C995 SQL for Data Analysis

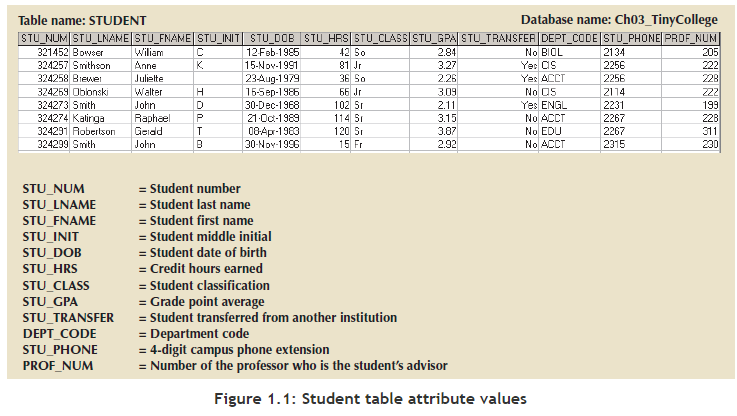
# 1 - Modeling data

## 1.0 Introduction

* A database logically is made up of **tables** **(≡ entities ≡ relations)** and their relationships.
* Two **views of a database**:
  1. **Logical** view of database
  2. **Physical** implementation of the database

## 1.1 Logical View of a Database

* **Table (≡ entities ≡ relations):** 2D structure composed of **rows (≡ tuples ≡ records)** and columns.
* **Tuple (≡ row ≡ record):** Represents a single entity occurrence within the **entity** **(≡ table ≡ relations)** set.
* **Column (≡ attribute ≡ field):** Each column represents an attribute and each column has a distinct name.
* Each **intersection** of a row and column represents a **single data value**.
* All **values in a column** must conform to the **same data format**.
* **(Attribute) domain**: The specific range of values of each column.
* The order of the rows and columns do not matter to the DBMS (Database Management System)
* Each **table (≡ entity ≡ relation)** must have an **attribute (≡ column ≡ field)** or combination of **attributes (≡ columns ≡ fields)** that uniquely identifies each row.



## 1.2 Keys

* ensure each row in a table is uniquely identifiable.
* consist of one or more **attributes (≡ columns ≡ fields)** that determine other attributes.

The first thing we want to do with the table is to identify a primary key.

* **Primary key (PK):** is an **attribute (≡ column ≡ field)** or combination of attributes that **uniquely identifies any given row.** In this case, STU\_NUM (the student number) is the primary key. Using the data in Figure 1.1, observe that a student's last name (STU\_LNAME) would not be a good primary key because several students have the last name of Smith. Even the combination of the last name and first name (STU\_FNAME) would not be an appropriate primary key because more than one student is named John Smith.

### Role of Keys

* **Determination:** knowing the value of one attribute makes it possible to determine the value of another.

*Example:* If you look at Figure 1.1, you can see that you if you are given STU\_NUM, you can determine the value for STU\_LNAME. This relationship is called **functional dependence** (the value of one or more **attributes (≡ column ≡ field)** determines the value of one or more other **attribute (≡ column ≡ field)**.



* + **Determinant:** The **attribute (≡ column ≡ field)** whose value determines another (STU\_NUM).
  + **Dependent:** The **attribute (≡ column ≡ field)** whose value is determined by the other attribute (STU\_LNAME).
  + STU\_NUM **functionally determines** STU\_LNAME  
    STU\_LNAME is **functionally dependent** on STU\_NUM
* Keys are determinants in functional dependencies.

### Types of Keys

* **Composite Key:** A key composed of more than one **attribute (≡ column ≡ field)**.
  + An **attribute (≡ column ≡ field)**  that is part of a key is called a **key attribute**.

*Example 1:*



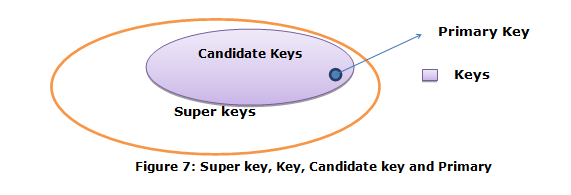
STU\_NUM is a key composed of only one key attribute.

*Example 2:*



(STU\_LNAME, STU\_FNAME, STU\_INIT, STU\_PHONE) is a composite key composed of four key **attributes (≡ columns ≡ fields)**.

* **Superkey**: A key that can uniquely identify any row in the table. A superkey functionally determines every attribute in the row. An **attribute** **(≡ columns ≡ fields)** or combination of attributes that uniquely identifies each row in a table. A superkey can in combination with other columns **(≡ attributes ≡ fields)**, or they can stand alone. In the STUDENT table, STU\_NUM is a superkey, as are the composite keys (STU\_NUM, STU\_LNAME), (STU\_NUM, STU\_LNAME, STU\_INIT), and (STU\_LNAME, STU\_FNAME, STU\_INIT, STU\_PHONE). In fact, because STU\_NUM alone is a superkey, any composite key that has STU\_NUM as a key attribute will also be a superkey. Be careful, however, because not all keys are superkeys.



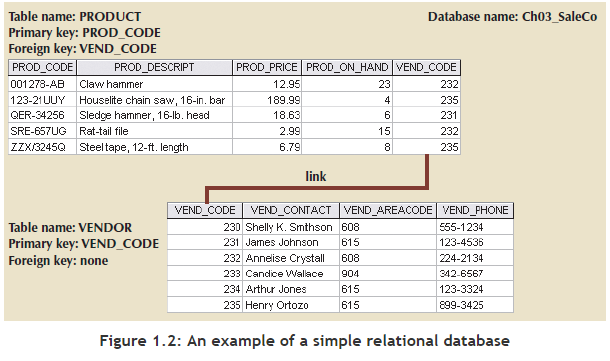
***Note:***

* + You cannot define a table with a superkey.
  + A superkey is a theoretical construct only.
* **Candidate key:** A column or set of columns in a table that uniquely identifies any database row without referring to any other data. A **minimal superkey** without any unnecessary attributes. For example, STU\_NUM would be a candidate key, as would (STU\_LNAME, STU\_FNAME, STU\_INIT, STU\_PHONE). On the other hand, (STU\_NUM, STU\_LNAME) is a superkey, but it is not a candidate key because STU\_LNAME could be removed and the key would still be a superkey.

A table can have many different candidate keys. If the STUDENT table also included the students' Social Security numbers as STU\_SSN, then it would appear to be a candidate key. Candidate keys are called candidates because they are the eligible options from which the designer will choose when selecting the primary key. The primary key is the candidate key chosen to be the primary means by which the rows of the table are uniquely identified.

***Note:***

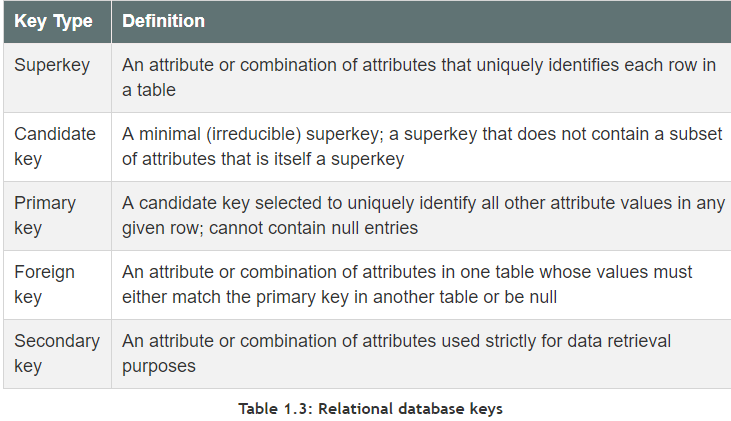
* + A candidate key is a theoretical construct only.
* **Entity Integrity**: The condition in which each **row** (**≡ entity instance)** in the table has its own unique identity. To ensure integrity, the primary key has two requirements:
  + All of the values in the primary key must be unique.
  + No key attribute in the primary key can contain a null.
* **Null:** Absence of any data value. A null could represent any of the following:
  + An unknown attribute value
  + A known, but missing, attribute value
  + A "not applicable" condition
* **Foreign key (FK):** is the primary key of one table that has been placed into another table to create a common attribute. A field or set of fields in one table that individually identifies a **tuple (≡ row ≡ record)** of another table.



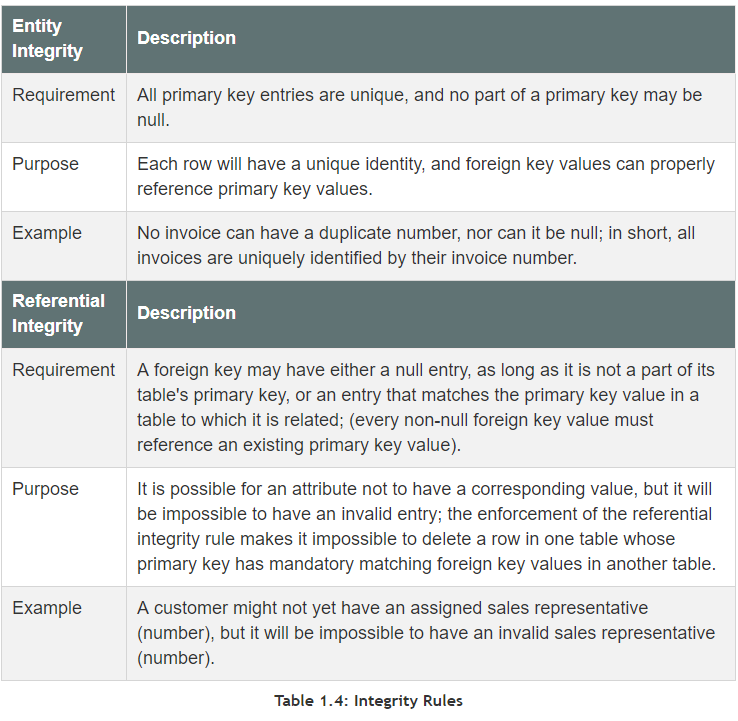
* + Foreign keys are used to ensure **referential integrity.**
* **Referential Integrity**: The condition in which every reference to an **entity instance (≡ row)** by another entity instance is valid, i.e. every foreign key must either be null or a valid value in the primary key of the related table.
* **Secondary key**: a key used strictly for data retrieval purposes (it is kind of an index). Suppose that customer data is stored in a CUSTOMER table in which the customer number is the primary key. Do you think that most customers will remember their numbers? Data retrieval for a customer is easier when the customer's last name and phone number are used. In that case, the primary key is the customer number; the secondary key is the combination of the customer's last name and phone number. Keep in mind that a secondary key does not necessarily yield a unique outcome.

***Note:***

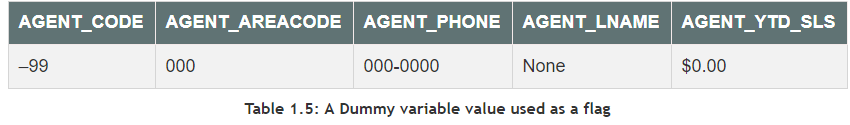
* + A secondary key is a theoretical construct only.



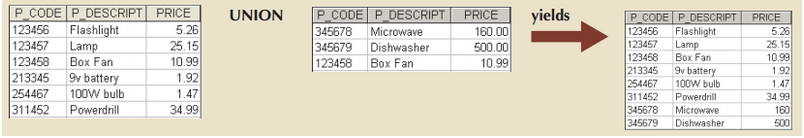
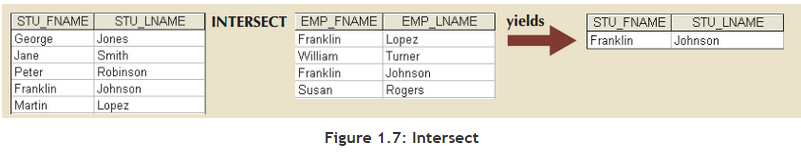
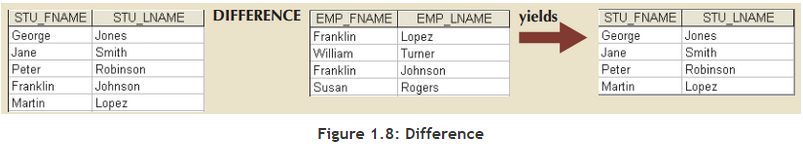
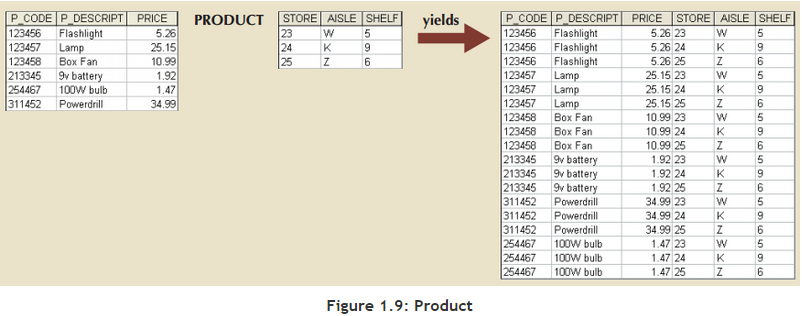
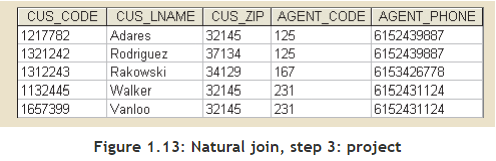
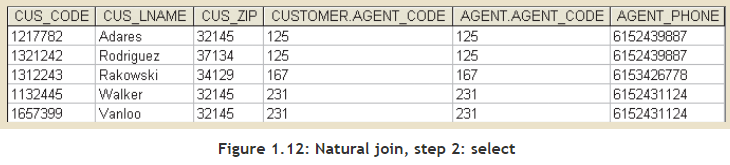
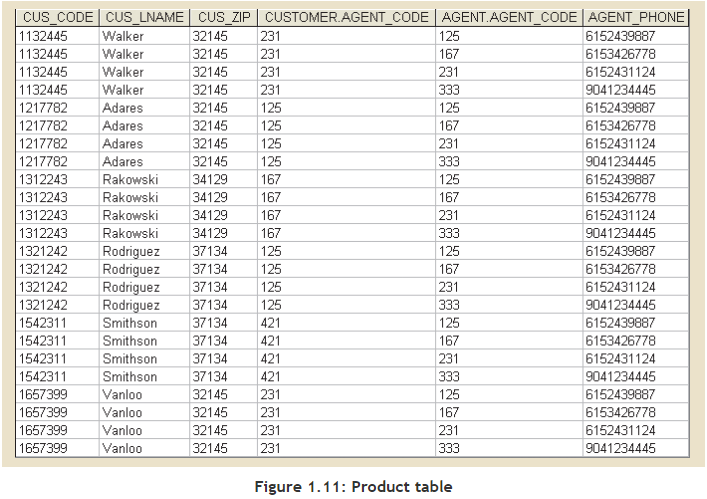
## 1.3 Integrity Rules

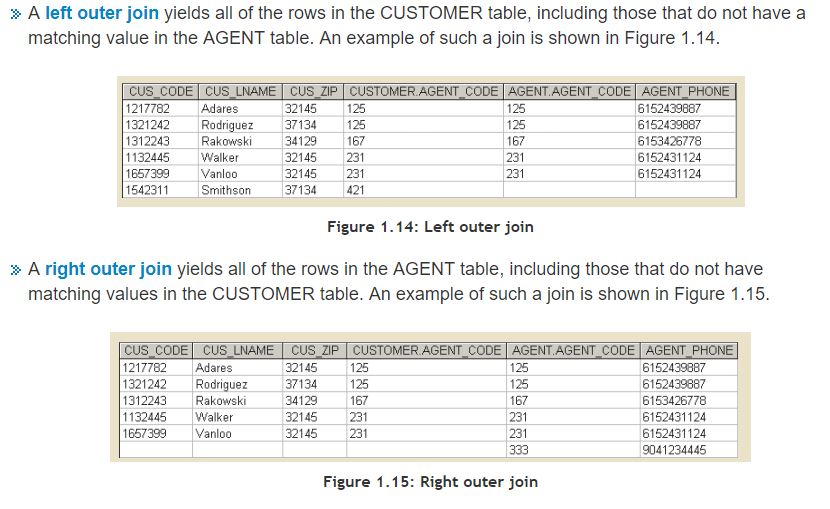


* **Flags:** used to avoid nulls and indicate the absence of some value.



## 1.4 Relational Algebra

* **SELECT (σ) (≡ RESTRICT):** Yields values for all rows found in the table that satisfy a given condition.  
  
* **PROJECT (π):** Selects selected columns  
  
* **UNION (∪):** Combines all rows from two tables, **excluding duplicate** rows. Tables must have compatible domains and columns (**union-compatible**).  
    
  
* **INTERSECT (∩):** Yields only the rows that appear on both tables. Tables must be union-compatible (e.g. both numeric or both character-based).  
    
  
* **DIFFERENCE (-):** Yields all rows in one table that are not found in the other table, i.e. it subtracts one table from the other. Tables must be union-compatible.  
    
  
* **PRODUCT (x):** Yields all possible pairs of rows from two tables (also known as **Cartesian product**). Therefore, if one table has 6 rows and the other has 3 rows, the PRODUCT yields a list composed of 18 rows. 
* **JOIN:** Allows information to be intelligently combined from two or more tables. 
  + **Natural Join (⨝):** **links tables by selecting only rows with common values in their common attributes( ≡ columns ≡ fields).**   
    
  + **Equijoin (⨝θ):** Links tables based on an equality condition that compares specified columns of each table. Equijoin does not eliminate duplicate columns, and the condition used to join the tables must be explicitly defined. In fact, the result of an equijoin looks just like Figure 1.12 of a natural join. The equijoin takes its name from the equality comparison operator (=) used in the condition. If any other comparison operator is used, the join is called a **theta join**.
  + **Inner Join:** Only returns matched records from the tables that are being joined.
  + **Outer Join:** The matched pairs would be retained, and any unmatched values in the other table would be left null.

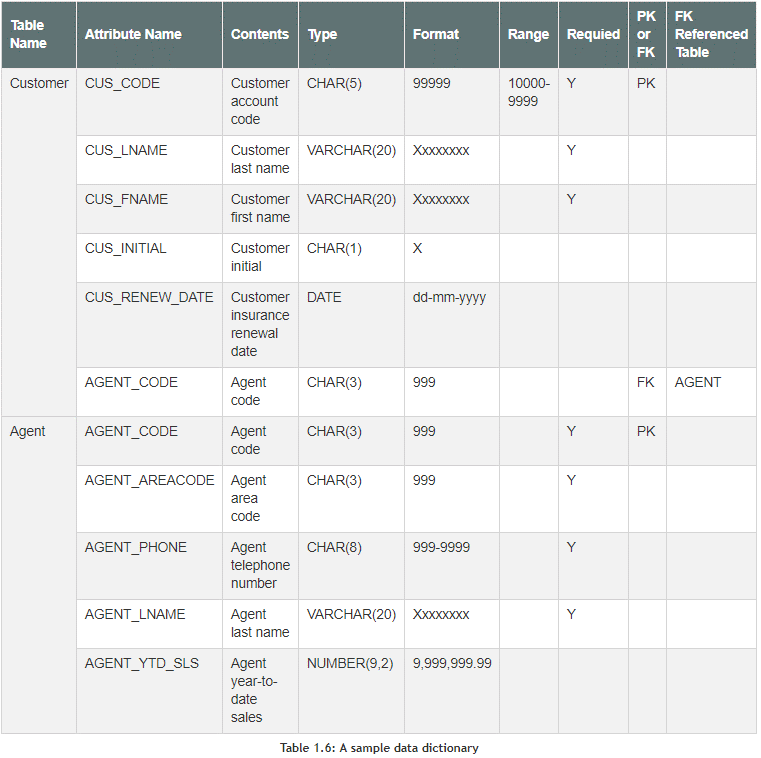


* **Divide:** Used to answer questions about one set of data being associated with all values of data in another set of data. The DIVIDE operation uses one 2-column table (Table 1) as the dividend and one single-column table (Table 2) as the divisor. For example, Figure 1.16 shows a list of customers and the products purchased in Table 1 on the left. Table 2 in the center contains a set of products that are of interest to the users. A DIVIDE operation can be used to determine which customers, if any, purchased every product shown in Table 2. In the figure, the dividend contains the P\_CODE and CUS\_CODE columns. The divisor contains the P\_CODE column. The tables must have a common column—in this case, the P\_CODE column. The output of the DIVIDE operation on the right is a single column that contains all values from the second column of the dividend (CUS\_CODE) that are associated with every row in the divisor.

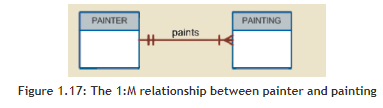
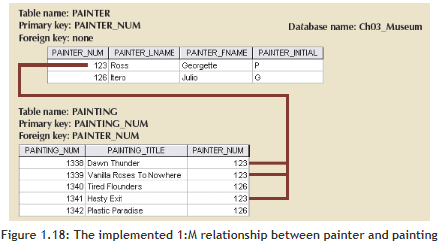
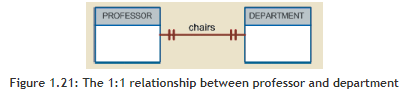
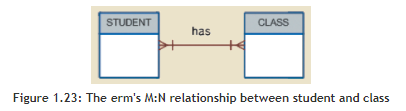
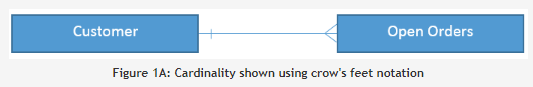
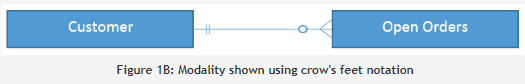


## 1.5 The Data Dictionary ≡ System Catalog

* **Data Dictionary (≡ System Catalog**): Provides a detailed description of all tables in the database created by the user and designer. It contains at least all the **attribute** **(≡ column ≡ field)** names and characteristics for each table in the system. Therefore, it contains metadata – data about data.
* **System Catalog (≡ Data Dictionary):** A detailed system data dictionary that describes all objects within the database, including data about table names, table’s creator and creation date, number of columns in each table, data type corresponding to each column, index filenames, index creator, authorized users, and access privileges.



## 1.6 Relationships within the Relational Database

* The **1:M relationship** is the **relational modeling ideal**. Therefore, this relationship type should be the norm in any relational database design.  
    
  As you examine the PAINTER and PAINTING table contents in Figure 1.18, note the following features:
  + Each painting was created by one and only one painter, but each painter could have created many paintings. Note that painter 123 (Georgette P. Ross) has three works stored in the PAINTING table.
  + There is only one row in the PAINTER table for any given row in the PAINTING table, but there may be many rows in the PAINTING table for any given row in the PAINTER table.  
    
* The **1:1 relationship** should be rare in any relational database design. One entity in a 1:1 relationship can be related to only one other entity, and vice versa. For example, one department chair—a professor—can chair only one department, and one department can have only one department chair.  
  
* **M:N relationships:** A many-to-many (M:N) relationship is not supported directly in the relational environment. However, M:N relationships can be implemented by creating a new entity in 1:M relationships with the original entities.  
  
  + Each CLASS can have many STUDENTs, and each STUDENT can take many CLASSes.
  + There can be many rows in the CLASS table for any given row in the STUDENT table, and there can be many rows in the STUDENT table for any given row in the CLASS table.
  + Problems inherent in the M:N relationships can be avoided by creating a **composite entity (≡ bridge entity ≡ associative entity**).
* The **degree of a relationship** is the number of entity types that participate in the relationship. The three most common relationships in ER models are Binary, Unary and Ternary.
* **A binary relationship** is when two entities participate and is the most common relationship degree.
* **A unary relationship** is when both participants in the relationship are the same entity.
* **A ternary relationship** is when three entities participate in the relationship.
* **Cardinality:** How many instances of one object are related to instances of another object. One order can be assigned to only one customer but a customer can have multiple orders (1:M). Cardinality can be shown in crow's feet notations, utilizing a straight line for 1 and a crow's foot for many.  
  
* **Modality:** Denotes an instance of a specific entity is optional or mandatory in a relationship. For instance, an open order must have a customer associated with it, but every customer does not need to have an open order. Modality is expressed with a straight line for modality 1 or a circle for modality 0. These are drawn in the relationship inside of the cardinality symbols.  
  

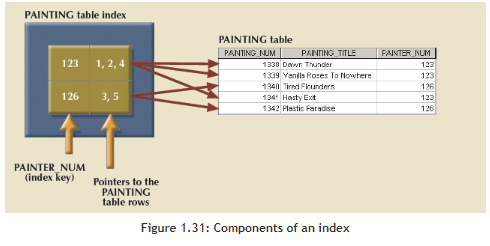
In other words**, modality** refers to the **minimum** number of times an instance of an entity can be associated with an instance in a related entity while **cardinality** refers to the **maximum** number of times this relationship may occur.

## 1.7 Data Redundancy Revisited

* Data redundancy leads to data anomalies, which can destroy the effectiveness of the database and the relational database makes it possible to control data redundancies by using common attributes that are shared by tables, called foreign keys.

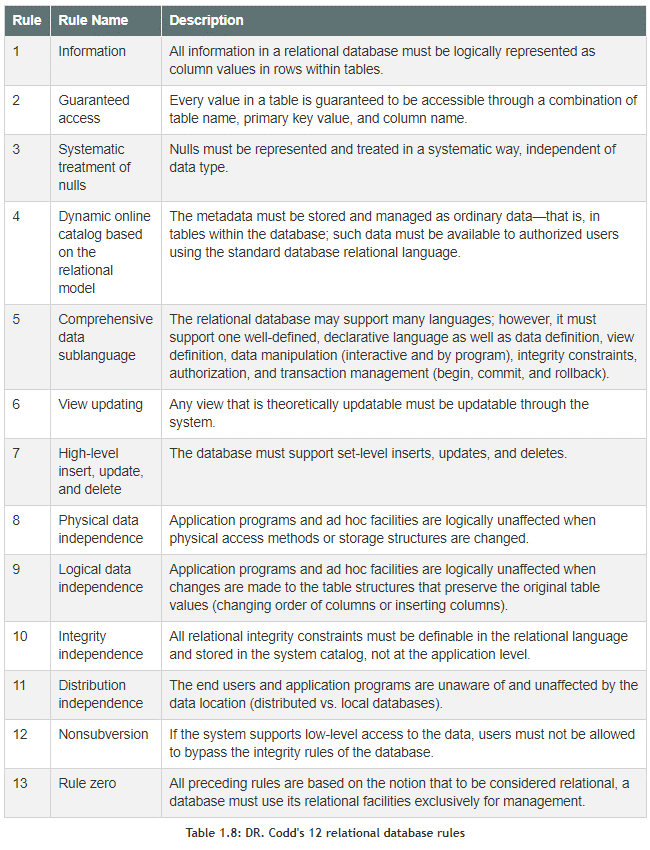
## 1.8 Indexes

* **Index:** An orderly arrangement used to logically access rows in a table. It is composed of an index key and a set of pointers.
* **Index Key:** The index’s reference point; an ordered arrangement of keys and pointers.



* **Unique Index:** The index key can only have one pointer value (row) associated with it.

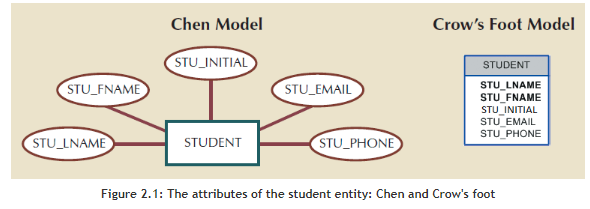
## 1.9 Codd’s Relational Database Rules

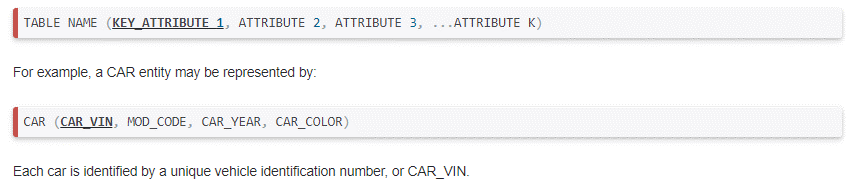


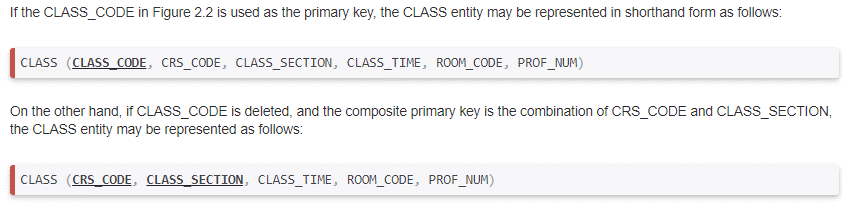
# 2 - ER-Modeling

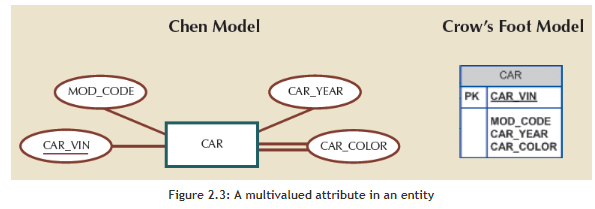
## 2.1 The Entity Relationship Model (ERM)

* The Chen notation favors conceptual modeling.
* The Crow's Foot notation favors a more implementation-oriented approach.
* The UML notation can be used for both conceptual and implementation modeling

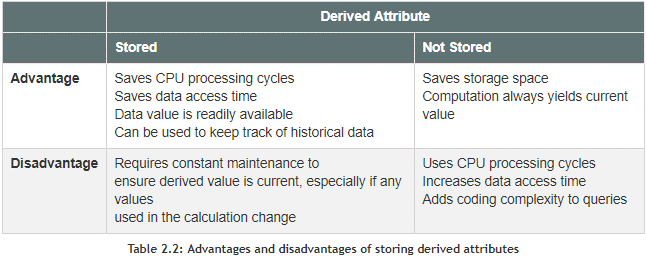


* **Required and Optional Attributes:** A required attribute is an attribute that must have a value; in other words, it cannot be left empty.
* An **optional attribute** is an attribute that does not require a value; therefore, it can be left empty.
* **Domains:** Attributes have a domain. **A domain** is the set of **possible values for a given attribute**. For example, the domain for a grade point average (GPA) attribute is written (0,4) because the lowest possible GPA value is 0 and the highest possible value is 4. The domain for a company's date of hire attribute consists of all dates that fit in a range (for example, company startup date to current date).
* **Identifiers (Primary Keys):** The ERM uses identifiers—one or more attributes that uniquely identify each entity instance. In the relational model, entities are mapped to tables, and the entity identifier is mapped as the table's primary key (PK). Identifiers are underlined in the ERD. Key attributes are also underlined in a frequently used shorthand notation for the table structure, called a relational schema, that uses the following format:  
  
* **Composite Identifiers:** Ideally, an entity identifier is composed of only a single attribute. For example, the table in Figure 2.2 uses a single-attribute primary key named CLASS\_CODE. However, it is possible to use a composite identifier, a primary key composed of more than one attribute.

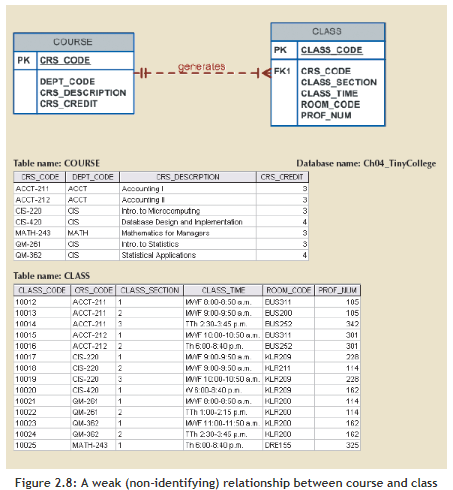
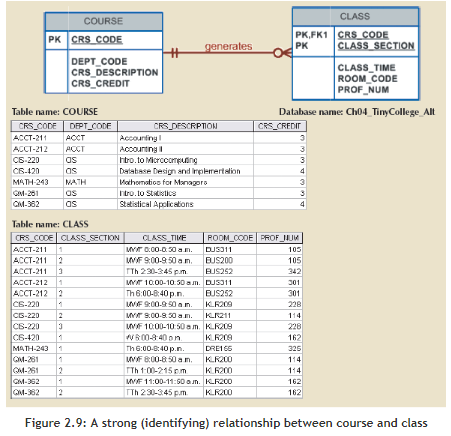


* **Composite and Simple Attributes:** Attributes are classified as simple or composite. A composite attribute, not to be confused with a composite key, is an attribute that can be further subdivided to yield additional attributes. For example, the attribute ADDRESS can be subdivided into street, city, state, and zip code. A simple attribute is an attribute that cannot be subdivided. For example, age, sex, and marital status would be classified as simple attributes.
* **Single-Valued Attributes:** A single-valued attribute is an attribute that can have only a single value. For example, a person can have only one Social Security number, and a manufactured part can have only one serial number. Keep in mind that a single-valued attribute is not necessarily a simple attribute. For instance, a part's serial number (such as SE-08-02-189935) is single-valued, but it is a composite attribute because it can be subdivided into the region in which the part was produced (SE), the plant within that region (08), the shift within the plant (02), and the part number (189935)
* **Multivalued Attributes:** Multivalued attributes are attributes that can have many values. For instance, a person may have several college degrees, and a household may have several different phones, each with its own number. Similarly, a car's color may be subdivided into many colors for the roof, body, and trim. In the Chen ERM, multivalued attributes are shown by a double line connecting the attribute to the entity. The Crow's Foot notation does not identify multivalued attributes.  
  

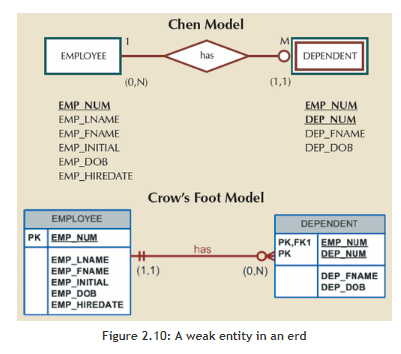


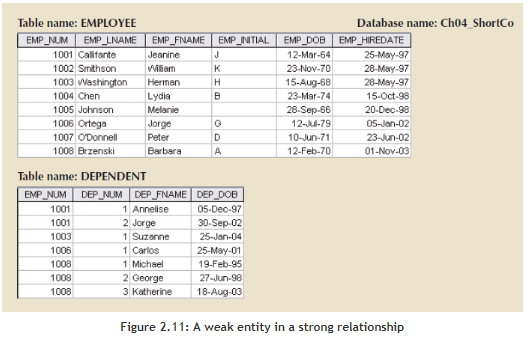
* **Derived Attributes:** Finally, a derived attribute is an attribute whose value is calculated (derived) from other attributes. The derived attribute need not be physically stored within the database; instead, it can be derived by using an algorithm. For example, an employee's age, EMP\_AGE, may be found by computing the integer value of the difference between the current date and the EMP\_DOB. If you use Microsoft Access, you would use the formula INT((DATE() - EMP\_DOB)/365).  
  

### 2.1.1 Relationships

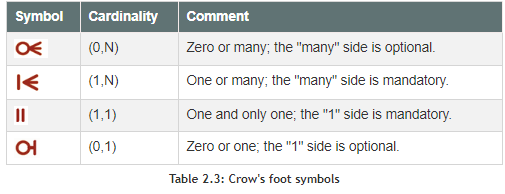
* **Participants:** the entities that participate in a relationship. **Relationships** between entities always **operate in both directions**. To define the relationship between the entities named CUSTOMER and INVOICE, you would specify that:
  + A CUSTOMER may generate many INVOICEs.
  + Each INVOICE is generated by one CUSTOMER.
  + The relationship above is 1:M.
* **Existence Dependence**
  + **Existence dependence:** An entity that can exist in the database only when it is associated with another related entity occurrence.
  + **Existence independence:** An entity that can exist apart from all its related entities. It is also referred to as a **strong entity** or **regular entity**.
* **Weak (Non-identifying) Relationships:** A weak relationship (≡ non-identifying relationship) exists if the primary key of the related entity does not contain a primary key component of the parent entity.   
    
    
  In this case, a weak relationship exists between COURSE and CLASS because CRS\_CODE (the primary key of the parent entity) is only a foreign key in the CLASS entity. In this example, the CLASS primary key did not inherit a primary key component from the COURSE entity. Figure 2.8 shows how the Crow's Foot notation depicts a weak relationship by placing a dashed relationship line between the entities. The tables shown below the ERD illustrate how such a relationship is implemented.
* **Strong (Identifying) Relationships:** Exists when the primary key of the related entity contains a primary key component of the parent entity.  
    
    
  In this case, the CLASS entity primary key is composed of CRS\_CODE and CLASS\_SECTION. Therefore, a strong relationship exists between COURSE and CLASS because CRS\_ CODE (the primary key of the parent entity) is a primary key component in the CLASS entity. In other words, the CLASS primary key did inherit a primary key component from the COURSE entity. (Note that the CRS\_CODE in CLASS is also the FK to the COURSE entity.)
* **Weak Entities**
  1. The entity is existence-dependent; it cannot exist without the entity with which it has a relationship.
  2. The entity has a primary key that is partially or totally derived from the parent entity in the relationship.

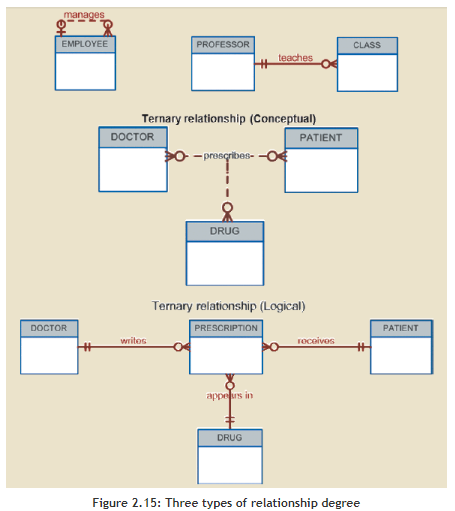
For example, a company insurance policy insures an employee and any dependents. For the purpose of describing an insurance policy, an EMPLOYEE might or might not have a DEPENDENT, but the DEPENDENT must be associated with an EMPLOYEE. Moreover, the DEPENDENT cannot exist without the EMPLOYEE; that is, a person cannot get insurance coverage as a dependent unless the person is a dependent of an employee. DEPENDENT is the weak entity in the relationship "EMPLOYEE has DEPENDENT."



Note that the Chen notation in Figure 2.10 identifies the weak entity by using a double-walled entity rectangle. The Crow's Foot notation generated by Visio Professional uses the relationship line and the PK/FK designation to indicate whether the related entity is weak. A strong (identifying) relationship indicates that the related entity is weak. Such a relationship means that both conditions have been met for the weak entity definition—the related entity is existence-dependent, and the PK of the related entity contains a PK component of the parent entity.  


* Figure 2.11 illustrates the implementation of the relationship between the weak entity (DEPENDENT) and its parent or strong counterpart (EMPLOYEE). Note that DEPENDENT's primary key is composed of two attributes, EMP\_NUM and DEP\_NUM, and that EMP\_NUM was inherited from EMPLOYEE.
* **Optional participation** means that one entity occurrence does not require a corresponding entity occurrence in a particular relationship. For example, in the "COURSE generates CLASS" relationship, you noted that at least some courses do not generate a class. In other words, an entity occurrence (row) in the COURSE table does not necessarily require the existence of a corresponding entity occurrence in the CLASS table.
* **Mandatory participation** means that one entity occurrence requires a corresponding entity occurrence in a particular relationship. If no optionality symbol is depicted with the entity, the entity is assumed to exist in a mandatory relationship with the related entity. If the mandatory participation is depicted graphically, it is typically shown as a small hash mark across the relationship line, similar to the Crow's Foot depiction of a connectivity of 1. The existence of a mandatory relationship indicates that the minimum cardinality is at least 1 for the mandatory entity.



* A **relationship degree** indicates the number of **entities(≡ tables ≡ relations)**or participants associated with a relationship. A **unary relationship** exists when an association is maintained within a single entity**(≡ table ≡ relation)**. A **binary relationship** exists when two **entities(≡ table ≡ relations)** are associated. A **ternary relationship** exists when three entities **(≡ table ≡ relations)** are associated.  
    
  

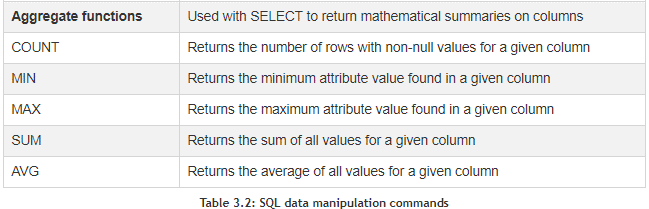
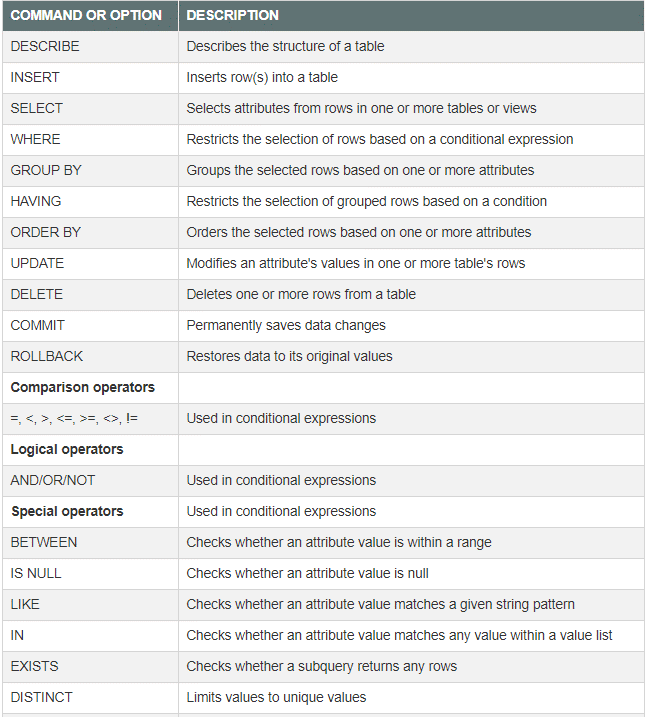
# 3 - Basic SQL

## 3.1 Introduction to SQL

SQL functions fit into two broad categories:

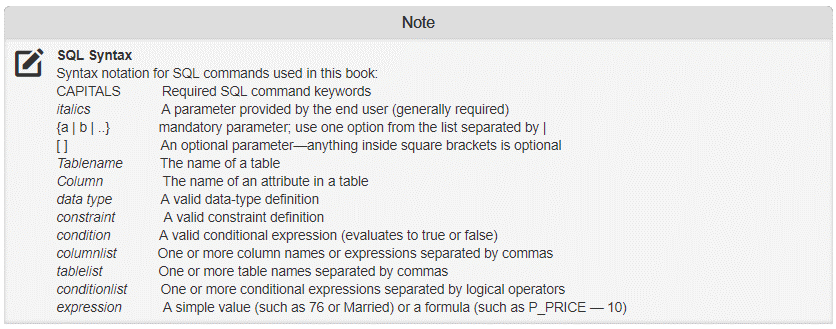
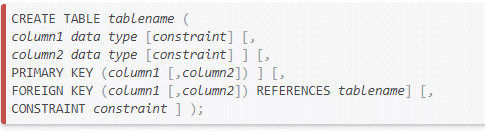
1. **Data Definition Language (DDL):** SQL includes commands to **create database objects** such as tables, indexes, and views, as well as commands to define access rights to those database objects.
2. **Data Manipulation Language (DML):** QL includes commands to **insert, update, delete**, and **retrieve** data within the database tables.

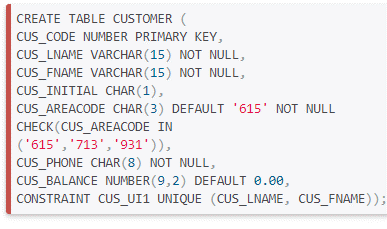




## 3.2 Data Definition Commands

- A schema is a logical group of database objects—such as tables and indexes—that are related to each other. Usually, the schema belongs to a single user or application. A single database can hold multiple schemas that belong to different users or applications. Schemas are useful in that they group tables by owner (or function) and enforce a first level of security by allowing each user to see only the tables that belong to that user.





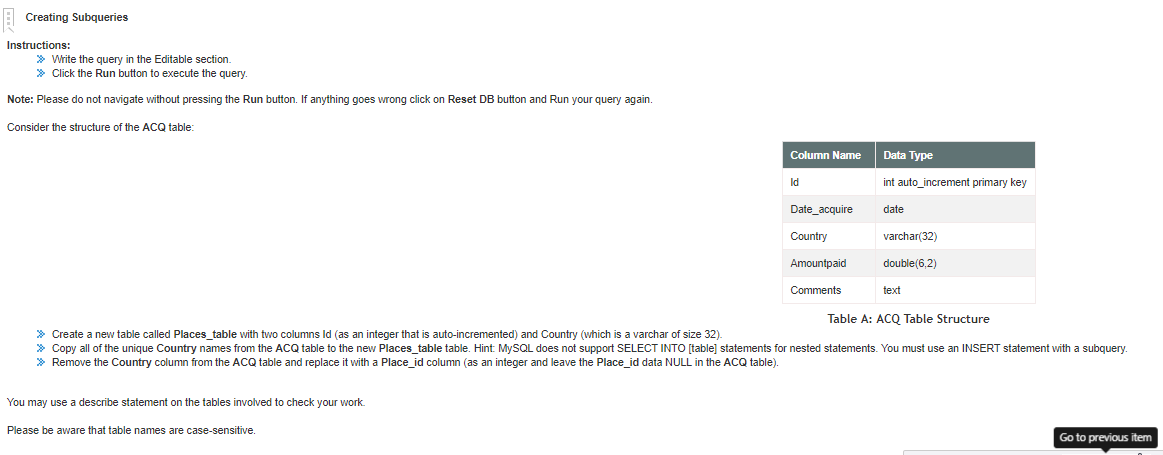
## 3.3 Data Manipulation Commands

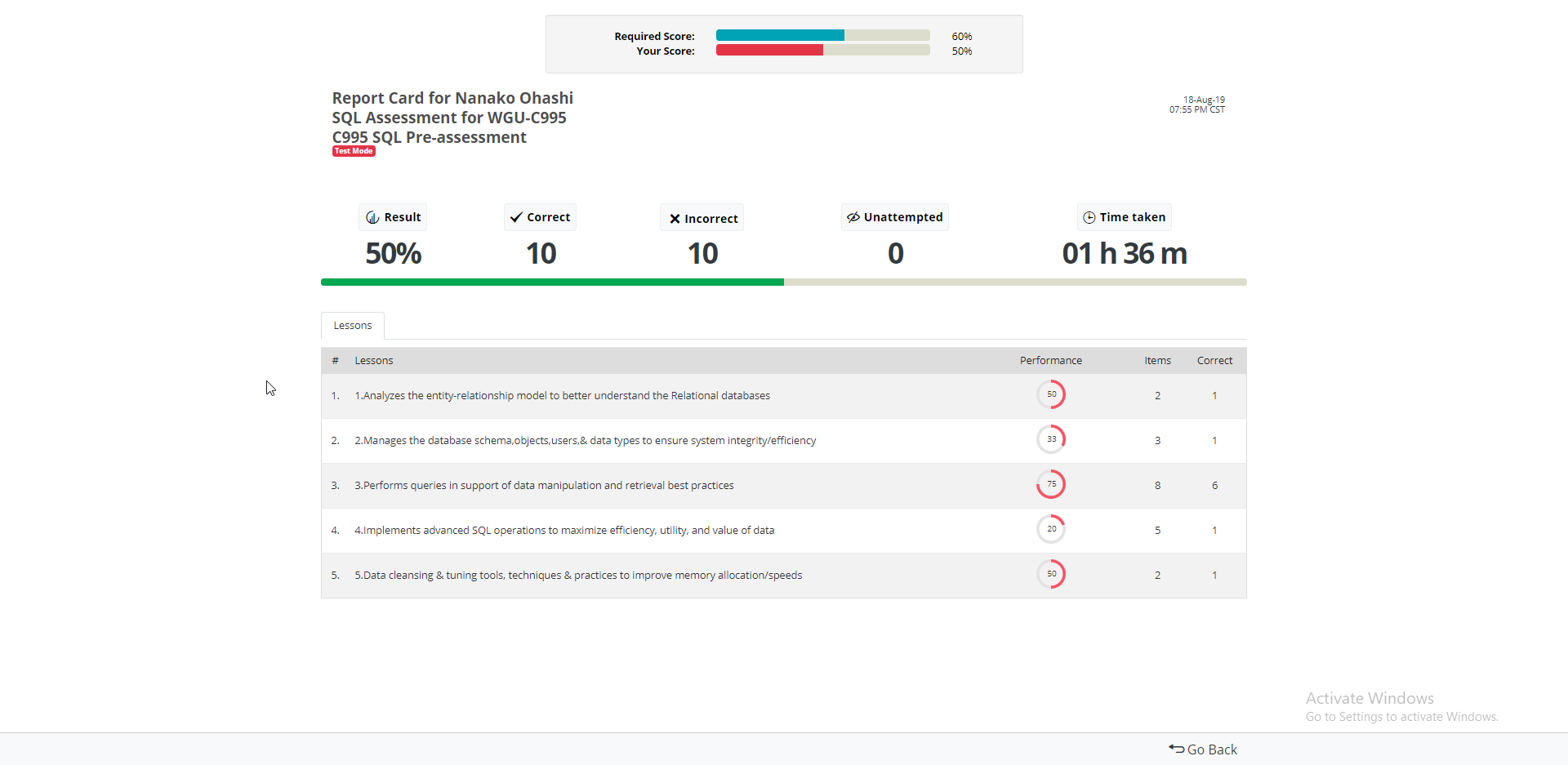
### 3.3.1 Adding Table Rows



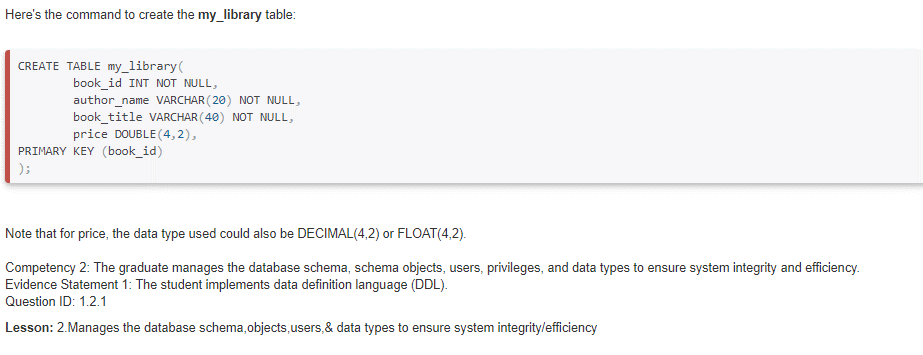
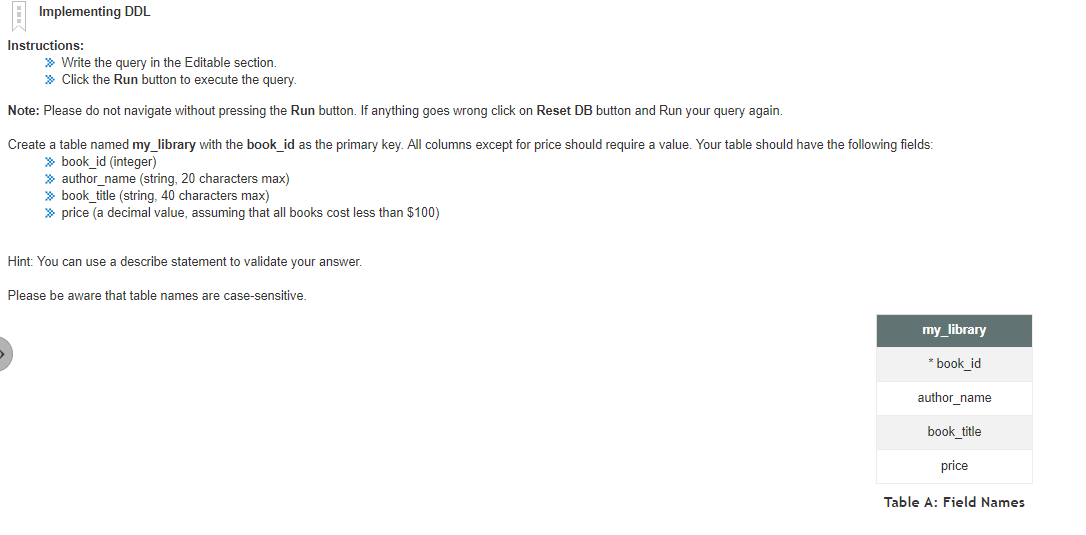
# Attempt 1

## Creating Subqueries

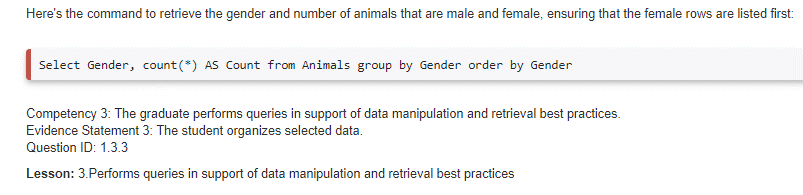
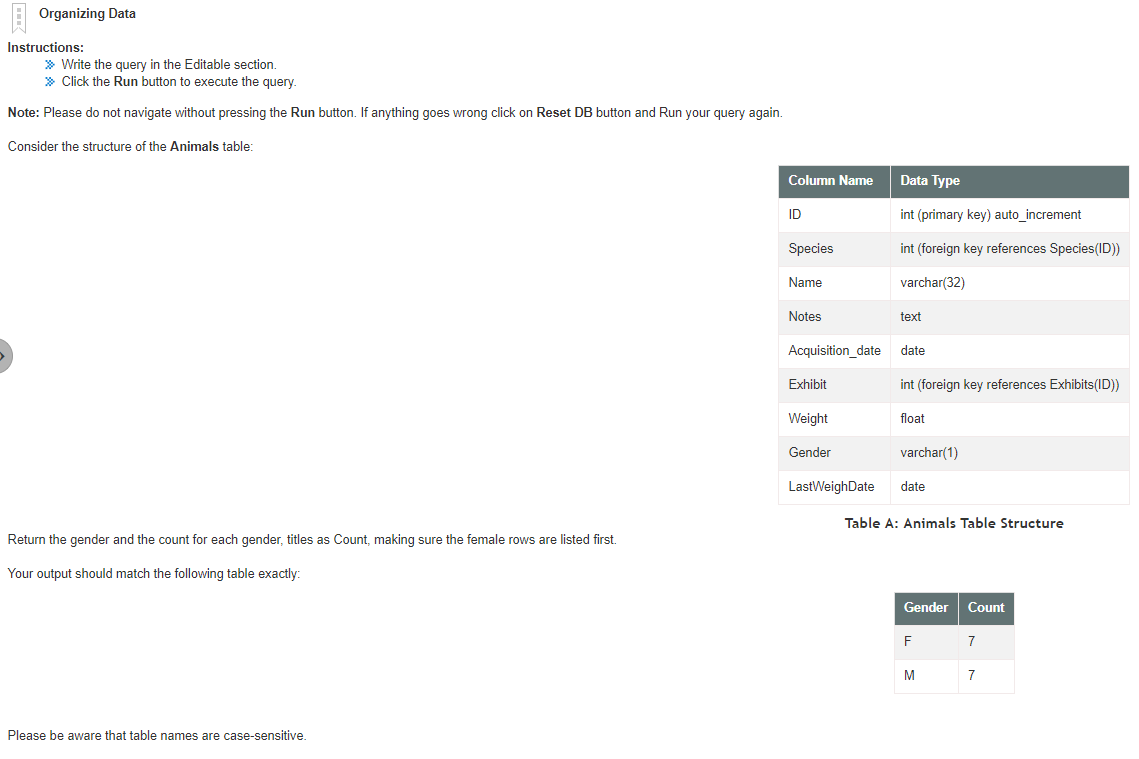




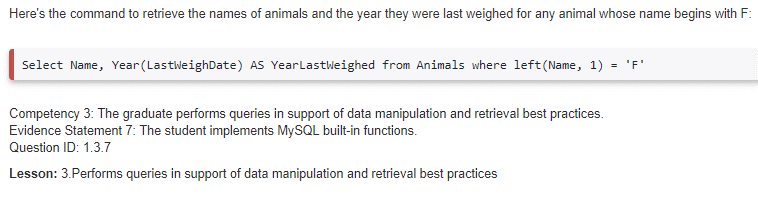
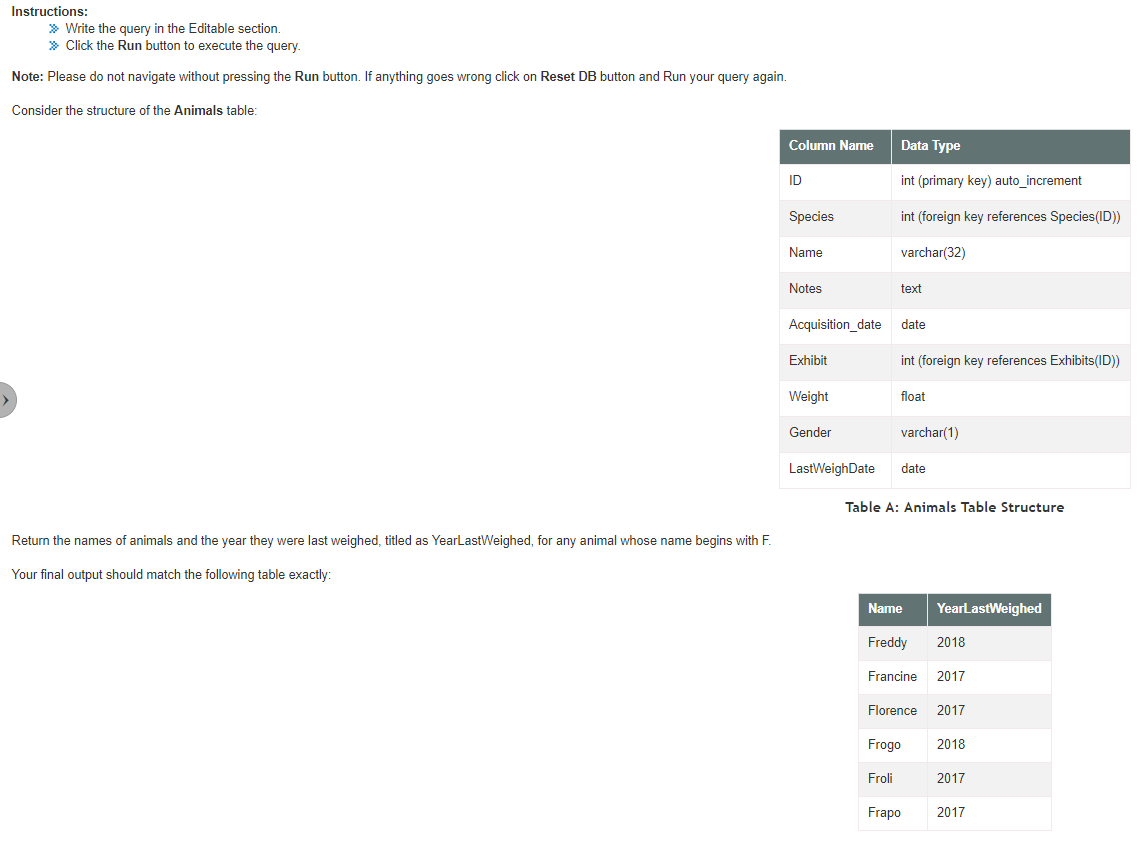
## Implementing DDL

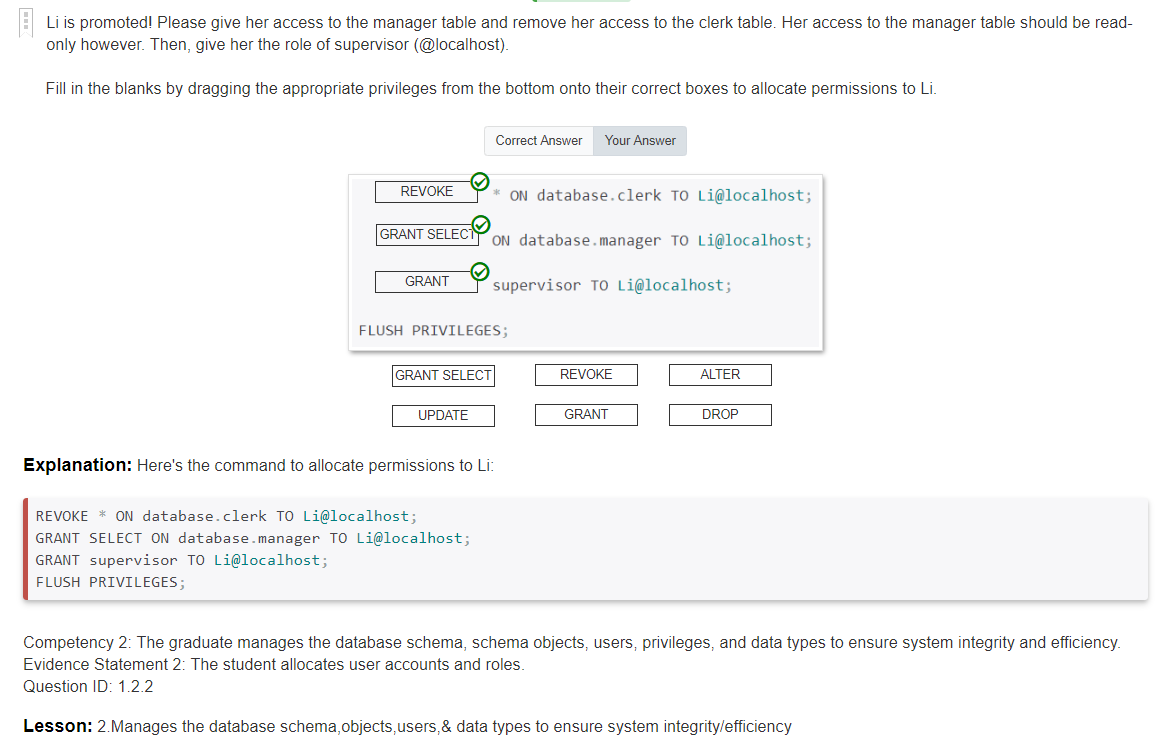
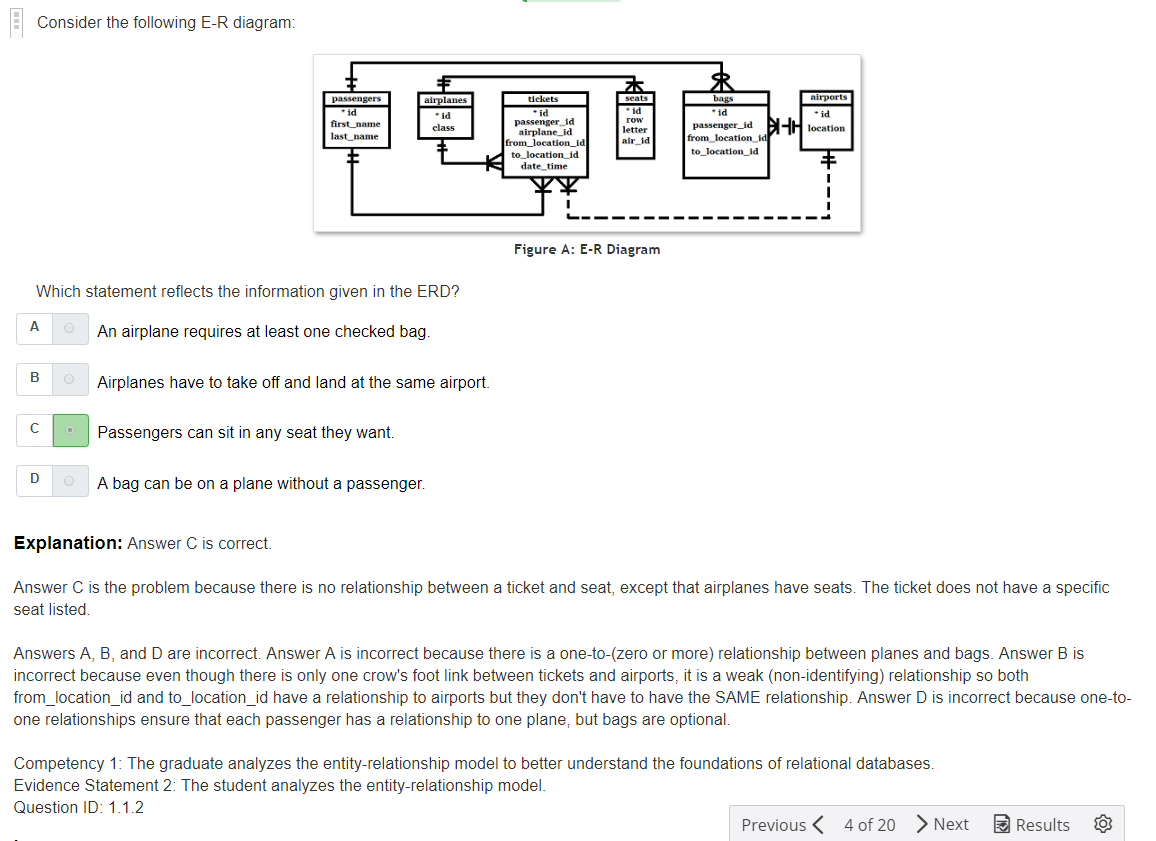


## Organizing Data

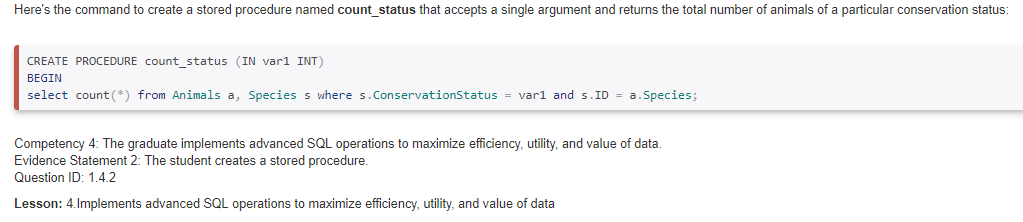
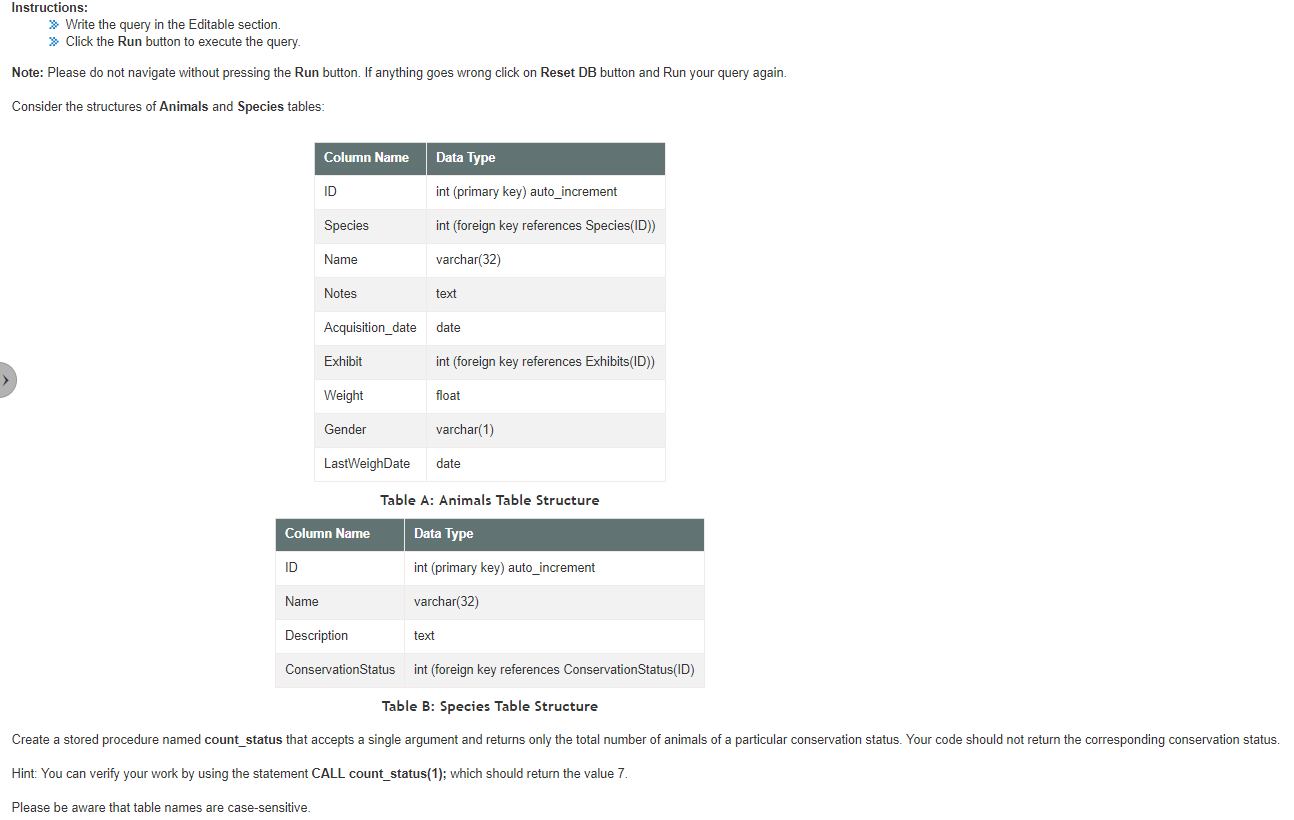


## Preforming Complex Table Operations





## Creating a Stored Procedure



**Formula IN Parameter**

DELIMITER $$

CREATE PROCEDURE *name of the procedure* (IN *name of IN parameter of stored procedure* TYPE)

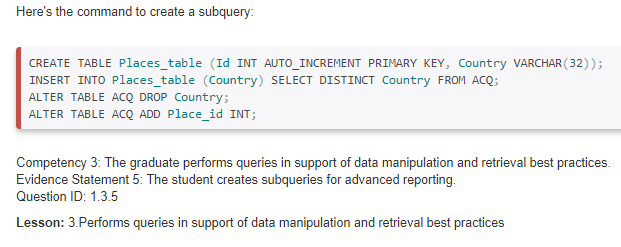
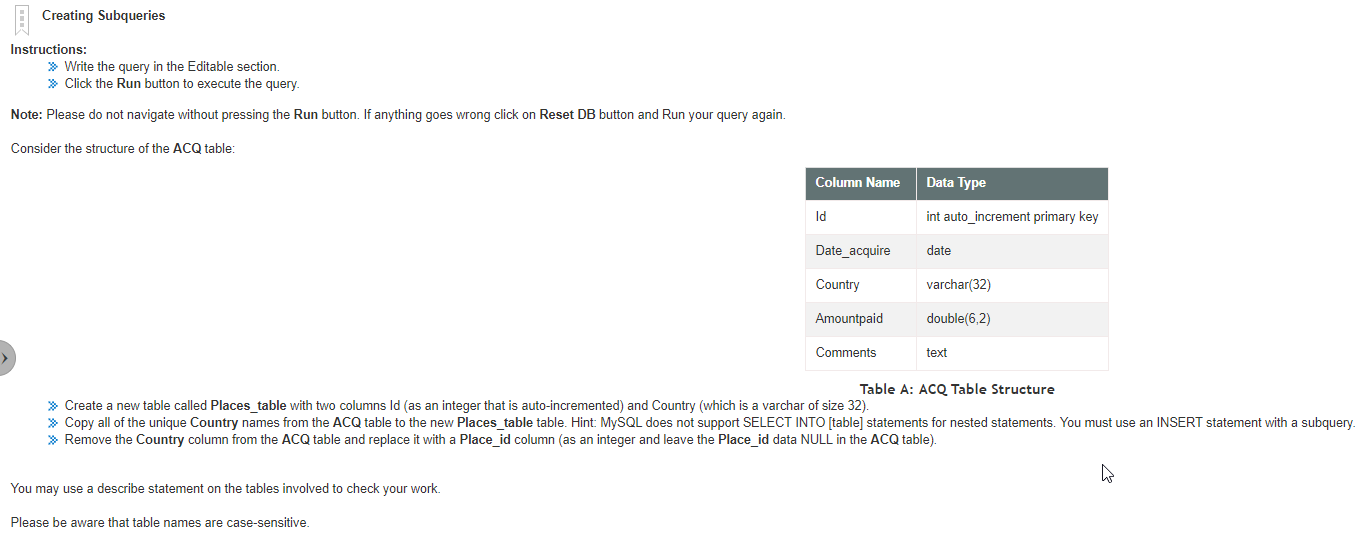
BEGIN

Here you put the function

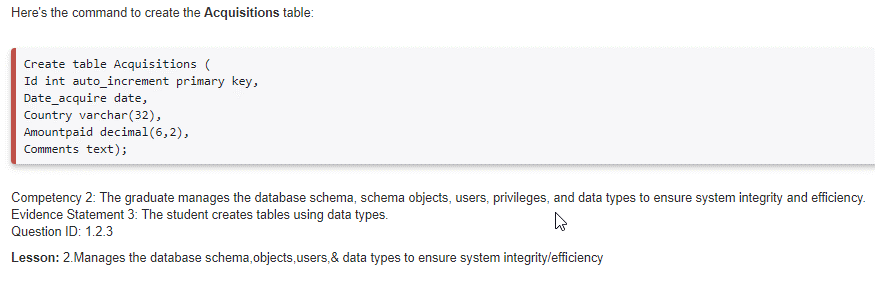
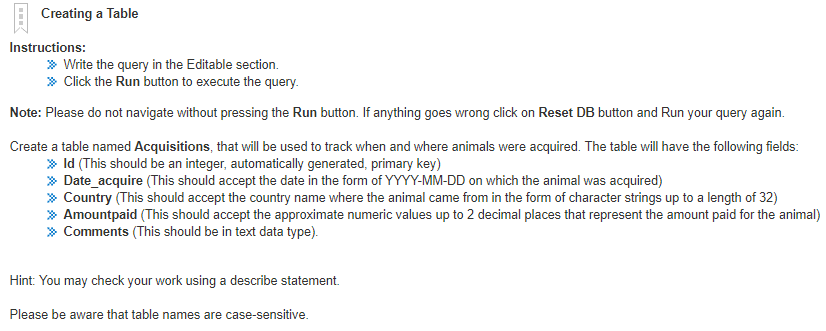
END

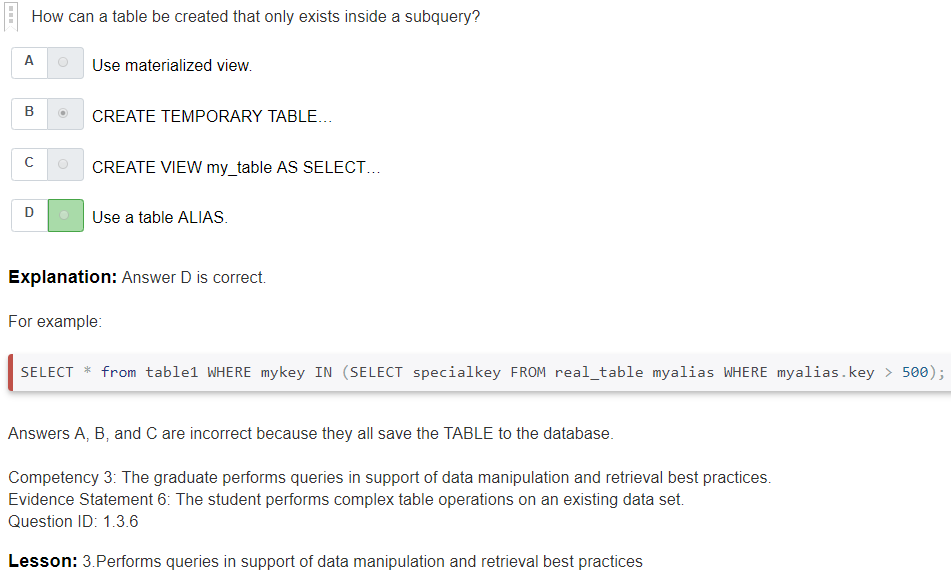
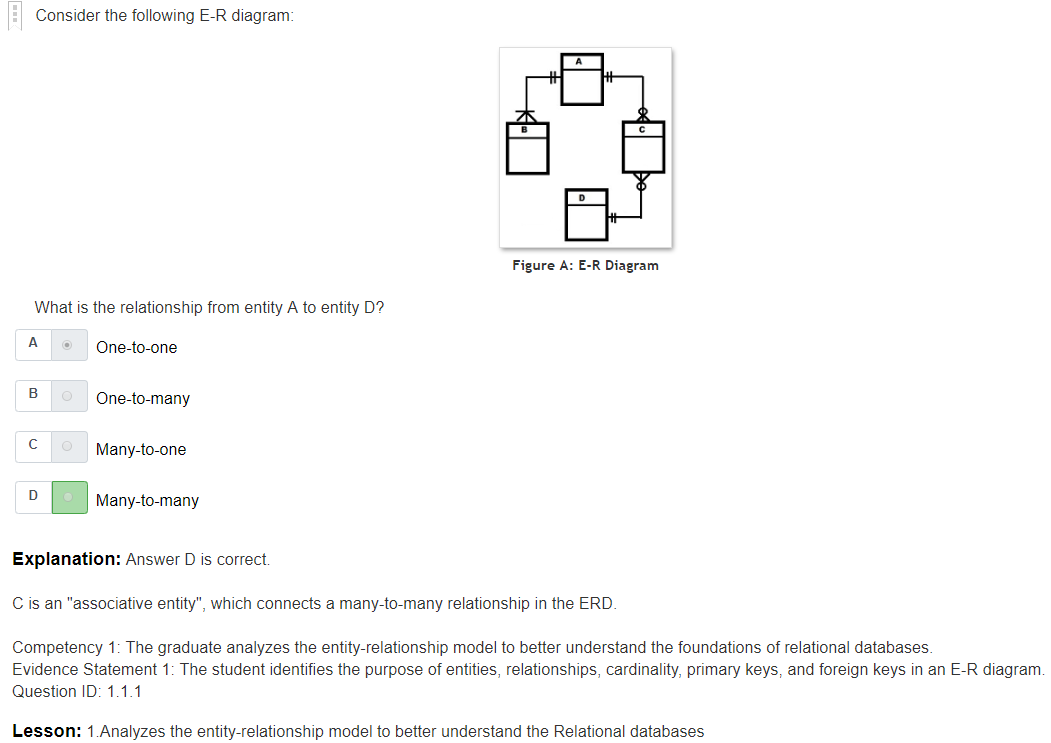
DELIMITER ;

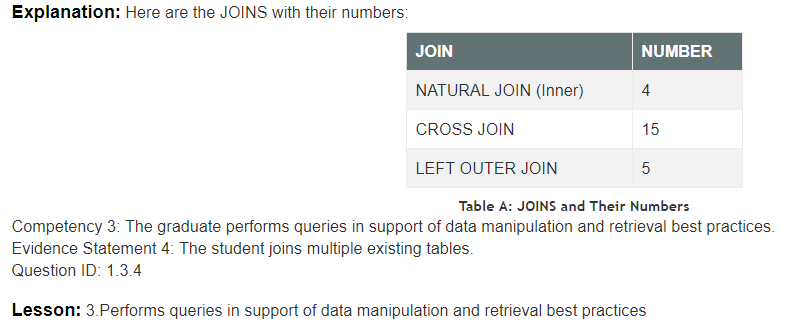
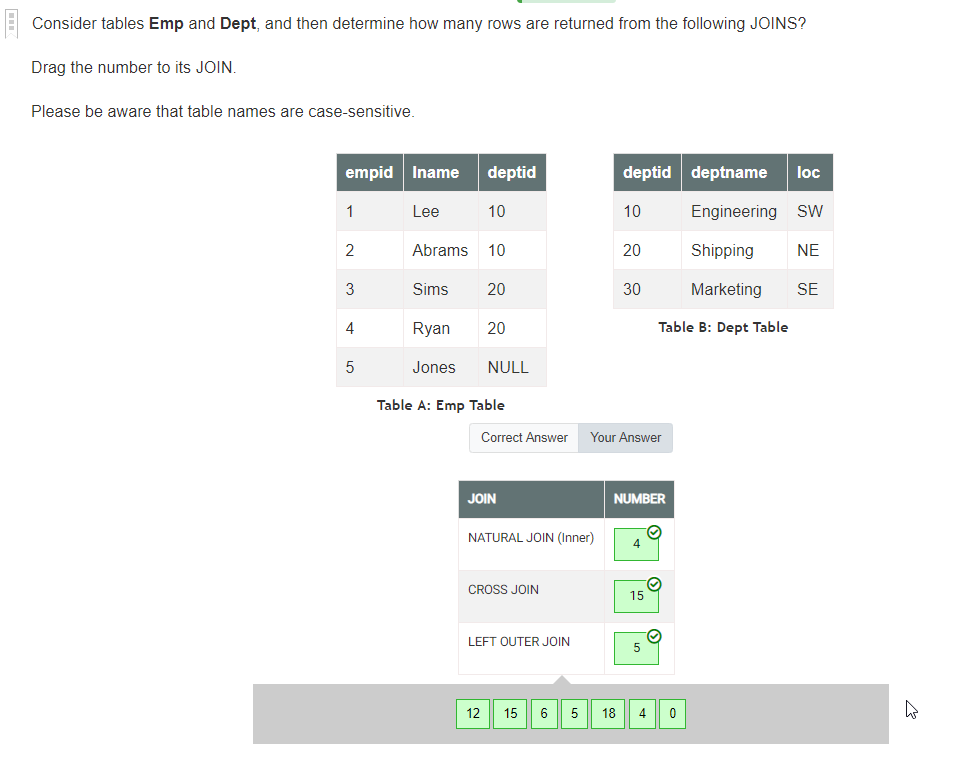
## Creating Subqueries



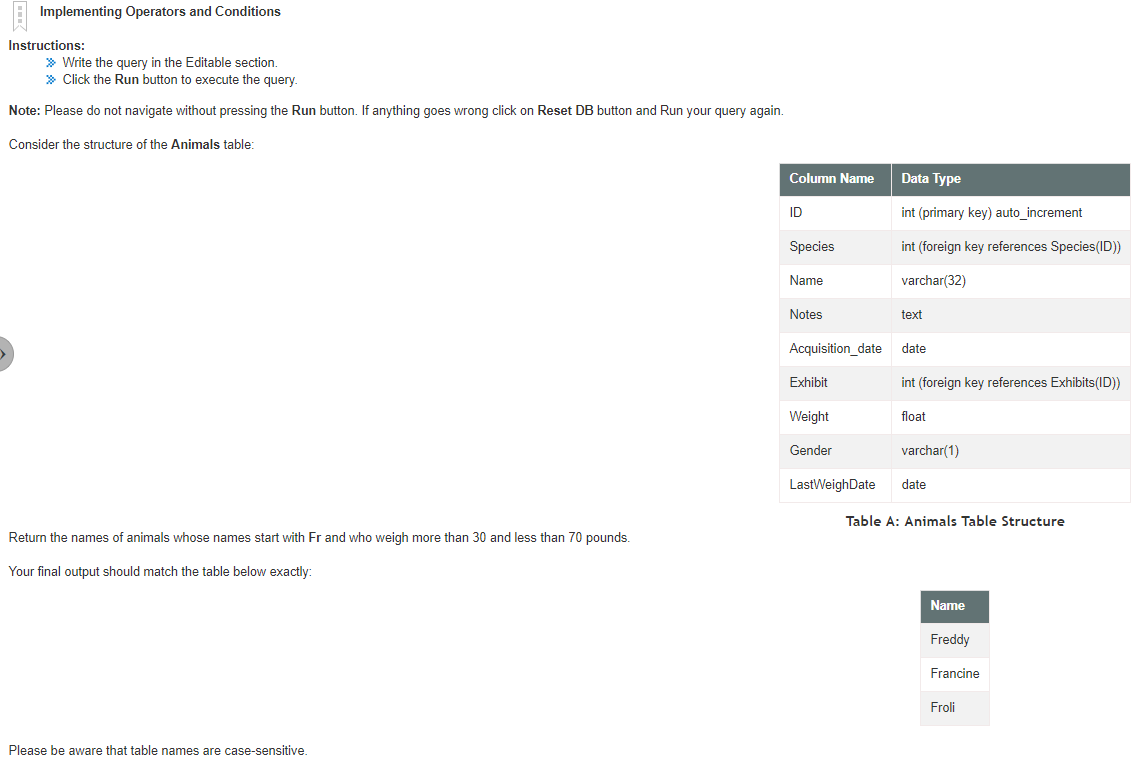
## Creating a Table

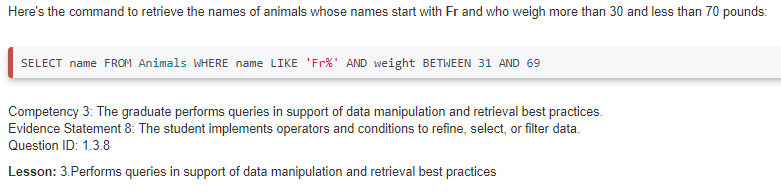


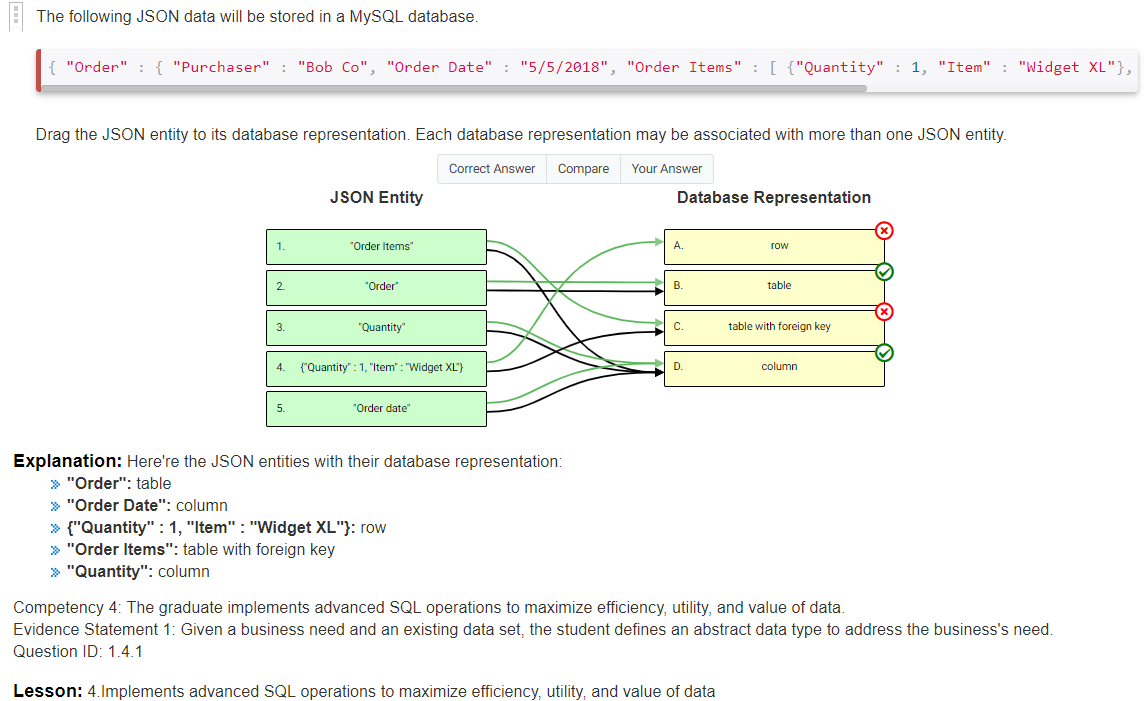
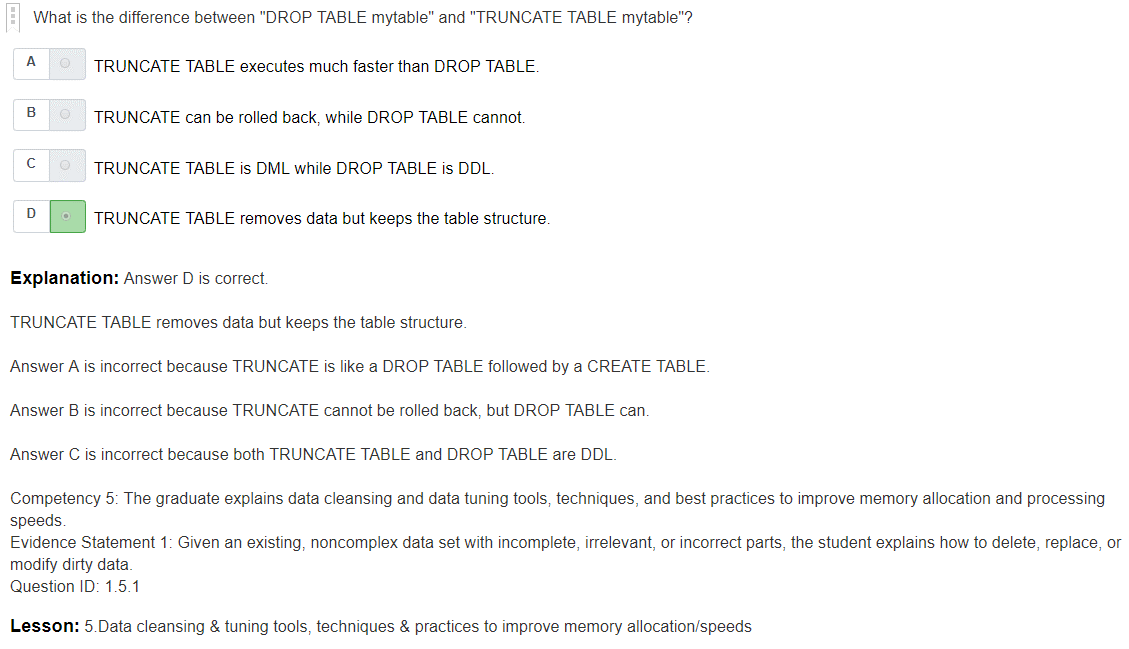


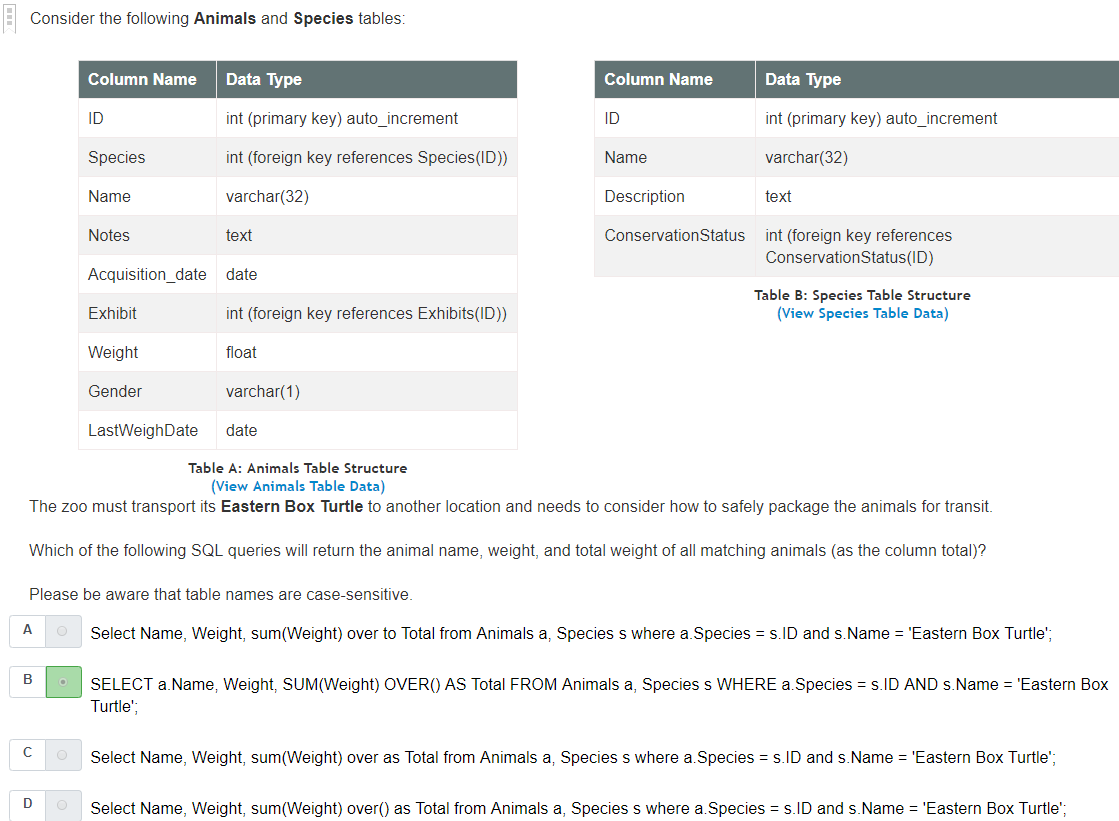


## Implementing Operators and Conditions



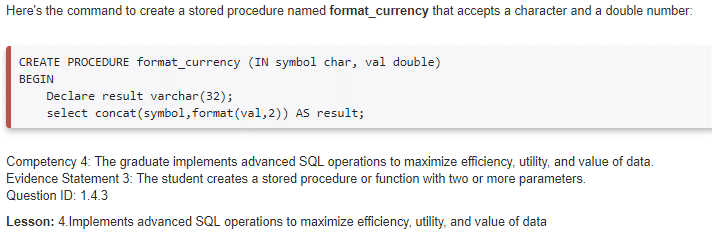




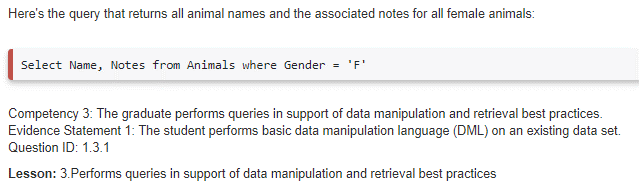
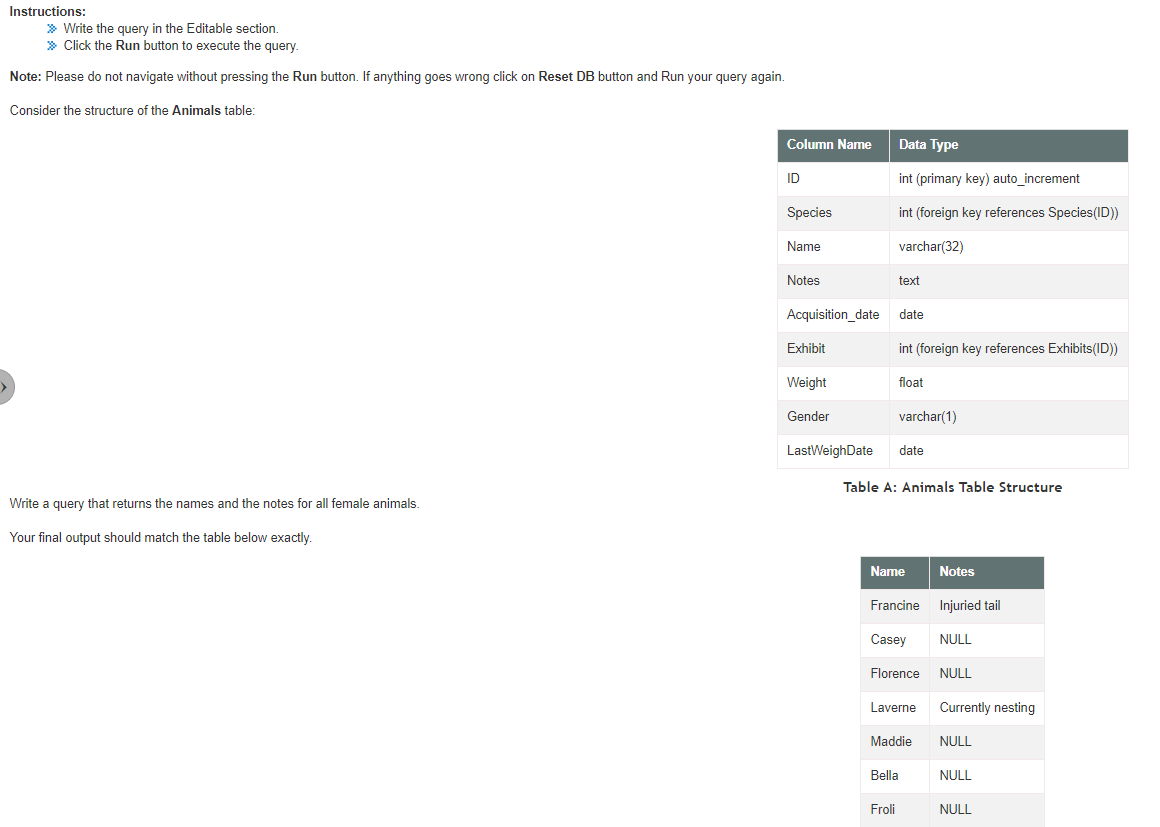


## Adding New Data

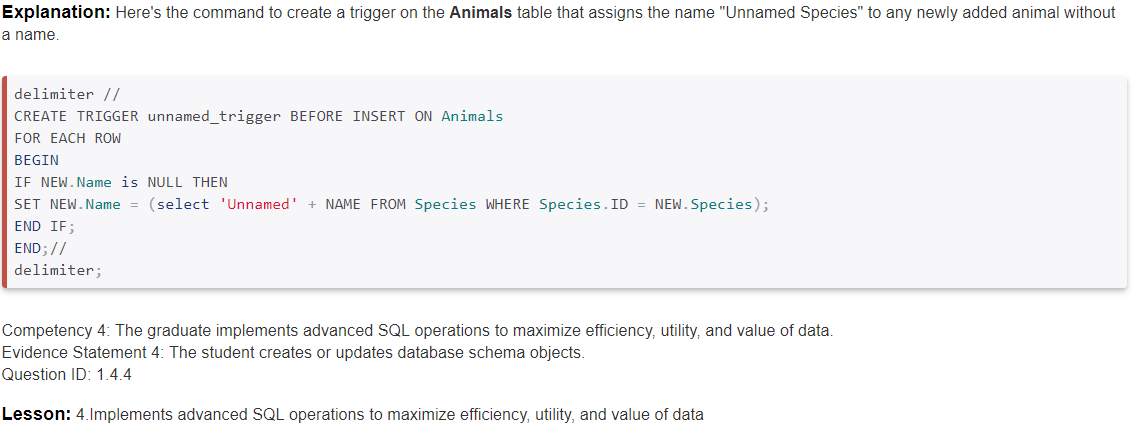
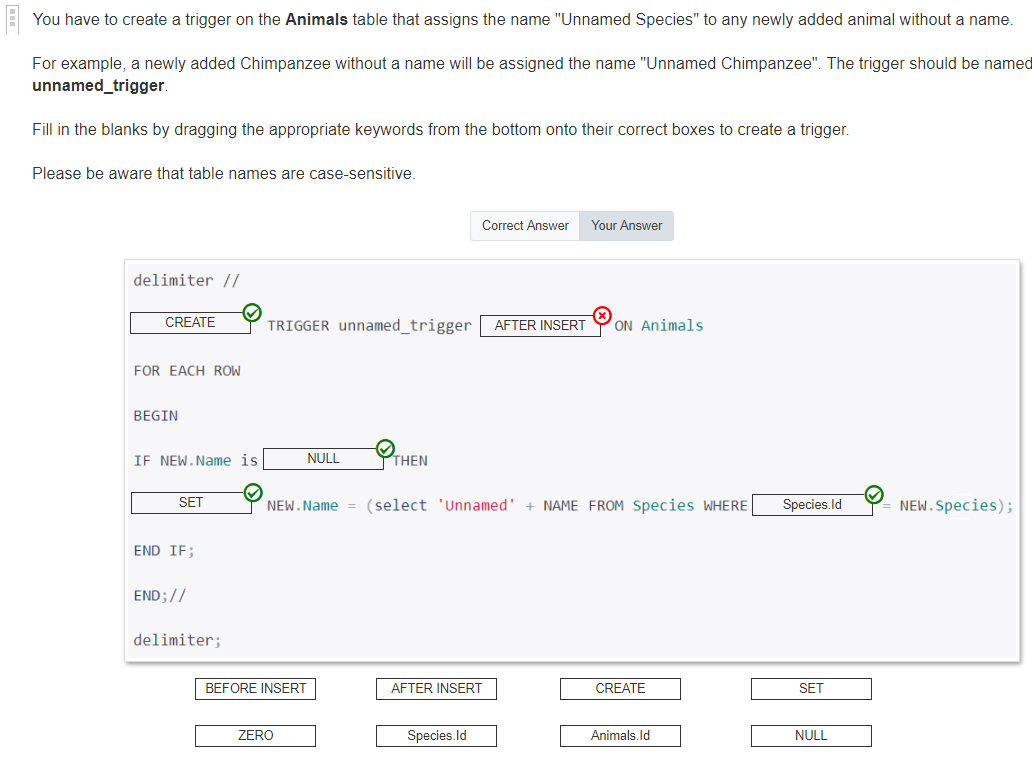
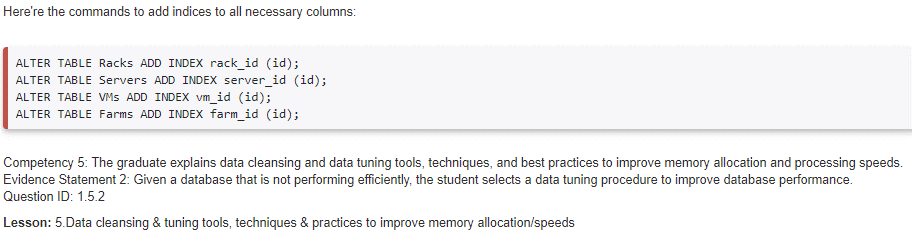
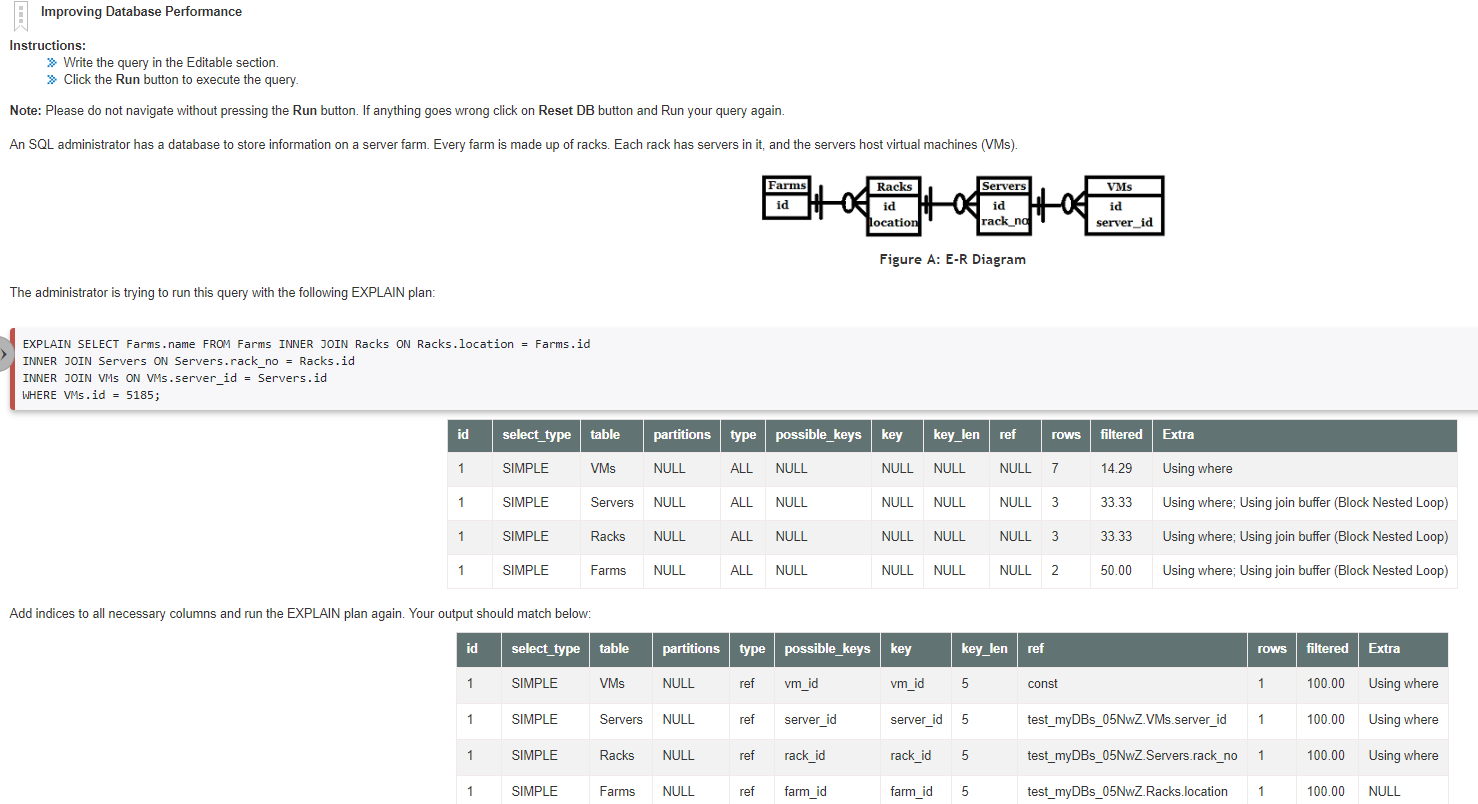
## Creating a Stored Procedure with Two or More Parameters



## Performing DML



## Improving Database Performance



# Stored Function



# Database Trigger

* + - Stored object
    - Executes based on certain conditions



Types of Trigger

* + - Before Insert
    - Before Update
    - Before Delete
    - After Insert
    - After Update
    - After Delete

# Code to know:

## ORDER BY

SELECT Name   
FROM Country   
ORDER BY Name DESC;

## WHERE

SELECT Name, Continent, Population   
FROM Country   
WHERE Population <100000 OR Population IS NULL  
ORDER BY Population DESC;

## LIKE & IN

SELECT Name, Continent, Population   
FROM Country   
WHERE Name LIKE ‘%island%’   
ORDER BY Name;

SELECT Name, Continent, Population   
FROM Country   
WHERE Name LIKE ‘\_a%’ *#finds entries with a as their second character*  
ORDER BY Name;

SELECT Name, Continent, Population   
FROM Country   
WHERE Continent IN (‘Europe’, ‘Asia’)

## INSERT INTO

CREATE TABLE test (a INT, b TEXT, c TEXT);  
INSERT INTO test VALUES (1, ‘This’, ‘Right here!’);  
SELECT \* FROM test;

INSERT INTO test (b, c) VALUES (‘That’, ‘Over there!’) *#insert entries to specific columns*

INSERT INTO test (a, b, c) SELECT id, name, description FROM item; *#inserts values from other table to new table*

## UPDATE rows

UPDATE test   
SET c= ‘Extra funny.’   
WHERE a = 2;

## DELETE rows

DELETE FROM test  
WHERE a = 2;

## DROP TABLE

DROP TABLE test;

## IS NULL

SELECT \* FROM test WHERE c IS NULL;

## CREATE DATABASE

CREATE DATABASE foo;

## CREATE TABLE

CREATE TABLE test (  
id INTEGER,  
name VARCHAR(255),  
address VARCHAR(255),  
city VARCHAR (255),  
state CHAR(2), *#fixed length*  
zip CHAR(10)  
);

## CREATE INDEX

CREATE TABLE test (  
id INTEGER,  
a VARCHAR(255),  
b VARCHAR(255),  
INDEX(a) *#(a) is the column you want to index*  
);

## CREATE PRIMARY KEY & AUTO\_INCREMENT

CREATE TABLE test (  
id INTEGER AUTO\_INCREMENT PRIMARY KEY,  
a VARCHAR (255)  
b VARCHAR (255)  
);

## ALTER TABLE

ALTER TABLE test ADD d VARCHAR (10); *#add column*ALTER TABLE test DROP d; *#drop column*ALTER TABLE test ADD bb VARCHAR (10) AFTER a; *#add column in specific location*ALTER TABLE test

## STRING FUNCTIONS

SELECT CHAR\_LENGTH(Name) *#length of characters in Name column*

SELECT Name, LEFT(Name, 3) *#left 3 names* , RIGHT(Name, 3), MID(Name, 2, 3) *# 3 characters starting from the second character position*, LEFT(Name, 3), CONCAT(Name, LocalName) *#combine strings by Column,* CONCAT\_WS(‘, ’ Name, LocalName, Region, Continent) *#combine with separator*

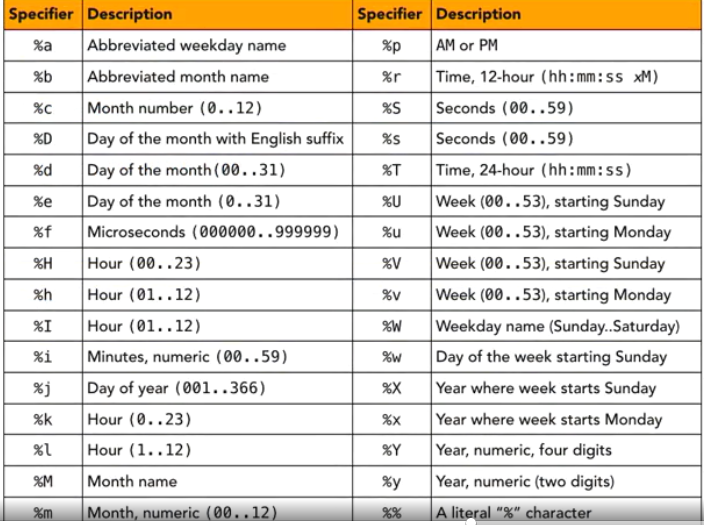
## NUMERIC FUNCTIONS

SELECT 7/3;   
SELECT 7 DIV 3; #integer  
SELECT 7 % 3; #modulus  
SELECT POWER(7,3); #7 to the power of 3  
SELECT ABS(-7); #absolute value  
SELECT SIGN(-7); #positive or negative  
SELECT CONV(57, 10, 16); #converting from base to base  
SELECT ROUND(PI(), 3); #round to 3 decimals.  
SELECT Name, RAND() FROM Country LIMIT 5; #select 5 rows of random numbers

## DATE AND TIME FUNCTIONS

SELECT NOW(); #current time stamp  
SELECT DAYNAME(NOW()); #Day of week  
SELECT DAYOFMONTH(NOW()); #Day of month  
SELECT DAYOFYEAR(NOW()); #day of year  
SELECT MONTH(NOW()); #month  
SELECT ADDTIME(‘1:30:00’, ‘ 00:15:00’); #add time  
SELECT SUBTIME(‘1:30:00’, ‘ 00:15:00’); #subtract time  
SELECT ADDDATE(‘2008-01-02’, INTERVAL 31 DAY);  
SELECT SUBDATE(‘2008-01-02’, INTERVAL 31 DAY);

## FORMAT DATE

SELECT DATE\_FORMAT(NOW(), ‘%W, %D OF %M, %Y’); #  


## AGGREGATE FUNCTIONS

SELECT COUNT(\*) FROM Country;  
SELECT Continent, COUNT(\*) AS Count FROM Country GROUP BY Continent ORDER BY Count DESC;  
SELECT COUNT(DISTINCT Continent) FROM Country; *# distinct is the same as nunique()*

# Triggers

## CREATE TRIGGER

CREATE TRIGGER newWidgetSale AFTER INSERT ON widgetSale  
FOR EACH ROW  
UPDATE widgetCustomer SET last\_order\_id = New.id WHERE id = New.customer\_id

## Prevent automatic updates with a trigger

DELIMITER // *#allows the use of ; in statements without terminating the statement*  
CREATE TRIGGER updateWidgetSale BEFORE UPDATE ON widgetSale  
FROM EACH ROW  
BEGIN  
IF (SELECT reconciled FROM widgetSale WHERE id = New.id) > 0 THEN   
SIGNAL SQLSTATE ‘45000’ SET MESSAGE\_TEXT = ‘ Error: cannot update reconciled row in widgetSale’;  
END IF;   
END  
//  
DELIMITER

# Stored Routines:

1. **Stored Function:** return a value and are used in the context of an expression  
   CREATE FUNCTION
2. **Stored Procedure:** called separately using the CALL statement and may return result sets or set variables.  
   CREATE PROCEDURE

## STORED FUNCTIONS

CREATE FUNCTION track\_len( seconds INT)  
RETURNS VARCHAR(16) DETERMINISTIC  
RETURN CONCAT\_WS(‘:’, seconds DIV 60, LPAD(seconds MOD 60, 2, ‘0’));  
SELECT title, track\_len(duration) FROM track;

## STORED PROCEDURE

DELIMITER //  
CREATE PROCEDURE list\_albums()  
BEGIN  
SELECT \* FROM album;  
END //  
DELIMITER ;