Introduction to SQL

SQL (pronounced “sequel”), the Structured Query Language, is the most common language for interacting with relational database systems. Most database queries and updates can be implemented with simple SQL commands, but SQL also supports many sophisticated features including recursion and nested subqueries. Unlike familiar imperative and functional languages, SQL is a declarative language: a SQL query (or update) defines the logic of the computation without defining the control-flow of the program, and the database itself determines the control-flow of the program when it selects an algorithm and data access method to implement your query.

The SQL Data Definition Language

In addition to queries and updates, SQL includes language features for defining the relations -- or data schema -- of a database. Consider the following Django Models (the `__unicode__` functions are not shown):

```
class Student(models.Model):
      andrew_id = models.CharField(max_length=20, primary_key=True)
      first_name = models.CharField(max_length=200)
      last_name = models.CharField(max_length=200)
class Course(models.Model):
      name = models.CharField(max_length=200)
```

In Postgresql a database with these relations could be created as:

```
> create database uni;
> connect uni
> create table uni_student (
      andrew_id varchar(20) not null primary key,
      first_name varchar(200) not null,
      last_name varchar(200) not null);
> create table uni_course (
      id serial not null primary key,
      name varchar(200) not null);
```

SQL statements are terminated with a semi-colon, and the language (including variables) is not case sensitive. Databases usually support the familiar data types -- integers, floating point numbers, strings, dates, etc -- although the type names and representations are different in SQL than in most programming languages. Here, the `CharField` fields of the Django Models are represented in the database as `varchar`s, variable-length character strings. All of the fields in the database are declared as `not null` to prevent the database from allowing null values; by default Django fields are marked as `not null`. Also notice the `id` primary key of the course table. Django implicitly creates an integer `id` as the primary key if you do not explicitly declare a primary key, and the `id` field is declared as `serial` so the database will automatically assign the `id` of a new record to the next-higher value (compared to the current `id`s in the table) if the `id` is not explicitly specified.

Simple SQL queries and updates

After the database and student and course tables have been created, you can insert and access records in the database with SQL updates and queries. For example:

```
> insert into uni_student (andrew_id, first_name, last_name)
      values ('cgarrod', 'Charles', 'Garrod');
> insert into uni_student (andrew_id, first_name, last_name)
      values ('mjs', 'Mark', 'Stehlik');
> insert into uni_course (name) values ('Web Application Development');
```

After these insertions you can search for records in the database with the select command:

```
> select * from uni_student where last_name = 'Garrod';
```

would return:

```
+-----------+------------+-----------+
| andrew_id | first_name | last_name |
+-----------+------------+-----------+
| cgarrod   | Charles    | Garrod    |
+-----------+------------+-----------+
```

```
> select first_name, last_name from uni_student where last_name =
'Stehlik';
```

would return:

```
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| Mark       | Stehlik   |
+------------+-----------+
```

```
> select * from uni_course;
```

would return:

```
+----+----------------------------+
| id | name                       |
+----+----------------------------+
|  1 | Web Application Development |
+----+----------------------------+
```

You can easily update or delete current records:

```
> update uni_student set first_name = 'Charlie' where andrew_id =
'cgarrod';
> delete from uni_student where andrew_id = 'mjs';
```

Django relationship fields and SQL join queries

A Django `ForeignKey` relationship is stored by just including the primary key of the related model in the current model's table. Django `ManyToManyField` and `OneToManyField` relationships are more complicated; the relationship between the models is stored in a separate database table. Suppose we introduced a `ManyToManyField` into our `Course` model:

```
class Student(models.Model):
      andrew_id = models.CharField(max_length=20, primary_key=True)
      first_name = models.CharField(max_length=200)
      last_name = models.CharField(max_length=200)
class Course(models.Model):
      name = models.CharField(max_length=200)
      students = models.ManyToManyField(Student)
```

A database schema to represent these models might be:

```
> create table uni_student (
      andrew_id varchar(20) not null primary key,
      first_name varchar(200) not null,
      last_name varchar(200) not null);
> create table uni_course (
      id serial not null primary key,
      name varchar(200) not null);
> create table uni_enrollments (
       id serial not null primary key,
      course_id integer references uni_course(id),
      student_id varchar(20) references uni_student(andrew_id));
```

Notice that the `ManyToManyField` in the `Course` model does not require an extra column in the `uni_course` table. Instead, the `uni_enrollments` table represents the `students` field of the `Course` model; it is literally the enrollments of the various students in the various courses. The data types of the `uni_enrollments` table match the primary key types of the `uni_course` and `uni_student` tables, and the SQL `references` keyword causes Postgres to enforce the restriction that records in the `uni_enrollments` table must match some pre-existing records in the `uni_course` and `uni_student` tables. (Note that Django would actually call the `uni_enrollments` table `uni_course_students`. We chose the name `uni_enrollments` for clarity.)

With this data representation it is easy to add or remove students from courses by just inserting or deleting a row from the enrollments table. If `webapps` is a `Course` instance and `charlie` is a `Student` instance for the insertions above, then the Django ORM statement `webapps.students.add(charlie)` would be implemented in SQL as:

```
> insert into uni_enrollments (course_id, student_id)
                          values (1, 'cgarrod');
```

because `1` is the primary key value for `webapps` and `'cgarrod'` is the primary key for `charlie`.

The Django ORM's `RelatedManager` class and Query API allow you to directly obtain

information about the relationship in Python. To reconstruct complex information about the enrollments in SQL you need to combine, or join, information from multiple tables with a SQL *join* query. For example, you can use the Django ORM to retrieve all information about courses containing a student named Charlie:

```
> Course.objects.filter(students__first_name__exact='Charlie')
```

To similarly implement this in SQL you might execute:

```
> select uni_course.* from uni_student, uni_course, uni_enrollments
      where uni_student.andrew_id = uni_enrollments.student_id
        and uni_enrollments.course_id = uni_course.id
        and uni_student.first_name = 'Charlie';
```

Conceptually, this query combines every row of the student table with every row of the course table and every row of the enrollments table, and then filters out (combined) rows that don't match enrollments of students named Charlie. Mathematically, a join query without constraints (i.e., without a `where` clause) is the Cartesian product of the tables being joined.

Summary

This document is just a very brief introduction to basic SQL data definitions, queries, and updates, and joins. To use SQL effectively you will need to practice and learn more, using external resources. Most database systems (including Postgres) contain language references specifically for their system. SQL, unfortunately, is not completely standardized and there are minor differences between database systems; the examples in this document will work in Postgres but require slight modification to work in MySQL or another database system.