Sample labs for “Advanced SQL”

By Patrick Kellogg

For http://www.wgu.edu

Lab #1:

DDL stands for Data Definition Language (or some say Data Description Language). These commands all deal with defining data structures such as tables, views, and indexes, among other schema objects.

The DDL commands are:

CREATE

DROP

ALTER

And

TRUNCATE

Take these one at a time:

CREATE is often used to define a new empty SQL table that is ready to accept data. The format of a CREATE TABLE statement is:

CREATE TABLE [table name] ( [column definitions] [table parameters] )

1. Create a table of foods for a diet called diet\_plan. The columns are food\_name (which should be a VARCHAR(20)), calories (as INT), and food\_group (1 = dairy group, 2 = meat group, 3 = fruits and vegetables, 4 = bread group, 5 = misc, which is an INT)

Answer:

CREATE TABLE diet\_plan (food\_name VARCHAR(20), calories INT, food\_group INT);

Next is the DROP command:

2. Give up on your diet. Use the “DROP TABLE [table name]” command to completely delete the diet\_plan table.

Answer:

DROP TABLE diet\_plan;

Now use the ALTER command. First, create a table to hold a set of exercises that you can do to lose weight and get fit.

Type this:

CREATE TABLE exercise\_routine (id INT PRIMARY KEY, exercise\_name VARCHAR(20), num\_sets INT, num\_reps INT);

This is a great start for a table to hold your exercise routine. For example, you could store the following information:

“Dumbbell Bench Press”, 1, 8

“Dumbbell Bench Press”, 1, 10

“Dumbbell Bench Press”, 1, 12

“Lat Pulldown”, 2, 10

This means your workout would be three sets of bench presses (of increasing reps) and two final sets of pulldowns.

However, that is an “arm/ chest workout”. What about “leg day”? Even after the workout information has been added to the table, you can still add (or remove) columns to the table.

3. Add a new column called “day\_of\_the\_week” to the exercise\_routine table. This will be a char to store “M”, “T” or “F”.

Answer:

ALTER TABLE exercise\_routine ADD day\_of\_the\_week char;

However, there is a mistake. The day of the week is usually a TWO-letter abbreviation like “Th” for Thursday, or “Sa” for Saturday. So, we need to fix the new column that you just added.

You can drop a column of a table by using the ALTER command again:

ALTER TABLE table\_name DROP COLUMN column\_name;

4. Run two DDL commands one after the other. The first one should delete the day\_of\_the\_week column of the exercise\_routine table. The second DDL command should create the day\_of\_the\_week column again, but this time with the correct VARCHAR(2) data type.

Answer:

ALTER TABLE exercise\_routine DROP day\_of\_the\_week;

ALTER TABLE exercise\_routine ADD day\_of\_the\_week VARCHAR(2);

One nice option for adding a column in MySQL is the ability to make the new ALTER column the FIRST one in the table, or to put it AFTER an existing column. You can do it like this:

ALTER TABLE table\_name ADD column\_name column\_data\_type FIRST;

or

ALTER TABLE table\_name ADD column\_name column\_data\_type AFTER exercise\_name;

You can also use the ALTER command to add a comment to the entire table itself:

ALTER TABLE table\_name COMMENT ‘This is a comment'

5. Add a comment to exercise\_routine that says ‘Exercise routine for January 1 to January 20’

Answer:

ALTER TABLE exercise\_routine COMMENT ‘Exercise routine for January 1 to January 20’

Finally, you can use the TRUNCATE command to delete all the data in a table but leave the structure intact, so the table is empty and ready to hold new data in the future.

TRUNCATE TABLE table\_name;

6. TRUNCATE exercise\_routine so you can start over with a new exercise plan

Answer:

TRUNCATE TABLE exercise\_routine;

Lab #2:

Create a table to store encryption keys for the data center model. The table should be named Keys and have the following data:

automatically generated integer primary key (id)

the name of the key up to a length of 32 (required, name)

multi-line text field for the key (key)

Create table Keys (id int auto\_increment primary key, name varchar(32) not null, key text)

Change the key field to hold binary data up to 64 KB.

Alter table Keys modify key blob not null

Add a “created” field to the Keys table that automatically holds the timestamp of when the row was created.

Alter table Keys add created timestamp default current\_timestamp

Change the key column to include the comment “Key in DER format”.

Alter table Keys modify key blob not null comment = ‘Key in DER format’

Lab #3:

DML stands for “Data Manipulation Language” and involves entering, removing, and modifying data in a database.

The four DML commands are:

\* SELECT

\* INSERT

\* UPDATE

\* DELETE

For example, say that there is a table to store the inventory of classic cars. There are only three columns: the “make” of the car, the “model”, and the year it was built. Here is the table structure:

DESCRIBE garage;

Field Type  Null Key Default Extra

make  varchar(10) YES      NULL

model varchar(10) YES      NULL

year  int(11) YES      NULL

3 rows in set

You can enter a new car into the garage by typing:

INSERT INTO garage VALUES ("Ford", "Fairlaine", 1961);

1. Enter a new car, a 1971 Chevy Camaro

Answer:

INSERT INTO garage VALUES ("Chevrolet", "Camaro", 1971);

You can see all the cars in the garage table by typing:

SELECT \* FROM garage;

Or, you can see only certain cars by typing:

SELECT \* FROM garage WHERE (make = "Chevrolet");

make  model year

Chevrolet   Corvette 1964

Chevrolet   Corvette 1967

Chevrolet   Camaro 1966

Chevrolet   Camaro 1969

4 rows in set

2. Select all the “Ferrari” cars in the garage

Answer:

SELECT \* FROM garage WHERE (make = "Ferrari");

You can also select by the year

SELECT \* FROM garage WHERE (year = 1967);

Or select only part of the data

SELECT year FROM garage WHERE (make = "Porsche");

Or select with wildcards:

SELECT make, model FROM garage WHERE (make like "F%");

3. Select all the Ferraris

Answer:

SELECT \* FROM garage WHERE (make = "Ferrari");

Now, you can delete all the Ferraris from the database by typing:

DELETE FROM garage WHERE (make = "Ferrari");

Do a “SELECT \* from garage;” and you will see that we only have 16 cars in the garage now instead of 20

Now, instead of deleting any more cars, say that we decide to sell them. There is a for\_sale table already created but it is empty:

DESCRIBE for\_sale;

SELECT \* FROM for\_sale;

We can INSERT INTO with a subquery, which is like doing a SELECT and an INSERT in the same command. For example, SELECT \* from garage WHERE (make = "Chevrolet"); will create a list of all Chevy cars in the garage. But an INSERT INTO can do this:

INSERT INTO for\_sale (make, model, year) (SELECT \* FROM garage WHERE (make = "Chevrolet"));

SELECT \* from for\_sale;

Note that 4 Chevrolet cars are in \*both\* tables. Doing the INSERT INTO did not change or delete any of the information in the garage table.

4. Also copy all of the “Alfa Romeo” cars from the garage table to the for\_sale table

Answer:

INSERT INTO for\_sale (make, model, year) (SELECT \* FROM garage WHERE (make = "Alfa Romeo"));

Now we have six cars in the for\_sale table:

SELECT \* FROM for\_sale;

make  model year  price

Chevrolet   Corvette 1964  NULL

Chevrolet   Corvette 1967  NULL

Chevrolet   Camaro 1966  NULL

Chevrolet   Camaro 1969  NULL

Alfa Romeo  33 Stradale 1967  NULL

Alfa Romeo  33 Stradale 1970  NULL

6 rows in set

Note that the for\_sale column is NULL for all the cars; we have not set any prices yet. We can list all the cars on sale for $10,000 by using the UPDATE statement:

UPDATE for\_sale SET price = 10000;

5. Change the price on all the cars to a more respectable $50,000

Answer:

UPDATE for\_sale SET price = 50000;

You can change the price of a single car by doing a WHERE statement:

UPDATE for\_sale SET price = 80000 WHERE (model = "Camaro" and year = 1966);

6. Update the price for all the Alfa Romeos to $40,000

Answer:

UPDATE for\_sale SET price = 40000 WHERE (make = "Alfa Romeo");

Precondition:

A valid garage table

CREATE TABLE garage (make VARCHAR(20), model VARCHAR(20), year INT);

A valid for\_sale table that has an extra column

CREATE TABLE for\_sale (make VARCHAR(20), model VARCHAR(20), year INT, price DOUBLE (10,2));

Entries in the garage table

INSERT INTO garage VALUES ("Ford", "Fairlaine", 1961);

INSERT INTO garage VALUES ("Ferrari", "250 GTO", 1962);

INSERT INTO garage VALUES ("Ferrari", "250 GTO", 1963);

INSERT INTO garage VALUES ("Alfa Romeo", "33 Stradale", 1967);

INSERT INTO garage VALUES ("Alfa Romeo", "33 Stradale", 1970);

INSERT INTO garage VALUES ("Jaguar", "XJ13", 1965);

INSERT INTO garage VALUES ("Ferrari", "330 P4", 1967);

INSERT INTO garage VALUES ("Jaguar", "E-Type", 1961);

INSERT INTO garage VALUES ("Jaguar", "E-Type", 1964);

INSERT INTO garage VALUES ("Jaguar", "E-Type", 1967);

INSERT INTO garage VALUES ("Porsche", "550", 1953);

INSERT INTO garage VALUES ("Porsche", "550", 1954);

INSERT INTO garage VALUES ("Chevrolet", "Corvette", 1964);

INSERT INTO garage VALUES ("Chevrolet", "Corvette", 1967);

INSERT INTO garage VALUES ("Lamborghini", "Miura", 1967);

INSERT INTO garage VALUES ("Ferrari", "Dino", 1968);

INSERT INTO garage VALUES ("Mercedes-Benz", "300SL", 1955);

INSERT INTO garage VALUES ("Chevrolet", "Camaro", 1966);

INSERT INTO garage VALUES ("Chevrolet", "Camaro", 1969);

Lab #4:

TCL stands for “Transaction Control Language” and it handles transactions: which are a series of DML commands.

For example, imagine that your part of job consists of entering orders into a database. People hand you paid orders for widgets, and you enter them into the widgets table:

DESCRIBE widgets;

Field Type  Null Key Default Extra

num\_sold int(11) YES      NULL

customer varchar(40) YES      NULL

paid\_amt double(10,2) YES      NULL

3 rows in set

You can enter orders like this:

INSERT INTO widgets VALUES (5, “Bob’s Construction”, 56.00);

1. Insert an order for “Mary Wills” for 3 widgets at a total of $33.33

Answer:

INSERT INTO widgets VALUES (3, "Mary Wills", 33.33);

Now your boss comes up to you and says that those orders are incorrect: the piece of paper you were given is old and that those orders shouldn’t have been entered into the widgets database.

You can delete the first order by typing:

DELETE FROM widgets WHERE customer = "Bob's Construction";

Or better yet, you can specify \*all\* the fields to make sure that you only delete that unique row and now all the previous orders from Bob’s Construction:

DELETE FROM widgets WHERE customer = "Bob's Construction" AND num\_sold = 5 AND paid\_amt = 56.12;

2. Delete the order for Mary Wills

Answer:

DELETE FROM widgets WHERE customer = "Mary Wills";

or

DELETE FROM widgets WHERE customer = "Mary Wills" AND num\_sold = 3 AND paid\_amt = 33.33;

Unfortunately, your boss comes by a \*third\* time and says that the orders were in fact correct, and you should enter them into the widgets database again.

This time you use TCL and create a TRANSACTION to mark the beginning and end of a set of SQL commands.

Type:

START TRANSACTION;

SAVEPOINT disputed\_orders;

INSERT INTO widgets VALUES (5, "Bob's Construction", 56.12);

INSERT INTO widgets VALUES (3, "Mary Wills", 33.33);

Now when your boss comes back a \*fourth\* time and says, “No, I guess those orders are bad after all…” you can simply type:

ROLLBACK TO disputed\_orders;

3. Start a transaction, insert the first order for “Bob’s Construction”, \*then\* create a SAVEPOINT named marys\_order, insert Mary’s order, and then rollback so that only Bob’s order remains:

Answer:

START TRANSACTION;

INSERT INTO widgets VALUES (5, "Bob's Construction", 56.12);

SAVEPOINT marys\_orders;

INSERT INTO widgets VALUES (3, "Mary Wills", 33.33);

ROLLBACK TO marys\_orders;

You can finalize the set of SQL commands by typing:

COMMIT;

That will save all the transactions to the database up to that point. Then you can create another SAVEPOINT for a new set of SQL commands.

Precondition:

A valid widgets table that has an extra column

CREATE TABLE widgets (num\_sold INT, customer VARCHAR(20), paid\_amt DOUBLE (10,2));

Lab #5:

Here is a table of places around the world:

DESCRIBE Places;

35 rows in set

Each record has the continent where the place is located, the country, and the city name. The final column describes the larger region where the city is located, regardless if the country calls it a “state”, “province”, or “area”.

We can list all the regions and cities that are in Mexico in a simple SQL command:

SELECT region, city FROM Places WHERE country = "Mexico";

But we can also list the cities in reverse alphabetical order by adding an ORDER BY clause:

SELECT region, city FROM Places WHERE country = "Mexico" ORDER BY city DESC;

1. List all the cities and regions (i.e. provinces) in Canada, but list them by region in reverse alphabetical order

Answer:

SELECT region, city FROM Places WHERE country = "Canada" ORDER BY region DESC;

To make things confusing, we can list \*both\* the region and city in reverse order:

SELECT region, city FROM Places WHERE country = "Canada" ORDER BY region DESC, city DESC;

Or, the region can be ascending and the city descending:

SELECT region, city FROM Places WHERE country = "Canada" ORDER BY region ASC, city DESC;

2. List the city and states in the United States, but make the state list descending and the cities within each state ascending

Answer:

SELECT region, city FROM Places WHERE country = "United States" ORDER BY region DESC, city ASC;

Doing an ORDER BY for two columns means that the list will be sorted by the first column (either ASC or DESC) and then the second column (ASC or DESC). This has the nice side effect of bunching up cities together that are in the same state. To do this explicitly, there is a GROUP BY clause that is very useful when doing other SQL commands.

For example, we can GROUP BY the region, and then do a COUNT to see how many regions have more than one city in the database:

SELECT region, country, COUNT(region) FROM Places GROUP BY region, country;

3. Do a COUNT for how many cities are in each country

Answer:

SELECT country, COUNT(city) FROM Places GROUP BY country;

You can also add a WHERE clause before the GROUP BY clause. For example, here is a list of the count of cities only in the provinces of Ontario and British Columbia in Canada:

SELECT region, COUNT(city) FROM Places WHERE region = "Ontario" OR region = "British Columbia" GROUP BY region;

4. Do a count of all the cities in each state, but only for the United States

SELECT region, COUNT(city) FROM Places WHERE country = "United States" GROUP BY region;

Finally, it may be important to limit the results by the COUNT. For example, here is a list of all the regions that have more that a single city in that region:

SELECT country, region FROM Places GROUP BY country, region

HAVING COUNT(city)>1;

5. List all the regions that have more than two cities in them

SELECT region FROM Places GROUP BY region HAVING COUNT(city)>2;

Other useful functions can be used with GROUP BY. For example “SUM(mycolumn) GROUP BY mycolumn” will return the sum for each group. Likewise, “MAX(mycolumn)” will find the maximum value for each group, and “MIN(mycolumn)” will find the minimum.

Precondition:

CREATE TABLE Places (continent VARCHAR(30), country VARCHAR(30), city VARCHAR(30), region VARCHAR(30));

INSERT INTO Places VALUES ("North America", "United States", "New York City", "New York");

INSERT INTO Places VALUES ("North America", "United States", "San Francisco", "California");

INSERT INTO Places VALUES ("North America", "United States", "Austin", "Texas");

INSERT INTO Places VALUES ("North America", "United States", "Seattle", "Washington");

INSERT INTO Places VALUES ("North America", "United States", "Los Angeles", "California");

INSERT INTO Places VALUES ("North America", "United States", "Boston", "Massachusetts");

INSERT INTO Places VALUES ("North America", "United States", "Denver", "Colorado");

INSERT INTO Places VALUES ("North America", "United States", "Houston", "Texas");

INSERT INTO Places VALUES ("North America", "United States", "Portland", "Oregon");

INSERT INTO Places VALUES ("North America", "United States", "San Diego", "California");

INSERT INTO Places VALUES ("North America", "United States", "San Jose", "California");

INSERT INTO Places VALUES ("North America", "United States", "San Antonio", "Texas");

INSERT INTO Places VALUES ("North America", "United States", "Paris", "Texas");

INSERT INTO Places VALUES ("North America", "Canada", "Toronto", "Ontario");

INSERT INTO Places VALUES ("North America", "Canada", "Vancouver", "British Columbia");

INSERT INTO Places VALUES ("North America", "Canada", "Montreal", "Quebec");

INSERT INTO Places VALUES ("North America", "Canada", "Ottowa", "Ontario");

INSERT INTO Places VALUES ("North America", "Canada", "Calgary", "Alberta");

INSERT INTO Places VALUES ("North America", "Canada", "Edmonton", "Alberta");

INSERT INTO Places VALUES ("North America", "Canada", "Quebec City", "Quebec");

INSERT INTO Places VALUES ("North America", "Mexico", "Ciudad Juárez", "Chihuahua");

INSERT INTO Places VALUES ("North America", "Mexico", "Acapulco", "Guerrero");

INSERT INTO Places VALUES ("North America", "Mexico", "Guadalajara", "Jalisco");

INSERT INTO Places VALUES ("North America", "Mexico", "Monterrey", "Nuevo León");

INSERT INTO Places VALUES ("North America", "Mexico", "Cancún", "Quintana Roo");

INSERT INTO Places VALUES ("North America", "Mexico", "Veracruz", "Veracruz");

INSERT INTO Places VALUES ("North America", "Mexico", "Tijuana", "Baja California");

INSERT INTO Places VALUES ("Europe", "France", "Paris", "Île-de-France");

INSERT INTO Places VALUES ("Europe", "France", "Toulouse", "Occitanie");

INSERT INTO Places VALUES ("Europe", "France", "Bordeaux", "Nouvelle-Aquitaine");

INSERT INTO Places VALUES ("Europe", "France", "Lille", "Hauts-de-France");

INSERT INTO Places VALUES ("Europe", "Germany", "Stuttgart", "Baden-Württemberg");

INSERT INTO Places VALUES ("Europe", "Germany", "Wiesbaden", "Hesse");

INSERT INTO Places VALUES ("Europe", "Germany", "Düsseldorf", "North Rhine-

Westphalia");

INSERT INTO Places VALUES ("Europe", "Germany", "Dresden", "Saxony");

Lab #6:

Here is a set of tables to map a social network. Americans is a list of people from America, Germans is a list of people from Germany, and Friends is an associative table that joins the two.

SELECT \* from Americans;

SELECT \* from Germans;

SELECT \* from Friends;

For example, this SQL query provides a list of two people who are friends with each other across country boundaries:

SELECT Americans.name, Germans.name FROM Americans INNER JOIN Friends ON Americans.id = Friends.id\_a INNER JOIN Germans ON Friends.id\_g = Germans.id ORDER by Americans.ID, Germans.id;

Note: and INNER JOIN is the same as a JOIN… an INNER JOIN is the default kind of JOIN.

1. Retry the above query replacing “INNER JOIN” with simple JOIN and then run it

Answer:

SELECT Americans.name, Germans.name FROM Americans JOIN Friends ON Americans.id = Friends.id\_a JOIN Germans ON Friends.id\_g = Germans.id;

You can get a list of all the Americans with the simple query:

SELECT Americans.name FROM Americans;

However, let’s say you want a list of all Americans \*and\* their friends, but you also want to show the Americans that have no friends:

SELECT Americans.name, Americans.id, Friends.id\_g, Germans.name FROM Americans LEFT JOIN Friends ON Americans.id = Friends.id\_a LEFT JOIN Germans ON Friends.id\_g = Germans.id ORDER BY Americans.id, Germans.id;

Note that this list show two Americans, Carlos and Sebastian, who have NULL as their id\_g German friend id. They are still included in this query result, where the INNER JOIN before excluded them.

2. Repeat the query, but this time returning all Germans and their American friends, but also include Germans with NULL friends (Note: use LEFT JOIN)

Answer:

SELECT Germans.name, Germans.id, Friends.id\_a, Americans.name FROM Germans LEFT JOIN Friends ON Germans.id = Friends.id\_g LEFT JOIN Americans ON Friends.id\_a = Americans.id ORDER BY Germans.id, Americans.id;

Again, two people do not have any friends, Frieda and Bertha. However, it may be confusing to have to swap the American and German table when you retyped that query. There is also a RIGHT JOIN:

SELECT Germans.name, Germans.id, Friends.id\_a, Americans.name FROM Friends RIGHT JOIN Germans ON Germans.id = Friends.id\_g LEFT JOIN Americans on Friends.id\_a = Americans.id ORDER BY Germans.id, Americans.id;

Germans -> LEFT JOIN ->  Friends

Is equivalent to

Friends -> RIGHT JOIN -> Germans

3. Repeat the first query with a RIGHT JOIN:

SELECT Americans.name, Americans.id, Friends.id\_g,

Germans.name FROM Americans LEFT JOIN Friends ON

Americans.id = Friends.id\_a LEFT JOIN Germans ON

Friends.id\_g = Germans.id ORDER BY Americans.id, Germans.id;

Note: You should only have to change three words in the query above to turn it into a RIGHT JOIN

Friends -> RIGHT JOIN -> Americans

Answer:

SELECT Americans.name, Americans.id, Friends.id\_g, Germans.name FROM Friends RIGHT JOIN Americans ON Americans.id = Friends.id\_a LEFT JOIN Germans on Friends.id\_g = Germans.id ORDER BY Americans.id, Germans.id;

Many SQL languages also support an OUTER JOIN (also known as a FULL OUTER JOIN). This gives you a list of all the American-German friend pairs as well as all people with NULL as a friend. However, MySQL does not currently support OUTER JOINs (as of version 8.0).

SELECT Americans.name, Germans.name FROM Americans OUTER JOIN Friends ON Americans.id = Friends.id\_a OUTER JOIN Germans ON Friends.id\_g = Germans.id ORDER by Americans.ID, Germans.id;

(Note: this will NOT run in MySQL)

One way to cheat is to do a UNION of the two queries we did before:

SELECT Americans.name, Germans.name FROM Americans LEFT JOIN Friends ON Americans.id = Friends.id\_a LEFT JOIN Germans ON Friends.id\_g = Germans.id

UNION ALL

SELECT Americans.name, Germans.name FROM Germans LEFT JOIN Friends ON Germans.id = Friends.id\_g LEFT JOIN Americans ON Friends.id\_a = Americans.id;

However, keep in mind that UNIONs can be \*very\* expensive for both processor time and disk space.

Plus, note that 50 rows are returned, many with with same pair of American and German friends, since we simply mashed the two queries together.

Note:

4. Run the UNION query above with UNION instead of UNION ALL. Now hom may rows are returned?

Answer:

SELECT Americans.name, Germans.name FROM Americans LEFT JOIN Friends ON Americans.id = Friends.id\_a LEFT JOIN Germans ON Friends.id\_g = Germans.id

UNION

SELECT Americans.name, Germans.name FROM Germans LEFT JOIN Friends ON Germans.id = Friends.id\_g LEFT JOIN Americans ON Friends.id\_a = Americans.id;

26 rows in set

There are also other types of JOINs. A CROSS JOIN forms the cartesian product of all the rows in each table.

For example, if you wanted to print out all possible combination of American and German friends \*regardless\* of the Friends associated table, you could type:

SELECT Americans.name, Germans.name FROM Americans CROSS JOIN Germans;

Note that 64 rows are returned, because that is the product of (8 American people) \* (8 German people).

Finally, there is a self join used to combine a table with itself. For example, imagine that Americans can be friends with any other American \*or\* themselves. This SQL statement shows all the possible combinations:

SELECT a.name, b.name FROM Americans a, Americans b;

5. Do a self join on the German table with itself

Answer:

SELECT a.name, b.name FROM Germans a, Germans b;

Precondition:

Valid tables and prepopulated data for Americans, Germans, and their associated table Friends

Lab #7:

We want to automate our shipping department. We want to SELECT a record in our Orders table based on order\_num. There is a column in that table called shipping\_method.

\* If the method is “domestic” then you should INSERT a record in the Shipped table with the order\_num , customer’s name, and address (as the data column)

\* If the method is “pickup”, then you should INSERT a record in the Counter table with the order\_num, customer’s name, and the date\_packaged which should be today’s DATETIME that it is right NOW()

We want to do this all in one SQL statement. However, let’s break this up into parts:

1. Create the INSERT statement to insert name, address, and order\_num into the Shipped table for the “domestic” orders

Answer:

INSERT INTO Shipped (order\_num, name, data) VALUES (order\_num, name, address);

2. Do an INSERT statement to insert name, order\_num, and the current date and time into the Shipped table for “pickup” orders

INSERT INTO Shipped (order\_num, name, data) VALUES (order\_num, name, NOW());

3. Do the SELECT statement to get the name, address, and shipping\_method of the customer in the Order table where the order\_num = 100

SELECT name, address, shipping\_method FROM Orders WHERE order\_num = 100;

4. Combine parts 1 and 3. That is, INSERT a record with order\_num, name, and address into Shipped, but get the values in a subquery from Orders where order\_num = 100 (i.e. “domestic”):

Note: You don’t need the VALUES command with a subquery

Answer:

INSERT INTO Shipped (order\_num, name, data) SELECT order\_num, name, address FROM Orders WHERE order\_num = 100;

5. Do the same for Counter. In other words, INSERT a record with order\_num, name, and address into Shipped, but get the values in a subquery from Orders where order\_num = 101:

Answer:

INSERT INTO Shipped (order\_num, name, data) SELECT order\_num, name, NOW() FROM Orders WHERE order\_num = 101;

6. Now you need to SELECT either (order\_num, name, address) for “domestic” (like order\_num=100) \*or\* (order\_num, name, NOW()) for “pickup” (like order\_num\_101). Use a CASE statement. The syntax is:

SELECT (field1, field2,

CASE field\_to\_compare

 WHEN condition1 THEN result1

 WHEN condition2 THEN result2

 WHEN conditionN THEN resultN

 ELSE result

END as field3

FROM mytable

WHERE mycondition;

Answer:

SELECT

order\_num,

name,

CASE shipping\_method

WHEN 'domestic' THEN address

WHEN 'pickup' THEN NOW()

END as data

FROM Orders

WHERE order\_num = 100;

7. Now that we have the three fields we want, we can finish the INSERT statement, and then we have done a subquery and CASE statement on the MySQL command

Answer:

INSERT INTO Shipped

SELECT

order\_num,

name,

CASE shipping\_method

WHEN 'domestic' THEN address

WHEN 'pickup' THEN NOW()

END as data

FROM Orders

WHERE order\_num = 100;

Lab #8:

A company has an Employee table for every person who works at the company. Only a few people in the HR department have the privileges to INSERT, DELETE, and UPDATE the table.

However, the Employees table also includes information on the birth day and month and year of each employee. There is a new requirement that a few people on the “Birthday Squad” should have access to SELECT from the Employee table but to \*only\* see the employee’s name and birth day and month, and \*not\* their salary or the year they were born.

You can do this by creating a VIEW on the Employees table. That will allow you to create a VIEW that only has the first\_name, last\_name, birth\_month, and birth\_day and nothing else.

Since you have all privileges, you can see the entire Employees table:

DESCRIBE Employees;

SELECT \* from Employees;

Field Type  Null Key Default Extra

first\_name  varchar(20) YES      NULL

last\_name   varchar(20) YES      NULL

address varchar(40) YES      NULL

salary   double(10,2) YES      NULL

birth\_month int(11) YES      NULL

birth\_day   int(11) YES      NULL

birth\_year  int(11) YES      NULL

7 rows in set

first\_name  last\_name address salary   birth\_month birth\_day   birth\_year

Cheryl   Miller   102 Oak Street, Hillsbury, MA 01001 45100.00 12 5   1960

Steve    Johnson 55 Smith Avenue, Cranville MA 01002 39450.00 4 12   1957

Roger    Beatnik 9915 1st Street #7A, Hillsbury, MA 01001   21760.00 3   3 1984

Alicia   Vasquez 390 Krups Drive, Scranly, MA 01007 27900.00 7 31 1977

Pedro Monoton 777 Spryce Boulevard, Unita, MA 01005   61000.00 1 18 1971

5 rows in set

1. Create a VIEW called BirthdaySquad that only include first\_name, last\_name, birth\_month, and birth\_day from the Employees table

The syntax is:

CREATE VIEW view\_name AS SELECT statement;

Answer:

CREATE VIEW BirthdaySquad AS SELECT first\_name, last\_name, birth\_month, birth\_day FROM Employees;

Now we can run the query:

SELECT \* FROM BirthdaySquad;

To see the results:

first\_name  last\_name birth\_month birth\_day

Cheryl   Miller   12 5

Steve    Johnson 4 12

Roger    Beatnik 3 3

Alicia   Vasquez 7 31

Pedro Monoton 1 18

5 rows in set

To clean this up, we could create a lookup table to turn the month “12” into “December” to improve readability. Also, we could create a new role “BirthdaySquadRole” to make sure that the BirthdaySquad can only have SELECT permission to the new VIEW and not be able to DELETE or UPDATE any of the underlying table.

However, for right now, we are going to go on to a second task

There are also two other tables: Buildings and Services

DESCRIBE Buildings;

SELECT \* from Buildings;

DESCRIBE Services;

SELECT \* from Services;

We want to figure out the “burn rate” of our company. A burn rate is the cost per month of all things necessary for a company to exist every month. For our example, we need to list the monthly salary of all employees, as well as the monthly rent of all buildings and the monthly cost of all services.

We could create a complex JOIN to combine all the financial data from the Employees, Buildings, and Service tables, but instead we will do a UNION

The syntax is:

CREATE VIEW view\_name AS

SELECT column\_list

UNION [DISTINCT | ALL]

SELECT column\_list

UNION [DISTINCT | ALL]

SELECT column\_list

3. SELECT just the building\_name and monthly\_rent FROM Buildings

Answer:

SELECT building\_name, monthly\_rent FROM Buildings;

4. SELECT just the company and monthly\_fee FROM Buildings

Answer:

SELECT company, monthly\_fee FROM Services;

5. This one is the trickiest. SELECT each employees first\_name and second name and concatenate (CONCAT) them together to form the first column. Then, select as the second column and divide it by 12 to get the monthly salary for each employee. Finally, call that second column monthly\_cost so we can use it later

Answer:

SELECT CONCAT(first\_name, " ", last\_name), (salary/12) AS monthly\_cost FROM Employees;

6. Now, using UNION, create a BurnRate VIEW that has fist and second columns from the Employees, Buildings, and Service table that you just created

Answer:

CREATE VIEW BurnRate AS

SELECT CONCAT(first\_name, " ", last\_name), (salary/12) AS monthly\_cost FROM Employees

UNION

SELECT building\_name, monthly\_rent FROM Buildings

UNION

SELECT company, monthly\_fee FROM Services;

Now, we can execute this SQL query using the new BurnRate VIEW:

SELECT SUM(monthly\_cost) FROM BurnRate;

And you can see that we need $25,557 each month to keep our company in business

Precondition:

An Employee table populated with data

CREATE TABLE Employees (id INT, first\_name VARCHAR(20), last\_name VARCHAR(20), address VARCHAR(40), salary DOUBLE(10,2), birth\_month INT, birth\_day INT, birth\_year INT);

INSERT INTO Employees VALUES (1, "Cheryl ", "Miller ", "102 Oak Street, Hillsbury, MA 01001", 45100, 12, 5, 1960);

INSERT INTO Employees VALUES (2, "Steve ", "Johnson", "55 Smith Avenue, Cranville MA 01002", 39450, 4, 12, 1957);

INSERT INTO Employees VALUES (3, "Roger ", "Beatnik", "9915 1st Street #7A, Hillsbury, MA 01001 ", 21760, 3, 3, 1984);

INSERT INTO Employees VALUES (4, "Alicia ", "Vasquez", "390 Krups Drive, Scranly, MA 01007", 27900, 7, 31, 1977);

INSERT INTO Employees VALUES (5, "Pedro", "Monoton", "777 Spryce Boulevard, Unita, MA 01005", 61000, 1, 18, 1971);

A Buildings table populated with data

CREATE TABLE Buildings (id INT, building\_name, address VARCHAR(40), monthly\_rent DOUBLE(10,2));

INSERT INTO Buildings VALUES (101, "Main office", "6640 Main Street, Hillsbury, MA 01001", 7300);

INSERT INTO Buildings VALUES (102, "Storage facility", "6645 Main Street, Hillsbury, MA 01001", 450);

A Services table populated with data

CREATE TABLE Services (id INT, company VARCHAR(40), monthly\_fee DOUBLE(10,2));

INSERT INTO Services VALUES (1001, "McDougal Landscaping", 500);

INSERT INTO Services VALUES (1002, "Ajax Cleaning Service", 620);

INSERT INTO Services VALUES (1003, "Constance & Phillip’s Catering", 250);

INSERT INTO Services VALUES (1004, "Speedy Delivery Vehicles", 170);

Lab #8:

There are many reasons that a database may need to be normalized. There are update, insertion, and deletion anomalies that may occur when a database is designed badly.

For example, here is a table that stores objects and what color they are:

DESCRIBE Things;

17 rows in set

Now, this table is perfectly valid. Needing to normalize a database table doesn’t often mean that there is an error; it means that the database might not be operating as efficiently as possible.

For example, even in the simple Things table, there are multiple problems that could occur:

\* A user could insert an object with the color “oranj”. The SQL INSERT statement would execute correctly, but that record will \*never\* show up on any SELECT statement that searches by color

\* Also, the color column is wasting a lot of disk space. Every record is taking up a VARCHAR(20). The word “yellow” is in the table three times. A better idea would be to break out the color column into its own lookup table.

1. Select (distinctly) all of the colors in the Things table

Answer:

SELECT DISTINCT color FROM Things;

First create a new table called ColorLookup that has an id integer and the name of a color

CREATE TABLE ColorLookup (id INT AUTO\_INCREMENT PRIMARY KEY, color VARCHAR(20));

2. Now, take those values and INSERT them into the new ColorLookup

The syntax is:

INSERT INTO first\_table\_name [(column1, column2, ... columnN)]

   SELECT column1, column2, ...columnN

   FROM second\_table\_name

   [WHERE condition];

Answer:

INSERT INTO ColorLookup (color) SELECT DISTINCT color FROM Things;

3. Add a new column to the Things table. It should be called color\_int and be an integer

The syntax is:

ALTER TABLE table

ADD [COLUMN] column\_name column\_definition [FIRST|AFTER existing\_column];

Answer:

ALTER TABLE Things ADD color\_int INT;

Now we can update the new color\_int column with the correct id from the ColorLookup table:

UPDATE Things INNER JOIN ColorLookup ON Things.color = ColorLookup.color SET Things.color\_int = ColorLookup.id;

SELECT \* FROM Things;

4. Drop the Things.color column now that it is no longer needed

The syntax is:

ALTER TABLE table\_name

DROP column\_name;

Answer:

ALTER TABLE Things DROP color;

Now the following JOIN should work:

SELECT Things.object, ColorLookup.color FROM Things INNER JOIN ColorLookup ON Things.color\_int = ColorLookup.id;

And the database takes up less space and there is now a useful color lookup table that can be used by other tables in the database.

Precondition:

A valid Things table that is populated with data:

CREATE TABLE Things (object VARCHAR(20), color VARCHAR(20));

INSERT INTO Things VALUES ("banana", "yellow");

INSERT INTO Things VALUES ("sun", "yellow");

INSERT INTO Things VALUES ("lemon", "yellow");

INSERT INTO Things VALUES ("orange", "orange");

INSERT INTO Things VALUES ("pumpkin", "orange");

INSERT INTO Things VALUES ("crab", "orange");

INSERT INTO Things VALUES("basketball", "orange");

INSERT INTO Things VALUES("carrot", "orange");

INSERT INTO Things VALUES("ocean", "blue");

INSERT INTO Things VALUES("sky", "blue");

INSERT INTO Things VALUES("blueberry", "blue");

INSERT INTO Things VALUES("bluebird", "blue");

INSERT INTO Things VALUES("cherry", "red");

INSERT INTO Things VALUES("tomato", "red");

INSERT INTO Things VALUES("stop sign", "red");

INSERT INTO Things VALUES("strawberry", "red");

INSERT INTO Things VALUES("ruby", "red");

Lab #9:

Our company’s workflow is kind of confusing. Every order is in an Orders table. This links to an OrderList table with a breakdown of all the products ordered. So, for example, Orders.id #5000 might have four products in it, #100, #101, and #102. Then you can look up to see what the name of product #100 is in the Products lookup table. The tables look like this:

 

DESCRIBE OurProducts;

SELECT \* FROM OurProducts;

DESCRIBE OrderList;

SELECT \* FROM OrderList;

DESCRIBE Orders;

SELECT \* FROM Orders;

1. SELECT all the Product.desc that are in Order.id = 501

Answer:

You need a JOIN to do this:

SELECT OurProducts.descript FROM OurProducts

JOIN OrderList ON OurProducts.id = OrderList.proditem\_id

JOIN Orders ON OrderList.id = Orders.orderlist\_id

WHERE Orders.id = 501;

There is also a Commissions table that holds the amount that each salesperson made from the sale. This table has strict privileges on it because we don’t want every employee to be able to see how much each salesperson made. However, you have full SELECT, INSERT, and DELETE access to this table.

There is also a Salespeople table that provides information. The two new tables look like this:

 

DESCRIBE Commissions;

SELECT \* FROM Commissions;

DESCRIBE Salespeople;

SELECT \* FROM Salespeople;

2. Find the salesperson’s name who sold order #501

Answer:

Again, do a JOIN:

SELECT Salespeople.name FROM Salespeople

JOIN Commissions ON Salespeople.id = Commissions.salesperson\_id

JOIN Orders ON Commissions.order\_id = Orders.id

WHERE Orders.id = 501;

3. Now put them together. Who are all the salespeople who ever sold a hammer?

SELECT Salespeople.name FROM Salespeople

JOIN Commissions ON Salespeople.id = Commissions.salesperson\_id

JOIN Orders ON Commissions.order\_id = Orders.id

JOIN OrderList ON Orders.orderlist\_id = OrderList.id

JOIN OurProducts ON OrderList.proditem\_id = OurProducts.id

WHERE OurProducts.descript = "hammer";

However, this SQL statement is taking a lot of resources because the tables aren’t set up efficiently. For example, do an EXPLAIN plan:

4. Run an EXPLAIN PLAIN on the SELECT statement you created

Answer:

EXPLAIN

SELECT Salespeople.name FROM Salespeople

JOIN Commissions ON Salespeople.id = Commissions.salesperson\_id

JOIN Orders ON Commissions.order\_id = Orders.id

JOIN OrderList ON Orders.orderlist\_id = OrderList.id

JOIN OurProducts ON OrderList.proditem\_id = OurProducts.id

WHERE OurProducts.descript = "hammer";

   id select\_type table partitions  type possible\_keys key   key\_len ref   rows  filtered Extra

1 1 SIMPLE   Salespeople NULL  ALL NULL NULL NULL  NULL 1 100   NULL

2 1 SIMPLE   Commissions NULL  ALL NULL NULL NULL  NULL 1 100   Using where; Using join buffer (Block Nested Loop)

3 1 SIMPLE   Orders   NULL  ALL NULL  NULL NULL NULL  1 100   Using where; Using join buffer (Block Nested Loop)

4 1 SIMPLE   OrderList   NULL ALL NULL  NULL NULL NULL 4 25   Using where; Using join buffer (Block Nested Loop)

5 1 SIMPLE   OurProducts NULL  ALL NULL NULL NULL  NULL 6 16,67   Using where; Using join buffer (Block Nested Loop)

None of our tables are indexed. Nor do they have primary keys. So, the JOIN statement is showing computationally-expensive join buffers for every step.

5. Index all relevant columns for the four tables

ALTER TABLE OurProducts ADD INDEX (id);

ALTER TABLE OrderList ADD INDEX (id);

ALTER TABLE Orders ADD INDEX (id);

ALTER TABLE Commissions ADD INDEX (salesperson\_id);

ALTER TABLE Commissions ADD INDEX (order\_id);

ALTER TABLE Salespeople ADD INDEX (id);

Run the EXPLAIN plan again:

   id select\_type table partitions  type possible\_keys key   key\_len ref   rows  filtered Extra

1 1 SIMPLE   Salespeople NULL  ALL id NULL  NULL NULL  1 100   Using where

2 1 SIMPLE   Commissions NULL  ref salesperson\_id,order\_id   salesperson\_id   5 rextester.Salespeople.id 1 100   Using where

3 1 SIMPLE   Orders   NULL  ref id id 5   rextester.Commissions.order\_id 1 100   NULL

4 1 SIMPLE   OurProducts NULL  ALL id NULL  NULL NULL  6 16,67   Using where; Using join buffer (Block Nested Loop)

5 1 SIMPLE   OrderList   NULL ALL id NULL  NULL NULL  4 25   Using where; Using join buffer (Block Nested Loop)

This is better, but we still aren’t using primary keys

6. Add a PRIMARY KEY for every table that has a column with unique data that supports it

ALTER TABLE OurProducts ADD PRIMARY KEY (id);

ALTER TABLE Orders ADD PRIMARY KEY (id);

ALTER TABLE Salespeople ADD PRIMARY KEY (id);

Why not add a primary key to the Commissions or OrderList tables? This is because those columns do not have unique data. However we can still index it.

7. Add non-primary indexed for the columns that need it

ALTER TABLE OrderList ADD INDEX (id);

ALTER TABLE Commissions ADD INDEX (salesperson\_id);

ALTER TABLE Commissions ADD INDEX (order\_id);

Run the EXPLAIN plan a final time:

   id select\_type table partitions  type possible\_keys key   key\_len ref   rows  filtered Extra

1 1 SIMPLE   Salespeople NULL  ALL PRIMARY,id NULL  NULL NULL 1 100   NULL

2 1 SIMPLE   Commissions NULL  ref salesperson\_id,order\_id,salesperson\_id\_2,order\_id\_2 salesperson\_id   5 rextester.Salespeople.id 1 100   Using where

3 1 SIMPLE   Orders   NULL  eq\_ref   PRIMARY,id  PRIMARY 4   rextester.Commissions.order\_id 1 100   NULL

4 1 SIMPLE   OurProducts NULL  ALL PRIMARY,id NULL  NULL NULL 6 16,67 Using where; Using join buffer (Block Nested Loop)

5 1 SIMPLE   OrderList   NULL ALL id,id\_2 NULL  NULL NULL  4 25   Using where; Using join buffer (Block Nested Loop)

Now every step of the SQL statement has possible keys that can be used. So, why isn’t MySQL using them? For datasets this small, it’s actually faster for MySQL to \*Not\* use certain indexes. However, the code is working correctly.

Preconditions:

Six tables set up correctly and populated with data:

CREATE TABLE OurProducts (id INT, prodtype\_id INT, descript VARCHAR(20));

CREATE TABLE OrderList (id INT, proditem\_id INT);

CREATE TABLE Orders (id INT, orderlist\_id INT, total\_price DOUBLE(8,2));

CREATE TABLE Commissions (salesperson\_id INT, order\_id INT, commission DOUBLE(8,2));

CREATE TABLE Prodtype (id INT, descript VARCHAR(20));

CREATE TABLE Salespeople (id INT, name VARCHAR(20), prodtype\_id INT);

INSERT INTO OurProducts VALUES (101, 1, "hammer");

INSERT INTO OurProducts VALUES (102, 1, "saw");

INSERT INTO OurProducts VALUES (103, 1, "screwdriver");

INSERT INTO OurProducts VALUES (104, 1, "level");

INSERT INTO OurProducts VALUES (105, 1, "pliers");

INSERT INTO OurProducts VALUES (106, 1, "wire cutters");

INSERT INTO OrderList VALUES (1001, 101);

INSERT INTO OrderList VALUES (1001, 103);

INSERT INTO OrderList VALUES (1001, 104);

INSERT INTO OrderList VALUES (1001, 105);

INSERT INTO Orders VALUES (501, 1001, 631.56);

INSERT INTO Commissions VALUES (8001, 501, 63.15);

INSERT INTO Prodtype VALUES (1, "hardware");

INSERT INTO Salespeople VALUES (8001, "Sully Johnston", 1);