CS 325 - SQL Reading Packet 2: "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.
- Sharon Tuttle, “Database Design”, Lecture readings, Cal Poly Humboldt.

The basic SQL `select` statement syntax and semantics

In the reading packet introducing the relational model, we discussed the most important **relational operations**, from relational algebra. In this packet, 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, allowing you to ask questions about the data in a database. 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 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 <1 or more expressions representing relations, sep'd by commas>
[where <search-condition>];
```

We'll call the part of this consisting of the keyword `select` and the expressions following it the **select clause**, the part of this consisting of the keyword `from` and the expressions following it the **from**

clause, and the (optional) part of this consisting of the keyword `where` and the search condition following it the **where clause**.

So, a SQL `select` statement 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 a SQL `select` statement is written across, although blank lines within a `select` statement should be avoided since SQL*Plus considers a blank line to mean that a SQL statement is finished! 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 relations listed in the `from` clause is computed;
2. a relational selection of this Cartesian product is computed, selecting only those rows for which the `where` clause search condition is true;
3. a not-necessarily-"pure" 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, a DBMS does not always perform a "pure" relational projection -- it does not always perform the final step of removing any duplicate rows in the result. It only removes any duplicate rows from the result if the optional keyword `DISTINCT` is included in the `select` clause, after the keyword `select` and before the expressions whose results are to be projected.

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 in the previous SQL reading packet?

```
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), a `*` in the `select` clause is a shorthand meaning one wants to project 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 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 `insert` statements

Before we continue with additional examples, we need to set up some example tables. The home page for the public course web site should include a "References" section, within which should be 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-projects`, paste in this posted link's contents, and save your resulting SQL script file. Alternately, I've placed a copy of this script file

on Canvas and so you should be able to make transfer it to your own directory on nrs-project with some UNIX command.

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 SQL reading packet. You can either look at the posted version, or open the file using vim or nano, 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, the cascade constraints clause makes 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.

The earlier create table statement examples showed declaring a column by giving its name, and giving a type to serve as a physical domain for the values that column is allowed to contain. But SQL allows you to include some additional clauses to further limit, or constrain, the values considered to be part of a column's domain -- it allows you to add some additional **constraints** on a column's domain. You see some of the additional constraints in set-up-ex-tbls.sql's create table statements.

For example, the dept table's create table statement is using an additional constraint, NOT NULL, in the column definitions for dept_name and dept_loc. 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 -- the domains for dept_name and dept_loc do not include NULL.

We discussed foreign keys in a previous reading packet; 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, although instead of a domain constraint on a column, it is a table 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. (We'll introduce how to specify a default value for a column a little later in this reading packet.)

So,

```
insert into empl(empl_num, empl_last_name, job_title, mgr, hiredate,
                 salary, dept_num)
values
('7934', 'Miller', 'Clerk', '7782', '23-Jan-2016', 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 about how you can specify a default value for a column? You can see that in the declaration for table `customer`'s `cust_balance` column:

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

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

You are encouraged to try out the SQL statements discussed in this reading packet, and see for yourself that they work as described -- you are also encouraged to try out additional SQL statements, because practicing writing SQL statements is the most effective way to become better at writing them!

But, so that the example queries might make more sense, here are the contents of the example tables after `set-up-ex-tbls.sql` has been run:

The `empl` table contains:

```
EMPL EMPL_LAST_NAME  JOB_TITLE   MGR  HIREDATE      SALARY COMMISSION DEP
---- --------------- ---------- ---- --------- ---------- ---------- ---
7839 King            President       17-NOV-11       5000            500
7566 Jones           Manager    7839 02-APR-12       2975            200
7698 Blake           Manager    7839 01-MAY-13       2850            300
7782 Raimi           Manager    7839 09-JUN-12       2450            100
7902 Ford            Analyst    7566 03-DEC-12       3000            200
7369 Smith           Clerk      7902 17-DEC-12        800            200
7499 Michaels        Sales      7698 20-FEB-18       1600        300 300
7521 Ward            Sales      7698 22-FEB-19       1250        500 300
7654 Martin          Sales      7698 28-SEP-18       1250       1400 300
7788 Scott           Analyst    7566 09-NOV-18       3000            200
7844 Turner          Sales      7698 08-SEP-19       1500          0 300

EMPL EMPL_LAST_NAME  JOB_TITLE   MGR  HIREDATE      SALARY COMMISSION DEP
```

```
---- -------------- ---------- ---- --------- ---------- ---------- ---
7876 Adams          Clerk      7788 23-SEP-18       1100            400
7900 James          Clerk      7698 03-DEC-17        950            300
7934 Miller         Clerk      7782 23-JAN-16       1300            100

14 rows selected.
```

(Above, you are seeing the results using Oracle SQL*Plus's default display settings -- we'll be discussing how to change these default display settings later in the semester. In the meantime, I'll just note that the column headings are repeated about every 11 rows of output by default, and that SQL*Plus sometimes does not show all of a column's name -- above, note that column `dept_num` has a SQL*Plus default display of `DEP`! And a column with no value for a particular row -- a column whose value is null for that row -- is displayed as blank, as you can see for most employees' `commission` column and for President King's `mgr` column.)

The `dept` table contains:

```
DEP DEPT_NAME       DEPT_LOC
--- --------------- ---------------
100 Accounting      New York
200 Research        Dallas
300 Sales           Chicago
400 Operations      Boston
500 Management      New York
```

And the `customer` table contains:

```
CUST_I CUST_LNAME           CUST_FNAME      EMPL CUST_STREET
------ -------------------- --------------- ---- -------------------------
CUST_CITY       CU CUST_ZIP   CUST_BALANCE
-------------- -- ---------- ------------

100001 Firstly              First           7499 1111 First Street
Fortuna         CA 95520           1111.11

100002 Secondly             Second          7654 2222 Second Street
McKinleyville   CA 95523             222.2

100003 Thirdly              Third           7499 333 Third Street
Arcata          CA 95519-1234             0
```

(Because the default linesize in SQL*Plus is shorter than the default display width of the `customer` table's rows, this appears in SQL*Plus with the ugly formatting shown above. Again, we'll discuss how to change SQL*Plus's display default settings later in the semester.)

Using SQL `select` statements for the classic relational operations

Relational projection with a SQL `select` statement

You can use a SQL `select` statement 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:

This statement results in 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;
```

...with the results (for `empl` as filled in `set-up-ex-tbls.sql`):

```
EMPL_LAST_NAME      SALARY HIREDATE
--------------- ---------- ---------
Michaels              1600 20-FEB-18
Ward                  1250 22-FEB-19
Turner                1500 08-SEP-19
Blake                 2850 01-MAY-13
James                  950 03-DEC-17
King                  5000 17-NOV-11
Ford                  3000 03-DEC-12
Smith                  800 17-DEC-12
Martin                1250 28-SEP-18
Adams                 1100 23-SEP-18
Miller                1300 23-JAN-16

EMPL_LAST_NAME      SALARY HIREDATE
--------------- ---------- ---------
Raimi                 2450 09-JUN-12
Scott                 3000 09-NOV-18
Jones                 2975 02-APR-12

14 rows selected.
```

This statement results in the relational projection of the `job_title` column of the `empl` table:

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

...with the results:

```
JOB_TITLE
----------
Manager
Analyst
Clerk
President
Sales
```

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.

Consider these two SQL queries:

```
select distinct job_title, dept_num
from   empl;

select job_title, dept_num
from   empl;
```

The results for:

```
select distinct job_title, dept_num
from   empl;
```

...are:

```
JOB_TITLE  DEP
---------- ---
Manager    100
Sales      300
Clerk      100
Manager    200
Manager    300
Clerk      400
President  500
Analyst    200
Clerk      200
Clerk      300

10 rows selected.
```

And, the results for:

```
select job_title, dept_num
from   empl;
```

...are:

```
JOB_TITLE  DEP
---------- ---
President  500
Manager    200
Manager    300
```

```
Manager    100
Analyst    200
Clerk      200
Sales      300
Sales      300
Sales      300
Analyst    200
Sales      300

JOB_TITLE  DEP
---------- ---
Clerk      400
Clerk      300
Clerk      100

14 rows selected.
```

Be sure you understand why the results of these two queries 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 a SQL `select` statement as follows to specify a relational selection:

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

Here are some example relational selections:

This is the relational selection of the rows of the `empl` table where `job_title = 'Manager'`:

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

...with the results:

```
EMPL EMPL_LAST_NAME  JOB_TITLE  MGR  HIREDATE      SALARY COMMISSION DEP
---- --------------- ---------- ---- --------- ---------- ---------- ---
7566 Jones           Manager    7839 02-APR-12       2975            200
7698 Blake           Manager    7839 01-MAY-13       2850            300
7782 Raimi           Manager    7839 09-JUN-12       2450            100
```

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 that no rows result. (Note that, mathematically, the empty table is a relation, too! It is a set with no tuples/rows.) This is because the only place that SQL is case-sensitive is within string literals, so `'Manager'` is not equal to `'manager'`.

(The output you see in SQL*Plus:

```
no rows selected
```

...is called **feedback** in Oracle SQL*Plus, and that is another SQL*Plus environment option that the user can set: the user can specify that they also want the number of rows selected output to the screen along with a SQL query's output. Why do you not always see it, by default? Because feedback's default setting in Oracle SQL*Plus is to display when a result has no rows, or has 6 or more rows. We'll also talk about how to modify this later in the semester.)

As a non-empty selection example, this is the relational selection of the rows of the `empl` table in which the `hiredate` is after June 1, 2013:

```
select *
from   empl
where  hiredate > '01-JUN-2013';
```

...with the results:

```
EMPL EMPL_LAST_NAME  JOB_TITLE   MGR  HIREDATE     SALARY COMMISSION DEP
---- --------------- ---------- ---- --------- ---------- ---------- ---
7499 Michaels             Sales  7698 20-FEB-18       1600        300 300
7521 Ward                 Sales  7698 22-FEB-19       1250        500 300
7654 Martin               Sales  7698 28-SEP-18       1250       1400 300
7788 Scott              Analyst  7566 09-NOV-18       3000            200
7844 Turner               Sales  7698 08-SEP-19       1500          0 300
7876 Adams                Clerk  7788 23-SEP-18       1100            400
7900 James                Clerk  7698 03-DEC-17        950            300
7934 Miller               Clerk  7782 23-JAN-16       1300            100

8 rows selected.
```

Relational equi-join with a SQL `select` statement

There is more than one way to write equi-joins and natural joins in Oracle's implementation of SQL, but they are not all created equal! For CS 325, you are expected to use one of the two variants described below for equi-joins and natural joins (others exist, but, for various reasons we won't go into right now, these will be considered poor style and against course coding standards). When we discuss several other types of join later in the semester, we'll add the acceptable syntax for those joins.

If you consider our semantic definition of equi-join from the reading packet introducing the relational model, it should be pretty clear that you can use a SQL `select` statement as follows to specify an equi-join:

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

After all, this specifies (conceptually -- the actual algorithm is more efficient than this sounds!) taking the Cartesian product of `tbl1` and `tbl2`,then selecting only those rows for which `tbl1.join-col` and `tbl2.join-col` are equal, then projecting all of the columns in the result.

There is also an alternate way of expressing this in a SQL `select` statement -- a so-called ANSI join, because this syntax was added in the SQL-92 ANSI revision of the SQL standard:

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

My understanding is that the Oracle DBMS converts either of these to the same executable code, so both should be equivalent in terms of performance.

Note that, using either style, when you are doing an equi-join or a natural join of two tables `tbl1` and `tbl2`, you are expected to always explicitly include the equality condition specifying which column you are joining the two tables on -- you **always** need the `<tbl1.join-col> = <tbl2.join-col>`. We'll call this the **join condition**.

One advantage of the ANSI variant is that you are less likely to accidentally write a Cartesian product when you meant to write an equi-join by leaving out the join condition!

Here are some equi-join examples:

Each of the following is 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;

select *
from  empl join dept
      on empl.dept_num = dept.dept_num;
```

These have exactly the same result -- because the default linesize in SQL*Plus is shorter than the default display width of the equi-join's resulting rows, this appears in SQL*Plus about as follows:

```
EMPL EMPL_LAST_NAME  JOB_TITLE  MGR  HIREDATE      SALARY COMMISSION DEP DEP
---- --------------- ---------- ---- --------- ---------- ---------- --- ---
DEPT_NAME       DEPT_LOC
--------------- ---------------
7839 King            President       17-NOV-11       5000            500 500
Management      New York

7566 Jones           Manager    7839 02-APR-12       2975            200 200
Research        Dallas

7698 Blake           Manager    7839 01-MAY-13       2850            300 300
Sales           Chicago
```

```
EMPL EMPL_LAST_NAME  JOB_TITLE  MGR  HIREDATE      SALARY COMMISSION DEP DEP
---- --------------- ---------- ---- --------- ---------- ---------- --- ---
DEPT_NAME       DEPT_LOC
--------------- ---------------
7782 Raimi           Manager    7839 09-JUN-12       2450            100 100
Accounting      New York

7902 Ford            Analyst    7566 03-DEC-12       3000            200 200
Research        Dallas

7369 Smith           Clerk      7902 17-DEC-12        800            200 200
Research        Dallas


EMPL EMPL_LAST_NAME  JOB_TITLE  MGR  HIREDATE      SALARY COMMISSION DEP DEP
---- --------------- ---------- ---- --------- ---------- ---------- --- ---
DEPT_NAME       DEPT_LOC
--------------- ---------------
7499 Michaels        Sales      7698 20-FEB-18       1600        300 300 300
Sales           Chicago

7521 Ward            Sales      7698 22-FEB-19       1250        500 300 300
Sales           Chicago

7654 Martin          Sales      7698 28-SEP-18       1250       1400 300 300
Sales           Chicago


EMPL EMPL_LAST_NAME  JOB_TITLE  MGR  HIREDATE      SALARY COMMISSION DEP DEP
---- --------------- ---------- ---- --------- ---------- ---------- --- ---
DEPT_NAME       DEPT_LOC
--------------- ---------------
7788 Scott           Analyst    7566 09-NOV-18       3000            200 200
Research        Dallas

7844 Turner          Sales      7698 08-SEP-19       1500          0 300 300
Sales           Chicago

7876 Adams           Clerk      7788 23-SEP-18       1100            400 400
Operations      Boston


EMPL EMPL_LAST_NAME  JOB_TITLE  MGR  HIREDATE      SALARY COMMISSION DEP DEP
---- --------------- ---------- ---- --------- ---------- ---------- --- ---
DEPT_NAME       DEPT_LOC
--------------- ---------------
7900 James           Clerk      7698 03-DEC-17        950            300 300
Sales           Chicago

7934 Miller          Clerk      7782 23-JAN-16       1300            100 100
Accounting      New York
```

Here are the same results, reformatted for this reading packet to make sure it is clear what relation is resulting from this equi-join:

empl_num	**empl_last_name**	**job_title**	**mgr**	hiredate	**salary**	**commission**	**empl.dept_num**	**dept.dept_num**	**dept_name**	**dept_loc**
7839	King	President		17-NOV-11	5000		500	500	Management	New York
7566	Jones	Manager	7839	02-APR-12	2975		200	200	Research	Dallas
7698	Blake	Manager	7839	01-MAY-13	2850		300	300	Sales	Chicago
7782	Raimi	Manager	7839	09-JUN-12	2450		100	100	Accounting	New York
7902	Ford	Analyst	7566	03-DEC-12	3000		200	200	Research	Dallas
7369	Smith	Clerk	7902	17-DEC-12	800		200	200	Research	Dallas
7499	Michaels	Sales	7698	20-FEB-18	1600	300	300	300	Sales	Chicago
7521	Ward	Sales	7698	22-FEB-19	1250	500	300	300	Sales	Chicago
7654	Martin	Sales	7698	28-SEP-18	1250	1400	300	300	Sales	Chicago
7788	Scott	Analyst	7566	09-NOV-18	3000		200	200	Research	Dallas
7844	Turner	Sales	7698	08-SEP-19	1500	0	300	300	Sales	Chicago
7876	Adams	Clerk	7788	23-SEP-18	1100		400	400	Operations	Boston
7900	James	Clerk	7698	03-DEC-17	950		300	300	Sales	Chicago
7934	Miller	Clerk	7782	23-JAN-16	1300		100	100	Accounting	New York

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 using the first style demonstrated above:

```
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 a `select` statement's `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`. (Yes, that includes within the `select` clause! You'll see an instance of this in the next example.)

Relational natural join with a SQL `select` statement

There is no reasonable short-cut to easily get a natural join using a SQL `select` statement -- it is like the `equi-join`, but you have to explicitly 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>;

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

As an example, each of the following results in 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;

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

The result of either of these should display as:

```
EMPL EMPL_LAST_NAME  JOB_TITLE  MGR  HIREDATE      SALARY COMMISSION DEP
---- --------------- ---------- ---- --------- ---------- ---------- ---
DEPT_NAME       DEPT_LOC
--------------- ---------------
7839 King            President       17-NOV-11       5000            500
Management      New York

7566 Jones           Manager    7839 02-APR-12       2975            200
```

```
Research        Dallas

7698 Blake           Manager    7839 01-MAY-13       2850            300
Sales           Chicago


EMPL EMPL_LAST_NAME  JOB_TITLE  MGR  HIREDATE      SALARY COMMISSION DEP
---- --------------- ---------- ---- --------- ---------- ---------- ---
DEPT_NAME       DEPT_LOC
-------------- --------------
7782 Raimi           Manager    7839 09-JUN-12       2450            100
Accounting      New York

7902 Ford            Analyst    7566 03-DEC-12       3000            200
Research        Dallas

7369 Smith           Clerk      7902 17-DEC-12        800            200
Research        Dallas


EMPL EMPL_LAST_NAME  JOB_TITLE  MGR  HIREDATE      SALARY COMMISSION DEP
---- --------------- ---------- ---- --------- ---------- ---------- ---
DEPT_NAME       DEPT_LOC
-------------- --------------
7499 Michaels        Sales      7698 20-FEB-18       1600        300 300
Sales           Chicago

7521 Ward            Sales      7698 22-FEB-19       1250        500 300
Sales           Chicago

7654 Martin          Sales      7698 28-SEP-18       1250       1400 300
Sales           Chicago


EMPL EMPL_LAST_NAME  JOB_TITLE  MGR  HIREDATE      SALARY COMMISSION DEP
---- --------------- ---------- ---- --------- ---------- ---------- ---
DEPT_NAME       DEPT_LOC
-------------- --------------
7788 Scott           Analyst    7566 09-NOV-18       3000            200
Research        Dallas

7844 Turner          Sales      7698 08-SEP-19       1500          0 300
Sales           Chicago

7876 Adams           Clerk      7788 23-SEP-18       1100            400
Operations      Boston


EMPL EMPL_LAST_NAME  JOB_TITLE  MGR  HIREDATE      SALARY COMMISSION DEP
---- --------------- ---------- ---- --------- ---------- ---------- ---
DEPT_NAME       DEPT_LOC
-------------- --------------
7900 James           Clerk      7698 03-DEC-17        950            300
Sales           Chicago

7934 Miller          Clerk      7782 23-JAN-16       1300            100
Accounting      New York
```

Here are the same results, reformatted for this reading packet to make sure it is clear what relation is

resulting from this natural join:

empl_num	**empl_last_name**	**job_title**	**mgr**	hiredate	**salary**	**commission**	**empl.dept_num**	**dept_name**	**dept_loc**
7839	King	President		17-NOV-11	5000		500	Management	New York
7566	Jones	Manager	7839	02-APR-12	2975		200	Research	Dallas
7698	Blake	Manager	7839	01-MAY-13	2850		300	Sales	Chicago
7782	Raimi	Manager	7839	09-JUN-12	2450		100	Accounting	New York
7902	Ford	Analyst	7566	03-DEC-12	3000		200	Research	Dallas
7369	Smith	Clerk	7902	17-DEC-12	800		200	Research	Dallas
7499	Michaels	Sales	7698	20-FEB-18	1600	300	300	Sales	Chicago
7521	Ward	Sales	7698	22-FEB-19	1250	500	300	Sales	Chicago
7654	Martin	Sales	7698	28-SEP-18	1250	1400	300	Sales	Chicago
7788	Scott	Analyst	7566	09-NOV-18	3000		200	Research	Dallas
7844	Turner	Sales	7698	08-SEP-19	1500	0	300	Sales	Chicago
7876	Adams	Clerk	7788	23-SEP-18	1100		400	Operations	Boston
7900	James	Clerk	7698	03-DEC-17	950		300	Sales	Chicago
7934	Miller	Clerk	7782	23-JAN-16	1300		100	Accounting	New York

It 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 of those earlier sections was that you **can** use a SQL `select` statement to specify each "pure" basic relational operation, not that you are limited to those.) The SQL `select` statement makes such combinations very reasonable, especially once you are comfortable with its semantics as described earlier.

Note that SQL has a Boolean `AND` operation, for logical and, a Boolean `OR` operation, for logical or, and a Boolean `NOT` operation, for logical not. These can be used to build very sophisticated `where` clauses, where you can select a quite-specifically-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.

You can also 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 appropriate join condition(s)...!), select only the rows you want from any joins, and project only the columns you want from that selection.

```
select    <comma-separated desired expressions to project>
from      <tbl>
```

```
where     <condition to specify desired rows from tbl>;

select    <comma-separated desired expressions to project>
from      <tbl1>, <tbl2>
where     <tbl1.join-col> = <tbl2.join-col>
          and <condition to specify desired rows from join>;

select    <comma-separated desired expressions to project>
from      <tbl1> join <tbl2>
          on <tbl1.join-col> = <tbl2.join-col>
where     <condition to specify desired rows from join>;
```

That `where` clause's condition can range from a very simple condition to a quite-complex compound condition involving any number of `and`, `or`, and `not` operations -- we'll spend more than one future reading packet discussing just some of the available options for the `where` clause!

By the way -- we aren't actually limited to two tables in the `from` clause, nor are we limited to table names in the `from` clause. But we'll include examples of these in later packets. And, as mentioned previously, we'll be adding additional optional `select` statement clauses in later packets as well.

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;
```

...with the results:

```
JOB_TITLE  HIREDATE
---------- ---------
Sales      20-FEB-18
Sales      22-FEB-19
Sales      28-SEP-18
```

Note that, while there happen to be no duplicate rows in this result, for different `empl` table contents, this may produce duplicate rows -- there is no `distinct` in the `select` clause to prevent them. Also, are you surprised at the number of rows in this result? A column with no value is not the same as a column with a value of `0` -- the condition `commission > 0` is **not** true for a row whose commission is null.

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 June 1, 2018? Either of these will do so:

```
select empl_last_name, dept_name, dept_loc
from   empl, dept
where  empl.dept_num = dept.dept_num
and    hiredate > '01-JUN-2018';

select empl_last_name, dept_name, dept_loc
```

```
from   empl join dept
       on empl.dept_num = dept.dept_num
where  hiredate > '01-JUN-2018';
```

And, both of these have the result:

```
EMPL_LAST_NAME  DEPT_NAME       DEPT_LOC
--------------- --------------- ---------------
Scott           Research        Dallas
Turner          Sales           Chicago
Martin          Sales           Chicago
Ward            Sales           Chicago
Adams           Operations      Boston
```

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. (That is, if someone says they are projecting just some columns from a join of two tables, it should be safe to assume that equi-join or natural join is intended -- if someone means one of the other types of joins, they should specify that explicitly. And we'll do so when we introduce a few of these other types of joins later in the semester.)