 8/3/2025 – 8/6/2025
No discussion, collaboration, consultation, sharing, and Gen AI

Name: Nicolas Escudero

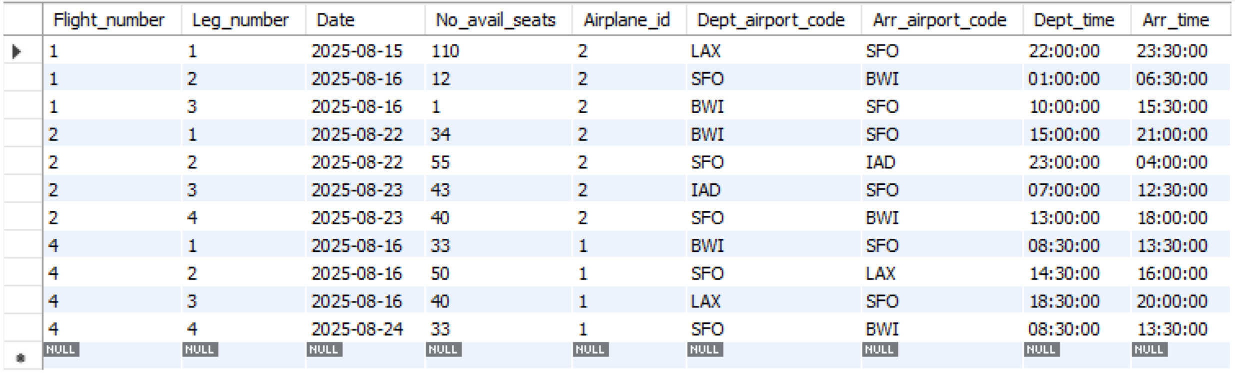
query to create a list of direct return flights from 'SFO' to 'BWI' on 8/23/2025.

First, I wrote a series of DML statements for this query case, all under LEG_INSTANCE. I also created two new AIRPORT entries to reflect the BWI and SFO airports:

LEG_INSTANCE

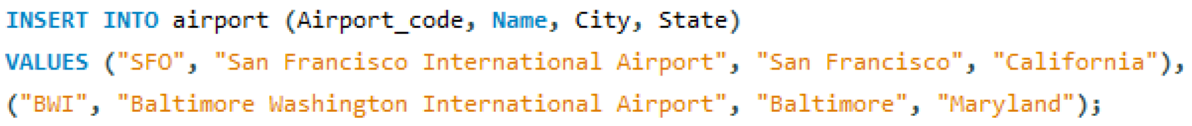


```
INSERT INTO leg_instance (Flight_number, Leg_number, Date, No_avail_seats, Airplane_id,
Dept_airport_code, Arr_airport_code, Dept_time, Arr_time)
VALUES (2,4, "2025-08-23", 40, 2, "SFO", "BWI", "13:00:00", "18:00:00"),
(2,3, "2025-08-23", 43, 2, "IAD", "SFO", "7:00:00", "12:30:00"),
(2,2, "2025-08-22", 55, 2, "SFO", "IAD", "23:00:00", "04:00:00"),
(2,1, "2025-08-22", 34, 2, "BWI", "SFO", "15:00:00", "21:00:00"),
(1, 1, "2025-08-15", 110, 2,"LAX", "SFO", "22:00:00", "23:30:00"),
(1, 2, "2025-08-16", 12, 2,"SFO", "BWI", "01:00:00", "6:30:00"),
(1, 3, "2025-08-16", 1, 2,"BWI", "SFO", "10:00:00", "15:30:00"),
(4, 1, "2025-08-16", 33, 1, "BWI", "SFO", "08:30:00", "13:30:00"),
(4, 2, "2025-08-16", 50, 1, "SFO", "LAX", "14:30:00", "16:00:00"),
(4, 3, "2025-08-16", 40, 1, "LAX", "SFO", "18:30:00", "20:00:00"),
(4, 4, "2025-08-24", 33, 1, "SFO", "BWI", "08:30:00", "13:30:00");
```



Flight_number	Leg_number	Date	No_avail_seats	Airplane_id	Dept_airport_code	Arr_airport_code	Dept_time	Arr_time
1	1	2025-08-15	110	2	LAX	SFO	22:00:00	23:30:00
1	2	2025-08-16	12	2	SFO	BWI	01:00:00	06:30:00
1	3	2025-08-16	1	2	BWI	SFO	10:00:00	15:30:00
2	1	2025-08-22	34	2	BWI	SFO	15:00:00	21:00:00
2	2	2025-08-22	55	2	SFO	IAD	23:00:00	04:00:00
2	3	2025-08-23	43	2	IAD	SFO	07:00:00	12:30:00
2	4	2025-08-23	40	2	SFO	BWI	13:00:00	18:00:00
4	1	2025-08-16	33	1	BWI	SFO	08:30:00	13:30:00
4	2	2025-08-16	50	1	SFO	LAX	14:30:00	16:00:00
4	3	2025-08-16	40	1	LAX	SFO	18:30:00	20:00:00
4	4	2025-08-24	33	1	SFO	BWI	08:30:00	13:30:00
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

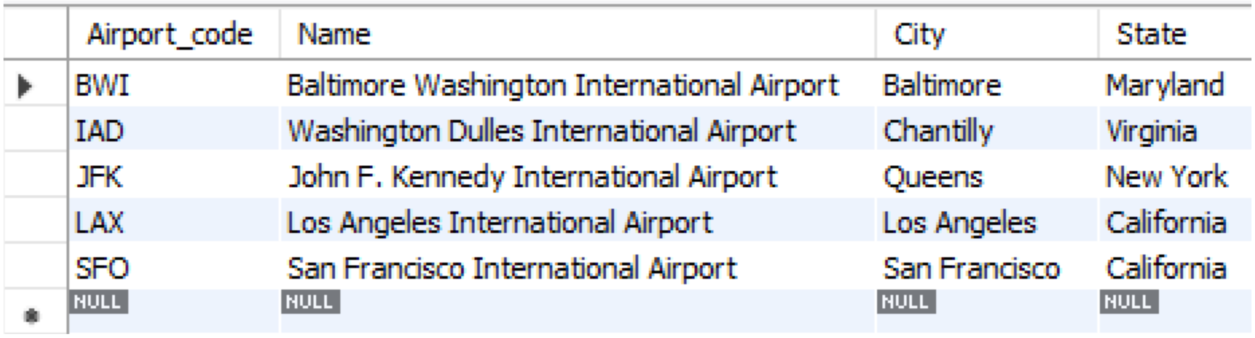
AIRPORT



```
INSERT INTO airport (Airport_code, Name, City, State)
VALUES ("SFO", "San Francisco International Airport", "San Francisco", "California"),
("BWI", "Baltimore Washington International Airport", "Baltimore", "Maryland");
```

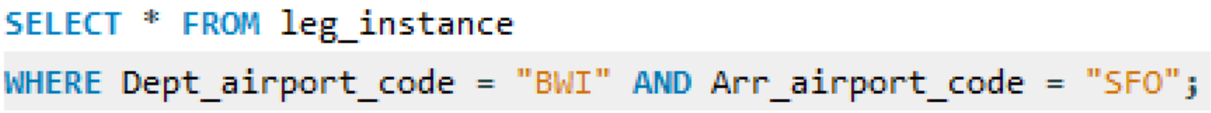
CS 605.641 Principles of Database Systems
Summer 2025 Final Examination
Dates: 8/3/2025 – 8/6/2025
No discussion, collaboration, consultation, sharing, and Gen AI

Name: Nicolas Escudero

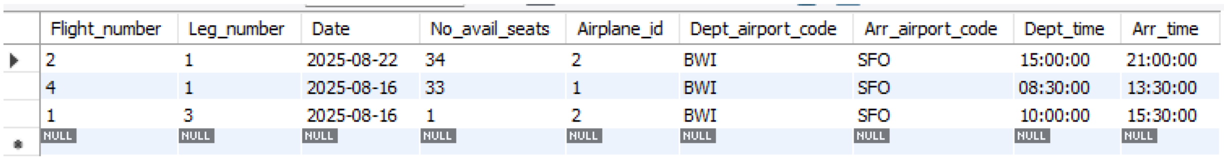


Airport_code	Name	City	State
BWI	Baltimore Washington International Airport	Baltimore	Maryland
IAD	Washington Dulles International Airport	Chantilly	Virginia
JFK	John F. Kennedy International Airport	Queens	New York
LAX	Los Angeles International Airport	Los Angeles	California
SFO	San Francisco International Airport	San Francisco	California
NULL	NULL	NULL	NULL

For the first part, I wrote a SELECT statement for the given criteria: Departing code = “BWI”, Arrival code = “SFO”, seats remaining > 2, and Date = “2025-08-16”. To demonstrate the selectivity of it, I first did a SELECT command with only the departing and arrival code specifications:

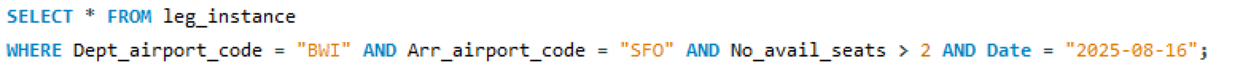


```
SELECT * FROM leg_instance
WHERE Dept_airport_code = "BWI" AND Arr_airport_code = "SFO";
```

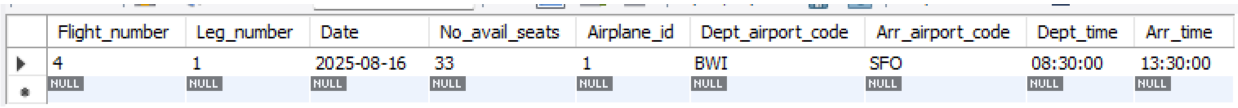


Flight_number	Leg_number	Date	No_avail_seats	Airplane_id	Dept_airport_code	Arr_airport_code	Dept_time	Arr_time
2	1	2025-08-22	34	2	BWI	SFO	15:00:00	21:00:00
4	1	2025-08-16	33	1	BWI	SFO	08:30:00	13:30:00
1	3	2025-08-16	1	2	BWI	SFO	10:00:00	15:30:00
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Then, I added the rest of the criteria for number of available seats and date, returning only one of the tuples:

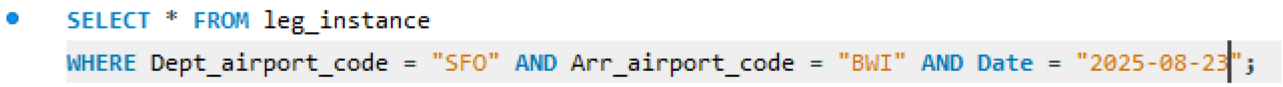


```
SELECT * FROM leg_instance
WHERE Dept_airport_code = "BWI" AND Arr_airport_code = "SFO" AND No_avail_seats > 2 AND Date = "2025-08-16";
```



Flight_number	Leg_number	Date	No_avail_seats	Airplane_id	Dept_airport_code	Arr_airport_code	Dept_time	Arr_time
4	1	2025-08-16	33	1	BWI	SFO	08:30:00	13:30:00
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

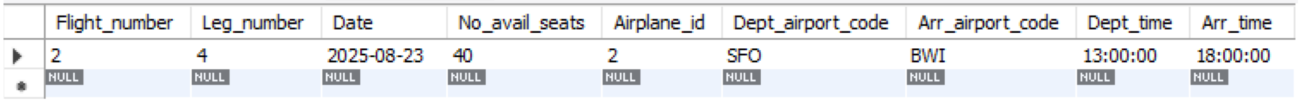
For the second part, I wrote a new query to select for criteria on just Departing code, arrival code, and date:



```
SELECT * FROM leg_instance
WHERE Dept_airport_code = "SFO" AND Arr_airport_code = "BWI" AND Date = "2025-08-23";
```

CS 605.641 Principles of Database Systems
Summer 2025 Final Examination
Dates: 8/3/2025 – 8/6/2025
No discussion, collaboration, consultation, sharing, and Gen AI

Name: Nicolas Escudero



	Flight_number	Leg_number	Date	No_avail_seats	Airplane_id	Dept_airport_code	Arr_airport_code	Dept_time	Arr_time
▶	2	4	2025-08-23	40	2	SFO	BWI	13:00:00	18:00:00
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL