CIS 315 - Reading Packet: "Set-theoretic operations, more on modifying data, SQL*Loader, and sequences"

Sources:

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

more SELECT operations: UNION, INTERSECT, and MINUS

When we first mentioned the operations that could be done on relations, we mentioned that some were based on set theory, and some were based on relational algebra. We then discussed the most-important relational operations.

Now we will discuss the most important set-theoretic operations, and how to implement them using the SQL SELECT statement.

The set-theoretic operations are set operations that can be done on relations because they are sets -- sets of tuples, sets of rows. The three we will be discussing are union, intersection, and difference.

The union operation

You probably remember talking about the union of two sets in some past math class -- if a set A is something like {1, 2, 3, 4, 5} and a set B is something like {2, 4, 6, 8}, then the union of sets A and B is the set of everything that is in either set -- that is,

$$A \cup B = \{1, 2, 3, 4, 5, 6, 8\}$$

So, what does union mean when the sets involved are relations? It means the relation that is the set of all tuples or rows that are in either of those relations; but, since the result of a relational operation has to still be a relation, you cannot perform the union of just any two relations. You can only perform a union of two so-called **union-compatible** relations: they have to have the same number of columns, with compatible domains. Them, the resulting set of rows can still be a relation.

That is, if you have a table A, with:

```
col1      col2
-----     -----
dog          13
cat          14
hamster      15
```

and a table B, with

```
col1      col2
-----     -----
chicken      18
cat          14
gerbil       20
```

...then $A \cup B$ would be the relation:

```
col1      col2
-----     -----
dog          13
cat          14
hamster      15
chicken      18
gerbil       20
```

The intersection operation

You probably remember the basic intersection operation on sets as well -- if a set A is something like {1, 2, 3, 4, 5} and a set B is something like {2, 4, 6, 8}, then the intersection of sets A and B is the set of everything that is in both sets -- that is,

$$A \cap B = \{2, 4\}$$

It turns out that all of the set-theoretic operations for relations only apply to relations that are so-called union-compatible (with the same number of columns, with compatible domains). For such relations, then, the intersection of those relations will be the the relation that is the set of all tuples or rows that are in both of those relations.

So, for table A with:

```
col1      col2
-----     -----
dog          13
cat          14
hamster      15
```

and table B with:

```
col1      col2
-----     -----
chicken      18
cat          14
gerbil       20
```

...then $A \cap B$ would be the relation:

```
col1      col2
-----     -----
cat          14
```

The difference operation

This is probably the least familiar of the three set-theoretic operations we will be discussing. In basic set theory, the difference of two sets are those elements in the first set that are not in the second. That is, for those sets A = {1, 2, 3, 4, 5} and B = {2, 4, 6, 8, 10},

A - B = {1, 3, 5}

and

B - A = {6, 8, 10}

And, again, difference on relations can only be done on relations that are so-called union-compatible, and then it means the relation consisting of those tuples or rows from the first relation that are not in the second relation.

So, for table A with:

```
col1      col2
-----     -----
dog          13
cat          14
hamster      15
```

and table B with:

```
col1      col2
-----     -----
chicken      18
cat          14
gerbil       20
```

...then A - B would be the relation:

```
col1      col2
-----     -----
dog          13
hamster      15
```

...and B - A would be the relation:

```
col1      col2
-----     -----
chicken      18
gerbil       20
```

How to write queries using these set-theoretic operations in SQL

Above, we described each of these set-theoretic operations in general terms. Now, we'll describe how you can write a SQL SELECT statement including these operations.

Basically, there is an operator for each of these three, and each expects to be surrounded by two union-compatible sub-selects. (You can choose to follow this with an ORDER-BY clause *following* and *outside of* the last of the sub-selects, if you wish.) The union operation can be performed using the UNION operator, the intersection operation can be performed using the INTERSECT operator, and the difference operation can be performed using the MINUS operator. That is, (using [ ] to indicate that the ORDER-BY is optional, NOT as part of the syntax):

(*sub-select*)
`UNION`
(*sub-select*)
`[order by ...];`

(*sub-select*)
`INTERSECT`
(*sub-select*)
`[order by ...];`

(*sub-select*)
`MINUS`
(*sub-select*)
`[order by ...];`

For example, the union of the department numbers of departments in Chicago and the department numbers of employees who are managers could be expressed as:

```
(select  dept_num
 from    dept
 where   dept_loc = 'Chicago')
union
(select  dept_num
 from    empl
 where   job_title = 'Manager');
```

What if you'd like to order the resulting union'ed rows in descending order of dept_num? Then that ORDER-BY clause needs to be at the very end, OUTSIDE of the parentheses for the second sub-select -- you are ordering the rows in the union'ed result, not of the second sub-select!:

```
(select dept_num
 from   dept
 where  dept_loc = 'Chicago')
union
(select dept_num
 from   empl
 where  job_title = 'Manager')
order by dept_num desc;
```

The intersection of the employee last names, dept_nums, and hiredates of employees hired after 7-1-1991 with the employee last names, dept_nums, and hiredates of employees located in Dallas, ordered by employee last name, could be expressed as:

```
(select  empl_last_name, dept_num, hiredate
 from    empl
 where   hiredate > '01-Jul-1991')
intersect
(select  empl_last_name, d.dept_num, hiredate
 from    empl e, dept d
 where   d.dept_num = e.dept_num
 and     dept_loc = 'Dallas')
order by empl_last_name;
```

And the difference of the employee last names, dept_nums, and hiredates of employees hired after 7-1-1991 with the employee last names, dept_nums, and hiredates of employees located in Dallas, ordered by employee last name, could be expressed as:

```
(select  empl_last_name, dept_num, hiredate
 from    empl
 where   hiredate > '01-Jul-1991')
minus
(select  empl_last_name, d.dept_num, hiredate
 from    empl e, dept d
 where   d.dept_num = e.dept_num
 and     dept_loc = 'Dallas')
order by empl_last_name;
```

UNION ALL

If you look at the results of the UNION examples thus far, you will hopefully notice that you never get duplicate rows in the results -- SQL's UNION operator, it turns out, results in a "true" set-theoretic union, and as true sets never have duplicate elements, a "true" union of relations never has duplicate tuples or rows, either.

Sometimes, though, when you write a query, you want duplicate rows (maybe you want to count something about them, for example). You can get a non-"pure" union by using **UNION ALL**.

Run each of the following in SQL, and you should observe this difference in action:

```
(select empl_last_name, dept_num, hiredate
 from   empl
 where  hiredate > '01-Jul-1991')
union all
(select empl_last_name, d.dept_num, hiredate
 from   empl e, dept d
 where  d.dept_num = e.dept_num
 and    dept_loc = 'Dallas')
order by empl_last_name;

(select empl_last_name, dept_num, hiredate
 from   empl
```

```
 where  hiredate > '01-Jul-1991')
union
(select empl_last_name, d.dept_num, hiredate
 from   empl e, dept d
 where  d.dept_num = e.dept_num
 and    dept_loc = 'Dallas')
order by empl_last_name;
```

Some additional notes on using set-theoretic operations

Note that you will receive an error message if you attempt these operations with relations that the DBMS can tell are obviously not union-compatible (for example, different numbers of columns between the two sub-selects, or "different-enough" domains). Unfortunately, it cannot really tell if two columns whose contents are of the same type really have the same *meaning* -- the same "true" domain. So, these can result in nonsense results if you use them on more-subtly inappropriate sub-selects.

For example, here is an attempted union that will fail, because the two sub-selects result in relations that are clearly not union-compatible, each having a different number of columns:

```
-- WILL FAIL!! not union-compatible!!

(select  dept_num, dept_name
 from    dept
 where   dept_loc = 'Chicago')
union
(select  dept_num
 from    empl
 where   job_title = 'Manager');
```

The above query results in the error message:

```
(select     dept_num, dept_name
 *
ERROR at line 1:
ORA-01789: query block has incorrect number of result columns
```

Likewise, this attempted intersection will fail, because even though the relations resulting from the two sub-selects have one column each, their domains are obviously different enough that Oracle can detect it:

```
-- WILL ALSO FAIL!! also not union-compatible!!

(select  dept_num
 from    dept
 where   dept_loc = 'Chicago')
union
(select  salary
 from    empl
 where   job_title = 'Manager');
```

...although the error message in this case is a bit different (since the reason for it being not-union-compatible is a bit different):

```
(select  dept_num
         *
ERROR at line 1:
ORA-01790: expression must have same datatype as corresponding expression
```

But the following, sadly, will give results, although they don't make much sense, because the SQL interpreter cannot tell if two "compatible" types are not compatible in terms of "true" meaning and "true" domain:

```
(select  dept_num
 from    dept
 where   dept_loc = 'Chicago')
union
(select  empl_num
 from    empl
 where   job_title = 'Manager');
```

...which results in:

```
DEPT
----
300
7566
7698
7782
```

Another note: the column names do not have to be the same in the sub-selects, as long as the number of columns and the types are compatible:

```
(select empl_last_name, salary
 from   empl
 where  commission is null)
union
(select empl_last_name, salary + commission
 from   empl
 where commission is not null);
```

...which, by the way, finally gives us a reasonable way to project "total" compensation for employees who just have salary and those who have both salary and commission! (Remember, if you try to just project salary + commission for everyone, you get a NULL result for those with a NULL commission...)

However, if you want to order the results, note that the ORDER-BY at the end "sees" the column names projected by the first sub-select -- you need to use whatever name that first sub-select uses for those projected columns:

```
(select empl_last_name, salary "Total compensation"
 from   empl
 where  commission is null)
union
(select empl_last_name, salary + commission
```

```
 from   empl
 where commission is not null)
order by "Total compensation";
```

(Had you noticed that Oracle SQL*Plus always gives the column labels of the first sub-select in the result?

Sometimes you can use UNION to get results that you cannot with the SELECT features we have discussed so far.

Assume that we added a new department to the **dept** table:

```
insert into dept
values
('600', 'Computing', 'Arcata');
```

If you wanted to project the number of employees in each department -- even new departments with now employees -- you might try:

```
select    dept_name, count(*)
from      empl e, dept d
where     e.dept_num = d.dept_num
group by  dept_name;
```

However, this won't work -- the Computing department won't show up. A natural join and equi-join will ALWAYS omit rows from one table that don't have a foreign key matching it in the other table. No empl has dept_num of 600, so Computing cannot show up in this query's result.

However, because you CAN project constants (as we saw in a previous week's lab), you could use a UNION to combine the above result with the results of a sub-select grabbing department names and the constant 0 for departments with NO employees:

```
(select    dept_name, count(*) "# of Employees"
 from      empl e, dept d
 where     e.dept_num = d.dept_num
 group by  dept_name)
union
(select    dept_name, 0 "# of Employees"
 from      dept d
 where     not exists
           (select 'a'
            from   empl e
            where  e.dept_num = d.dept_num))
order by "# of Employees" desc;
```

Of course, if you would prefer another means besides NOT EXISTS to see which departments have no employees, you could use MINUS for that, requesting the difference between the dept_names of all departments and the dept_names of the rows in the join of dept and empl:

```
(select dept_name
 from   dept)
```

```
minus
(select dept_name
 from   dept d, empl e
 where  d.dept_num = e.dept_num);
```

So, this could work to get counts for all departments, also: (note the careful use of parentheses here!) (and note that I had to give the 2nd sub-select a column alias to get this to work -- that wasn't true of the earlier example. IF you get an error regarding what you are ordering by, use the same column alias for ALL sub-selects involved...)

```
(select    dept_name, count(*) "# of Employees"
 from      empl e, dept d
 where     e.dept_num = d.dept_num
 group by  dept_name)
union
((select dept_name, 0 "# of Employees"
  from   dept)
 minus
 (select dept_name, 0
  from   dept d, empl e
  where  d.dept_num = e.dept_num))
order by "# of Employees" desc;
```

The ORDER BY issue is worth a few more words: when an order-by is at the end of a "regular" top-level select, you can order by any column, even if you aren't projecting that column -- that is, this works just fine:

```
select   empl_last_name
from     empl
order by salary;
```

However, when the ORDER-BY is ordering the results of sub-selects being UNION'ed or MINUS'ed or INTERSECT'ed, that outer-level ORDER-BY ONLY knows about the columns actually projected by the sub-selects. That is, this query will NOT work:

```
-- this will NOT work -- because the outer-level's order-by only
-- knows about the 3 columns projected by the minus'd sub-selects:

(select empl_last_name, dept_num, hiredate
 from   empl
 where  hiredate > '01-Jul-1991')
minus
(select empl_last_name, d.dept_num, hiredate
 from   empl e, dept d
 where  d.dept_num = e.dept_num
 and    dept_loc = 'Dallas')
order by salary;
```

It will complain that:

```
order by salary
         *
ERROR at line 9:
```

```
ORA-00904: "SALARY": invalid identifier
```

note on the "full" SELECT syntax

Having covered UNION, INTERSECT, and MINUS, we have now covered all of the major components of a SQL SELECT statement.

The posted '**"Full" SELECT statement summary**' summarizes the "full" SELECT syntax and its semantics; be sure to look over it, and let me know if you have any questions about it.

further manipulations of database contents: beyond INSERT

But, while querying a database is arguably the most important thing one does with a database, one also needs to insert, update, and manipulate the data within that database in appropriate ways between such queries. We've discussed basic row insertion into tables using the SQL INSERT statement; now we'll discuss updating existing rows, and deleting rows. We'll also talk about an Oracle database object, a sequence, that can make it easier to create suitable primary keys for tables over time, and briefly introduce SQL*Loader, another Oracle tool that can make insertions into databases easier when that data already happens to be contained within files.

brief aside: some demonstrations of Oracle DBMS support for domain integrity

Consider the following **parts** table:

```
drop table parts cascade constraints;

create table parts
(part_num               char(5),
 part_name              varchar2(25),
 quantity_on_hand       smallint,
 price                  decimal(6,2),
 level_code             char(3),        -- level code must be 3 digits
 last_inspected         date,
 primary key            (part_num)
);
```

Here is an example of a successful row insertion into this table:

```
insert into parts
values
('10601', '3/8 in lug nut', 1000, 0.02, '002', '09-SEP-2002');
```

And, here is an example of at least partial domain integrity support in action: the following insertion will NOT work, because the given part name is longer than the attribute declaration for part_name allows:

```
insert into parts
values
('10602', '5/8 in lug nut from Argentina or Brazil', 16, 4.50, '105',
```

```
 '04-SEP-2000');
```

Here's the error message that Oracle SQL*Plus gives when this is attempted:

```
('10602', '5/8 in lug nut from Argentina or Brazil', 16, 4.50, '105',
          *
ERROR at line 3:
ORA-12899: value too large for column "ST10"."PARTS"."PART_NAME" (actual: 39,
maximum: 25)
```

As another example, this insertion will fail because the price is too large for that attribute's declaration:

```
insert into parts
values
('10602', '5/8 in lug nut', 16, 10000.00, '105', '04-SEP-2000');
```

...resulting in the error message:

```
('10602', '5/8 in lug nut', 16, 10000.00, '105', '04-SEP-2000')
                                *
ERROR at line 3:
ORA-01438: value larger than specified precision allows for this column
```

But all of these will succeed, and will help us in setting up our upcoming UPDATE and DELETE examples:

```
insert into parts
values
('10603', 'hexagonal wrench', 13, 9.99, '003', '05-SEP-2000');

insert into parts
values
('10604', 'tire', 287, 39.99, '333', '06-SEP-2000');

insert into parts
values
('10605', 'hammer', 30, 9.99, '003', '01-SEP-2000');

insert into parts
values
('10606', '3/8 in bolt', 5000, 0.03, '005', '04-SEP-2000');

insert into parts
values
('10607', '7/8 in bolt', 2655, 0.04, '005', '02-SEP-2000');
```

SQL UPDATE command

The SQL INSERT command is used, as you know, for adding a new row to a table. What if you want to change something, however, about a row that is already in a table? Then you can use the SQL **UPDATE** command to do so.

Here is a first, simple example of the basic UPDATE command syntax:

```
update tbl_name
set    attrib1 = expression1
where  attrib2 = expression2;
```

The semantics, or meaning, of this is that, in every row of `tbl_name` for which `attrib2 = expression2`, `attrib1` will be changed to the value of `expression1`. (So, note that more than one row might be changed as the result of a single UPDATE command.)

Also, it is important to realize that `expression1` and `expression2` can be as complex as you'd like -- indeed, the WHERE clause here can be every bit as complex as a SELECT statement's WHERE clause, with nested sub-selects, various operators, etc.

Here are a few examples of UPDATE commands:

```
update  parts
set     price = 66.66
where   part_num = '10604';
```

Only one row is changed by this command, since only one row in parts has part_num of `'10604'`. And now the price for that particular row has been changed to `66.66`.

Given the rows we just inserted into parts, the following will end up updating two rows:

```
update parts
set    quantity_on_hand = 0
where  price = 9.99;
```

...because both the hexagonal wrench and the hammer have price of 9.99. And both of these rows now have a quantity_on_hand of 0.

What do you think happens if you have no WHERE clause in an UPDATE command? Well, consider what happens in the SELECT statement in such a case: all rows of the specified table (or of the specified Cartesian product!) are selected. Likewise, if you leave off the WHERE clause in an UPDATE statement, then EVERY row in the specified table will have that modification made to it.

So, the following will change the last_inspected attribute of ALL rows currently in the parts table to contain the current date (since **sysdate** is an Oracle date function that "returns the current date and time set for the operating system on which the database resides" [Oracle Database SQL Reference, http://download.oracle.com/docs/cd/B19306_01/server.102/b14200/functions172.htm])

```
update  parts
set     last_inspected = sysdate;
```

...updating all six rows currently in the parts table to now have the same last_inspected value, the date that this command is run.

Here is an example giving at least a suggestion that the SET and WHERE clauses can get more interesting:

```
update parts
set    last_inspected = (select max(hiredate)
                         from empl)
where  quantity_on_hand < (select quantity_on_hand
                           from   parts
                           where  part_num = '10607');
```

...which actually does update four of parts' rows to now have a last_inspected value of January 23, 1992.

SQL DELETE command

The SQL UPDATE command can change the values of the attributes within a row, but it cannot get rid of an entire row. The SQL DROP TABLE command can get rid of an entire table, including all of its rows, but it cannot get rid of its rows and keep the table.

No; if you want to get rid of one or more rows (but keep the table), then you need the SQL DELETE command.

Here is a first, simple example of the basic DELETE command syntax:

```
delete from tbl_name
where  attrib1 = expression;
```

The semantics, or meaning, of this is that, for every row of `tbl_name` for which `attrib1 = expression`, that row will be removed from the table. (So, note that more than one row might be deleted as the result of a single DELETE command.)

And, as for UPDATE, it is important to realize that the DELETE command's `expression` can be as complex as you'd like -- that the DELETE command's WHERE clause, too, can be every bit as complex as a SELECT statement's WHERE clause, with nested sub-selects, various operators, etc.

Here are a few examples of DELETE commands:

```
delete from parts
where  price = 66.66;
```

This deletes one row from parts, the one for part 10604, which happens to be the only one right now that had a price of 66.66.

```
delete from parts
where level_code = '005';
```

Two rows happened to have a level_code of '005', for parts 10606 and 10607 -- if you look, you'll see that both are indeed gone after this statement has been executed.

And, as for UPDATE, if you omit the WHERE clause in a DELETE statement, you will delete ALL of the specified table's rows -- the table will still exist, but it will have no rows (it will have 0 rows).

```
delete from parts;
```

After the above command, the parts table will be empty.

Putting back some rows for another DELETE example:

```
insert into parts
values
('10601', '3/8 in lug nut', 1000, 0.02, '002', '09-SEP-2002');

insert into parts
values
('10603', 'hexagonal wrench', 13, 9.99, '003', '05-SEP-2000');

insert into parts
values
('10604', 'tire', 287, 39.99, '333', '06-SEP-2000');

insert into parts
values
('10605', 'hammer', 30, 9.99, '003', '01-SEP-2000');

insert into parts
values
('10606', '3/8 in bolt', 5000, 0.03, '005', '04-SEP-2000');

insert into parts
values
('10607', '7/8 in bolt', 2655, 0.04, '005', '02-SEP-2000');
```

...here's a final example with a more interesting WHERE clause:

```
delete from parts
where quantity_on_hand > (select avg(quantity_on_hand)
                          from parts);
```

...now two of those 6 new rows are gone again.

brief aside: some demonstrations of Oracle DBMS support for referential integrity

Since we have this lovely parts table available, let's create a **part_orders** table, which has a foreign key referencing the parts table, so we can demonstrate some of Oracle's support for referential integrity.

```
drop table part_orders cascade constraints;

create table part_orders
(order_num  char(6),
 cust_num   char(8),
 part_num   char(5),
 order_date date,
 quantity   integer,
 order_code char(1),
```

```
 primary key      (order_num),
 foreign key      (part_num) references parts
);
```

So, because part_orders has a foreign key referencing parts, then since Oracle does support referential integrity, no row can be inserted into the child table part_orders unless there is a corresponding row in parent table parts with the same part_num as the proposed child part_orders row. Likewise, you will now not be able to delete a row from the parent table parts if there is a child table in part_orders whose part_num is the same as parent row to be deleted.

The following insertion into part_orders will work, since it is a part order for a currently-existing part:

```
insert into part_orders
values
('111111', '11111111', '10601', '01-Feb-2000', 6, 'B');
```

The following insertion into part_orders will NOT work, since it is for a part whose number is NOT in the parts table:

```
insert into part_orders
values
('111112', '11111111', '10106', '01-Feb-2000', 6, 'B');
```

Here is the Oracle error message:

```
insert into part_orders
*
ERROR at line 1:
ORA-02291: integrity constraint (ST10.SYS_C0084605) violated - parent key not
found
```

Likewise, the following deletion will fail, since it is attempting to delete a part for which there is a part_orders row:

```
delete from parts
where part_num = '10601';
```

...and here is the Oracle error message:

```
delete from parts
*
ERROR at line 1:
ORA-02292: integrity constraint (ST10.SYS_C0084605) violated - child record
found
```

Here's a further example of referential integrity support: you cannot update a part_orders row to have a non-existent part, either:

```
update part_orders
set   part_num = '13'
where part_num = '10601';
```

...resulting in the Oracle error message:

```
update part_orders
*
ERROR at line 1:
ORA-02291: integrity constraint (ST10.SYS_C0084605) violated - parent key not
found
```

...nor can you change a part_num for a part if there's a part_order involving that part_num:

```
update parts
set    part_num = '13'
where  part_num = '10601';
```

...resulting in the Oracle error message:

```
update parts
*
ERROR at line 1:
ORA-02292: integrity constraint (ST10.SYS_C0084605) violated - child record
found
```

YET ANOTHER brief aside: MORE demonstrations of Oracle DBMS support for referential integrity

Oracle may not take domain integrity support as far as it might, but here are some additional means of constraining/specifying attribute domains that it DOES support:

```
-- maxpoint     integer      not null,        -- this column MUST have a value
-- quantity     integer      default 1,       -- put 1 in if NO value is
--                                            -- inserted EXPLICITLY for this
--                                            -- column
-- car_color    varchar2(10) check(car_color IN ('red', 'green', 'white')),
-- quiz_grade   integer      check(quiz_grade >= 0 AND quiz_grade <= 100),
-- quiz_grade   integer      check(quiz_grade between 0 and 100),
```

Let's use some of these in a new version of table part_orders:

```
drop table part_orders cascade constraints;

create table part_orders
(order_num      char(6),
 cust_num       char(8)       not null,
 part_num       char(5)       not null,
 order_date     date,
 quantity       integer       default 1 not null,
 order_code     char(1)       check(order_code in ('B',
                                'I',
                                'G')),
 delivery_code  char(1)       check(delivery_code in
                                ('U', 'F', 'P')) not null,
 primary key    (order_num),
 foreign key    (part_num) references parts
```

```
);
```

Now for some insertions:

```
insert into part_orders
values
('111111', '11111111', '10601', '01-Feb-2000', 6, 'B', 'U');
```

Even though order_code has a check clause, it can still be NULL:

```
insert into part_orders(order_num, cust_num, part_num, order_date, quantity,
                        delivery_code)
values
('333333', '33333333', '10601', '01-Feb-2000', 8, 'F');

insert into part_orders(order_num, part_num, cust_num, order_date, quantity,
                        delivery_code)
values
('222222', '10605', '22222222', '1-Jan-00', 4, 'P');
```

Here's a demonstration that the DEFAULT clause works for the quantity attribute if NO value is explicitly specified for that attribute:

```
insert into part_orders(order_num, part_num, cust_num, order_date, delivery_code)
values
('444444', '10601', '22222222', '1-Feb-00', 'U');
```

...but, be careful! EXPLICIT insertion of NULL overrides the DEFAULT for an attribute; so this insertion FAILS because it ends up violating the not-null constraint that quantity also has:

```
insert into part_orders
values
('555555', '44444444', '10601', '3-Mar-98', NULL, 'G', 'U');
```

...resulting in the error message:

```
('555555', '44444444', '10601', '3-Mar-98', NULL, 'G', 'U')
                                            *
ERROR at line 3:
ORA-01400: cannot insert NULL into ("ST10"."PART_ORDERS"."QUANTITY")
```

Here are some more "bad" insertions that won't be allowed:

The order_code HAS to be 'B', 'I', or 'G':

```
insert into part_orders
values
('666666', '44444444', '10601', '25-Dec-99', 5, 'b', 'P');
```

...with the complaint:

```
insert into part_orders
*
```

```
ERROR at line 1:
ORA-02290: check constraint (ST10.SYS_C0084610) violated
```

The cust_num CANNOT be null, because it was specified as NOT NULL:

```
insert into part_orders(order_num, part_num, delivery_code)
values
('777777', '10601', 'U');
```

...with the complaint:

```
insert into part_orders(order_num, part_num, delivery_code)
*
ERROR at line 1:
ORA-01400: cannot insert NULL into ("ST10"."PART_ORDERS"."CUST_NUM")
```

a command you SHOULDN'T need often: the ALTER command

Note that changing a table's CONTENTS is different from changing a table's STRUCTURE; the DELETE command deletes a table's rows, but the table (even if it is empty) remains. To get rid of a whole table structure, you use the DROP TABLE command.

Likewise, UPDATE lets you change the contents of an existing row or rows, but if you want to change an existing table's structure, you must use a different command: the ALTER command.

You should not regularly have to alter tables after the fact, if they are designed well. But, every so often, it is helpful to be able to do so. Here are a few examples, just in case.

For example, this would add a new attribute to the parts table, a supplier attribute:

```
alter table parts
add
(supplier   varchar2(20)
);
```

If you'd like to see the new attribute in parts' structure, try the SQL*Plus describe command:

```
describe parts
```

And doing:

```
select  *
from    parts;
```

...will show that the value for supplier for all of the existing rows is NULL. (and, yes, this result is rather ugly, because the line is too long and is wrapping...)

You can, of course, use UPDATE to now modify the supplier attribute for these existing rows as desired:

```
update parts
```

```
set    supplier = 'Acme'
where  part_num in ('10603', '10604');
```

Note that Oracle may restrict you from making some alterations; for example, you can make an existing attribute "bigger", but you may not be able to make it "smaller" if any existing rows would not "fit" in the new "smaller" attribute.

Introduction to Sequences

A *sequence* is an Oracle database object provided for convenience: it generates, literally, a sequence of values. This can be useful for generating sound, non-duplicating primary keys for new rows over time.

Here are some tables to help us in playing with sequences:

```
drop table painter cascade constraints;

create table painter
(ptr_num        char(3),
 ptr_lname      varchar2(30) not null,
 ptr_fname      varchar2(15),
 primary key    (ptr_num)
);

drop table painting cascade constraints;

create table painting
(ptg_title      varchar2(30),
 ptr_num        char(3),
 primary key    (ptg_title),
 foreign key    (ptr_num) references painter
);
```

Let's say that I decide to create a sequence to help me to set good primary keys for the painter table over time. Then:

```
drop sequence painter_seq;

-- sequence painter_seq will start at 100, the next will be 102,
--    the next will be 104, etc.
-- (the increment and start clauses are optional --
--    the sequence increments by 1 if not specified,
--    and I THINK it starts at 1 if not specified...)

create sequence painter_seq
increment by    2
start with      100;
```

For a sequence object, adding **.nextval** after the name of the sequence gets you the next value of that sequence. So, here are some insertions into painter that make use of this:

```
insert into painter
values
(painter_seq.nextval, 'Van Gogh', 'Vincent');
```

```
insert into painter
values
(painter_seq.nextval, 'Monet', 'Claude');

insert into painter
values
(painter_seq.nextval, 'Da Vinci', 'Leonardo');
```

And if I look at the contents of painter now:

```
select  *
from    painter;
```

...I will see:

```
PTR PTR_LNAME                      PTR_FNAME
--- ------------------------------ ---------------
100 Van Gogh                       Vincent
102 Monet                          Claude
104 Da Vinci                       Leonardo
```

If I use **.currval** after the name of a sequence, it should give you the sequence's CURRENT value. If I know that a painting I'm adding is by the "latest" painter added, then I can do something like this:

```
insert into painting
values
('Mona Lisa', painter_seq.currval);

select  *
from    painting;
```

...and this select would result in:

```
PTG_TITLE                      PTR
------------------------------ ---
Mona Lisa                      104
```

I've had little luck using sequences in queries; this fails, for example:

```
select *
from   painter
where  ptr_num = painter_seq.currval;
```

...with the error message:

```
where ptr_num = painter_seq.currval
                             *
ERROR at line 3:
ORA-02287: sequence number not allowed here
```

But if you just want to see the current value of a sequence, you can project it -- DUAL is a built-in Oracle "dummy" table with 1 row and 1 column that is useful for such a query:

```
select  painter_seq.currval
from    dual;
```

...resulting in:

```
   CURRVAL
----------
       104
```

Now, even though sequences are typically used to generate primary keys, they don't HAVE to be. Here's a silly example demonstrating this:

```
insert into parts
values
('10614', 'stuff' || painter_seq.currval,
  painter_seq.currval, .13, '005', sysdate, 'Harry');
```

...which can be seen to result in inserting the following row:

```
PART_ PART_NAME  QUANTITY_ON_HAND PRICE LEV LAST_INSP   SUPPLIER
----- ---------  --------------   ----- --- ---------   --------
10614 stuff104    104             .13   005 07-APR-09   Harry
```

Brief Introduction to SQL*Loader

SQL*Loader is a program that comes with the Oracle DBMS to more conveniently import data that happens to already be in file format into database tables. I mention this as an example of a kind of tool that a DBMS might provide to make such "large-scale" insertion less work.

SQL*Loader actually has many options, far beyond what I want to introduce here. These are just three simple examples; Google "oracle sql loader documentation" if you are interested in learning more about SQL*Loader.

Basically, you set up a file with the suffix .ctl (control) as a control file for SQL*Loader, that says where the data is and how to handle it and into what tables to try to insert it. You then run the SQL*Loader program with that control file and (possibly) with data files.

Posted along with this lab exercise are the files:

```
empltemp_data1.ctl
empltemp_data2.ctl and empltemp_data2.dat
painter.ctl and painter.dat
```

Each of these control files contain extensive comments, explaining them and how to run them. Note the SQL*Loader program, called **sqlldr**, is run at the *UNIX* command line -- it is a separate program from sqlplus. However, this is also a convenient place to mention how the SQL*Plus **host** command can let you run an operating system command within SQL; this can be used to run the **sqlldr** command within a SQL script:

For example, if you have a SQL script containing the following, it would run `sqlldr` with the `painter.ctl` control file (if that is in the current working directory!); note that you would be asked to type in your Oracle username and password by the sqlldr program.

```
-- this will run sqlldr with the painter.ctl control file;
--    note that you need to type in your Oracle username and password
--    when requested

prompt ABOUT TO CALL sqlldr via host command;
prompt    (type Oracle username and password when prompted)

host sqlldr control=painter.ctl log=painter.log

prompt painter table after call to sqlldr:

select *
from   painter;
```