```
USE test2;

    a. Make test2

SELECT DATABASE(

    a. Show the c

DROP DATABASE IF

    a. Delete the

    b. Slide abou

CREATE TABLE stu
rst_name VARCHAR(
st_name VARCHAR(3
ail VARCHAR(60) N
reet VARCHAR(50)
ty VARCHAR(40) NO
ate CHAR(2) NOT N
p MEDIUMINT UNSIG
one VARCHAR(20) N
rth_date DATE NOT
x ENUM('M', 'F')
te_entered TIMEST
nch_cost FLOAT NULL,
udent_id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY
```

student
first_name
last_name
email
street
city
state
zip
phone
birth_date
sex
date_entered
lunch_cost
student_id

class
name
class_id

test
date
type
class_id
test_id

score
student_id
event_id
score
event_id, student_id

absence
student_id
date
student_id, date

```
VARCHAR(30) : Characters with an expected max length of 30
```

[MySQL Tutorial Video by Derek Banas]

1. Login to MySQL

    a. mysql5 -u mysqladmin -p

2. quit

    a. Quit MySQL

3. show databases;

    a. Display all databases

4. CREATE DATABASE test2;

    a. Create a database

5. USE test2;

    a. Make test2 the active database

6. SELECT DATABASE();

    a. Show the currently selected database

7. DROP DATABASE IF EXISTS test2;

    a. Delete the named database

    b. Slide about building tables (2)

8. CREATE TABLE student(

```
first_name VARCHAR(30) NOT NULL,
last_name VARCHAR(30) NOT NULL,
email VARCHAR(60) NULL,
street VARCHAR(50) NOT NULL,
city VARCHAR(40) NOT NULL,
state CHAR(2) NOT NULL DEFAULT "PA",
zip MEDIUMINT UNSIGNED NOT NULL,
phone VARCHAR(20) NOT NULL,
birth_date DATE NOT NULL,
sex ENUM('M', 'F') NOT NULL,
date_entered TIMESTAMP,
lunch_cost FLOAT NULL,
student_id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY
);
```

a. VARCHAR(30) : Characters with an expected max length of 30

b. NOT NULL : Must contain a value

c. NULL : Doesn't require a value

d. CHAR(2) : Contains exactly 2 characters

e. DEFAULT "PA" : Receives a default value of PA

f. MEDIUMINT : Value no greater then 8,388,608

g. UNSIGNED : Can't contain a negative value

h. DATE : Stores a date in the format YYYY-MM-DD

i. ENUM('M', 'F') : Can contain either a M or F

j. TIMESTAMP : Stores date and time in this format YYYY-MM-DD-HH-MM-SS

k. FLOAT: A number with decimal spaces, with a value no bigger than 1.1E38 or smaller than -1.1E38

l. INT : Contains a number without decimals

m. AUTO_INCREMENT : Generates a number automatically that is one greater than the previous row

n. PRIMARY KEY (SLIDE): Unique ID that is assigned to this row of data

    I. Uniquely identifies a row or record

    II. Each Primary Key must be unique to the row

    III. Must be given a value when the row is created and that value cannot be NULL

    IV. The original value cannot be changed It should be short

    V. Itâ''s probably best to auto increment the value of the key

o. Atomic Data & Table Templating

As your database increases in size, you are going to want everything to be organized, so that it can perform your queries quickly. If your tables are set up properly, your database will be able to crank through hundreds of thousands of bits of data in seconds.

How do you know how to best set up your tables though? Just follow some simple rules:

Every table should focus on describing just one thing. Ex. Customer Table would have name, age, location, contact information. It shouldnâ''t contain lists of anything such as interests, job history, past address, products purchased, etc.

After you decide what one thing your table will describe, then decide what things you need to describe that thing. Refer to the customer example given in the last step.

Write out all the ways to describe the thing and if any of those things requires multiple inputs, pull them out and create a new table for them. For example, a list of past employers.

Once your table values have been broken down, we refer to these values as being atomic. Be careful not to break them down to a point in which the data is harder to work with. It might make sense to create a different variable for the house number, street name, apartment number, etc.; but by doing so you may make yourself more work? That decision is up to you?

p. Some additional rules to help you make your data atomic: Donâ''t have multiple columns with the same sort of information. Ex. If you wanted to include a employment history you should create job1, job2, job3 columns. Make a new table with that data instead.

Donâ''t include multiple values in one cell. Ex. You shouldnâ''t create a cell named jobs and then give it the value: McDonalds, Radio Shack, Walmart,â'¦ Normalized Tables

q. What does normalized mean?

Normalized just means that the database is organized in a way that is considered standardized by professional SQL programmers. So if someone new needs to work with the tables theyâ''ll be able to understand how to easily.

Another benefit to normalizing your tables is that your queries will run much quicker and the chance your database will be corrupted will go down.

r. What are the rules for creating normalized tables:

The tables and variables defined in them must be atomic Each row must have a Primary Key defined. Like your social security number identifies you, the Primary Key will identify your row.

You also want to eliminate using the same values repeatedly in your columns. Ex. You wouldnâ''t want a column named instructors, in which you hand typed in their names each time. You instead, should create an instructor table and link to itâ''s key.

Every variable in a table should directly relate to the primary key. Ex. You should create tables for all of your customers potential states, cities and zip codes, instead of including them in the main customer table. Then you would link them using foreign keys. Note: Many people think this last rule is overkill and can be ignored!

No two columns should have a relationship in which when one changes another must also change in the same table. This is called a Dependency. Note: This is another rule that is sometimes ignored.

------------ Numeric Types ------------

TINYINT: A number with a value no bigger than 127 or smaller than -128

```
SMALLINT: A number with a value no bigger than 32,768 or smaller than -32,767
MEDIUM INT: A number with a value no bigger than 8,388,608 or smaller than -8,388,608
INT: A number with a value no bigger than 2^31 or smaller than 2^31 â'' 1
BIGINT: A number with a value no bigger than 2^63 or smaller than 2^63 â'' 1
FLOAT: A number with decimal spaces, with a value no bigger than 1.1E38 or smaller than
-1.1E38
DOUBLE: A number with decimal spaces, with a value no bigger than 1.7E308 or smaller than
-1.7E308

------------ String Types ------------

CHAR: A character string with a fixed length
VARCHAR: A character string with a length thatâ''s variable
BLOB: Can contain 2^16 bytes of data
ENUM: A character string that has a limited number of total values, which you must define.
SET: A list of legal possible character strings. Unlike ENUM, a SET can contain multiple
values in comparison to the one legal value with ENUM.

------------ Date & Time Types ------------

DATE: A date value with the format of (YYYY-MM-DD)
TIME: A time value with the format of (HH:MM:SS)
DATETIME: A time value with the format of (YYYY-MM-DD HH:MM:SS)
TIMESTAMP: A time value with the format of (YYYYMMDDHHMMSS)
YEAR: A year value with the format of (YYYY)

9. DESCRIBE student;

      a. Show the table set up

10. INSERT INTO student VALUES('Dale', 'Cooper', 'dcooper@aol.com',
      '123 Main St', 'Yakima', 'WA', 98901, '792-223-8901', "1959-2-22",
      'M', NOW(), 3.50, NULL);

      a. Inserting Data into a Table

      b. INSERT INTO student VALUES('Harry', 'Truman', 'htruman@aol.com',
      '202 South St', 'Vancouver', 'WA', 98660, '792-223-9810', "1946-1-24",
      'M', NOW(), 3.50, NULL);

      INSERT INTO student VALUES('Shelly', 'Johnson', 'sjohnson@aol.com',
      '9 Pond Rd', 'Sparks', 'NV', 89431, '792-223-6734', "1970-12-12",
      'F', NOW(), 3.50, NULL);

      INSERT INTO student VALUES('Bobby', 'Briggs', 'bbriggs@aol.com',
      '14 12th St', 'San Diego', 'CA', 92101, '792-223-6178', "1967-5-24",
      'M', NOW(), 3.50, NULL);

      INSERT INTO student VALUES('Donna', 'Hayward', 'dhayward@aol.com',
```

```
      '120 16th St', 'Davenport', 'IA', 52801, '792-223-2001', "1970-3-24",
      'F', NOW(), 3.50, NULL);

      INSERT INTO student VALUES('Audrey', 'Horne', 'ahorne@aol.com',
      '342 19th St', 'Detroit', 'MI', 48222, '792-223-2001', "1965-2-1",
      'F', NOW(), 3.50, NULL);

      INSERT INTO student VALUES('James', 'Hurley', 'jhurley@aol.com',
      '2578 Cliff St', 'Queens', 'NY', 11427, '792-223-1890', "1967-1-2",
      'M', NOW(), 3.50, NULL);

      INSERT INTO student VALUES('Lucy', 'Moran', 'lmoran@aol.com',
      '178 Dover St', 'Hollywood', 'CA', 90078, '792-223-9678', "1954-11-27",
      'F', NOW(), 3.50, NULL);

      INSERT INTO student VALUES('Tommy', 'Hill', 'thill@aol.com',
      '672 High Plains', 'Tucson', 'AZ', 85701, '792-223-1115', "1951-12-21",
      'M', NOW(), 3.50, NULL);

      INSERT INTO student VALUES('Andy', 'Brennan', 'abrennan@aol.com',
      '281 4th St', 'Jacksonville', 'NC', 28540, '792-223-8902', "1960-12-27",
      'M', NOW(), 3.50, NULL);

11. SELECT * FROM student;

      a. Shows all the student data

12. CREATE TABLE class(
      name VARCHAR(30) NOT NULL,
      class_id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY);

      a. Create a separate table for all classes

13. show tables;

      a. Show all the tables

14. INSERT INTO class VALUES
('English', NULL), ('Speech', NULL), ('Literature', NULL),
('Algebra', NULL), ('Geometry', NULL), ('Trigonometry', NULL),
('Calculus', NULL), ('Earth Science', NULL), ('Biology', NULL),
('Chemistry', NULL), ('Physics', NULL), ('History', NULL),
('Art', NULL), ('Gym', NULL);

      a. Insert all possible classes

      b. select * from class;

15. CREATE TABLE test(
```

```
      date DATE NOT NULL,
      type ENUM('T', 'Q') NOT NULL,
      class_id INT UNSIGNED NOT NULL,
      test_id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY);
```

a. class_id is a foreign key

I. Used to make references to the Primary Key of another table

II. Example: If we have a customer and city table. If the city table had a column which listed the unique primary key of all the customers, that Primary Key listing in the city table would be considered a Foreign Key.

III. The Foreign Key can have a different name from the Primary Key name.

IV. The value of a Foreign Key can have the value of NULL.

V. A Foreign Key doesnâ''t have to be unique

16. CREATE TABLE score(

```
      student_id INT UNSIGNED NOT NULL,
      event_id INT UNSIGNED NOT NULL,
      score INT NOT NULL,
      PRIMARY KEY(event_id, student_id));
```

a. We combined the event and student id to make sure we don't have duplicate scores and it makes it easier to change scores

b. Since neither the event or the student ids are unique on their own we are able to make them unique by combining them

17. CREATE TABLE absence(

```
      student_id INT UNSIGNED NOT NULL,
      date DATE NOT NULL,
      PRIMARY KEY(student_id, date));
```

a. Again we combine 2 items that aren't unique to generate a unique key

18. Add a max score column to test

a. ALTER TABLE test ADD maxscore INT NOT NULL AFTER type;

b. DESCRIBE test;

19. Insert Tests

a. INSERT INTO test VALUES
('2014-8-25', 'Q', 15, 1, NULL),

```
('2014-8-27', 'Q', 15, 1, NULL),
('2014-8-29', 'T', 30, 1, NULL),
('2014-8-29', 'T', 30, 2, NULL),
('2014-8-27', 'Q', 15, 4, NULL),
('2014-8-29', 'T', 30, 4, NULL);
```

b. select * FROM test;

20. ALTER TABLE score CHANGE event_id test_id
    INT UNSIGNED NOT NULL;

    a. Change the name of event_id in score to test_id

    b. DESCRIBE score;

21. Enter student scores

    a. INSERT INTO score VALUES

```
(1, 1, 15),
(1, 2, 14),
(1, 3, 28),
(1, 4, 29),
(1, 5, 15),
(1, 6, 27),
(2, 1, 15),
(2, 2, 14),
(2, 3, 26),
(2, 4, 28),
(2, 5, 14),
(2, 6, 26),
(3, 1, 14),
(3, 2, 14),
(3, 3, 26),
(3, 4, 26),
(3, 5, 13),
(3, 6, 26),
(4, 1, 15),
(4, 2, 14),
(4, 3, 27),
(4, 4, 27),
(4, 5, 15),
(4, 6, 27),
(5, 1, 14),
(5, 2, 13),
(5, 3, 26),
(5, 4, 27),
(5, 5, 13),
(5, 6, 27),
```

```
    (6, 1, 13),
    (6, 2, 13),
    # Missed this day (6, 3, 24),
    (6, 4, 26),
    (6, 5, 13),
    (6, 6, 26),
    (7, 1, 13),
    (7, 2, 13),
    (7, 3, 25),
    (7, 4, 27),
    (7, 5, 13),
    # Missed this day (7, 6, 27),
    (8, 1, 14),
    # Missed this day (8, 2, 13),
    (8, 3, 26),
    (8, 4, 23),
    (8, 5, 12),
    (8, 6, 24),
    (9, 1, 15),
    (9, 2, 13),
    (9, 3, 28),
    (9, 4, 27),
    (9, 5, 14),
    (9, 6, 27),
    (10, 1, 15),
    (10, 2, 13),
    (10, 3, 26),
    (10, 4, 27),
    (10, 5, 12),
    (10, 6, 22);

22. Fill in the absences

    a. INSERT INTO absence VALUES
    (6, '2014-08-29'),
    (7, '2014-08-29'),
    (8, '2014-08-27');

23. SELECT * FROM student;

    a. Shows everything in the student table

24. SELECT FIRST_NAME, last_name
    FROM student;

    a. Show just selected data from the table (Not Case Sensitive)

25. RENAME TABLE
    absence to absences,
```

```
    class to classes,
    score to scores,
    student to students,
    test to tests;

    a. Change all the table names SHOW TABLES;

26. SELECT first_name, last_name, state
    FROM students
    WHERE state="WA";

    a. Show every student born in the state of Washington

27. SELECT first_name, last_name, birth_date
    FROM students
    WHERE YEAR(birth_date) >= 1965;

    a. You can compare values with =, >, <, >=, <=, !=

    b. To get the month, day or year of a date use MONTH(), DAY(), or YEAR()

27. SELECT first_name, last_name, birth_date
    FROM students
    WHERE MONTH(birth_date) = 2 OR state="CA";

    a. AND, && : Returns a true value if both conditions are true

    b. OR, || : Returns a true value if either condition is true

    c. NOT, ! : Returns a true value if the operand is false

28. SELECT last_name, state, birth_date
    FROM students
    WHERE DAY(birth_date) >= 12 && (state="CA" || state="NV");

    a. You can use compound logical operators

29. SELECT last_name
    FROM students
    WHERE last_name IS NULL;

    SELECT last_name
    FROM students
    WHERE last_name IS NOT NULL;

    a. If you want to check for NULL you must use IS NULL or IS NOT NULL

30. SELECT first_name, last_name
    FROM students
```

```
      ORDER BY last_name;

      a. ORDER BY allows you to order results. To change the order use
      ORDER BY col_name DESC;

31. SELECT first_name, last_name, state
      FROM students
      ORDER BY state DESC, last_name ASC;

      a. If you use 2 ORDER BYs it will order one and then the other

32. SELECT first_name, last_name
      FROM students
      LIMIT 5;

      a. Use LIMIT to limit the number of results

33. SELECT first_name, last_name
      FROM students
      LIMIT 5, 10;

      a. You can also get results 5 through 10

34. SELECT CONCAT(first_name, " ", last_name) AS 'Name',
      CONCAT(city, ", ", state) AS 'Hometown'
      FROM students;

      a. CONCAT is used to combine results

      b. AS provides for a way to define the column name

35. SELECT last_name, first_name
      FROM students
      WHERE first_name LIKE 'D%' OR last_name LIKE '%n';

      a. Matches any first name that starts with a D, or ends with a n

      b. % matches any sequence of characters

36. SELECT last_name, first_name
      FROM students
      WHERE first_name LIKE '___y';

      a. _ matches any single character

37. SELECT DISTINCT state
      FROM students
      ORDER BY state;
```

```
    a. Returns the states from which students are born because DISTINCT
    eliminates duplicates in results

38. SELECT COUNT(DISTINCT state)
    FROM students;

    a. COUNT returns the number of matches, so we can get the number
    of DISTINCT states from which students were born

39. SELECT COUNT(*)
    FROM students;

    SELECT COUNT(*)
    FROM students
    WHERE sex='M';

    a. COUNT returns the total number of records as well as the total
    number of boys

40. SELECT sex, COUNT(*)
    FROM students
    GROUP BY sex;

    a. GROUP BY defines how the results will be grouped

41. SELECT MONTH(birth_date) AS 'Month', COUNT(*)
    FROM students
    GROUP BY Month
    ORDER BY Month;

    a. We can get each month in which we have a birthday and the total
    number for each month

42. SELECT state, COUNT(state) AS 'Amount'
    FROM students
    GROUP BY state
    HAVING Amount > 1;

    a. HAVING allows you to narrow the results after the query is executed

43. SELECT
    test_id AS 'Test',
    MIN(score) AS min,
    MAX(score) AS max,
    MAX(score)-MIN(score) AS 'range',
    SUM(score) AS total,
    AVG(score) AS average
    FROM scores
    GROUP BY test_id;
```

a. There are many math functions built into MySQL. Range had to be quoted because it is a reserved word.

b. You can find all reserved words here http://dev.mysql.com/doc/mysqld-version-reference/en/mysqld-version-reference-reservedwords-5-5.html

44. The Built in Numeric Functions (SLIDE)

ABS(x) : Absolute Number: Returns the absolute value of the variable x.

ACOS(x), ASIN(x), ATAN(x), ATAN2(x,y), COS(x), COT(x), SIN(x), TAN(x) :Trigonometric Functions : They are used to relate the angles of a triangle to the lengths of the sides of a triangle.

AVG(column_name) : Average of Column : Returns the average of all values in a column. SELECT AVG(column_name) FROM table_name;

CEILING(x) : Returns the smallest number not less than x.

COUNT(column_name) : Count : Returns the number of non null values in the column. SELECT COUNT(column_name) FROM table_name;

DEGREES(x) : Returns the value of x, converted from radians to degrees.

EXP(x) : Returns e^x

FLOOR(x) : Returns the largest number not grater than x

LOG(x) : Returns the natural logarithm of x

LOG10(x) : Returns the logarithm of x to the base 10

MAX(column_name) : Maximum Value : Returns the maximum value in the column. SELECT MAX(column_name) FROM table_name;

MIN(column_name) : Minimum : Returns the minimum value in the column. SELECT MIN(column_name) FROM table_name;

MOD(x, y) : Modulus : Returns the remainder of a division between x and y

PI() : Returns the value of PI

POWER(x, y) : Returns x ^ Y

RADIANS(x) : Returns the value of x, converted from degrees to radians

RAND() : Random Number : Returns a random number between the values of 0.0 and 1.0

ROUND(x, d) : Returns the value of x, rounded to d decimal places

SQRT(x) : Square Root : Returns the square root of x

STD(column_name) : Standard Deviation : Returns the Standard Deviation of values in the column. SELECT STD(column_name) FROM table_name;

SUM(column_name) : Summation : Returns the sum of values in the column. SELECT SUM(column_name) FROM table_name;

TRUNCATE(x) : Returns the value of x, truncated to d decimal places

45. SELECT * FROM absences;

```
DESCRIBE scores;

SELECT student_id, test_id
FROM scores
WHERE student_id = 6;

INSERT INTO scores VALUES
(6, 3, 24);

DELETE FROM absences
WHERE student_id = 6;
```

a. Look up students that missed a test

b. Look up the specific test missed by student 6

c. Insert the make up test result

d. Delete the record in absences

46.
```
ALTER TABLE absences
    ADD COLUMN test_taken CHAR(1) NOT NULL DEFAULT 'F'
    AFTER student_id;
```

a. Use ALTER to add a column to a table. You can use AFTER or BEFORE to define the placement

47.
```
ALTER TABLE absences
    MODIFY COLUMN test_taken ENUM('T','F') NOT NULL DEFAULT 'F';
```

a. You can change the data type with ALTER and MODIFY COLUMN

48.
```
ALTER TABLE absences
    DROP COLUMN test_taken;
```

```
   a. ALTER and DROP COLUMN can delete a column

49. ALTER TABLE absences
      CHANGE student_id student_id INT UNSIGNED NOT NULL;

      a. You can change the data type with ALTER and CHANGE

50. SELECT *
    FROM scores
    WHERE student_id = 4;

      UPDATE scores SET score=25
      WHERE student_id=4 AND test_id=3;

      a. Use UPDATE to change a value in a row

51. SELECT first_name, last_name, birth_date
      FROM students
      WHERE birth_date
      BETWEEN '1960-1-1' AND '1970-1-1';

      a. Use BETWEEN to find matches between a minimum and maximum

52. SELECT first_name, last_name
      FROM students
      WHERE first_name IN ('Bobby', 'Lucy', 'Andy');

      a. Use IN to narrow results based on a predefined list of options

53. SELECT student_id, date, score, maxscore
      FROM tests, scores
      WHERE date = '2014-08-25'
      AND tests.test_id = scores.test_id;

      a. To combine data from multiple tables you can perform a JOIN
      by matching up common data like we did here with the test ids

      b. You have to define the 2 tables to join after FROM

      c. You have to define the common data between the tables after WHERE

54. SELECT scores.student_id, tests.date, scores.score, tests.maxscore
      FROM tests, scores
      WHERE date = '2014-08-25'
      AND tests.test_id = scores.test_id;

      a. It is good to qualify the specific data needed by proceeding
      it with the tables name and a period
```

```
      b. The test_id that is in scores is an example of a foreign key, which
      is a reference to a primary key in the tests table

55. SELECT CONCAT(students.first_name, " ", students.last_name) AS Name,
      tests.date, scores.score, tests.maxscore
      FROM tests, scores, students
      WHERE date = '2014-08-25'
      AND tests.test_id = scores.test_id
      AND scores.student_id = students.student_id;

      a. You can JOIN more than 2 tables as long as you define the like
      data between those tables

56. SELECT students.student_id,
      CONCAT(students.first_name, " ", students.last_name) AS Name,
      COUNT(absences.date) AS Absences
      FROM students, absences
      WHERE students.student_id = absences.student_id
      GROUP BY students.student_id;

      a. If we wanted a list of the number of absences per student we
      have to group by student_id or we would get just one result

57. SELECT students.student_id,
      CONCAT(students.first_name, " ", students.last_name) AS Name,
      COUNT(absences.date) AS Absences
      FROM students LEFT JOIN absences
      ON students.student_id = absences.student_id
      GROUP BY students.student_id;

      a. If we need to include all information from the table listed
      first "FROM students", even if it doesn't exist in the table on
      the right "LEFT JOIN absences", we can use a LEFT JOIN.

58. SELECT students.first_name,
      students.last_name,
      scores.test_id,
      scores.score
      FROM students
      INNER JOIN scores
      ON students.student_id=scores.student_id
      WHERE scores.score <= 15
      ORDER BY scores.test_id;

      a. An INNER JOIN gets all rows of data from both tables if there
      is a match between columns in both tables

      b. Here I'm getting all the data for all quizzes and matching that
```

data up based on student ids

59. One-to-One Relationship (SLIDE)

a. In this One-to-One relationship there can only be one social security number per person. Hence, each social security number can be associated with one person. As well, one person in the other table only matches up with one social security number.

b. One-to-One relationships can be identified also in that the foreign keys never duplicate across all rows.

c. If you are confused by the One-to-One relationship it is understandable, because they are not often used. Most of the time if a value never repeats it should remain in the parent table being customer in this case. Just understand that in a One-to-One relationship, exactly one row in a parent table is related to exactly one row of a child table.

60. One-to-Many Relationship

a. When we are talking about One-to-Many relationships think about the table diagram here. If you had a list of customers chances are some of them would live in the same state. Hence, in the state column in the parent table, it would be common to see a duplication of states. In this example, each customer can only live in one state so their would only be one id used for each customer.

b. Just remember that, a One-to-Many relationship is one in which a record in the parent table can have many matching records in the child table, but a record in the child can only match one record in the parent. A customer can choose to live in any state, but they can only live in one at a time.

61. Many-to-Many Relationship

a. Many people can own many different products. In this example, you can see an example of a Many-to-Many relationship. This is a sign of a non-normalized database, by the way. How could you ever access this information:

b. If a customer buys more than one product, you will have multiple product idâ''s associated with each customer. As well, you would have multiple customer idâ''s associated with each product.