# CIS 315 - Reading Packet: "Writing relational operations using SQL"

## SOURCES:

* Oracle9i Programming: A Primer, Rajshekhar Sunderraman, Addison Wesley.
* Classic Oracle example tables **empl** and **dept**, adapted somewhat over the years

## The basic SQL SELECT statement syntax and semantics

In lecture, we discussed the most important **relational operations**, from relational algebra. Today, we're going to discuss how these relational operations, and combinations of these relational operations, can be expressed in SQL.

In particular, we are going to be discussing Oracle SQL's **select** statement, which "provides a simple and powerful way of expressing **ad hoc** queries against the database." Really, it is the basic **query** statement in SQL. One can use it "to extract the specified data from the database and present it to the user in an easy-to-read format" (or in the form of a table, anyway).

Here's the confusing part: the relational operations are expressed in SQL using the SQL SELECT statement. What's confusing about that? Well, you should recall that the most important relational operations are selection, projection, equi-join, and natural-join (and that you have to understand Cartesian product to understand the equi-join and natural join, even though you rarely want Cartesian product by itself). You should **not** assume that the SQL SELECT statement is only for the relational selection operator! You use it to express selections **and** projections **and** equi-joins **and** natural joins, and even Cartesian products (although rarely intentionally!).

You need to become very comfortable expressing these relational operations, and combinations of these operations, using the SQL SELECT statement.

(Note: remember that SQL is NOT case-sensitive; it does not matter if you type `SELECT` or `select` or `Select`, or even `sElEcT` (although that would be hard to read!). In SQL, case only matters within string literals -- `'Hi'` is *not* equal to `'hi'`. You'll find that I tend to type SQL in lowercase, although sometimes I might write some keywords in uppercase for emphasis. I don't mind what case you use, as long as you are consistent about it within a given script.)

We're going to find out that the SQL SELECT statement has a number of optional clauses. Ignoring most of those optional clauses for the moment, here is the **basic SQL SELECT statement syntax** (where < > and [ ] are **not** part of the syntax, but < > is used to describe parts the user chooses, and [ ] is used to indicate optional parts):

```
select [distinct] <one or more expressions, separated by commas>
from <one or more expressions representing tables, separated by commas>
[where <search-condition>];
```

So, a SQL SELECT must always have a **select-clause** and a **from-clause**; optionally, it may have a **where-clause** (and it frequently does). It may also have additional optional clauses that we will discuss later. SQL*Plus does not care how many lines this is written across (although blank lines within a SELECT should be avoided, as discussed in last week's lab!). However, it will be a **course style standard** that the select-clause, the from-clause, and the where-clause will start on **separate** lines.

Here are the **semantics** of the SQL SELECT statement: conceptually (although the algorithm may be much more efficient in reality):

1. the **Cartesian product** of the tables listed in the **from-clause** is computed;
2. a relational **selection** of this Cartesian product is computed, selecting those rows for which the **where-clause** search condition is true;
3. a relational **projection** of #2's selection is computed, projecting only the expressions (often column names) from the **select-clause**.

A table results from this, although that table is not saved, and it may not always be a true relation, because, for efficiency reasons, Oracle does not always perform the final step of removing any duplicate rows in the tabular result. It only removes any duplicate rows from the result if the optional keyword **DISTINCT** is included in the **select-clause** as shown.

Understanding these semantics will help you see how the SQL SELECT statement can be used to specify desired combinations of relational operations.

Remember the very simple **select** statement we used last week?

```
select *
from   <tablename>;
```

Now we can see that this (1) computes the Cartesian product of the tables in the from-clause -- but as there is just one table in that clause, the result is just the rows of that table. Then, (2) there is no where-clause, so all of those rows are selected. Finally, (3) * in the select-clause is a shorthand meaning all of the columns in all of the tables in the from-clause, and so all of the columns in that table are projected to result in the final table result. Thus we see all of the columns of all of the rows of <tablename> as a result of this **select** statement.

## Interlude: some example tables, and a few words on foreign keys, other table constraints, and inserts

Before we continue with today's examples, we need to set up some example tables. Along with this lab you will find a link to a SQL script **set_up_ex_tbls.sql**, which sets up and populates three tables, **empl**, **dept**, and **customer**. You can create a file **set_up_ex_tbls.sql** on nrs-labs, paste in this posted link's contents (omitting the "Please send questions to..." line and straight line after that...!), and save your resulting SQL script file. Alternately, I've placed a copy of this script file at **~st10/315lab04/set_up_ex_tbls.sql**, and so you should be able to make your own copy in your current directory with the UNIX command: (note the space and period at the end, they are important!!)

```
cp  ~st10/315lab04/set_up_ex_tbls.sql .
```

Once you have the **set_up_ex_tbls.sql** SQL script, you should execute it within **sqlplus** to set up and populate these tables on your Oracle account.

Let's look at this script for a moment, however, as it happens to include some features not discussed in last week's introduction. You can either look at the posted version, or open the file using **pico** or **emacs** or **vi**, or you can even look at it on-screen under UNIX by using

```
     more set_up_ex_tbls.sql
```

Consider the **drop table** statements -- these now include the clause **cascade constraints**. This clause means to drop this table even if it is a "parent" table, a table referenced by foreign keys in other tables. (A table with such a foreign key is said to be a "child" table of this "parent" table that its foreign key references.) In Oracle, a table has to already exist before another table can specify a foreign key referencing that table; thus, "parent" tables must be created before "children" tables are created. But "parents" cannot be dropped if their "child" tables still exist -- "child" tables have to be dropped first. Since many programmers like to "pair" their drop-table and create-table statements within a script setting up a set of tables, **cascade constraints** make this possible. It should be used with some care, but if a script is going to completely recreate all of the tables in a collection, it should be safe to use it in this way.

We are using an additional **constraint**, **NOT NULL**, in the column definitions for **dept_name** and **dept_loc** in the **dept** table. This is asking the DBMS to ensure that rows inserted into **dept** must include values for these columns -- these columns should never be allowed to contain the special **NULL** value. That is, dept_name and dept_loc should not be permitted to be empty.

We discussed foreign keys in last week's lecture; you see the SQL syntax for specifying a foreign key in several of these create-table statements:

```
           foreign key (col1, col2, ...) references tbl
       or
           foreign key (col1, col2, ...) references tbl(diffname1, diffname2, ...)
```

This is actually another constraint on the table being created: it is saying that the column or columns specified are foreign keys referencing the specified table. If these columns in the referenced table have exactly the same names as in this table, you can use the first version above. Otherwise, you must use the second version.

By the way -- note **empl**'s foreign key **mgr**. It is indeed a foreign key referencing the **empl** table itself!

Also note the variation on the **insert** statement used for inserting rows into the **empl** table - this is the version you use if you only want to explicitly fill some of the columns in a new row, or if you want to specify the column values in an order different than their order in the table's create table statement. After the table name, you include a parenthesized list of what columns' values are to be specified and in what order, and after **values** you include a parenthesized list of exactly the values for those columns, in that order. What happens to the unspecified columns in the new row? They will either be NULL, or if the create-table statement specifies a default value for that column, that column will contain that default value.

So,

```
insert into empl(empl_num, empl_last_name, job_title, mgr, hiredate,
                 salary, dept_num)
values
('7934', 'Miller', 'Clerk', '7782', '23-Jan-1992', 1300.00, '100');
```

...is saying to insert a new row into empl with these values for empl_num, empl_last_name, job_title, mgr, hiredate, salary, and dept_num. Since column commission is not in that list, its value will be NULL for the new row.

Are you curious how you specify a default value for a column? You can see that in customer's cust_balance column:

```
 cust_balance  number(7, 2)   default 0.0,
```

...you include the constraint **default** followed by the desired default value.

## Using SQL SELECT for the classic relational operations

### *Relational PROJECTION with a SQL SELECT statement*

You can use SQL SELECT as follows to specify a "pure" relational projection:

```
select distinct <columns-to-project-separated-by-commas>
from   <tbl>;
```

Here are some example "pure" relational projections:

The relational projection of the empl_last_name, salary, and hiredate columns of the empl table:

```
select distinct empl_last_name, salary, hiredate
from   empl;
```

The relational projection of the job_title column of the empl table:

```
select distinct job_title
from   empl;
```

Notice that you can project the desired columns in any order that you like, and you will see the values of these columns for all of the rows in the table. (But, because of the **distinct**, you will get a true relation as the result: any duplicate rows in the result will be removed.)

What happens if you omit the **distinct**? Then you may get ALMOST a "true" relational projection -- any duplicate rows will remain in the resulting table. Depending on what you are asking and why, sometimes you might want duplicate rows (even if that isn't a true relation), and SQL gives you the option, then. It is also a bit more efficient, since the DBMS doesn't have to do the work of checking for duplicate rows before displaying the result. You should use **distinct** when you know duplicate rows might occur and you don't want them.

Compare the results of:

```
select distinct job_title, dept_num
from   empl;

select job_title, dept_num
from   empl;
```

Be sure you understand why the results differ.

One more thought: if you are projecting (all of) a table's primary key, is it possible for there to be duplicate rows in the result? No, because primary keys are not permitted to be the same in any two rows. So there really isn't any need to use **distinct** if you are projecting a table's primary key, as the result is guaranteed to be the "true" relational projection without it.

## *Relational SELECTION with a SQL SELECT statement*

You can use SQL SELECT as follows to specify a relational selection:

```
select *
from   <tbl>
where  <condition-specifying-rows-to-select>;
```

Here are some example relational selections:

The relational selection of the rows of the empl table where job_title = 'Manager':

```
select *
from   empl
where  job_title = 'Manager';
```

SQL actually provides a rich set of ways specifying which rows to select -- for now, note that you can use =, as above, to specify that you want rows where a particular column's value is equal to the specified value. You can also use <, <=, >, >=, to indicate that you are interested in rows where a column's value is compared in these ways to some value, and there are two ways to indicate that you are interested in rows in which a column is not equal to some value: <> and **!= .**

Quick question: what rows do you think will result from the query:

```
select *
from   empl
where  job_title = 'manager';
```

Try it -- you'll see the **no** rows result. (The empty table is a relation, too!) This is because the only place that SQL is case-sensitive is within string literals: so `'Manager'` is not equal to `'manager'`.

The relational selection of the rows of the empl table in which the hiredate is after June 1, 1991:

```
select *
from   empl
where  hiredate > '01-JUN-1991';
```

### *Relational EQUI-JOIN with a SQL SELECT statement*

You can use SQL SELECT as follows to specify an equi-join:

```
select *
from   <tbl1, tbl2>
where  <tbl1.join-col = tbl2.join-col>;
```

Here is an example equi-join:

The equi-join of the tables empl and dept using the join condition empl.dept_num = dept.dept_num:

```
select *
from   empl, dept
where  empl.dept_num = dept.dept_num;
```

Consider the semantics of the basic SQL select statement -- they are exactly the steps we described for what an equi-join means: (conceptually) compute the Cartesian product of the two tables in the from-clause, select only those rows in that Cartesian product satisfying the join condition, and project all of the columns in the result.

### *Relational CARTESIAN PRODUCT with a SQL SELECT statement*

It is rare that you actually want a Cartesian product of tables, but it is a very common error to ask for one when you do not intend to. Consider what you get if you leave off the join condition in an attempted equi-join:

```
select *
from   <tbl1, tbl2>;
```

...where you really intended:

```
select *
from   <tbl1, tbl2>
where  <tbl1.join-col = tbl2.join-col>;
```

According to the basic SQL select statement semantics, that first statement above determines the Cartesian product of tbl1 and tbl2 -- and since there is no join condition, all of the rows of the Cartesian product are selected, and all of its columns projected. Thus, the result is just the Cartesian product of those two tables -- for a tbl1 with m rows and a tbl2 with n rows, all m*n rows of that Cartesian product!

If you are looking at an equi-join or natural join result, and realize there are way too many rows in it, the first thing you should suspect is an inadvertent Cartesian product, and you should check if you have left out the necessary join condition!

Quick note, before we go on: you know that computers do not handle ambiguity well. That applies to SQL as well. No two columns within the Cartesian product of the from-clause of a SQL select statement can have the same name. This isn't a problem, however, because columns in two tables that otherwise would have the same name are really considered to have the name <tbl-name>.<column-name>. So, when a from-clause has:

```
from empl, dept
```

...dept's dept_num column has the name `dept.dept_num`, empl's dept_num column has the name `empl.dept_num`, and there is no ambiguity.

However, when you are specifying columns in the select-clause or the where-clause, you must give unambiguous column names as well. So, if a column name appears in more than one table in the from-clause, you must precede that column name by the table name and a *period everywhere* else within that select statement -- as `empl.dept_num` or `dept.dept_num` rather than simply as `dept_num` .

***Relational NATURAL JOIN with a SQL SELECT statement***

There is no short-cut to easily get a natural join using a SQL select statement -- it is like the equi-join, but you have to exlicitly project in the select-clause all of the columns except the "duplicate" one you'd like to omit.

```
select <every-column-except-the-duplicate-column-you'd-like-to-omit>
from   <tbl1, tbl2>
where  <tbl1.join-col = tbl2.join-col>;
```

Here is an example natural join:

The natural join of the tables empl and dept using the join condition empl.dept_num = dept.dept_num:

```
select empl_num, empl_last_name, job_title, mgr, hiredate,
       salary, commission, empl.dept_num, dept_name, dept_loc
from   empl, dept
where  empl.dept_num = dept.dept_num;
```

It of course does not matter whether you choose to project the empl.dept_num column or the dept.dept_num column, as long as you only project one of them -- the result is still considered the natural join of these two tables on that join condition.

## COMBINATIONS of RELATIONAL OPERATIONS using a SQL SELECT statement

We often perform combinations of relational operations using a SQL select statement; we are not limited just to the individual relational operations we have just demonstrated. (The point was that you *can* use SQL select to specify each "pure" basic relational operation, not that you are limited to those.) SQL select makes such combinations very reasonable, especially once you are comfortable with its semantics that we described earlier.

Note that SQL has an **AND** operation, for logical and, an **OR** operation, for logical or, and a **NOT** operation, for logical not. This can be used to build very sophisticated where-clauses, where you can select a finely-requested choice of rows from the SQL select's Cartesian product, including selecting just some of the rows from an equi-join or natural join, if you use AND to select rows that meet the join condition AND some other condition.

You can decide to project just some of the columns from some selection or some equi-join or some natural join. So, in practice, you join only the tables you want (if you include an appropriate join condition...!), select only the rows you want from that join, project only the columns you want from that selection.

```
SELECT     <desired expressions to project>
FROM       <tbl, or tbls to be joined>
[WHERE     <join-condition-if-a-join>
            AND <condition to specify desired rows from join>];
```

...although, oddly enough, it is more common to indent this as:

```
SELECT     <desired expressions to project>
FROM       <tbl, or tbls to be joined>
[WHERE     <join-condition-if-a-join>
AND        <condition to specify desired rows from join>];
```

I'll accept either of the above indentation styles for SQL select statements.

And the where-clause is not limited to this; it can be any condition, involving as many AND's and OR's and NOT's associated with conditions as you want.

So, for example, what if you would like to project just the job_title and hiredate of empl rows whose commission is more than 0? This SQL select statement will do so:

```
select job_title, hiredate
from   empl
where  commission > 0;
```

(Note that this may produce duplicate rows -- there is no **distinct** in the select-clause to prevent them.)

And what if you'd like to project just the empl_last_name, dept_name, and dept_loc from the selection of rows from the equi-join of empl and dept on the join condition empl.dept_num = dept.dept_num for which the hiredate is later than December 1, 1991?

```
select empl_last_name, dept_name, dept_loc
from   empl, dept
where  empl.dept_num = dept.dept_num
and    hiredate > '01-DEC-1991';
```

Do you see that I could have described the above as being a further projection of either the equi-join or the natural join of empl and dept with that join condition? It is projecting just some of the columns from either one, after all. So, in practice many people just say they are projecting just certain columns from the **join** of these tables, in this case.