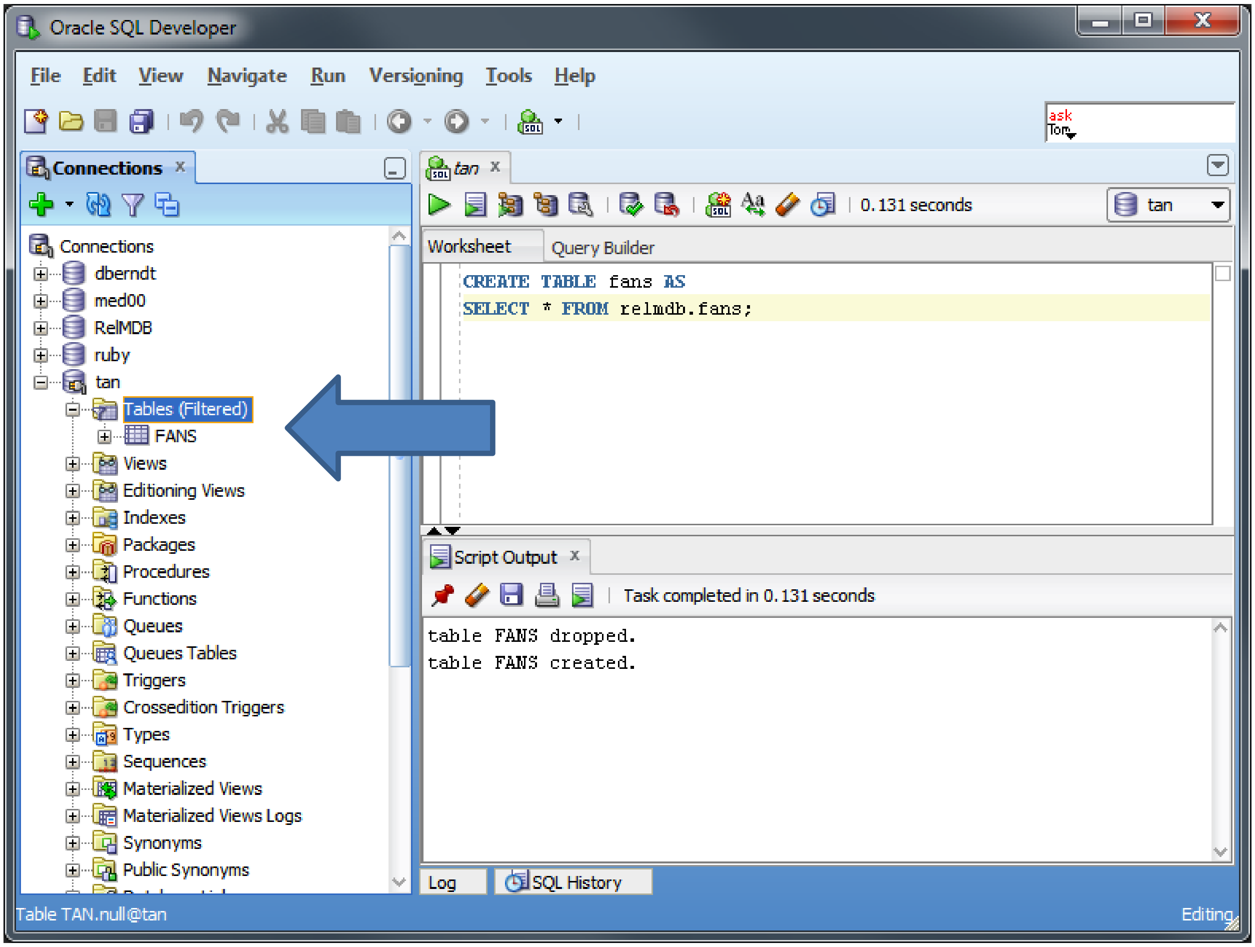
Assignment 1 “Cheat Sheet” for SQL Query Writing

Getting Started

This document is intended to help you complete the query writing assignment, especially if you a rusty or have not seen a lot of SQL before. Though, it also includes some basic features of the SQL Developer environment which all might find helpful.

First make a connection to the database using your user login (here I used TAN). All the folders you see below the user name contain database objects you own. So, these folders should be empty if you are using a fresh account. The CREATE TABLE AS SELECT (CTAS) statement shown here references the FANS table in the RELMDB schema and creates a copy here. Note: There is a FANS table in the folder after refreshing.



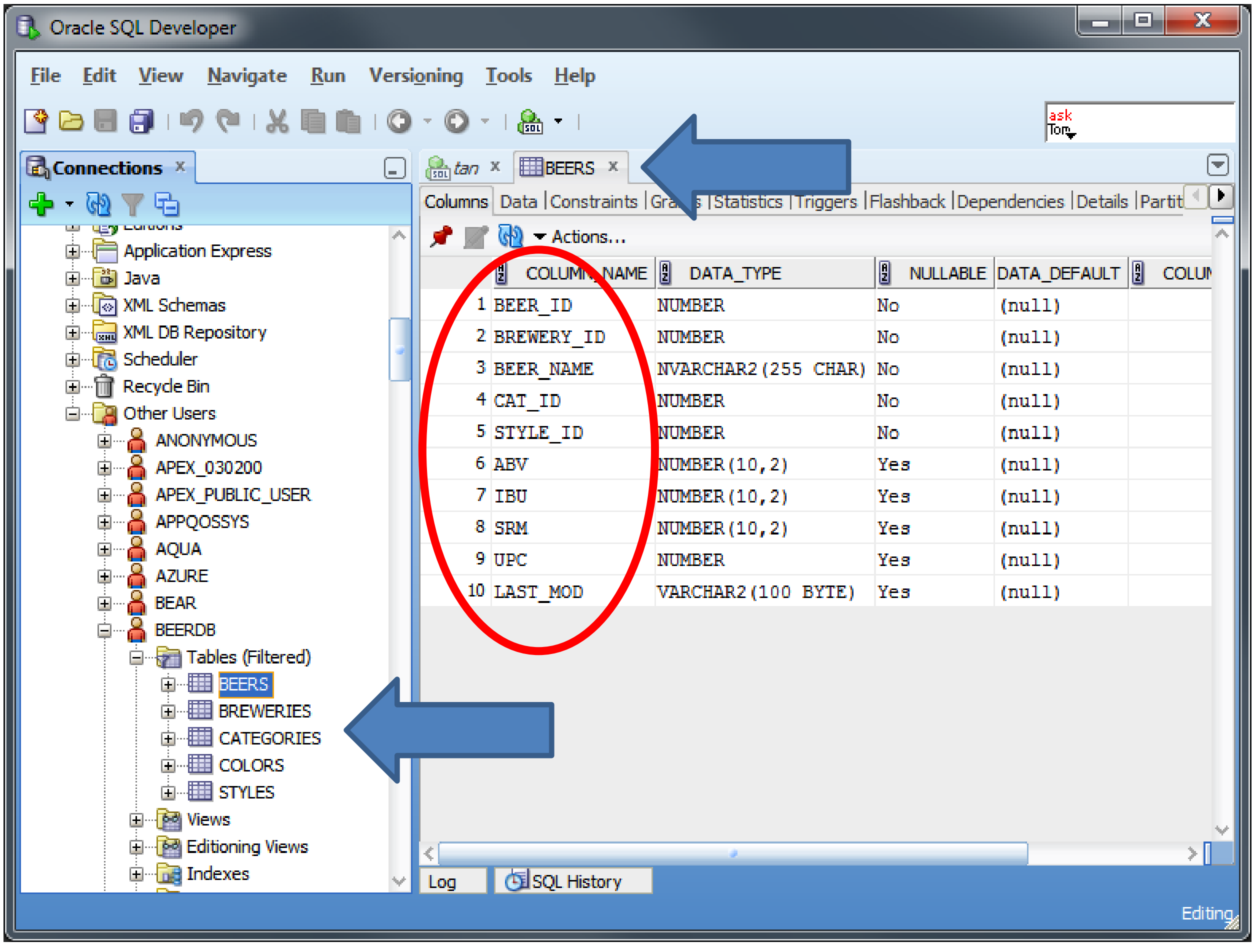
```
CREATE TABLE fans AS
SELECT * FROM relmdb.fans;
```

```
table FANS dropped.
table FANS created.
```

We really don’t need any tables of our own yet, since we can write queries against the movie database (RELMDB) and craft beer database (BEERDB).

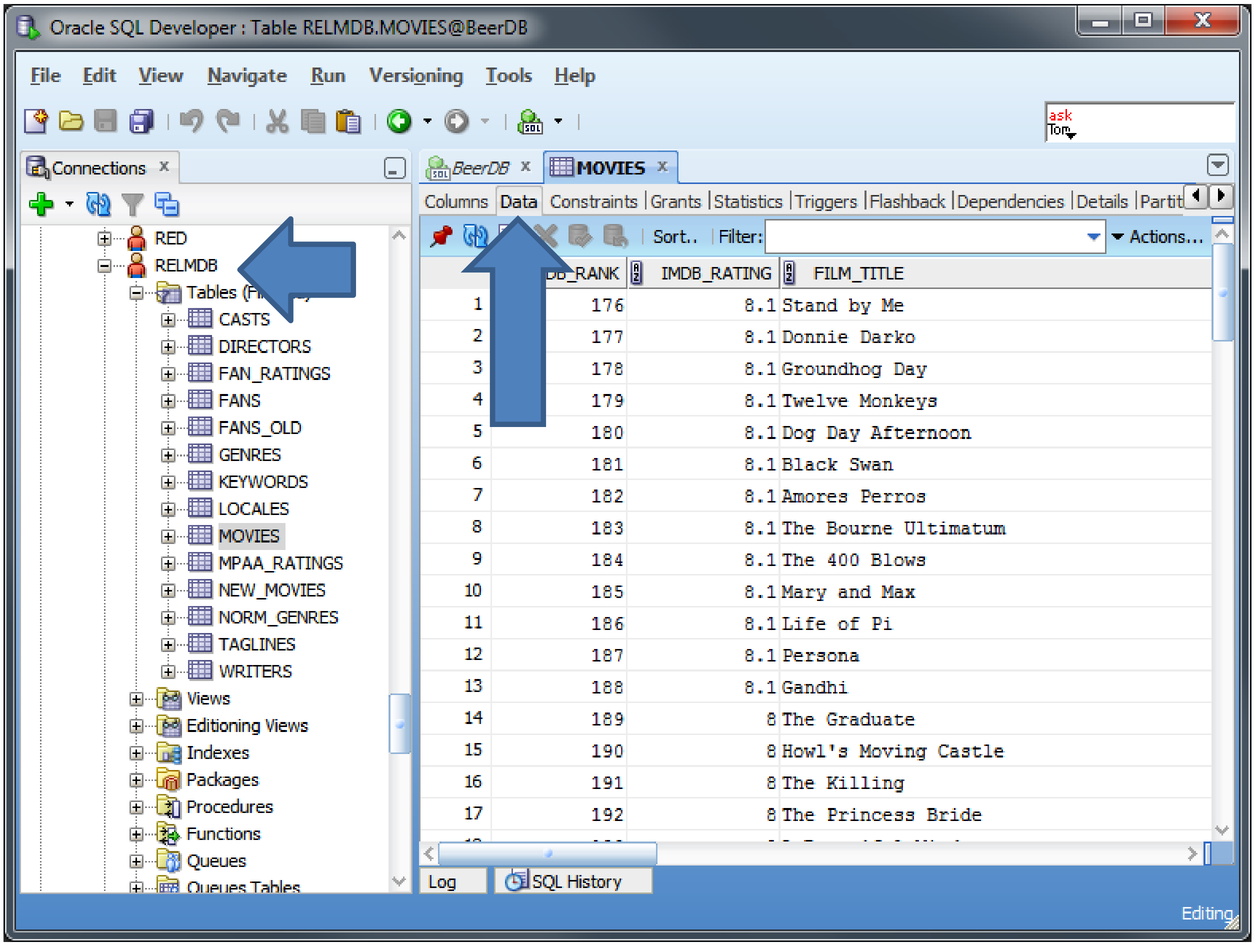
Where are the beer and movie databases?

The Oracle database instance includes many users and associated schemas. So, look for the “Other Users” folder and expand the list. You can browse any schema, but look specifically for the BEERDB and RELMDB users and expand those to see the tables. These are the tables you can query against by prefacing them with the schema name (such as BEERDB.BEERS). You can click on the table name to see the columns and other details about the table, as well as expand the table to see a column list. In this way, you can see what is available for query writing.



	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLU
1	BEER_ID	NUMBER	No	(null)	
2	BREWERY_ID	NUMBER	No	(null)	
3	BEER_NAME	NVARCHAR2(255 CHAR)	No	(null)	
4	CAT_ID	NUMBER	No	(null)	
5	STYLE_ID	NUMBER	No	(null)	
6	ABV	NUMBER(10,2)	Yes	(null)	
7	IBU	NUMBER(10,2)	Yes	(null)	
8	SRM	NUMBER(10,2)	Yes	(null)	
9	UPC	NUMBER	Yes	(null)	
10	LAST_MOD	VARCHAR2(100 BYTE)	Yes	(null)	

So, you can do the same with the RELMDB user to see the movie database. Again you can click on a table, such as the MOVIES table to see the details, including the actual data. This is a great way to get familiar with the data before writing any queries.



	DB_RANK	IMDB_RATING	FILM_TITLE
1	176	8.1	Stand by Me
2	177	8.1	Donnie Darko
3	178	8.1	Groundhog Day
4	179	8.1	Twelve Monkeys
5	180	8.1	Dog Day Afternoon
6	181	8.1	Black Swan
7	182	8.1	Amores Perros
8	183	8.1	The Bourne Ultimatum
9	184	8.1	The 400 Blows
10	185	8.1	Mary and Max
11	186	8.1	Life of Pi
12	187	8.1	Persona
13	188	8.1	Gandhi
14	189	8	The Graduate
15	190	8	Howl's Moving Castle
16	191	8	The Killing
17	192	8	The Princess Bride

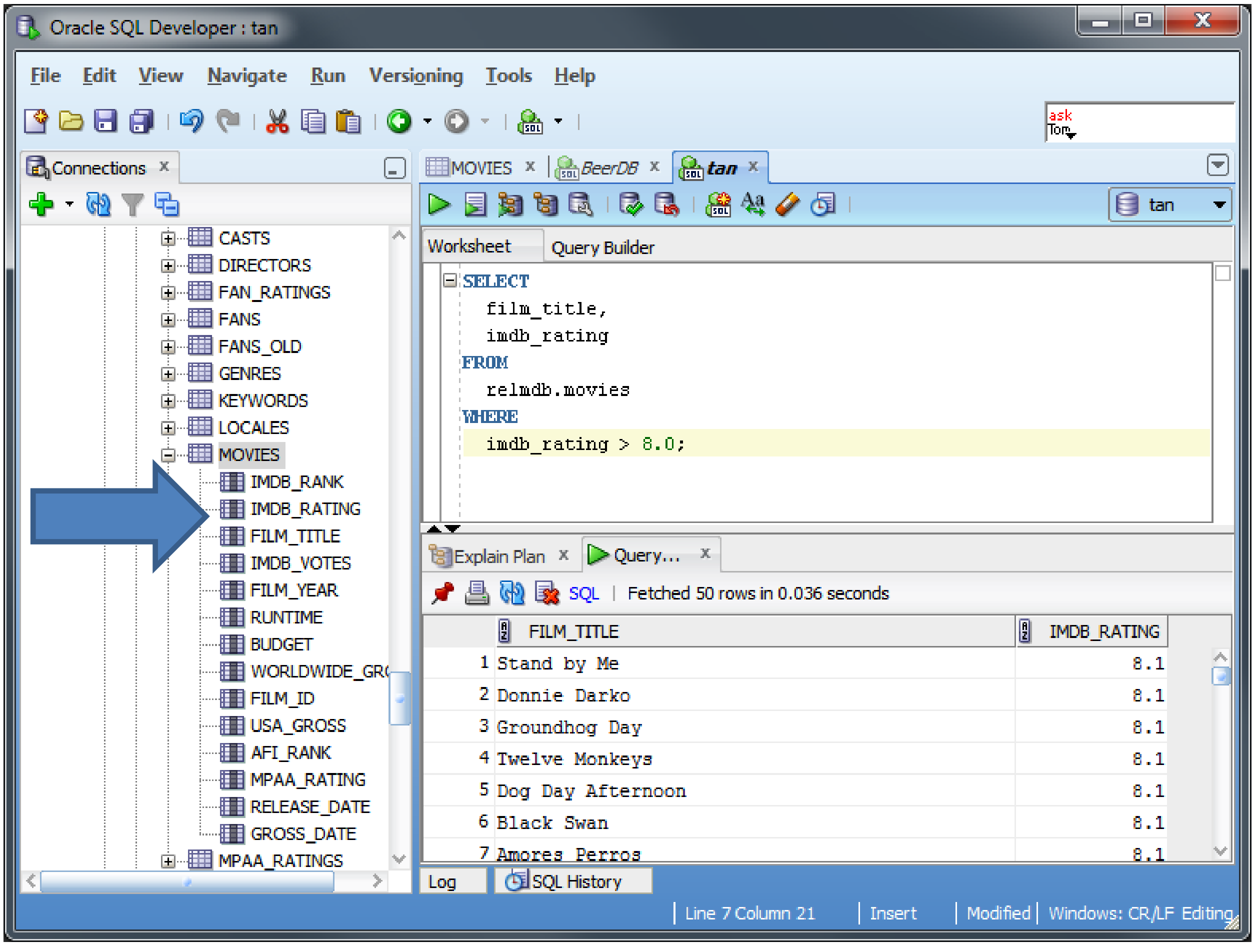
So, we are ready to write our first query for the assignment. Remember that as you type queries in the SQL worksheet, the tool is constantly checking the syntax (highlighting correct SQL keywords) and underlining problems. In addition, there will be suggestions for auto-completing parts of the queries.

Writing Our First Query

So, we can start with the first query against the movie database. The description of the query is as follows.

“Display the name of all movies that have an IMDB rating of at least 8.0, with more than 100,000 IMDB votes, and were released from 2007 to 2013. Show the movies with the highest IMDB ratings first.”

The best way to tackle SQL query writing is in steps, building and testing simpler components and then putting the whole query together. You can execute the query along the way to see if the results meet your expectations. So, we can start by building the basic SELECT-FROM-WHERE clause structure. The target of our query is the MOVIES table, so that is referenced in the FROM clause as RELMDB.MOVIES. We will want to see the FILM_TITILE and the IMDB_RATING since these are integral to the query. Next, add the WHERE clause to find only those movies with IMDB_RATING > 8.0. Go ahead and run the query and make sure that all is well. You can see the film titles and note that all the IMDB_RATING values are above 8.0; so far so good.

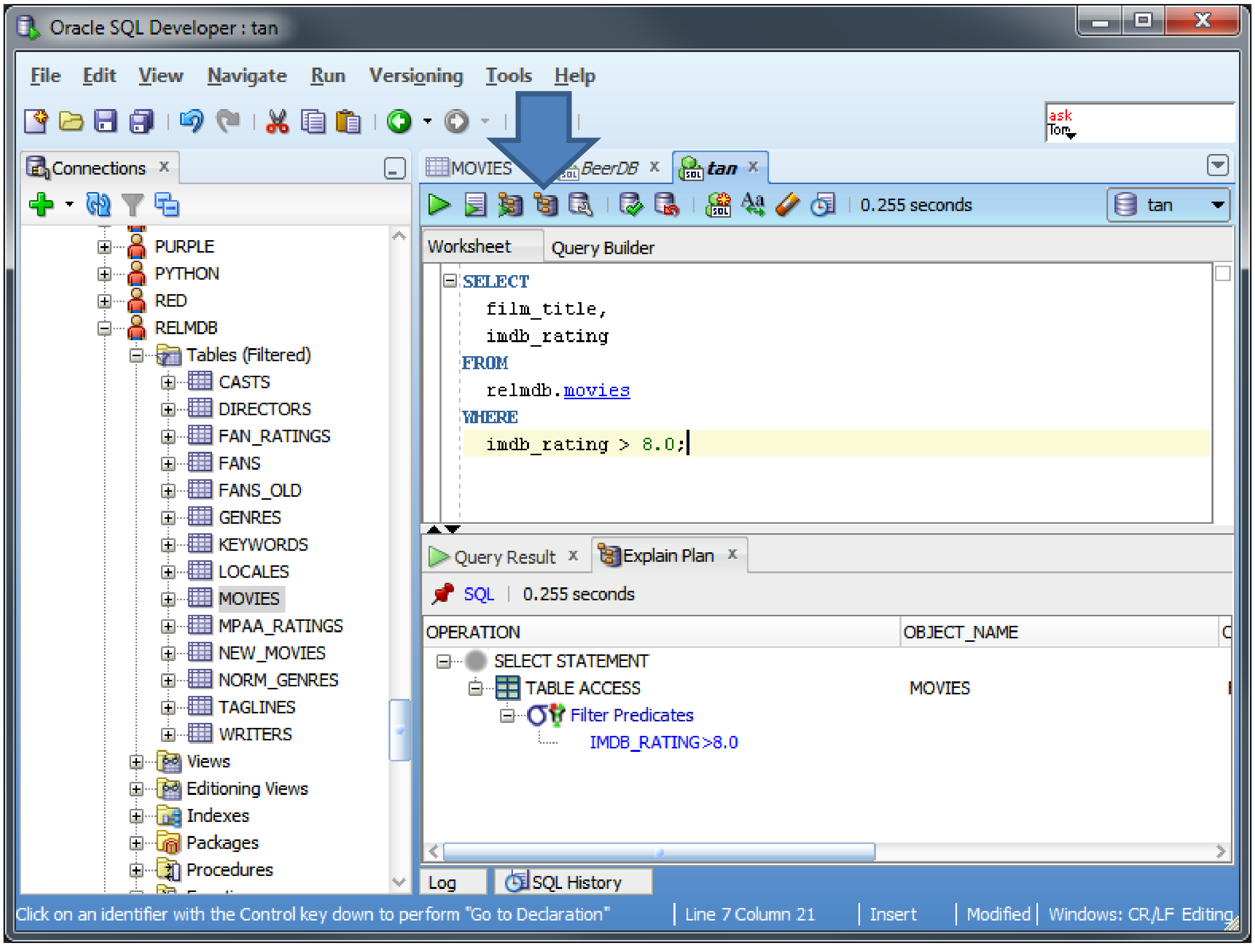


```
SELECT
  film_title,
  imdb_rating
FROM
  relmdb.movies
WHERE
  imdb_rating > 8.0;
```

Fetched 50 rows in 0.036 seconds

	FILM_TITLE	IMDB_RATING
1	Stand by Me	8.1
2	Donnie Darko	8.1
3	Groundhog Day	8.1
4	Twelve Monkeys	8.1
5	Dog Day Afternoon	8.1
6	Black Swan	8.1
7	Amores Perros	8.1

Remember that SQL is a declarative language. You are really just describing the result you would like to see. You are asking to see the FILM_TITLE and IMDB_RATING attributes for just those movies with an IMDB_RATING above 8.0. The database query optimizer figures out how to compute the desired result set using a reasonably efficient execution plan. The “Explain Plan” button will show you exactly how your query is being executed. Here, the execution plan notes that it is a SELECT and that it is accessing the table MOVIES (with a FULL scan). It also includes the filter predicate IMDB_RATING > 8.0. We will look at execution plans in more depth as the course progresses.

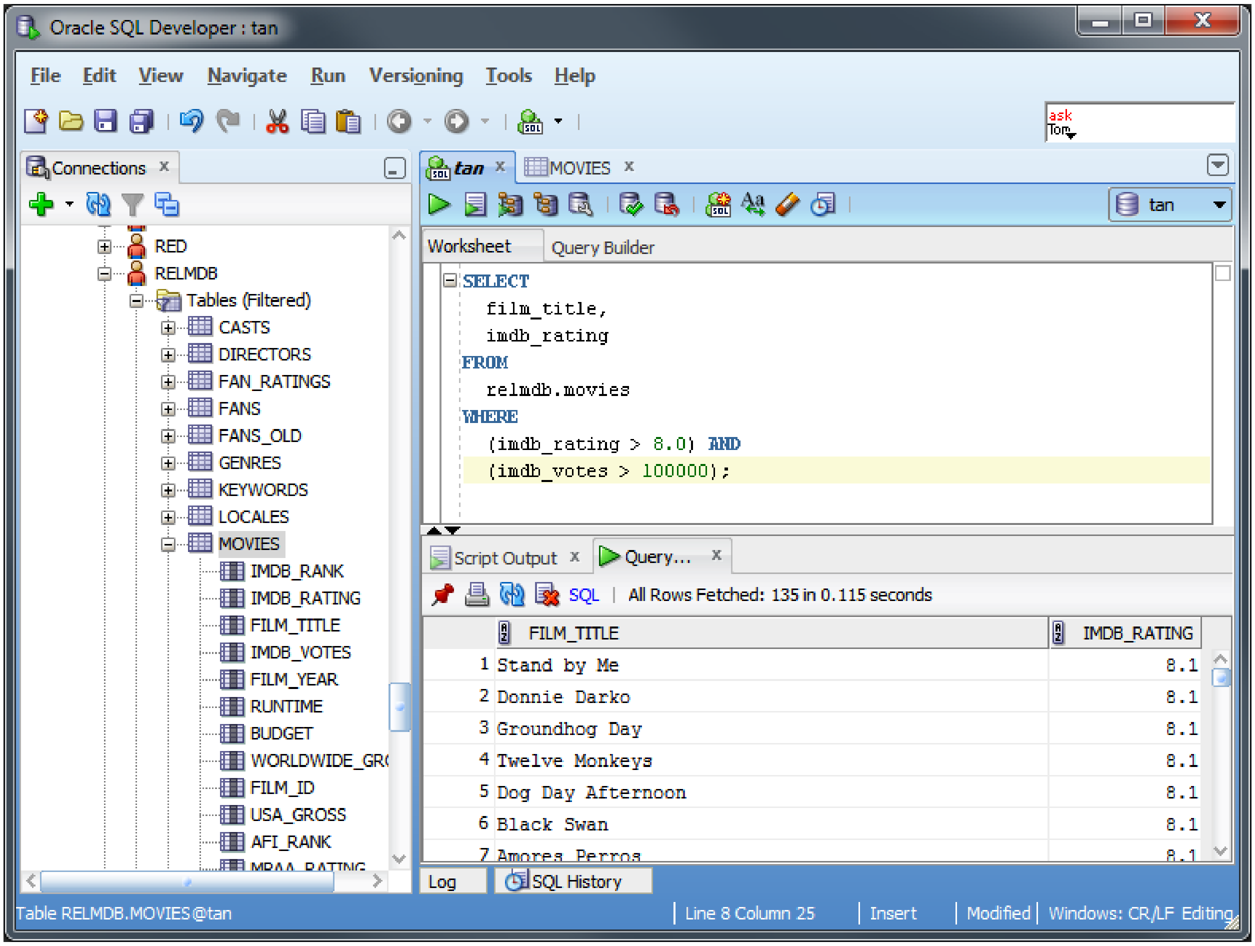


```
SELECT
  film_title,
  imdb_rating
FROM
  relmdb.movies
WHERE
  imdb_rating > 8.0;
```

```
OPERATION                          OBJECT_NAME
SELECT STATEMENT
  TABLE ACCESS                     MOVIES
    Filter Predicates
      IMDB_RATING>8.0
```

Click on an identifier with the Control key down to perform "Go to Declaration" | Line 7 Column 21 | Insert | Modified | Windows: CR/LF Editing

We can continue building the query by adding the second WHERE predicate. We want movies that also have over IMDB_VOTES > 100000. Note that we are capitalizing SQL keywords and writing table and column names in lower case. That is just a convention and many database programmers have other favorite style elements. Indentation and parentheses also help make the query easier to read. All the capitalization and whitespace is ignored by the database engine. Now our WHERE clause includes two predicates with AND requiring that both evaluate to TRUE. Go ahead and run the query again. The result set should be smaller since we are being more restrictive.



```
SELECT
  film_title,
  imdb_rating
FROM
  relmdb.movies
WHERE
  (imdb_rating > 8.0) AND
  (imdb_votes > 100000);
```

All Rows Fetched: 135 in 0.115 seconds

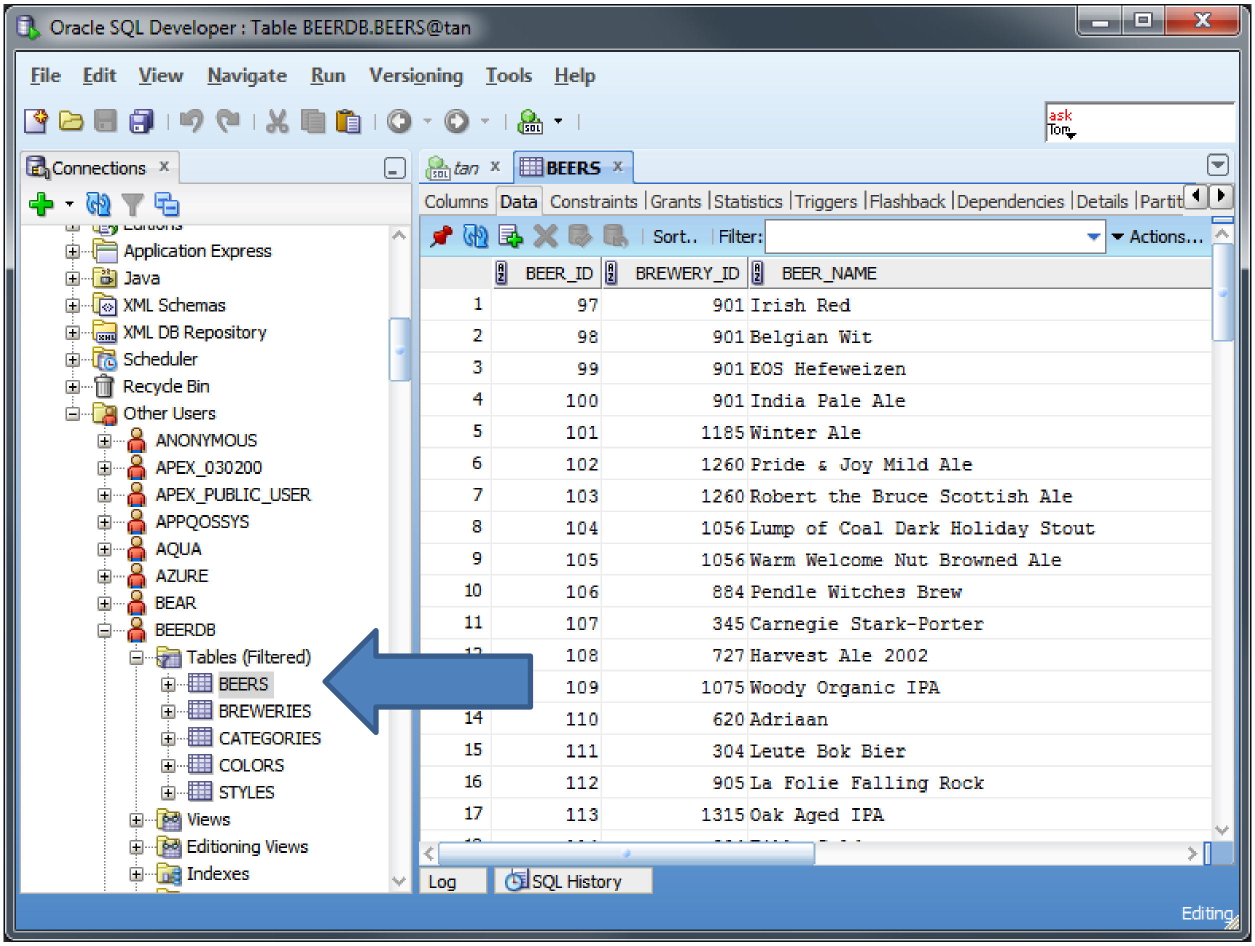
```
   FILM_TITLE              IMDB_RATING
 1 Stand by Me                     8.1
 2 Donnie Darko                    8.1
 3 Groundhog Day                   8.1
 4 Twelve Monkeys                  8.1
 5 Dog Day Afternoon               8.1
 6 Black Swan                      8.1
 7 Amores Perros                   8.1
```

Table RELMDB.MOVIES@tan | Line 8 Column 25 | Insert | Modified | Windows: CR/LF Editing

To finish up, the query also asks for those movies that were released between 2007 and 2013. How should we add the additional predicates to the WHERE clause? You could add something like (FILM_YEAR >= 2007) AND (FILM_YEAR <= 2013) or look at the BETWEEN operator that seems like a perfect fit. That last specification is to show the “highest IMDB ratings first,” which means we need to sort the result set. The ORDER BY clause provides that functionality. We will need to add ORDER BY IMDB_RATING DESC to the query, yielding a SELECT-FROM-WHERE-ORDER BY structure. The sort order can be specified as ascending (ASC - the default) or descending (DESC). Here we need to use DESC to see the highest values first. Go ahead and complete the query.

Writing Our Second Query

Let's try to write a second query, this time targeting the beer database (BEERDB). Again, looking at the main BEERS table, we can get a sense of the data. We can see that there is a BREWERY_ID column that would be useful for joining with the BREWERIES table. So, we can start here.

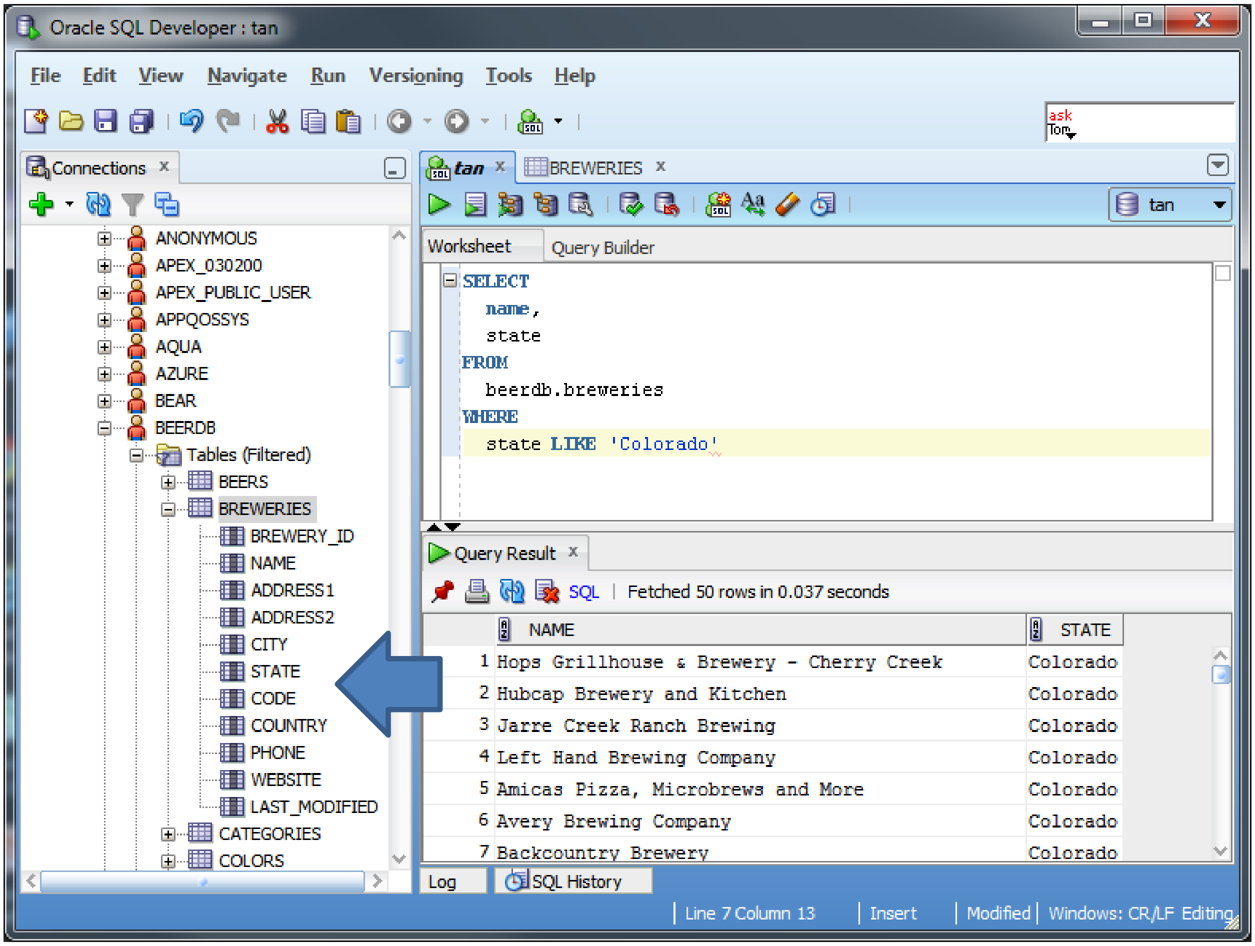


	BEER_ID	BREWERY_ID	BEER_NAME
1	97	901	Irish Red
2	98	901	Belgian Wit
3	99	901	EOS Hefeweizen
4	100	901	India Pale Ale
5	101	1185	Winter Ale
6	102	1260	Pride & Joy Mild Ale
7	103	1260	Robert the Bruce Scottish Ale
8	104	1056	Lump of Coal Dark Holiday Stout
9	105	1056	Warm Welcome Nut Browned Ale
10	106	884	Pendle Witches Brew
11	107	345	Carnegie Stark-Porter
	108	727	Harvest Ale 2002
	109	1075	Woody Organic IPA
14	110	620	Adriaan
15	111	304	Leute Bok Bier
16	112	905	La Folie Falling Rock
17	113	1315	Oak Aged IPA

First, we can try a simple single table query against the BREWERIES table. Find all the breweries in the state of Colorado. Later, we can join to the BEERs table to find out how many beers each brewery produces. So the query would be as follow (feel free to add this one to the assignment).

"List all the breweries in Colorado, along with the beers they produce. Sort the results by brewery, so the beers appear together in the result set."

Here is the SQL for the first step, along with the result set. Our goal is to simply list the breweries in Colorado. On the left, we can see the available columns for the BREWERIES table, including STATE. There are definitely some interesting breweries.

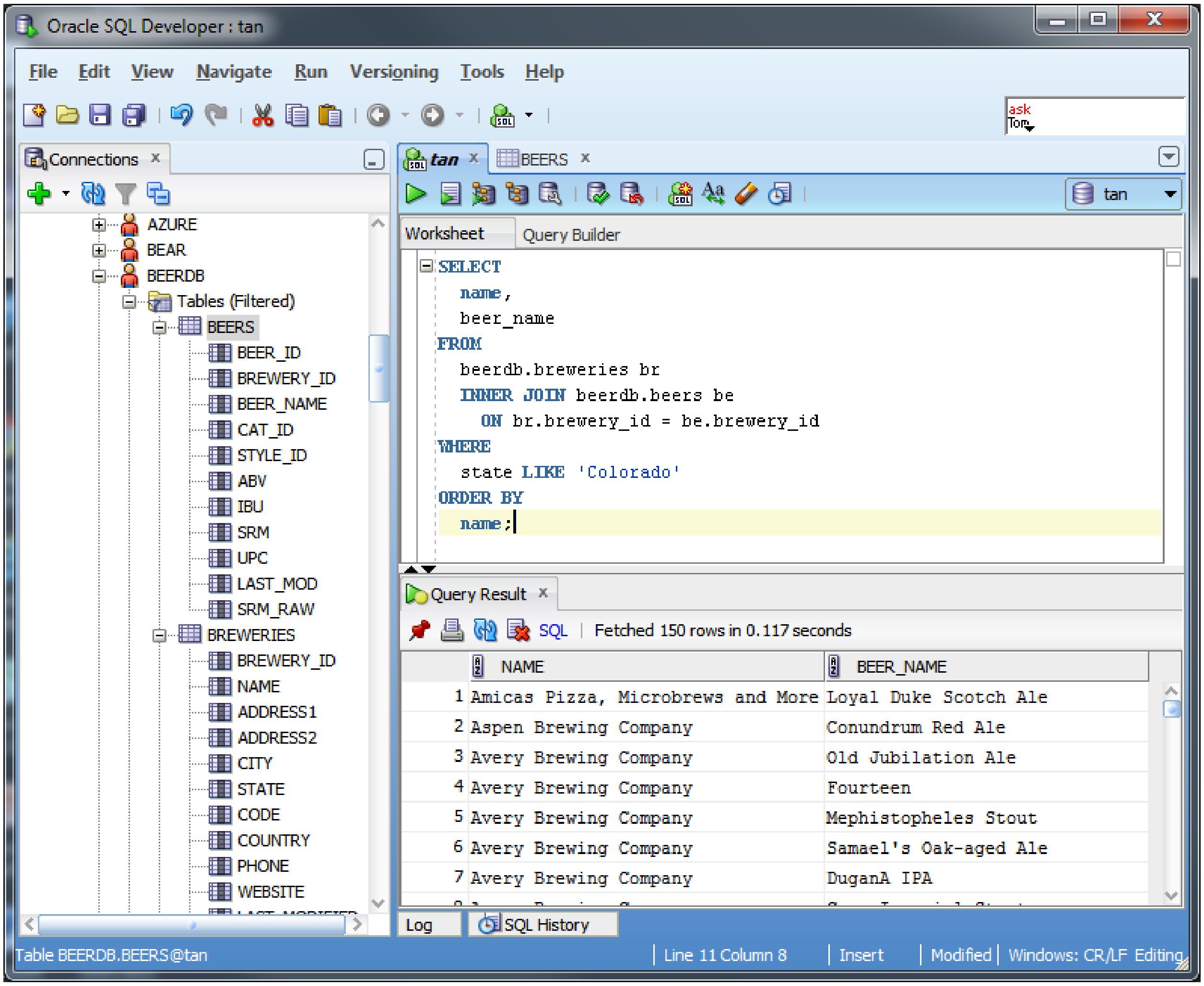


```
SELECT
  name,
  state
FROM
  beerdb.breweries
WHERE
  state LIKE 'Colorado'
```

Fetched 50 rows in 0.037 seconds

	NAME	STATE
1	Hops Grillhouse & Brewery - Cherry Creek	Colorado
2	Hubcap Brewery and Kitchen	Colorado
3	Jarre Creek Ranch Brewing	Colorado
4	Left Hand Brewing Company	Colorado
5	Amicas Pizza, Microbrews and More	Colorado
6	Avery Brewing Company	Colorado
7	Backcountry Brewery	Colorado

Again, we are building our query in steps. So, next we will need to join to the BEERS table to find all the beers produced. The BREWERY_ID column is the obvious choice for joining the two tables. It also makes sense to use a basic INNER JOIN, the most widely used method for combining tables.



```sql
SELECT
  name,
  beer_name
FROM
  beerdb.breweries br
  INNER JOIN beerdb.beers be
    ON br.brewery_id = be.brewery_id
WHERE
  state LIKE 'Colorado'
ORDER BY
  name;
```

Fetched 150 rows in 0.117 seconds

	NAME	BEER_NAME
1	Amicas Pizza, Microbrews and More	Loyal Duke Scotch Ale
2	Aspen Brewing Company	Conundrum Red Ale
3	Avery Brewing Company	Old Jubilation Ale
4	Avery Brewing Company	Fourteen
5	Avery Brewing Company	Mephistopheles Stout
6	Avery Brewing Company	Samael's Oak-aged Ale
7	Avery Brewing Company	DuganA IPA

So, the query now uses an INNER JOIN to combine BREWERIES and BEERS. Notice that table aliases ("br" and "be") are used to save some typing in the ON clause. This is called explicit join notation since INNER JOIN is specifically stated along with the join criteria in the ON clause. Nothing about the join is put into the WHERE clause. This is how you should write your joins.

Another interesting query could be built from here (please feel free to include this one in your assignment).

"List all the breweries in Colorado, along with the number of beers they produce. Sort the results in descending order by the number of beers, so the most prolific breweries appear first."

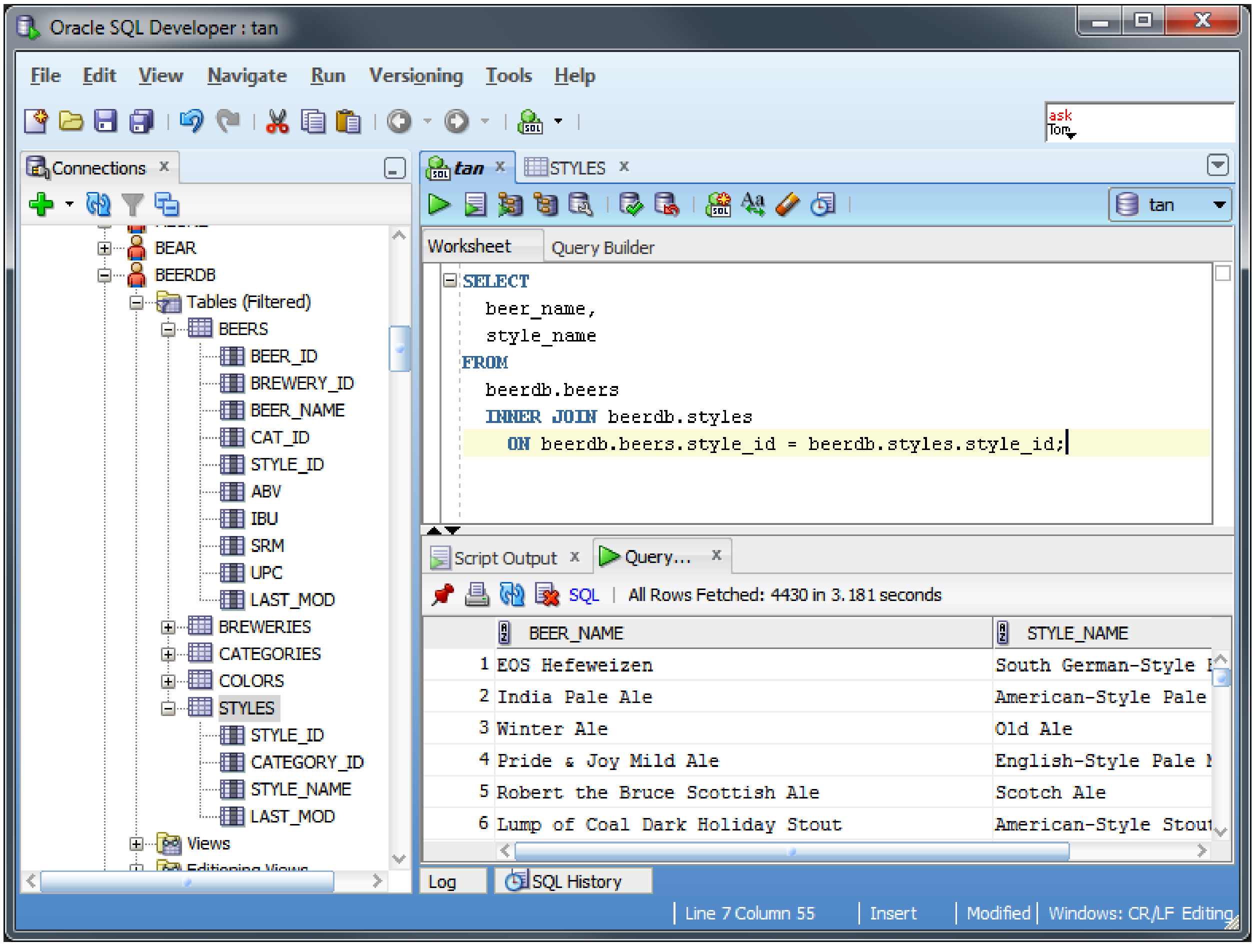
To write this query, you will need to replace the beer names with COUNT(beer_name) to get the number (not a list of the beers). An aggregation operator like COUNT is typically used with a GROUP BY clause. Here you would want to GROUP BY the brewery name. Try to complete this query and include it in your assignment!

Writing One More Beer Database Query

We can start on another query from the beer database to illustrate some more query writing skills. For example, the following query takes us further into different join types.

"Display each beer's name and style name. A beer should be displayed regardless of whether a style name exists or not."

We could start by simply joining BEERS with STYLES following the form outlined in the previous query. The query is fine and produces a reasonable answer set. However, the specification states that we should see a beer even if a style name does not exist. How do we handle this requirement?



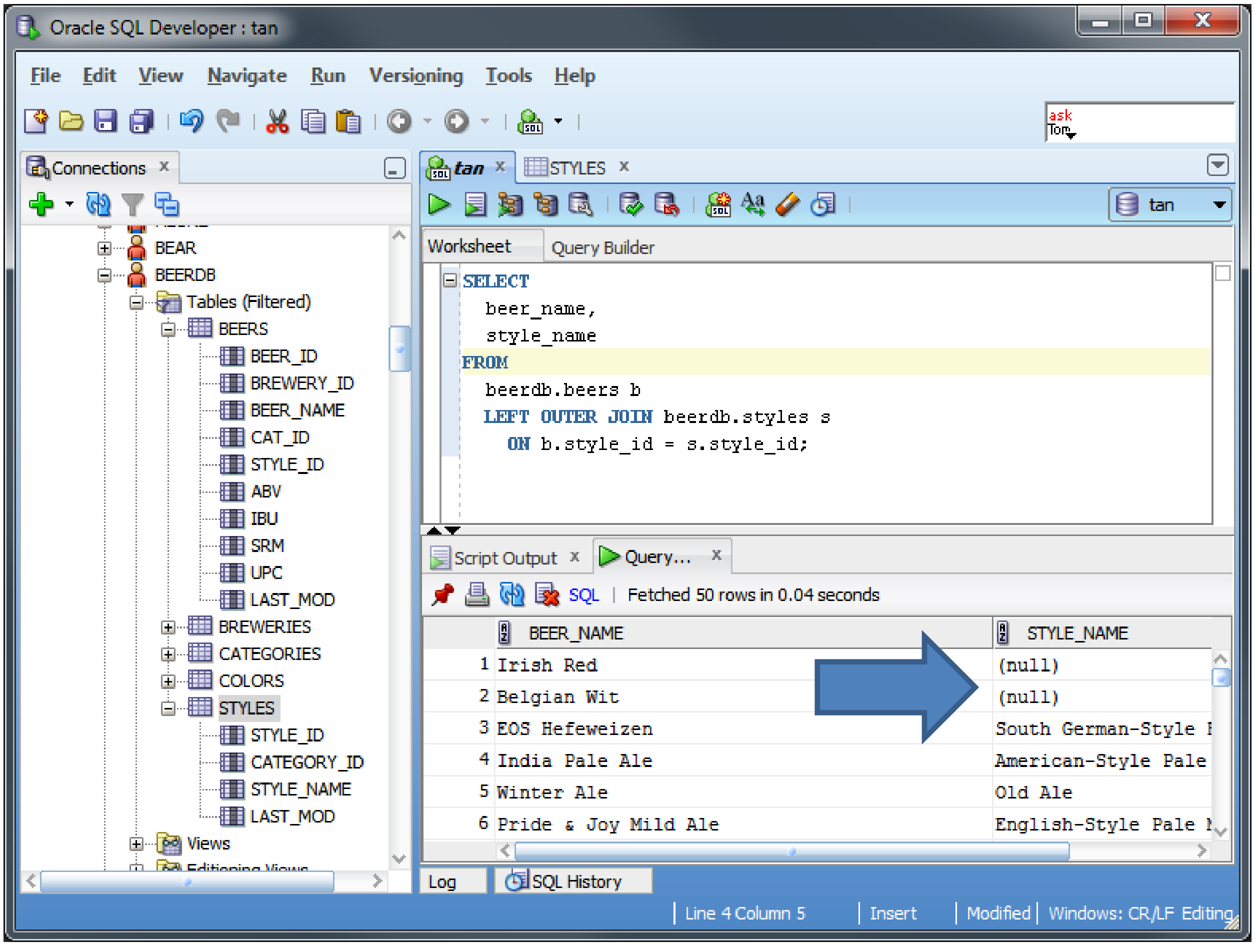
```
SELECT
  beer_name,
  style_name
FROM
  beerdb.beers
  INNER JOIN beerdb.styles
    ON beerdb.beers.style_id = beerdb.styles.style_id;
```

All Rows Fetched: 4430 in 3.181 seconds

	BEER_NAME	STYLE_NAME
1	EOS Hefeweizen	South German-Style H
2	India Pale Ale	American-Style Pale
3	Winter Ale	Old Ale
4	Pride & Joy Mild Ale	English-Style Pale M
5	Robert the Bruce Scottish Ale	Scotch Ale
6	Lump of Coal Dark Holiday Stout	American-Style Stout

There are different join types. The most commonly used type is the INNER JOIN, which means that the condition must be met in both tables. Here this means that there must be a STYLE_ID in the beers table and a corresponding entry in the STYLES table. The OUTER JOIN is used when you want to include entries from one table even when not paired with an entry from the other table. A LEFT OUTER JOIN (or just LEFT JOIN) includes rows from the left-hand side even if not paired with an entry from the other table. A RIGHT OUTER JOIN just comes at it from the other side. A FULL OUTER JOIN includes rows from either side, even if there is no paired row from the other table. So, which one do we want here?

We want to use the LEFT OUTER JOIN as shown below, so that all the beer names from the BEERS table (the left-hand side) show in the query results. You can see here that some of the style names have NULL values. These are the rows that contain a beer name, but no corresponding style name. This version meets the query specification to show beer names "regardless of whether a style name exists or not."



```
SELECT
  beer_name,
  style_name
FROM
  beerdb.beers b
  LEFT OUTER JOIN beerdb.styles s
    ON b.style_id = s.style_id;
```

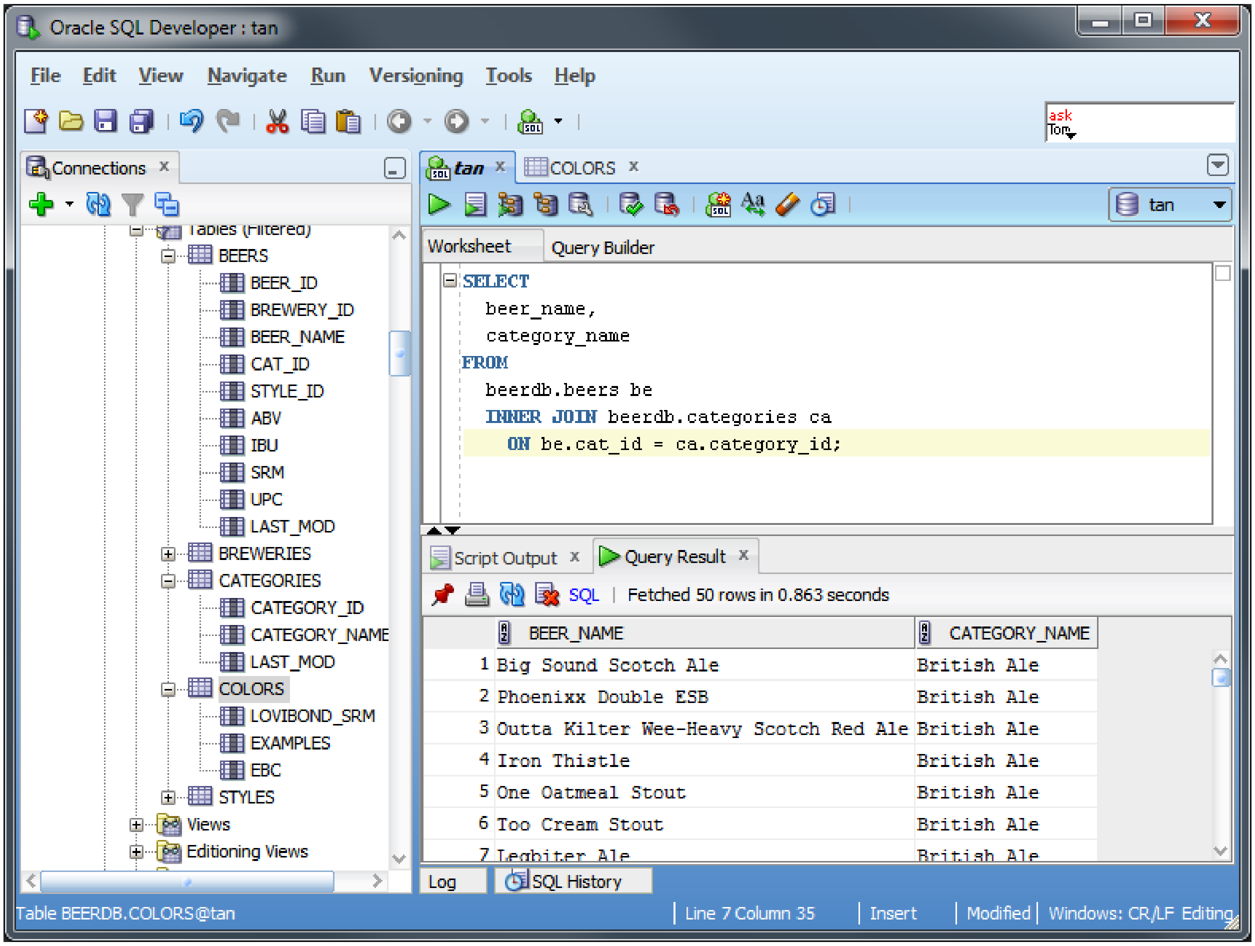
Fetched 50 rows in 0.04 seconds

	BEER_NAME	STYLE_NAME
1	Irish Red	(null)
2	Belgian Wit	(null)
3	EOS Hefeweizen	South German-Style I
4	India Pale Ale	American-Style Pale
5	Winter Ale	Old Ale
6	Pride & Joy Mild Ale	English-Style Pale M

Writing a Final Beer Database Query

We can tackle one final beer database query to demonstrate multiple joins. The query specification is as follows. Clearly, we need to combine BEERS with CATEGORIES and COLORS.

"Display each beer's name, category name, color example, and style name, for all beers that have values for category name, color example, and style name."



```
SELECT
  beer_name,
  category_name
FROM
  beerdb.beers be
  INNER JOIN beerdb.categories ca
    ON be.cat_id = ca.category_id;
```

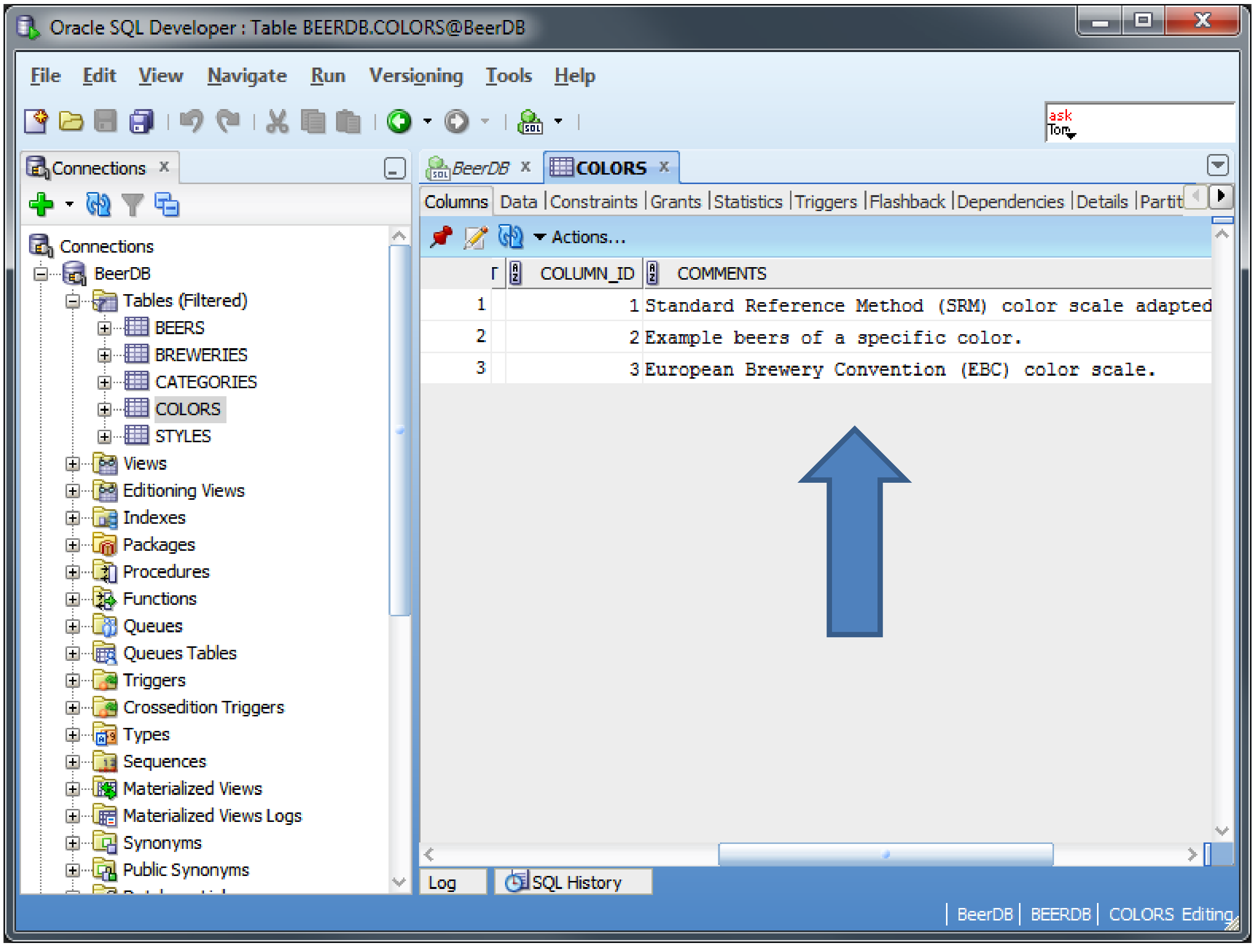
Fetched 50 rows in 0.863 seconds

	BEER_NAME	CATEGORY_NAME
1	Big Sound Scotch Ale	British Ale
2	Phoenixx Double ESB	British Ale
3	Outta Kilter Wee-Heavy Scotch Red Ale	British Ale
4	Iron Thistle	British Ale
5	One Oatmeal Stout	British Ale
6	Too Cream Stout	British Ale
7	Leghiter Ale	British Ale

We start by joining the BEERS table with the CATEGORIES table using an INNER JOIN on CATEGORY_ID. Note that the column names can differ, with CAT_ID in the BEERS table and CATEGORY_ID in the CATEGORIES table. That is fine as long as there really is a foreign key relationship between these tables and columns. Secondly, we again use table aliases ("be" and "ca") to save some typing and make the query easier to interpret.

Next we have to add another join to handle the COLORS table. Remember that a join takes two relations and creates a single output relation. So, we take the output of the first join and pair that with another table (COLORS).

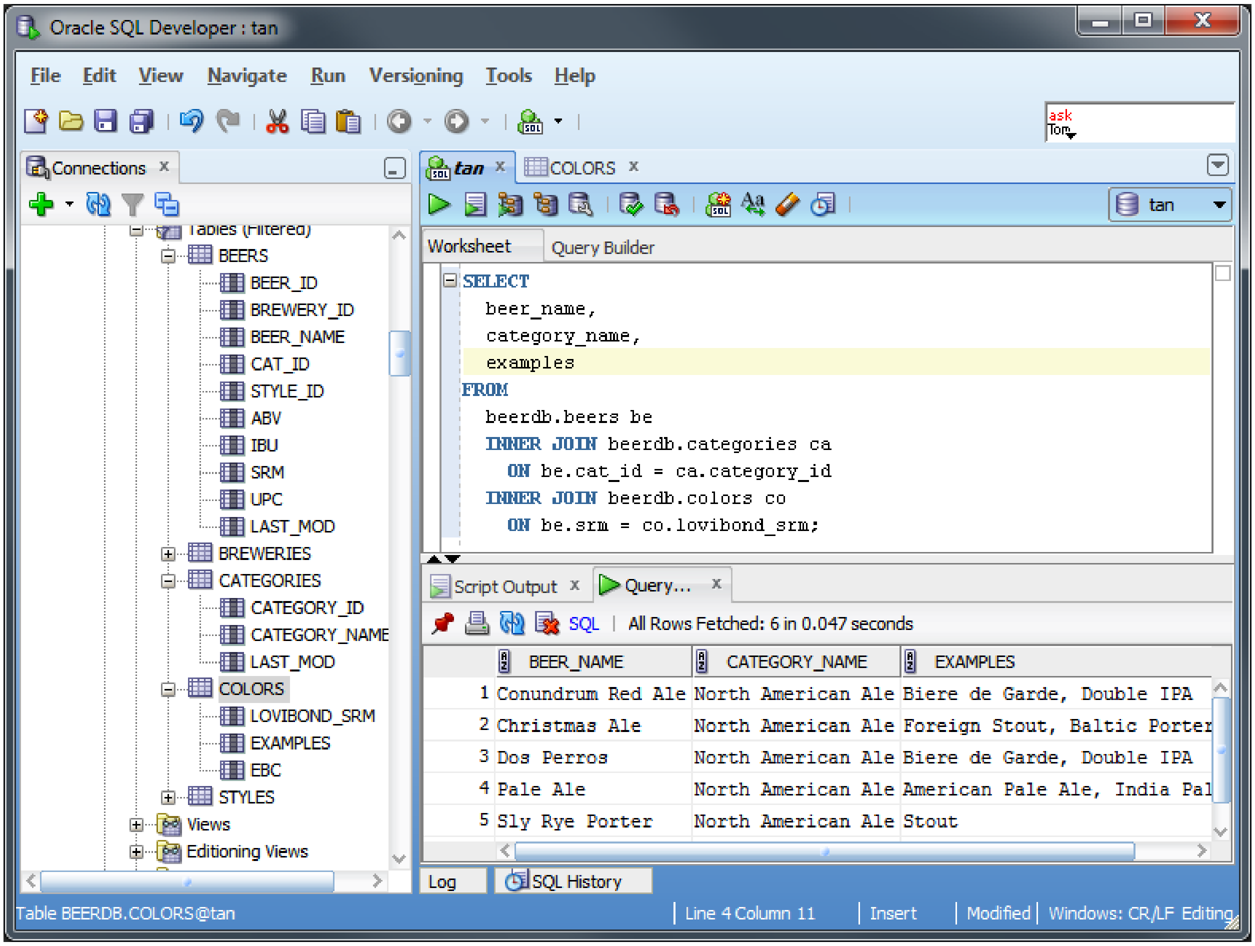
Before we proceed, we can explore the COLORS table a bit. Clicking on the COLORS table shows some details. If you scroll over on the "Columns" tab, you will see a COMMENTS column. This is metadata about the table. The Oracle DBMS provides a COMMENT statement for specifying table or column comments. This is a very nice way of documenting a database and helping future query writers. Here we can see that the SRM column refers to the Standard Reference Method (SRM) used to specify the color of beer. This also takes the mystery out of EBC, an alternate European Brewery Convention for beer color.



	COLUMN_ID	COMMENTS
1	1	Standard Reference Method (SRM) color scale adapted
2	2	Example beers of a specific color.
3	3	European Brewery Convention (EBC) color scale.

Now let's get back to writing the query. The next step is to add another join, bring the COLORS table into the mix. Again, we use an INNER JOIN.

This SQL code below shows the two joins (and three tables). Note that again the COLORS table is joined with columns that have slightly different names, SRM and LOVIBOND_SRM. Now we know why from the column comments above. The EXAMPLES column shows the types of beers that typify this color. This is a helpful way of describing beer color. Not many beers in the database have color examples, but the query is kind of nice.



```
SELECT
  beer_name,
  category_name,
  examples
FROM
  beerdb.beers be
  INNER JOIN beerdb.categories ca
    ON be.cat_id = ca.category_id
  INNER JOIN beerdb.colors co
    ON be.srm = co.lovibond_srm;
```

All Rows Fetched: 6 in 0.047 seconds

	BEER_NAME	CATEGORY_NAME	EXAMPLES
1	Conundrum Red Ale	North American Ale	Biere de Garde, Double IPA
2	Christmas Ale	North American Ale	Foreign Stout, Baltic Porter
3	Dos Perros	North American Ale	Biere de Garde, Double IPA
4	Pale Ale	North American Ale	American Pale Ale, India Pal
5	Sly Rye Porter	North American Ale	Stout

Table BEERDB.COLORS@tan | Line 4 Column 11 | Insert | Modified | Windows: CR/LF Editing

That's it for now. Hopefully, this "cheat sheet" helps you complete the query writing assignment and learn a little SQL along the way. Please feel free to ask further questions or suggest additions to this document as you work on other queries. Good luck.