Introduction to SQL for Data Scientists

Ben O. Smith*
College of Business Administration
University of Nebraska at Omaha

Learning Objectives

By the end of this document you will learn:

1. How to perform basic SQL commands
2. Understand the difference between "left", "right" and "full" joins
3. Understand groupings and how they connect to aggregation functions
4. Understand data type basics
5. How to perform basic subqueries

1 Introduction

In the information sciences, we commonly have data spread across multiple data sets or database sources. Thankfully, most database servers have an agreed upon a standard format to interact, merge and answer questions with that data: SQL (Structured Query Language). While subtle distinctions exists between database systems (SQL Server, SQLite, MySQL, Oracle and others), SQL is (mostly) a portable skill across server platforms.

1.1 Tables and Data Types

In most cases, you will be interacting with or creating tables of data. Each column in a table has a specific data type. These types not only indicate how the data is stored, but how *queries* (questions you ask) are executed. Let's suppose you are examining student data with three simple tables.

*bosmith@unomaha.edu

"student" data table

Column Name	Data Type
id*	Integer
name	Text

"term_gpa" data table

Column Name	Data Type
id*	Integer
term*	Integer
gpa	Float

"degrees" data table

Column Name	Data Type
id*	Integer
term	Integer
degree*	Char(5)

* Indicates primary key of the table

Amongst the seven columns there are four different data types: integer, float, char and text. Integer is arguably the simplest form of data a computer can store (ignoring booleans and tiny storage methods – which are just variations on the integer). Integers are stored by simply translating the number to binary. This means that a 32 bit unsigned integer can store any number between 0 and $4,294,967,295$ $(2^{32} - 1)$. Because of its very efficient method of storage, searching, ordering and grouping integers is almost always the best option (assuming you have an option).

A float is simply an integer with some part of the number indicating a position of the decimal place. For instance, traditionally in a 32 bit float, 24 bits are dedicated to the number itself (i.e. the integer) while the remaining bits are used to describe the position of the decimal place. For the purposes of database use, one should understand two basic ideas about floating points: they are an efficient form of storage, but not as good as integers and if you perform mathematical operations using both integers and floats all of the data will be converted to floats to execute the task.

Chars are a fixed-length string of text. For instance, char(5) would store 5 characters. Computers can't actually store characters directly, so something called a character map is used. So, suppose you have 256 different characters in an alphabet (which is generally considered the minimum), you could have a stream of ones and zeros stored somewhere where the stream is divided into groups of 8 (this would be an 8 bit character set). Under these conditions, each letter you store would take 8 bits of storage (so a char(5) would take 40 bits per row). In a fixed length text stream environment, the entire fixed length is stored even if only part of it is used.

While the text datatype stores each character the same way a char does, it doesn't have a predefined fixed length. So, there is two options: each row could be of differing lengths (this is what varchar does), or the data in this column is stored separate from the table (which is what text does). Both have bad performance results. If each row is of varying length, then aggregation

functions and groupings are far slower. But storing the data away from table isn't good either as that means searching involves a bunch of redirection operations. It is obviously important to store text data in databases, but you should always be clear on what the performance implications are, especially as you start performing joins and subqueries.

1.1.1 Primary Keys

Every row should have a unique value that identifies it known as a primary key (which is a type of index). Often times this is a single, non-repeating integer (as is the case with the student data table), but it doesn't have to be: as long as the combination of columns describe a unique value. In general, accessing a row by its primary key is the fastest method.

1.2 Data

To assist with our exploration of the SQL language we will define our data:

"student"

id	name
1	Edith Warton
2	Dorian Gray
3	Ophelia
4	Henry James
5	Harper Lee

"term_gpa"

id	term	gpa
1	2011	3.32
1	2012	3.51
2	2011	2.22
2	2013	1.70
3	2011	3.70
4	2011	3.10
4	2012	3.21
4	2013	3.30
5	2013	2.99

"degrees"

id	term	degree
1	2012	EconS
3	2011	MathS
3	2011	CompS
4	2012	EngLT

While your datasets will likely be very large, in the process of learning the language it is usually good to start with something that you can visually see the answer.

2 Select

A select statement (which is a form of query) has three basic parts: *what do want*, *where is it from* and *under what conditions*. So an extremely basic command might be:

```
1 SELECT s.id AS id, s.name AS name
2 FROM student AS s
```

```
3 WHERE s.id=1;
```

Which results in a single row:

id	name
1	Edith Warton

So what's happening here? We're grabbing the id and name data (*SELECT s.id AS id, s.name AS name*) from the student table (which we are renaming "s" using the "AS" statement – *FROM student AS s*) where the id column equals one (*WHERE s.id=1*). The condition (*WHERE s.id=1*) is actually executed first. Because id is the primary key, the database simply looks up the location of the row and pulls a single value (the rest of the rows are not examined).

Now, let's suppose you want to attach the GPA of a specific term to these results.

```
1 SELECT s.id AS id, s.name AS name, t.gpa AS gpa
2 FROM student AS s
3 JOIN term_gpa AS t ON s.id=t.id
4 WHERE s.id=1 AND t.term=2012;
```

Which results in a single row:

id	name	gpa
1	Edith Warton	3.51

Let's break this apart: there are actually two result sets that are then merged. The conditions on each table can be thought to be executed separately then joined on the "on" condition (the exact execution order is up the database optimizer).

What happens if I do this?

```
1 SELECT s.id AS id, s.name AS name, t.gpa AS gpa
2 FROM student AS s
3 JOIN term_gpa AS t ON s.id=t.id
4 WHERE s.id=1;
```

Which results in a two rows:

Now that's interesting, you might expect to get a single result, but because the results from term_gpa result in two rows, both of which match the same row in student, when the two result

id	name	gpa
1	Edith Warton	3.32
1	Edith Warton	3.51

sets are merged it results in a repeat of the student information.

What if we remove the where condition entirely? What happens then:

```
SELECT s.id AS id, s.name AS name, t.gpa AS gpa
FROM student AS s
JOIN term_gpa AS t ON s.id=t.id;
```

id	name	gpa
1	Edith Warton	3.32
1	Edith Warton	3.51
2	Dorian Gray	2.22
2	Dorian Gray	1.70
3	Ophelia	3.70
4	Henry James	3.10
4	Henry James	3.21
4	Henry James	3.30
5	Harper Lee	2.99

As you can see, each record is repeated for each combination of rows that meet the criteria of the join.

However, let's add another table, in this case degrees, where the id must mach the id in the student table and the term must mach the term_gpa table.

```
SELECT s.id AS id, s.name AS name, t.gpa AS gpa
FROM student AS s
JOIN term_gpa AS t ON s.id=t.id
JOIN degrees AS d ON d.id=s.id AND t.term=d.term;
```

id	name	gpa
1	Edith Warton	3.51
3	Ophelia	3.70
3	Ophelia	3.70
4	Henry James	3.21

That resulted in a lot less rows! But it makes sense: any student who didn't also exist in the degrees table (i.e. they didn't graduate), couldn't be merged with the results from the student and term_gpa table. Also note that the fact that we aren't actually displaying information from the degrees table is not relevant to the merge process.

2.1 Left and Right Joins

But sometimes you don't want to eliminate un-merged rows, instead you would prefer blanks when row can not be matched. This is where left and right joins come in. Consider:

```
SELECT s.id AS id, s.name AS name, t.gpa AS gpa, d.degree  AS degree
FROM student AS s
JOIN term_gpa AS t ON s.id=t.id
LEFT JOIN degrees AS d ON d.id=s.id AND t.term=d.term;
```

id	name	gpa	degree
1	Edith Warton	3.32	
1	Edith Warton	3.51	EconS
2	Dorian Gray	2.22	
2	Dorian Gray	1.70	
3	Ophelia	3.70	CompS
3	Ophelia	3.70	MathS
4	Henry James	3.10	
4	Henry James	3.21	EngLT
4	Henry James	3.30	
5	Harper Lee	2.99	

This is seemingly the same criteria as before, but this time if it can't find something in the degrees table it simply attaches null values (instead of eliminating the row). The "left" part of the left join indicates that every row found on the left side of the join (proceeding the join) should be shown. A right join is simply the opposite[1].

Often times you actually care about which values have null when you attempt to join. For instance, let's say you wanted to find all students who didn't graduate as well as their GPA:

```
SELECT s.id AS id, s.name AS name, t.gpa AS gpa
FROM student AS s
```

[1]While left and right joins are both in the SQL syntax, right joins are not included in all databases. Given they do the same thing (it is just a matter of code arrangement), you should use left joins.

```
3 JOIN term_gpa AS t ON s.id=t.id
4 LEFT JOIN degrees AS d ON d.id=s.id AND t.term<=d.term
5 WHERE d.id IS NULL;
```

id	name	gpa
2	Dorian Gray	2.22
2	Dorian Gray	1.70
4	Henry James	3.30
5	Harper Lee	2.99

So, what we've got here is people that are enrolled, but have not graduated, by term. Note how Henry James is still listed because he has enrolled in a semester after his last degree.

2.2 Grouping

Let's try to clean this last result up. Maybe I don't want each students GPA by term, maybe I want an average. I also might want one row to represent one student.

```
1 SELECT s.id AS id, MAX(s.name) AS name, AVG(t.gpa) AS gpa
2 FROM student AS s
3 JOIN term_gpa AS t ON s.id=t.id
4 LEFT JOIN degrees AS d ON d.id=s.id AND t.term<=d.term
5 WHERE d.id IS NULL
6 GROUP BY s.id;
```

id	name	gpa
2	Dorian Gray	1.96
4	Henry James	3.30
5	Harper Lee	2.99

The concept of grouping is that you have a result and you are going to group it on some criteria. Aggregation functions (e.g. *AVG*, *MAX*, *SUM*, *COUNT*) work in conjunction with the group. So, when you use an aggregation function, it is giving you the average, sum, whatever that are rolled-up for that particular row.

It also of note that you must use an aggregation function on any column in your select list that you are not grouping on. This makes sense: you don't need to use aggregation on a column where

you are grouping because they are all the same for each row (that's why they are grouped), but without the aggregation function on a non-grouped row, the database won't know which of the results for a given column/row to show. Admittedly, sometimes it looks a bit silly (as is the case with the "MAX" function, in the example, run on a string), but it is very consistent, which is what you want from a language.

2.3 Subquery

Suppose for a moment that you wanted to get the inflation adjusted GPA of each student in the graduation term (we're assuming that GPA is inflated by 0.04 points per year)[2]:

```
SELECT s.id AS id, MAX(s.name) AS name, MIN(mGPA.gpa) AS gpa
FROM student AS s
JOIN term_gpa AS t ON s.id=t.id
JOIN degrees AS d ON d.id=s.id AND t.term<=d.term
JOIN (
    SELECT id, (gpa-(.04*(term-2011))) AS gpa, term
    FROM term_gpa
) AS mGPA ON s.id=mGPA.id AND mGPA.term=d.term
GROUP BY s.id;
```

id	name	gpa
1	Edith Warton	3.47
3	Ophelia	3.70
4	Henry James	3.17

This query involves a *subquery*, which is a query in its own right that is executed as a part of the larger query. In this case, the subquery is executed and produces a set of results that are then merged through the join process (much like if that data existed as a separate table).

Subqueries don't have to be in any specific part of the main query. Let's say you you want to get the average GPA of each student and place it along with the students name:

```
SELECT id, name, (
    SELECT AVG(gpa)
    FROM term_gpa
```

[2]I'm aware that this query could be constructed using a standard join, I've constructed it in this form to demonstrate the use of a subquery

```
    WHERE id=s.id
) AS avg_gpa
FROM student AS s
ORDER BY name ASC;
```

id	name	avg_gpa
2	Dorian Gray	1.96
1	Edith Warton	3.42
5	Harper Lee	2.99
4	Henry James	3.20
3	Ophelia	3.70

And you can have subqueries of subqueries. Let's suppose you are trying categorize the students based on their average GPAs:

```
SELECT id, name, (
    SELECT
        CASE
        WHEN avg_gpa >=3.5 THEN 2
        WHEN avg_gpa <3.5 AND avg_gpa >=3.0 THEN 1
        ELSE 0
        END AS gpa_type FROM
        (
            SELECT AVG(gpa) AS avg_gpa
            FROM term_gpa
            WHERE id=s.id
        ) AS avg_gpa_table
) AS gpa_type
FROM student AS s
ORDER BY name ASC;
```

id	name	gpa_type
2	Dorian Gray	0
1	Edith Warton	1
5	Harper Lee	0
4	Henry James	1
3	Ophelia	2

So, subqueries are executed *as if* they are mathematical operations (from inside of the parentheticals out). So in the inner subquery we get the average GPA for that particular row, then we

classify it using as CASE statement which is returned to the main query.

2.4 Like and more complicated conditions

Sometimes we would like to match part of a string. While this is a relatively slow operation, it is sometimes necessary. Suppose I'm looking for a particular person in the data, but I only have their first name:

```
SELECT id, name
FROM student AS s
WHERE name LIKE 'Dorian%';
```

The character "%" states *match anything of any length* (and "Like" statements are not case sensitive in general). It is often useful to simply specify part of a string that can be different.

```
SELECT id, name
FROM student AS s
WHERE name LIKE 'Dor__n%';
```

In this context, "_" means *match any one character* (suppose you don't know the spelling of a name).

You can use like conditions in very complicated settings. Let's say you want a list of graduating students with a high GPA or are in a science (assuming all of the science majors end in "S").

```
SELECT s.id AS id, MAX(s.name) AS name
FROM student AS s
JOIN degrees AS d ON d.id=s.id
WHERE (
    (
        SELECT AVG(gpa) AS gpa
        FROM term_gpa WHERE s.id=id AND term<=d.term
    ) >= 3.3
OR
    d.degree LIKE '%S'
)
GROUP BY s.id;
```

Henry James would appear in these results if the GPA requirement was adjusted down to something like 3.1

id	name
1	Edith Warton
3	Ophelia

3 Indexes

Indexes are a copy of a subset of a particular object's (e.g. table) data that is stored in a particular order. Indexes are commonly something called a "b-tree", which is a generalization of the binomial search tree but allows for multiple child nodes.

In the context of this document, we'll focus on how to think about indexes instead of how they exactly work. Imagine a library with a card catalogue containing small amounts of information about each book (namely the book name and the authors). They are also in a specific order and reference the location of the "real" data in the books. A database index can be thought of much in this way.

An index can be on any one or more columns. This includes text columns (though you generally index a few characters per row instead of the entire column; which is what you do on numeric columns – if it is not obvious to you why, please reread section 1.1).

Let's suppose their is a multicolumn index on (term, gpa, id) in the term_gpa table and we run the following query:

```
1 SELECT t.id AS id, t.gpa AS gpa
2 FROM term_gpa AS t
3 WHERE t.term=2011 AND t.gpa > 3.0
4 GROUP BY t.id;
```

Ok, so the database is able to use the first two columns of the index to satisfy the where condition and use the third to satisfy the group by. Note that the order is important: the where condition is executed first, so those columns must be the first items in the index (followed by group by and order by). So, if the multicolumn index was (term, id, gpa), it would still be used to satisfy the term condition but the database would resort to scanning the remaining rows to examine the gpa and group the results.

However, (term, id, gpa) would be perfect for something like this:

```
SELECT t.id AS id, t.gpa AS gpa
FROM term_gpa AS t
WHERE t.term>2009
GROUP BY t.id, t.gpa;
```

Or something like this:

```
SELECT t.id AS id, t.gpa AS gpa
FROM term_gpa AS t
WHERE t.term>2009
ORDER BY t.id ASC, t.gpa DESC;
```

That is, as long as the order of the columns matches the order of execution, the index will be of some use to the database.

However, consider text content. Suppose we have an index on (name, id) in the student table and we execute the following from our discussion of like:

```
SELECT id, name
FROM student AS s
WHERE name LIKE 'Dorian%';
```

So, in this scenario, the index on name would be helpful to at least reduce the rows that would have to be examined by the database. However, consider this query:

```
SELECT id, name
FROM student AS s
WHERE name LIKE '%Gray';
```

Now, the index becomes completely useless (the database will have to examine all rows). This is because the index is ordered by the text at the beginning of the column and '%' can match anything.

4 Creating Reusable Code

Creating code that is reusable is extremely important, especially if you work with other people. Complete reusable code files allows other users, who may not understand your logic, to run your code while modifying one important aspect of the code.

Unfortunately, this is an area that the database providers have been less-good about providing a consistent interface across systems. Nonetheless, these concepts are similar and the syntax for MySQL and MS SQL are very similar, thus we will follow their syntax.

Remember this example?

```
SELECT id, name, (
    SELECT
        CASE
        WHEN avg_gpa >=3.5 THEN 2
        WHEN avg_gpa <3.5 AND avg_gpa >=3.0 THEN 1
        ELSE 0
        END AS gpa_type FROM
        (
            SELECT AVG(gpa) AS avg_gpa
            FROM term_gpa
            WHERE id=s.id
        ) AS avg_gpa_table
) AS gpa_type
FROM student AS s
ORDER BY name ASC;
```

Now, I want you to suppose that you don't want every student, but just those enrolled in a specific semester. I'm going to do this by joining the student table to the term_gpa table.

```
SELECT s.id AS id, s.name AS name, (
    SELECT
        CASE
        WHEN avg_gpa >=3.5 THEN 2
        WHEN avg_gpa <3.5 AND avg_gpa >=3.0 THEN 1
        ELSE 0
        END AS gpa_type FROM
        (
            SELECT AVG(gpa) AS avg_gpa
            FROM term_gpa
            WHERE id=s.id AND term<=2012
        ) AS avg_gpa_table
) AS gpa_type
FROM student AS s
JOIN term_gpa AS t ON s.id=t.id
WHERE t.term=2012
```

```
17 ORDER BY s.name ASC;
```

Great! But maybe this information has to be run every year. This is how you would solve that problem in MS SQL.

Listing 1: Term_Report.sql

```
1  DECLARE @term INT;
2
3  SET @term = 2012;
4
5  /* List of active students by GPA type */
6
7  SELECT s.id AS id, s.name AS name, (
8      SELECT
9          CASE
10         WHEN avg_gpa >=3.5 THEN 2
11         WHEN avg_gpa <3.5 AND avg_gpa >=3.0 THEN 1
12         ELSE 0
13         END AS gpa_type FROM
14         (
15             SELECT AVG(gpa) AS avg_gpa
16             FROM term_gpa
17             WHERE id=s.id AND term<=@term
18         ) AS avg_gpa_table
19 ) AS gpa_type
20 FROM student AS s
21 JOIN term_gpa AS t ON s.id=t.id
22 WHERE t.term=@term
23 ORDER BY s.name ASC;
```

Now, what's important to to see here is that we've declared a "variable" (@term) at the top, defined its type and set it to a particular value. Then in the query we just type in that variable. This entire query can be saved as a *some_name*.sql file (as noted by the title) that some other person can run.

Further, you aren't limited to a single query. Suppose, in addition to students by GPA type, you want to get a list of students who are active in the next term and another list of students who are both not active in the next term and didn't graduate.

Listing 2: Complete_Term_Report.sql

```
DECLARE @term INT;

SET @term = 2012;

/* List of active students by GPA type */

SELECT s.id AS id, s.name AS name, (
    SELECT
        CASE
        WHEN avg_gpa >=3.5 THEN 2
        WHEN avg_gpa <3.5 AND avg_gpa >=3.0 THEN 1
        ELSE 0
        END AS gpa_type FROM
        (
            SELECT AVG(gpa) AS avg_gpa
            FROM term_gpa
            WHERE id=s.id AND term<=@term
        ) AS avg_gpa_table
) AS gpa_type
FROM student AS s
JOIN term_gpa AS t ON s.id=t.id
WHERE t.term=@term
ORDER BY s.name ASC;

/* List of active students who are active in the next term */

SELECT s.id AS id, s.name AS name
FROM student AS s
JOIN term_gpa AS t ON s.id=t.id AND t.term=@term
JOIN term_gpa AS tt ON s.id=tt.id AND tt.term=(@term+1)
ORDER BY s.name ASC;

/* List of active students who are not active in the next term and didn't graduate */

SELECT s.id AS id, s.name AS name
FROM student AS s
JOIN term_gpa AS t ON s.id=t.id AND t.term=@term
LEFT JOIN term_gpa AS tt ON s.id=tt.id AND tt.term=(@term+1)
```

```
39 LEFT JOIN degrees AS d ON s.id=d.id AND d.term=@term
40 WHERE tt.id IS NULL AND d.id IS NULL
41 ORDER BY s.name ASC;
```

Now, what's really handy about this is that another person can simply change the value of @term and rerun this report (and the comments tell the person what each query does).

5 Further Reading

5.1 Online Resources

1. SQL Quick Reference (http://www.w3schools.com/sql/sql_quickref.asp)
2. MySQL Reference Manuals (http://dev.mysql.com/doc/#manual)
3. MS SQL Reference (http://msdn.microsoft.com/en-us/library/ms189826(v=sql.90).aspx)

5.2 Books

1. SQL Pocket Guide (http://shop.oreilly.com/product/0636920013471.do)
2. Learning SQL (http://shop.oreilly.com/product/9780596520847.do)
3. High Performance MySQL (http://shop.oreilly.com/product/0636920022343.do)