Relational Databases

Reading / References

- Concise Guide to Databases: A Practical Introduction[1] chapter 4
- Introduction to SQL: Mastering the Relational Database Language[2]
- MySQL[3]
- Stanford free online course on databases[4]

Notes

There are numerous relational database systems out there, all with similar functionalities, and all supporting the SQL standard. So talking to one of them is not all that dissimilar to talking to any of the others.

We will be using mySQL WorkBench to inteface with mySQL. You should find it under the Developer tab in your system's start menu. If you want to work on your own computers instead, you would need to install it manually, and possibly even install mySQL if it is not already installed.

The first thing we're going to need is a connection to a database. You all have such a connection set up in vault, which is the same place that you can store webapp-related information. We will tell mySQL WorkBench about it.

Setting up mySQLWorkbench

When you first open up mySQLWorkbench, there is a "connections" section on the top left, with a plus sign to create a new connection. Here's what you need to specify in it:

- Use any name you like for the connection.
- The connection method should be standard TCP/IP.
- The host name should be vault.hanover.edu
- Your username is your Hanover/email login.
- You do not need to specify a password right away, though you can if you want to. You will be asked for a password when you try to connect.
- When it asks you for a password, that password is HC_XXXXXX where the XXXXXX is replaced by your 6-digit id number. We will learn later how to change the password.
- The "default schema", another name for "database", should be your login name. Out of the many databases/schemas that are available in vault, this one has been created for you. You will be placing all your tables etc there.

[1] http://learning.acm.org/books/book_detail.cfm?id=2560109&type=24
[2] http://learning.acm.org/books/book_detail.cfm?id=1208031&type=safari
[3] http://learning.acm.org/books/book_detail.cfm?id=2484635&type=safari
[4] https://lagunita.stanford.edu/courses/Home/Databases/Engineering/about

- Double-Click on the connection to connect and start working with queries.

For now we will be using the workbench only as a place to write scripts. Later on we may do more with it.

If you don't have an open query page yet, open one via the File menu.

Basic MySQL commands

We will explore now various SQL commands. SQL is essentially just another programming language, though it differs from most programming languages in that it is very **declarative** in nature. We tell it *what* we want it to return, but not *how* to do it.

Here is a list of the main SQL commands. For reference, here is also a quick cheatsheet[5].

SHOW TABLES Lists the tables present in the database/schema.
DESCRIBE TABLE Returns information about a table.
CREATE TABLE Used to create a new table.
DROP TABLE Used to remove/delete a whole table.
ALTER TABLE Used to make changes to a table's definition (e.g. add a new column, or set an index or add a constraint).
INSERT Used to insert new values/rows into a table
SELECT Probably the most used of all the commands. Returns some results according to a query. Can also be used as part of other commands.
UPDATE Used to change particular parts of particular rows.
DELETE Used to delete whole rows based on some query.

We will now consider most of these in greater detail.

CREATE TABLE We will create a new table as an example of using the CREATE TABLE command. We will use it to showcase some of the different variable types in SQL. Before we proceed, it is customary to use all capitals for the various SQL commands, and to use lowercase for anything else. It is purely a convenience for readability.

```
CREATE TABLE students (
    id      INT   UNIQUE    NOT NULL AUTO_INCREMENT,
    login VARCHAR(20) UNIQUE NOT NULL,
    first VARCHAR(20),
    last  VARCHAR(20),
    credits INT DEFAULT 0,
    gpa      DOUBLE DEFAULT 0,
    PRIMARY KEY (id)
);
```

[5] http://cse.unl.edu/~sscott/ShowFiles/SQL/CheatSheet/SQLCheatSheet.html

When you run this, it will create a new table, called students, with 6 different attributes/columns:

- An id, which is also the primary key, and must be an integer and "unique" (so that no two rows can have the same id). We also set that row to auto-increment, a feature specific to MySQL. This way we will not have to find the newest id to insert.
- A login, which must be not null and must also be unique. it is a character string of varied length, at most 20.
- Attributes first and last for the students' first and last name. They are allowed to be null.
- Number of credits the student has so far. It is an integer, and defaults to 0.
- The student's gpa, a double-precision number defaulting to 0.

Note that every SQL command ends with a semicolon. The whitespace is all optional.

For more practice let us also create courses:

```
CREATE TABLE courses (
    id      INT  UNIQUE   NOT NULL AUTO_INCREMENT,
    prefix CHAR(4) NOT NULL,
    no      INT NOT NULL,
    title   VARCHAR(55) NOT NULL,
    credits INT NOT NULL DEFAULT 4,
    UNIQUE KEY fullCode (prefix, no),
    PRIMARY KEY (id)
);
```

Some points to note:

- We made the prefix into a CHAR(4) type, which means that we cannot store more than 4 characters in it but that it should also use 4 bytes for it, even for shorter prefixes.
- We introduced a unique key for the pair of (prefix, no). We called that key fullCode, in case we want to later delete it. This will impose a constraint: The system will not allow two different courses to have the same prefix and number, and it will give us an error if we try to create one.

Now we come to the complicated part. We want to create the concept of students enrolled in courses. Since every student can enroll in many courses, and many courses can have many students in them, this is a many-to-many relationship. To express such a relationship we need a third **association table**. At its simplest this table will contain pairs of a student's id and a course's id. It may optionally contain more information, for instance whether the student is taking the course for credit, and what their grade in the course is (if the course is completed). We will create this new table, and link it to the other two via **foreign keys**.

```
CREATE TABLE enrollments (
    student_id INT NOT NULL,
    course_id INT NOT NULL,
```

```
    letter_grade  CHAR(2),
    point_grade   DOUBLE,
    FOREIGN KEY (student_id) REFERENCES students(id) ON DELETE CASCADE,
    FOREIGN KEY (course_id)  REFERENCES courses(id) ON DELETE CASCADE,
    PRIMARY KEY (student_id, course_id)
);
```

The new item here is the FOREIGN KEY line, which forces the corresponding column in enrollment to reference the id column in the students table. You would not be allowed to add an enrollment for a student id that doesn't exist. Also we have added ON DELETE CASCADE, which means that if a student is deleted from the students table, that deletion will cascade to the enrollments table, and all the enrollments of that student in courses will also be deleted.

We will only mention in passing the other commands related to tables. DROP TABLE simply removes a table from existence, and you permanently lose that table's contents. So be careful with it. ALTER TABLE can be used to make changes to a table, such as adding a new constraint or creating a new column. We will not discuss these further.

INSERT The INSERT command is used to add new values into a table. We will use it now to add numerous students and courses into the system. We will revisit it later when we combine it with SELECT queries and use the result of a SELECT query as the input to an INSERT.

The basic syntax of INSERT looks as follows:

```
INSERT INTO students (login, first, last) VALUES
    ("somebodyj1", "Joe", "Somebody"),
    ("somebodyj2", "Joel", "Somebody"),
    ("otherp1", "Peter", "Other"),
    ("otherm1", "Mary", "Other"),
    ("doem1", "Mary", "Doe"),
    ("doep1", "Peter", "Doe"),
    ("doed1", "David", "Doe");
```

So we have to indicate the table we want to insert to, and after that you typically include a list of which columns you will be specifying with the values. If you omit it, then the system would expect you to provide values for all attributes, and in the order in which they appear in the definition. It's always a good idea to specify them like we did above.

Let's add some courses:

```
INSERT INTO courses (prefix, no, title) VALUES
    ("MAT", 121, "Calculus 1"),
    ("CS", 220, "Intro to CS"),
    ("MAT", 122, "Calculus 2"),
    ("MAT", 221, "Calculus 3"),
    ("CS", 223, "Data Structures");
```

We will leave the enrollment of students to classes for later, after we discuss SELECT queries.

SELECT SELECT is the main way to read information out of the database. The simplest call is one that returns all entries from a table:

```
SELECT * FROM students;
```

This will print out the entire student table. Instead of an asterisk, we can specify which columns we want to to show:

```
SELECT first, last FROM students;
SELECT last FROM students;
```

Notice that with that last one we saw the same values multiple times. Sometimes you want that to happen, and sometimes you don't. You can control that behavior via the keyword DISTINCT.

```
SELECT DISTINCT last FROM students;
```

SELECT extras There are a couple of extra clauses we can add to a SELECT clause. One is a WHERE clause. For instance we can get the first names of all those whose last name is Somebody:

```
SELECT first FROM students
WHERE last = "Somebody";
```

Let's go further, and add a second restriction for the first name:

```
SELECT login, first, last FROM students
WHERE first = "Joe"
AND last = "Somebody";
```

We can also add an ordering:

```
SELECT first, last FROM students
ORDER BY last, first;
```

The above line will order by last name and then break ties by first name. There is one more clause we can add to the SELECT, but we will look at that a bit later.

INSERTS with SELECT query Let's practice some more complex inserts where the values are determined via a SELECT query. The idea is that instead of listing tuples of values, we will be placing a SELECT query, like so:

```
INSERT INTO enrollments (student_id, course_id)
SELECT id, 1 FROM students;
```

We just enrolled all students to the course with id 1.

Let's go a bit further. We will now enroll all students with last name Somebody to all CS courses. We will also add a safeguard in case some of those already existed.

```
INSERT IGNORE INTO enrollments (student_id, course_id)
SELECT s.id, c.id
FROM students AS s, courses AS c
WHERE s.last = "Somebody"
AND c.prefix = "CS";
```

So let's talk about this one, as it is considerably more complicated:

- The word IGNORE says if any problems are encountered in some rows, they are to be ignored. So if there was a pair that already existed, it will not try to add it.
- In the FROM clause we actually have two tables, and we use the AS construct to give them shorthand names. When we put multiple tables in the FROM clause, the database will look at all possible combinations of values. So in this case it will start off with all pairs of a student and a course, then use the WHERE clause to filter some out.
- In the SELECT clause we use those shorthand names to make clear which "id" we are talking about.

Subordinate SELECT queries Let's take it up one more notch. We want to find all students that are not enrolled in any computer science courses, and enroll them into the introductory CS course. Let's start by finding out those students' ids. This is tricky. Here is a verbal description:

We want to select those students from s, whose id cannot be found in any enrollment where the course has prefix CS.

```
SELECT s.id
FROM students as s
WHERE NOT EXISTS (SELECT prefix
                  FROM enrollments as e, courses as c
                  WHERE e.course_id = c.id
                  AND e.student_id = s.id
                  AND c.prefix = "CS"
                 );
```

Note the inner SELECT query, where we look for pairs of enrollments and courses, and we tie them together via the IDs.

You might have been tempted to do the following instead, and it would have been wrong:

```
SELECT student_id
FROM enrollments as e, courses as c
WHERE e.course_id = c.id
AND c.prefix <> "CS";
```

This would not have been right: It looks for all enrollments of students in courses. So it would include a student as long as they are enrolled in at least one non-CS course, even if the same student is also enrolled in a CS course.

Now that we got the list of the student IDs, we need to use that whole thing inside an INSERT query. This is a common practice: Work out the SELECT first, and when you have that working then put it in the INSERT clause. In fact we'll take one more step before the insert:

```
SELECT s.id, c2.id
FROM students as s, courses as c2
WHERE c2.prefix = "CS"
AND c2.no = 220
```

```
AND NOT EXISTS (SELECT prefix
                FROM enrollments as e, courses as c
                WHERE e.course_id = c.id
                AND e.student_id = s.id
                AND c.prefix = "CS"
               );
```

We needed to add some extra steps to get the correct course. We could have looked at the courses list, found the id and used that directly, but this way is a bit more elegant.

Finally, adding the INSERT:

```
INSERT INTO enrollments (student_id, course_id)
SELECT s.id, c2.id
FROM students as s, courses as c2
WHERE c2.prefix = "CS"
AND c2.no = 220
AND NOT EXISTS (SELECT prefix
                FROM enrollments as e, courses as c
                WHERE e.course_id = c.id
                AND e.student_id = s.id
                AND c.prefix = "CS"
               );
```

Joins Many times in the earlier sections we have had the need to consolidate different tables across foreign keys. Let us look at one more example, where we want to show all enrollments in terms of student name and course info. We will do that in steps.

First, we will try to look at enrollments but instead of seeing student ids, we want to see all their information. Our first attempt would be this:

```
SELECT *
FROM students s, enrollments e;
```

Try it out and look at the result. You should see first the columns for the students, and then the ones from the enrollments. But take a closer look at the id column, holding the student id, and the student_id column, holding the student id stored in the enrollment. They don't always match! Right now we have every student listed with every enrollment, not only theirs but of all the other students as well! That's clearly wrong. We need to make sure to only see the pair of a student and an enrollment if the enrollment corresponds to *that* student. We need a WHERE clause for that:

```
SELECT *
FROM students s, enrollments e
WHERE s.id = e.student_id;
```

Any time we bring together tables like that, it is called a **join**. SQL offers us an alternative way to describe such joins, like so:

```
SELECT *
FROM students s
JOIN enrollments e ON e.student_id = s.id;
```

If we had omitted the ON part it would have given us all pairs of students and enrollments. The ON part is the analog of the WHERE clause before.

This is often called an "inner" join. We can also have something called a "left join". Try it out and see the difference:

```
SELECT *
FROM students s
LEFT JOIN enrollments e ON e.student_id = s.id;
```

You see that this includes a row for a student that is not enrolled in any classes. There is also a "right join" that would have instead done the same thing for "enrollments", but that would not have given us anything new. There should also be a "full join" that preserves rows for values that appear in only one side, without a match on the other side, but MySQL does not support it. There are ways to emulate it however, and if you find yourself needing it just search online for the many answers.

Here is an further example to incorporate the course info in the table:

```
SELECT first, last, prefix, no
FROM students as s
LEFT JOIN enrollments as e ON s.id = e.student_id
LEFT JOIN courses as c ON c.id = e.course_id
ORDER BY prefix, no;
```

Let us practice some more with joins. You can do these problems via either joins or just using a WHERE clause where appropriate.

- Find all pairs of students and courses where the student is enrolled in the course.
- Find all pairs of students and CS courses where the student is enrolled in the course.
- Find all pairs of ids for students that are in the same class. This would require joining two enrollments tables (i.e. joining the enrollments table with another copy of itself). You should not include pairs that consist of the same student twice.
- Find all pairs of students with the same last name but different first names.
- In the two problems above, find a way to make it so that pairs only appear once, i.e. if we have students s and t we would NOT see both the pair (s,t) and the pair (t,s).

SQL functions SQL contains a number of built-in functions, and even allows you to create your own, though that is a more advanced process. You can find a full list of the available mySQL functions here[6]. We will highlight a few:

- AVG() returns the average
- CONCAT() concatenates strings
- COUNT() and COUNT(DISTINCT) return counts of matches
- MAX() and MIN() find maximum and minimum values

[6] http://dev.mysql.com/doc/refman/5.7/en/func-op-summary-ref.html

- RAND() produces a random floating point value
- SUM() adds up all the values

Here is a simple example returning the concatenated first-last names of the students:

```
SELECT CONCAT(first, " ", last)
FROM students;
```

Let us look at how many students each course has. In order to do that we need to also learn about grouping. Look at the following:

```
SELECT course_id, COUNT(*)
FROM enrollments
GROUP BY course_id;
```

OK! So it shows us the three courses that have students enrolled in them, as well as how many there are there. GROUP BY tells it to group all those rows with the same course_id, and perform the described operation to them. In such a case you are restricted into what you can put in the SELECT part. They must be either the entries in the GROUP_BY clause or functions that aggregate across the entire list of entries, like COUNT.

More advanced queries This is a bit unsatisfactory however. First, we would like to see the course prefix plus number, not just the ids. Second, we would like to see the courses with 0 students included. This is trickier.

```
SELECT prefix, no, (SELECT COUNT(student_id)
                    FROM enrollments
                    WHERE course_id = id) AS enrollment
FROM courses;
```

This is a different example of a subordinate SELECT query. This time the whole SELECT query goes into one of the "column" in the outer SELECT. Within its form we can use the specific id for the course in that row. So what we are saying here is basically "for each course, count the number of enrollments with that course id, and these numbers form the basis for the enrollment column".

We may further want to order by that new column:

```
SELECT prefix, no, (SELECT COUNT(student_id)
                    FROM enrollments
                    WHERE course_id = id) AS enrollment
FROM courses
ORDER BY enrollment DESC;
```

DELETE We can use DELETE to remove entries. For instance let's remove all students with first name Peter from all classes. As this is a destructive operation, you want to do a SELECT query first to make sure you have the right cases. You should do this yourself: Find all enrollments where the student's name is "Joe".

```
SELECT * FROM enrollments
WHERE student_id IN (SELECT id from students
                     WHERE first = "Joe");
```

Now we change this into a DELETE. Remember we just want to delete the enrollment, not the student record:

```
DELETE FROM enrollments
WHERE student_id IN (SELECT id from students
                     WHERE first = "Joe");
```

Oops, did you see the error message? The system refused to do this. By default mysql will discourage you from deleting in any situation where the WHERE clause doesn't include conditions on all key attributes. In this case that would have had to be the combined student_id and course_id columns, but we did not use the course_id ones. This is called "safe update mode" and it is there to protect us from making too many changes by accident. One way around this is to disable the mode altogether, then reconnect to the server. Another is to add a clause for the missing key, with a condition that is always true:

```
DELETE FROM enrollments
WHERE student_id IN (SELECT id from students
                     WHERE first = "Joe")
AND course_id > 0;
```

UPDATE We can use the UPDATE method to set values within particular records. Some of the clauses in it are similar to that in a SELECT clause, where we need to identify which rows to act on. Here is an example. Let us say that we want to update the entry in a student's file that shows the number of credits the student has. This would entail adding up the credits from the various courses that the student is enrolled in. Let us first work out the SELECT query that would get that information for us. Do it yourself before reading on.

```
SELECT s.id, (SELECT SUM(c.credits)
              FROM courses AS c
              JOIN enrollments AS e on e.course_id = c.id
              WHERE e.student_id = s.id)
FROM students s;
```

When you run the above query you will notice that some of the sums are NULL. That is what happens for students that are not enrolled in any courses. We would rather have 0s there. We could do this using the function COALESCE that returns the first non-null argument in its list:

```
SELECT s.id, (SELECT COALESCE(SUM(c.credits), 0)
              FROM courses AS c
              JOIN enrollments AS e on e.course_id = c.id
              WHERE e.student_id = s.id)
FROM students AS s;
```

We could now turn this into an UPDATE query:

```
UPDATE students AS s
SET s.credits = (SELECT COALESCE(SUM(c.credits), 0)
                 FROM courses AS c
                 JOIN enrollments AS e on e.course_id = c.id
                 WHERE e.student_id = s.id)
WHERE s.id > 0;
```

The last line is needed for the same “safe update” reasons we ran into when we were trying to do a delete.

Let us modify the above query a bit. We should only count students as having credits in courses in which they have received a letter grade. This would require us to change the inner SELECT query:

```
UPDATE students AS s
SET s.credits = (SELECT COALESCE(SUM(c.credits), 0)
                 FROM courses AS c
                 JOIN enrollments AS e on e.course_id = c.id
                 WHERE e.student_id = s.id
                 AND e.letter_grade IS NOT NULL)
WHERE s.id > 0;
```

We should find all students to have 0 credits now, as there are no assigned grades so far.

For your homework, you will create a new table to hold the numeric correspondence of letter grades to numbers, and then you would be able to use that to update the gpa entries.