Week 5 Cliff Notes

In week 4, we covered how to use the SELECT clause and the ORDER BY clause. This week we are going over the many “single-row” functions; these are functions that are applied to each row, one at a time, as opposed to group functions which take in many rows and output a group result (we will cover that in week 6). The examples are copied from our [SQL Fiddle](http://sqlfiddle.com/#!9/60125).

Here is a summary of the “grammar” of sql that we have covered so far:

**SELECT** [either one or more columns followed by column names, the ‘\*’, or another sql]

**FROM** [a rowsource, like a table or a view]

**WHERE** [a list of conditions that evaluate to TRUE or FALSE for each row in the rowsource]

**ORDER BY** [a list of output columns that you would like to sort by]

Functions can be used in most parts of the query (SELECT, WHERE and ORDER BY); note that putting a function in the ORDER BY clause is not very common, so I will not focus on it much, but just so you know, it is perfectly valid. While there are many functions supported by each flavor of sql, my goal is to help you understand how to use any given function, using common functions as a touchpoint. In general, a function does something predictable given one or more inputs.

**String functions**

There are a ton of string functions you can use to alter the way text is presented. Let’s start with something simple, like the UPPER function - can you guess what it does?

select upper(title) from movies;



There is also a function called LOWER:

select title, lower(title) from movies;

|  |  |
| --- | --- |
| **title** | **lower(title)** |
| Annie Hall | annie hall |
| Blade Runner | blade runner |
| Bronenosets Potyomkin | bronenosets potyomkin |
| Casablanca | casablanca |
| Citizen Kane | citizen kane |

Note how the function works only the column listed in parenthesis. We could also use any old string value:

select lower('UPPER CASE') from dual;

|  |
| --- |
| **lower('UPPER CASE')** |
| upper case |

Like all other single-row functions, these do work on each and every single row. Let’s try something a little more challenging.

Another popular one is substring, which allows you cut off a portion of a value. The parameters of SUBSTRING are:

**string** the value you want to chop up

**offset** which character to start the cut

**length** an optional parameter to determine how many characters you will cut

Let’s get the first three characters (maybe including spaces as well) of each movie:

select substring(title, 1, 3) from movies;



substring is another function which accepts arguments. The first thing we told substring to do is work on the column “title” (that is the first argument). Next we told it to start at the first character (2nd argument is 1). The last thing we told substring to do is chop “title” after the 3rd character. So the movie title “Annie Hall” becomes “Ann”.

If we wanted to grab *everything* the 4th character, we would omit the last parameter and just use:

select title, substring(title, 5) from movies;

|  |  |
| --- | --- |
| **title** | **substring(title, 5)** |
| Annie Hall | e Hall |
| Blade Runner | e Runner |
| Bronenosets Potyomkin | enosets Potyomkin |
| Casablanca | blanca |
| Citizen Kane | zen Kane |

A little exercise that is common is to use substring to find all movies with the letter ‘e’ as the 5th character. How would you write that substring query? We know from the previous example that substring(title, 5) returns all movies, but with the first 4 characters missing from the title. So this is close. To change it to output only the 5th letter, we can make a simple change to the substring parameters and tell it that we want a length of 1:

select title, substring(title, 5, 1) from movies;

|  |  |
| --- | --- |
| **title** | **substring(title, 5, 1)** |
| Annie Hall | e |
| Blade Runner | e |
| Bronenosets Potyomkin | e |
| Casablanca | b |
| Citizen Kane | z |

But now we want to test that character. So put it in the WHERE clause and compare to ‘e’:

select title from movies where substring(title, 5, 1) = 'e';

|  |
| --- |
| **title** |
| Annie Hall |
| Blade Runner |
| Bronenosets Potyomkin |

Most functions can be nested as well. “Nesting” means you put one function inside another, or in other words, you pass the output of one function as the input of another function. Still confused? Let’s do an example. Here we are going to first take a substring, then we are going to use UPPER on the substring - I’ll show you in a progression:

select title, substring(title, 1,6), upper(substring(title, 1,6)) from movies;

|  |  |  |
| --- | --- | --- |
| **title** | **substring(title, 1,6)** | **upper(substring(title, 1,6))** |
| Annie Hall | Annie | ANNIE |
| Blade Runner | Blade | BLADE |
| Bronenosets Potyomkin | Bronen | BRONEN |
| Casablanca | Casabl | CASABL |
| Citizen Kane | Citize | CITIZE |

Just remember that these ‘simple’ functions “do stuff” for each and every single row; it doesn’t matter how many times you nest simple functions. Next week we talk about group functions, or functions that do stuff on more than one row at a time. Hence the synonym aggregate function.

I like how the SQL Server documentation categorizes various functions - beware, there are a LOT of them. Also, we will not talk about “rowset” or “ranking” functions in this course, but they are all useful for certain tasks - today we introduced what Microsoft calls “Scalar Functions”:

<https://msdn.microsoft.com/en-us/library/ms174318.aspx>

We have not covered any new grammar per se - the big jump comes in a couple weeks when we talk about JOINing tables. Next week when we go over group functions, we will introduce new grammar as well. But for now, we still have:

**SELECT** [either one or more columns and/or functions, or another sql]

**FROM** [a rowsource, like a table or a view]

**WHERE** [a list of conditions that evaluate to TRUE or FALSE for each row in the rowsource]

**ORDER BY** [a list of output columns that you would like to sort by]