Lecture Notes for
Database Management, using
Database Principles, Programming and Performance
Notes by Patrick E. O'Neil

Chapter 3 Idea of SQL. Computerized query language.

IBM developed SQL, and it became standard. Most queries (including updates, inserts, deletes) are very standard across relational products.

We will only use ORACLE in class to begin with, but will also talk about others. Try to keep to basic standard for now (called Basic SQL in the text). Will show some more advanced features of SQL-99 a bit later.

In Appendix A.2, to enter Oracle, tells you to login to UNIX, give command:

```
%sqlplus
```

Any problems, contact DBA for course.

You will need to start creating tables. Follow directions in assignment and in Appendix A.2.

Create table. Result is empty table.

```
create table customers (cid char(4) not null, cname varchar(13),
   city varchar(20), discnt real, primary key(cid));
```

Names and gives types to each of columns: Explain char(4), varchar(20).

Primary key clause means can't duplicate in table, foreign keys will refer to this column by default.

Note "not null" is an integrity constraint: must specify value on insert, can't leave null. More integrity constraints in Chapter 7, on DBA stuff.

Next, load from OS file into table. (Called sqlldr in ORACLE) No-one wants to type in one line at a time: everything is moved around on tape or disk (say from one DBMS to another). Note expected format:

```
c001,TipTop,Duluth,10.00          <-- comma separated, no spaces
```

c002,Basics,Dallas,12.00

You have to create a .ctl table for each table, custs.ctl, . . . Then use "sql-ldr" command (or sqlload in Oracle version 7) from the UNIX prompt (not from within SQL*PLUS).

After think have table loaded, use the sqlplus command and within that environment type:

select * from customers;\g

Should print out whole table. (* means all columns.)

Chapter 3.3. Simple Select Statements (Basically same as examples in the text, although starts somewhat different.)

Example 3.3.1. Find (JUST) names of agents based in New York.

rel. alg.: (AGENTS where city = 'New York')[aname]

SQL: *select* aname *from* agents *where* city = 'New York';

Keywords: SELECT, FROM (clause), WHERE (clause), end with semi-colon (;).

After SELECT have *select list*, resulting variables to report: *project on*.

The FROM clause names tables to take Cartesian product.

The WHERE clause connects and limits rows in Cartesian product. No join. (SQL-92 now allows join operation: Cover later.)

Note duplicate aname values **ARE** possible. SQL statement above **WILL** report duplicate names (Relational rule broken) unless write:

select *distinct* aname from agents where city = 'New York';

If want to emphasize that duplicates can arise, give name to default:

select *all* aname from agents where city = 'New York';

Recall we did: select * from customers; at end of table load.

Select * means all columns (no projection). There is no where clause so whole table is chosen. Like rel alg query: CUSTOMERS

Example 3.3.3. Select product ids of products for which orders are placed.

```
select distinct pid from orders;
```

Might there be duplicates in this? How avoid duplicates? DISTINCT.

Example 3.3.4. Retrieve all (cname, pname) pairs where the customer places an order for the product. If don't use JOIN operator in SQL, can't make mistake of joining on city column as well. In RELALG, need:

```
(C[cid, cname]  JOIN O JOIN  P)[cname, pname]
```

Recall that C is an alias for CUSTOMERS, O for ORDERS and P for PRODUCTS. Rather:

```
((C X O X P)
   where C.cid = O.cid and O.pid = P.pid)[cname, pname]
```

In SQL:

```
select distinct cname, pname from customers c, orders o, products p
   where c.cid = o.cid and o.pid = p.pid;
```

Note that all table alias names must be specified in single statement where they are used, in FROM clause following table name without comma intervening.

In ORACLE and INFORMIX, look up "alias, table". In DB2 US, say "correlation name", and "alias" is something else. (Also in INFORMIX, "correlation name" is something else!)

SQL is non-procedural to the greatest extent possible, so a statement can't depend on an earlier statement. Note too, could leave out alias in this example — just saves typing here.

Need qualifier when column names are ambiguous from one table to another; not otherwise (above cname, pname, but o.cid, c.cid). But it never hurts to qualify column names.

Conceptual process in Select. (1) Take product of tables in from clause, (2) apply selection in where clause, (3) project on attributes in select-list and report.

But this in not necessarily (usually not) the actual order of events. Query optimization is applied. If customers has 1000 rows, orders 100,000, and products 1000, then Cartesian product has 100 billion rows. But eventual answer will have at most 100,000 rows.

Can make this much more efficient: index customers by cid and products by pid; now go through orders one row at a time and put out cname, pname into temporary file. Finally, sort cname, pname pairs and cast out duplicates.

This is what the query optimizer might do in its query plan. Basically, a query plan is like a program that retrieves the data you want. You don't have to worry about it. Only type SQL, query optimizer makes the program.

Can perform calculations in select.

Example 3.3.5. retrieve a table based on orders, with columns ordno, cid, aid, pid, and profit on order.

```
select ordno, x.cid, x.aid, x.pid, 0.40*(x.qty*p.price)
   - 0.01*(c.discnt + a.percent)*(x.qty*p.price) as profit
   from orders x, customers c, agents a, products p
   where c.cid = x.cid and a.aid = x.aid and p.pid = x.pid;
```

(Profit is quantity * price, where subtract off 60% of cost, discnt for customer, and percent commission for agent. Expressions allow: +, -, *, / **, and functions: upper( ) or ceil( ) (Oracle, smalles integer greater than float -- functions with numeric arguments are not part of standard).

Without the "as profit" clause, Oracle will head the column with the expression. After clause put in, heading will be "profit" In ORACLE, this is called a column alias, comparable to the way table aliases are created.

In INFORMIX, it is called a "Display Label".

Table alias is also called correlation name (in DB2) and range variable. Sometimes not just shorthand, but necessary, as when join table with itself (need different qualifying names).

Example 3.3.6. List all pairs of customer cids based in the same city.

We have seen this kind of thing before in Rel Alg. Need to report distinct pairs of cid only once.

```
select [distinct?] c1.cid, c2.cid from customers c1, customers c2
   where c1.city = c2.city and c1.cid < c2.cid;
```

Note, could be upper case C1 and C2 column aliases; lower case is our general rule, but not crucial; just need to be consistent (case counts).

Do we need to say distinct above, class? (No. Because (c1, c2) is a set of unique pairs. Waste to use distinct when not necessary.)

Alias idea is to make two copies of all content of customers table as c1 and c2; then take Cartesian product, perform selection, and finally report out project onto c1, c2.

But what system really does is (draw customers table, show c1 and c2 as variables ranging over rows, perform nested loop — RANGE VARIABLES):

```
FOR c1 FROM ROWS 1 TO LAST OF customers
   FOR c2 FROM ROWS 1 TO LAST OF customers
      IF (c1.city = c2.city and c1.cid < c2.cid)
         PRINT OUT SELECT-LIST VALUES:  c1.cid, c2.cid
   END FOR c2
END FOR c1
```

Once again c1 and c2 do not represent copied tables, but variables taking on row values in the same table! Trick here is to NOT MATERIALIZE THE PRODUCT. And now see why aliases are sometimes called range variables: we can picture them as variables that range over the rows of a table.

Class 7. Due Class 9: Exercises through end of Section 3.5.

Example 3.3.7. Find pids of products ordered by at least two customers.

NOT EASY! How to do? Think of idea of two range variables alive at same time in one table. Picture in orders tables. Say pid the same and cid different for two.

```
select [distinct?]  x1.pid from orders x1, orders x2
   where x1.pid = x2.pid and x1.cid <> x2.cid;
```

Need distinct? Yes. Picture Cartesian product, pid p01 once with three different pid values, p01, p02, p03. What happens? There are three pairs of rows that x1, x2 can fall on: (p01, p02), (p01, p03), and (p02, p03), AND the REVERSE!

Can at least reduce to three times by changing <> above to <. Changes nested loop, assuming n rows in orders from:

```
FOR x1 FROM ROWS 1 TO n OF orders   -- PICTURE RANGE VARIABLES
   FOR x2 FROM ROWS 1 TO n OF orders
```

to:

```
FOR x1 FROM ROW 1 TO n OF orders
   FOR x2 FROM ROW x1 TO n OF orders
```

Obviously saves effort, since less looping.

Example 3.3.8. Get cids of customers ordering a product for which an order is placed by agent a06. Reminiscent of an example we had in rel alg. Note that a cid retrieved doesn't necessarily order any product through a06.

```
                        /--        Customers        CREATE PICTURE OF RANGE
    / --     Products      ---        who              VARIABLES AND HOW
TO
  a06 ---  ordered by  ---  place orders              SOLVE THIS THINKING IN
    \ -- agent a06          ---  for those        TERMS OF QBE;  THAT
                        \--     products        TRANSLATES DIRECTLY
                                                TO SQL STATEMENT
```

TWO connections through orders relation, so need two aliases. First products ordered by a06 (first cloud above):

```
select [distinct?] x.pid from orders x where x.aid = 'a06';
```

Would need distinct if wanted unique x.pid values. Next, customers who order those products:

```
select distinct y.cid from orders x, orders y          NESTED LOOP OF
   where x.aid = 'a06' and y.pid = x.pid;  (SLOW)      RANGE VARIABLES
```

x.pid reports back the first cloud above, and set y.pid = x.pid to get second cloud y.cid.

But an even MORE straightforward way to do that is to use a subquery (Section 3.4):

```
select distinct y.cid from orders y where y.pid in
   (select x.pid from orders x where x.aid = 'a06');
```

The "subquery" retrieves the first cloud, and then the outer select retrieves all y.cid where y.pid is "in" the first cloud.

The condition "in" is an operator expecting a variable on the left and a "set" of values on the right (from a subquery); it is true when the variable on the left is equal to any of the elements retrieved by the subquery.

Of course the outer Select retrieves all rows for which the condition is true.

For the first time, we are using some recursion based on the idea that a select retrieves a table. Now we can operate on that table as a set of rows (or in the case just covered, a set of values).

NOTE: For any type of query we have learned so far, if we can do it with join we can do with subquery and vice-versa. Subqueries are supposed to give a slightly more natural approach to some queries. BUT I will insist that you be able to translate Subqueries to joins in most cases.

Example 3.4.1. Retrieve cids of customers who place orders with agents in Los Angeles or Dallas. First find all agents in LA or Dallas, and make that a subquery:

```
select cid from orders where aid in (select aid from agents
   where city = 'Los Angeles' or city = 'Dallas');
```

Note don't need to qualify aid in subquery use or outside subquery: in each case there is only one table that is natural.

Conceptually, innermost subquery operates first, returns a set of values, then outer select runs through customer rows and tests whether cid is in this set. In fact that might NOT be the way it is done by the system (after query optimization). Equivalent to:

```
select [distinct?] x.cid from orders x, agents a where
   x.aid = a.aid and (a.city = 'Los Angeles' or a.city = 'Dallas');
```

Why ( )s? Because of order of operations: p and q or r is (p and q) or r, but that isn't what we want; want p and (q or r).

Otherwise, if r were true for any p, q, whole statement true; then if agents has some agent in Dallas, will get all cids reported (won't need x.aid = a.aid).

Skip a few simple examples. In particular, can construct your own set using a condition such as: . . . where . . . city in ('Duluth', 'Dallas'). Of course we could also do this using or,

All these variant ways of doing things don't necessarily help — just make user uncomfortable about having a complete grasp of the SQL syntax.

OK, now idea of correlated subquery. Recall in 3.4.1 didn't need qualified names because rows had natural home. But can refer to OUTER range variable from INNER subquery.

Example 3.4.4. (Variant) Find names of customers with discnt >= 10 who order product p05. To illustrate a point, using subquery:

```
select cname from customers where discnt >= 10 and
   'p05' in (select pid from orders where cid = customers.cid);
```

Yes, unusual to say where <const> in subquery (looks like a typo), but perfectly valid and as long as condition is true will retrieve cname.

Note that in subquery, the cid without a qualifier has home qualifier of orders, while customers.cid harks back to outer select. As if we wrote:

```
FOR c FROM ROW 1 TO LAST OF customers
   discard c if c.discnt < 10
```

```
    create L as empty list
    FOR x FROM ROW 1 TO LAST OF orders
      Add x.pid to list L if x.cid = c.cid    /* value set outside loop */
    END FOR x
    discard c if 'p05' not in list L
    otherwise place c.cname in ANSWER list
  END FOR c
  print out ANSWER list
```

Note can't precalculate the inner loop; need the outer loop to be set (This is called a correlated subquery). Of course, we could change query to:

```
  select cname from customers where discnt >= 10 and
    cid in (select cid from orders where pid = 'p05');
```

Can evaluate this with precalculation of subquery.

```
  create L as empty list
  FOR x FROM ROWS 1 TO LAST OF orders
    Add x.cid to list L if x.pid = 'p05'
  END FOR x
  FOR c FROM ROWS 1 TO LAST OF customers
    if (c.discnt < 10 or if c.cid not in list L)
      discard c and go to next in loop
    else place c.cname in ANSWER list
  END FOR c
  print out ANSWER list
```

OK, can refer to outer variable from inner loop. But can't do the REVERSE!!

Example 3.4.5. To get names of customers who order product p07 from agent a03, can't use query:

```
  select cname from customers where orders.aid = 'a03' and
    'p07' in (select pid from orders where cid = customers.cid);
  ***ILLEGAL***
```

The condition orders.aid = 'a03' can't be used outside subquery (doesn't exist yet -- scoping rule)

(** new **) Example. Find cids of customers who do not place any order through agent 'a03'. Can use not in, true exactly when in is false:

```
select cid from customers where cid not in (select cid from
   orders where orders.aid = 'a03'); /* example of minus operator */
```

Example 3.4.6. In older products, IN predicate only works for single values. Can't have condition ". . . where (cid, aid) in (select cid, aid from orders" . . .

This is part of the full SQL-92 standard (Note need (,)), and Oracle permits it.

Quantified Comparison Predicates. List:

=some	**<>some**	**<=some**	**>some**	**<some**	**>=some**
=any	**<>any**	**<=any**	**>any**	**<any**	**>=any**
=all	**<>all**	**<=all**	**>all**	**<all**	**>=all**

If comparison operator $\propto$ stands for operator in the set {<, <=. =, <>, >, >=},

v $\propto$some (subquery) is TRUE iff v $\propto$ s for **some** s in the subquery
v $\propto$any (subquery) is TRUE iff v $\propto$ s for **some** s in the subquery
v $\propto$all (subquery) is TRUE iff v $\propto$ s is true for **all** s in the subquery.

$\propto$some and $\propto$any mean same thing: $\propto$some is easier to remember.

Example 3.4.7. Find aid values of agents with a minimum percent commission. This was HARD in relational algebra. Now easy.

```
select aid from agents where percent <= (select percent from agents);
```

Class 8.

Exam 1 on Class 13 Responsible for most of Chapter 3.

OK, last time talking about Quantified Comparison Predicates Some, Any, or All. (You are responsible for this nomenclature, here and as in book.)

Example 3.4.8. Find all cids of customers who have the same discount as any customer in Dallas or Boston.

```
select cid from customers where discnt =some (select
   discnt from customers where city = 'Dallas' or city = 'Boston');
```

OK, =some same as =any or in. Could do away with in. But <some, etc. new.

New Example. Find cids of customers who do not place any order through agent 'a03'. Used not in, above:

```
select cid from customers where cid not in (select cid from
   orders where orders.aid = 'a03');
```

Since =any is same as in, is it true that <>any is same as not in?

```
select cid from customers where cid <>any (select cid from
   orders where orders.aid = 'a03');
```

NO! Doesn't work! <>any is true if cid = 'c001' and subquery is {'c001', 'c003'}. There IS ONE value it is not equal to. Really mean <>all. The reverse of =any is <>all. Exercise on this.

By the way, what do you think happens if you write:

```
select cid from customers where cid <> (select cid from
   orders where orders.aid = 'a03');
```

Just <>, not <>any or <>all? Does it even make sense if single value on left, set on right? Answer is, perfectly legal if set on right contains exactly one element. Otherwise, runtime error.

By the way, what happens if cid >= subquery, and subquery is an empty set? Is the result TRUE or FALSE? The answer is, result is UNKNOWN: almost like false, but not quite. More later.

Example 3.4.9. Get cid values of customers with discnt smaller than discnt of any customer who lives in Duluth. (Is the following right?)

```
select cid from customers where discnt <any
  (select discnt from customers where city = 'Duluth');
```

NO! WRONG EFFECT! **In English** here, "smaller than any" means "smaller than all". Why in SQL-92 replace any with some (less ambiguous).

The Exists predicate.

exists(subquery) is true iff subquery is a non-empty set.
not exists(subquery) is true iff subquery is an empty set.

Use not exists most often to handle FOR ALL conditions.

Example 3.4.10. Find cnames of customers who place an order through a05.

```
select cname from customers c where exists (select * from orders x
  where x.cid = c.cid and x.aid = 'a05');
```

OK, that works, but also:

```
select c.cname from customers c, orders x
  where x.cid = c.cid and x.aid = 'a05';
```

Interesting. Don't say anything explicit about x, don't say, "select c.cname if there exists an order x connecting c to 'a05'," but that's what we mean.

Picture this in terms of range variables on tables: x must exist connecting c.cid to 'a05'.

If we don't retrieve a column from a table range variable, that range variable is called unbound (predicate logic sense -- an unbound variable), and there is an implicit "exists" presumption that makes the result true

Same as saying range variables range independently and retrieve all rows of product where condition is true.

This basically means that to use the form "Exists" is unnecessary. Don't use Exists if don't need it.

not exists, on the other hand, is a different matter.

Example 3.4.12. Retrieve all customer names where the customer does not place an order through 'a05'. Recall that this doesn't work:

```
select c.cname from customers c, orders x
   where not (c.cid = x.cid and x.aid = 'a05');  *** WRONG EFFECT ***
```

Because customers c where c.cid = 'c001' and orders x where x.cid = 'c001' and x.aid = 'a01', so condition is false, not is true, and 'c001' is retrieved. But c001 does order through a05.

THERE IS NO WAY TO DO THIS WITHOUT A SUBQUERY! Want:

```
select c.cname from customers c where not exists
   (select * from orders x where c.cid = x.cid and x.aid = 'a05');
```

Recall we did something like this above also, with not in:

```
select cname from customers where cid not in
   (select cid from orders where aid = 'a05');
```

Is not exists more powerful than not in?

Example. Find all cid, aid pairs where the customer does not place an order through the agent:

```
select cid, aid from customers c, agents a where not exists
   (select * from orders x where x.cid = c.cid and x.aid = a.aid);
```

Note not in USED TO assume a simple value, so couldn't write:

```
select cid, aid from customers c, agents a where (cid, aid) not in
   (select cid, aid from orders x);
```

As a result, not exists was able to do things not in wasn't. But this is no longer true (SQL-92, Oracle V7.1, . . .)

Section 3.5. How to do FOR ALL (Very Hard) and Union (easy)

Example 3.5.2. Find cids of customers who place orders with ALL agents based in New York. In rel alg:

```
ORDERS[cid, aid]  DIVIDEBY  (AGENTS where city = 'New York')[aid]
```

Harder in SQL. Formulate as follows.

Want to retrieve cid such that FOR ALL agents based in New York, there EXISTS an order connecting cid to agent.

But we don't have a FORALL predicate. If we did, we'd like to write in SQL:

```
select cid from customers c where
   FORALL(select aid from agents a where city = 'New York',
   EXISTS(select * from orders x where x.aid = a.aid and
      c.cid = x.cid));
```

Where FORALL(subquery, condition) is true iff for all elements in the sub-query, the condition is true. (Being explicit about EXISTS here for a reason.)

OK, no FORALL predicate, so instead have to use a trick. FORALL (x, p) is equivalent to NOT EXISTS (x, NOT p) (there is no counterexample which would cause FORALL to fail).

In our case, we have FORALL(x, EXISTS (y, p)) is equivalent to NOT EXISTS (x, NOT EXISTS (y, p)). So statement above becomes:

```
select cid from customers c where
   not exists (select aid from agents a where city = 'New York' and
   not exists (select * from orders x where x.aid = a.aid and
      c.cid = x.cid));
```

Look at form above again for FORALL. In general, queries which obey a FORALL condition, have this form:

select . . . where not exists(select . . . where not exists(select . . .);

Hardest query you will be asked to pose. Last hard trick need to finish homework.

New Example. Get aids of agents who place orders for all customers who have discount greater than 8.

Assume we are thinking of retrieving aid for an agent that we're not sure fits the bill. How would we specify a counter-example using the exists concept? There exists something that would rule out aid. (Class?)

(Answer: There exists a customer with discnt > 8 for which aid does not place an order.)

Good: that would rule out aid. Assume we are inside a subquery, with a.aid specifying the outer aid to rule out: think of it as a constant like 'a02'. How would you select a customer cid with discnt > 8 for which a02 does not place an order?

```
select cid from customers c where c.discnt > 8 and  (Class?)
   not exists(select * from orders x where x.cid = c.cid
   and x.aid = 'a02');
```

Right? OK, now we want to make that x.aid = a.aid, where a.aid is in an outer select, and say that NO COUNTER-EXAMPLE CID EXISTS!

```
select aid from agents a where NO COUNTER-EXAMPLE EXISTS!

select aid from agents a where not exists ( subquery above, with
   'a02' replaced by a.aid);

select aid from agents a where not exists
   (select cid from customers c where c.discnt > 8 and
      not exists(select * from orders x where x.cid = c.cid
      and x.aid = a.aid);
```

When have to do FORALL: (Go through stepson pg. 110).

Example 3.5.5. Find cids for customers who order all products ordered by customer c006.

OK, we are retrieving ?.cid. (Don't know from what table, customers or orders) How disqualify it? (1) Give Counter example in English? (Class?)

There is a product ordered by customer c006 that our candidate customer ?.cid does not order.

(2) Select all counter examples. Start you off:

```
select pid from orders x where x.cid = 'c006' and
  -- products ordered by 'c006' and (Class?)
not exists (select * from orders y where x.pid= y.pid and y.cid= ?.cid);
```

(3) OK, now say no counter-example exists: not exists (subquery above)

(4) Let the row we want to retrieve be range variable in outer select with no counter-examples. (Usually choose range variable in smallest table.)

```
select c.cid from customers c where not exists
   (select pid from orders x where x.cid = 'c006' and not exists
   (select * from orders y where x.pid = y.pid and y.cid = c.cid));
```

OK, subtraction and union in Section 3.5 are very easy. Next time I will go on to 3.6.

Class 9.

Next hw, all undotted exercises through end of Chapter 3. Due class 13.

Exam 1, Class 14. Responsible for ALL of Chapter 2 and 3 (except no detail on Chapters 3.10 and 3.11 — Responsible for all homework and concepts covered in class).

Section 3.6

Start with Union, Difference, Intersect: how to do in SQL. We intend to add new operators to perform these operations in this Section. Already covered Union in Section 3.5.

These operators are NOT uniformly available in the different products.

New Example. List all cities that contain the products costing less than $1.00 or that contain agents who have made no sales.

It is easy to get cities associated with products or cities associated with agents. But you should see that no simple query of the form:

```
retrieve x.city from table1 x, table2 y, . . .
```

Is going to work, since x.city comes from one table only. Might conceivably do this by retrieving:

```
retrieve x.city, y.city from table1 x, table2 y, . . .
```

But then will be retrieving pairs, and that messes us up. What will we retrieve if there are NO cities of the second kind? (Class?) Nothing.

This is easy using UNION of relational algebra, and it turns out that standard SQL has a union form THAT ALL VENDORS HAVE IMPLEMENTED. Answer

```
select city from products where price < 1.00
union
select city from agents where aid not in
   (select aid from orders);
```

Now consider the *Subquery form*. Two forms only, Subquery and Select statement. Subquery is a select sql syntax that will go in (subquery).

Later, a Select statement adds a clause (ORDER BY) and cannot appear in (subquery). But until then, Subquery and Select statement are identical.

A Subquery form doesn't HAVE to appear as a subquery — just a prescription for a statement format. It can be an outer Select statement.

So here is the format for UNION, INTERSECT, and EXCEPT (relational algebra DIFFERENCE or MINUS)

```
subquery {UNION [ALL] | INTERSECT [ALL] | EXCEPT [ALL]}
```

Figure 3.10 Advanced SQL Subquery Form

Note that few products have implemented all of these possibilities; UNION [ALL] is part of Entry SQL-92 and implemented by all serious products.

Idea of ALL in these three cases. UNION ALL can cause several copies of a row to exist after being applied to multiple subqueries. Takes number into account.

So can have 3 rows to intersect with 2. Result : 2 (If use ALL!! One otherwise.) Can have 2 rows to subtract from 3. Result : 1
(If don't use ALL, would get result 0.)

We have already encountered the concept of subtraction using not exists.

Example 3.4.14. Find cids of customers who do not place any order through agent a03.

```
select cid from customers c where not exists
   (select * from orders where cid = c.cid and aid = 'a03');
```

This includes customers who have placed no orders. If we had asked for customers who placed some order but none through a03, would change customers c above to orders x.

Now the way we would write this with our new operators is:

```
select cid from customers except
   select cid from orders where cid = c.cid and aid = 'a03';
```

Recall, idea of too many equivalent forms. This makes it worse. ALL is a new capability, but is it ever wanted?

How would we implement "intersect"? Get list of cities that contain products AND agents.

```
select city from products where city in (select city from agents);
```

With use of INTERSECT, this becomes:

```
select city from products intersect select city from agents;
```

The CORRESPONDING clause is used to select a set of equivalently named columns on which to perform intersection (lazy about select_list).

OK, now expanded definition of FROM Clause. SQL-89 Definition was:

```
from tablename [[AS] corr_name] {, tablename [[AS] corr_name]...}
```

Implementation of these operators.

ORACLE Release 8 provides UNION, UNION ALL, INTERSECT and MINUS (variant of EXCEPT), but not INTERESECT ALL or MINUS ALL and provides no CORRESPONDING Clause.

DB2 Universal Database, Version 5, implements {UNION | INTERSECT | EXCEPT} [ALL], but no CORRESPONDING Clause.

INFORMIX basically implements only UNION [ALL] and recommends workarounds for the other operators.

To define SQL-92 FROM Clause general form, start with def of tableref:

```
-----------------------------------------------------------------
tableref::= tablename [[AS] corr_name] [(colname {, colname ...})]]
  | (subquery) [[AS] corr_name] [(colname {, colname ...})]
  | tableref1 [INNER | {LEFT | RIGHT | FULL} OUTER ]
          JOIN tableref2
     [ON search_condition | USING (colname {, colname . . .})]

FROM clause ::= FROM tableref {, tableref . . .}
-----------------------------------------------------------------
```

Figure 3.11 SQL-99 Recursive tableref Definition

Note that tableref is recursive: tableref can be any kind of join of two tablerefs.

Let's start with the fact that one can Select FROM a Subquery.

Example 3.6.3. Retrieve all customer names where the customer places at least two orders for the same product. Here's a new way:

```
select cname from
   (select o.cid as spcid from orders o, orders x
   where o.cid = x.cid and o.pid = x.pid and o.ordno <> x.ordno) as y,
   customers c where y.spcid = c.cid;
```

Note alias y for Subquery, alias spcid for column o.cid.

Foreshadowing in this query form of something called Views.

Next, talk about the various Join operators on lines 3 & 4 (and 5)

Can say: tableref1 CROSS JOIN tableref2

```
select cname from customers c cross join orders o where . . .
```

Same effect as

```
select cname from customers c, orders o where . . .
```

OK, now if we only use the JOIN keyword in the general form above, we can use one of the forms on line 5, either the ON search_condition or USING.

We call it a condition join when two different named columns are to be compared in an ON clause. E.g. if had cities table with columns named cityname, latitude and longitude, could write:

```
select city, latitude, longitude from customers c join cities x
   on (c.city = x.cityname);
```

OK, next is NATURAL JOIN.

```
select cname from customers natural join orders;
```

Just like in relational algebra (natural, equijoin). Can already do this:

```
select cname from customers c, orders o where c.cid = o.cid;
```

NATURAL JOIN uses all commonly named columns to force an equality match on the two argument tables.

Example 3.6.4 Retrieve all customers who purchased at least one product costing less than $0.50. In relational algebra, THIS is wrong:

```
((ORDERS JOIN PRODUCTS where price < 0.50) JOIN CUSTOMERS) [cname]
```

(Why, class?) Because cities of PRODUCTS and CUSTOMERS will match, becomes: Retrieve customers who purchased at least one product costing less than $0.50 that is stored in the same city as the customer.

Correct this with following form:

((ORDERS JOIN (PRODUCTS where price <0.50) [pid]) JOIN CUSTOMERS) [cname]

In SQL, using INNER JOIN, the default JOIN operator:

```
select cname from ((orders join
   (select pid from products where price < 0.60) p using (pid) ) op
      join customers using (cid);
```

or here's a condition join:

```
select cname
   from ((orders join
      (select pid from products where price < 0.60) p on o.pid = p.pid) op
      join customers c on op.cid = c.cid);
```

A USING clause limits the columns equated in a JOIN to a set of commonly named ones (i.e., same column name in both tables involved in the join.)

Since city is not named in the using clause or ON clause, we don't run into the trouble we had with the city column in relational algebra.

OK, next is **OUTER JOIN.**

Covered this in Chapter 2. Idea is that T1 FULL OUTER JOIN T2 (must say FULL, or LEFT or RIGHT, can't just say "OUTER JOIN"), if T1 has rows that don't match in the Join they are included in the answer with nulls for the columns from T2, and vice-versa.

We also have LEFT OUTER JOIN and RIGHT OUTER JOIN. LEFT means that will only preserver rows from the left-hand table in result, RIGHT analogously.

INNER JOIN means opposite of OUTER, what we have been using without thinking all this time: INNER is the default of OUTER and UNION JOIN.

UNION JOIN is something new, but unclear how useful. Kin of a generalization of OUTER JOIN. Say have two tables S and T

S

C	A
c1	a1
c3	a3
c4	a4

T

A	B
a1	b1
a2	b2
a3	b3

The FULL OUTER JOIN of S and T would give the following:

select * from S FULL OUTER JOIN T

S.C	S.A	T.B
c1	a1	b1
c3	a3	b3
c4	a4	null
null	a2	b2

But this is UNION JOIN. Don't even try to match.

select * from S FULL OUTER JOIN T

S.C	S.A	T.A	T.B
c1	a1	null	null
c3	a3	null	null

c4	a4	null	null
null	null	a1	b1
null	null	a2	b2
null	null	a3	b3

Join Forms in operational Database Systems

ORACLE Release 8 provides only Left and Right Outer Join, specified in Where Clause. General form:

SELECT . . . FROM T1, T2 WHERE {T1.c1 [(+)] = T2.c2 | T1.c1 = T2.c2 [(+)]};

Note only on one side. Plus in parentheses following the table means it will ACCEPT nulls in the other table -- will PRESERVE rows in other table.

Class 10.

Collect hw through 3.7. Next hw, all undotted exercises through end of Chapter 3

Exam 1, Class 14, Wed Oct 22. Responsible for ALL of Chapter 2 and 3 (except no detail on Chapters 3.10 and 3.11 — Responsible for all homework and concepts covered in class).

Section 3.6. Set Functions in SQL. count, max, min, sum, avg. Operates on sets of values, returns a single value (all except count, which can operate on sets of rows).

```
select sum(dollars) from orders where aid = 'a01';
```

Returns total dollar sales of agent a01 (note where clause gets applied first!!!). Looks like:

COL1
1400.00

Don't get confused, not like:

```
select sqrt(qty) from orders;
```

. . . which must return one value for each row in orders, operates on one row at a time. A Set function can operate on a SET of rows.

Note relational alg couldn't find the total dollar sales of an agent.

Lots of different terminology. X/OPEN and INGRES refer to set functions, ANSI says aggregate functions (aggregate: to bring together a number of objects into a single measure), ORACLE has group functions, DB2 says column functions. We say **set functions**. Here are standard ones.

Name	Argument type	Result type	Description
Count	any (can be *)	numeric	count of occurrences
sum	numeric	numeric	sum of arguments
avg	numeric	numeric	average of arguments
max	char or numeric	same as arg	maximum value

min	char or numeric	same as arg	minimum value

Figure 3.12 The set functions in SQL (pg. 124)

ORACLE has a few more, stddev and variance:

```
select variance(observations) from experiment;
select stddev(observations) from experiment;
```

Note max and min return char max and min.

```
select min(city) from products;
```

Note avg(x) has same value as sum(x)/count(x); probably more efficient.

Note that these three MIGHT NOT give same values:

```
select count(*) from customers;
select count(cid) from customers;      /* null values not counted      * /
     /* but cid not null in customers  * /
select count(city) from customers;     /* only if no null city values  * /
```

Surprising? Doesn't insist on different city values. But:

```
select count(distinct city) from customers;
```

is how we get count of distinct city names. Could also do:

```
select avg(distinct dollars) from orders;
```

But it would be VERY UNUSUAL if you really wanted this.

It is **not legal** to use an aggregate function directly in a where clause. E.g., try to list all customers with maximum discount.

```
select cid, cname from customers c where c.discnt = max(c.discnt);
/*** NOT LEGAL ***/
```

Problem is that range variable c is only supposed to range once over customers, not once outside and once inside max( ).

There is only one range variable in this expression and we need two, and need to evaluate max( ) first. How solve this? (Class?)

```
select cid, cname from customers where discnt =
   (select max(discnt) from customers);
```

Now customers inside subquery ranges separately from outside. Why can we use = above, rather than =any or =all? Because only one value returned!

REMEMBER THIS: MUST USE SUBQUERY IF SET FUNCTION IN WHERE!

Example 3.7.6. Find products ordered by at least two customers. Previously:

```
select distinct c1.pid from orders c1, orders c2
   where c1.pid = c2.pid and c1.cid < c2.cid;
```

New way (more efficient, generalizes to more than 2 without increasing number of alias joins):

```
select pid from products p where 2 <= (select count(distinct cid)
   from orders where pid = p.pid);
```

Handling Null Values.

Recall that a null value appears as a column value in a row when the value is either unknown (discnt for new customer) or inapplicable (employee manager for company president).

In passing, note that there is a proposal to have two different kinds of null values for these two cases.

If insert a row (insert statement, to come) without specifying some column values, nulls will be placed for those values

Unless column definition in create table specifies not null as for cid in Appendix A, pg. 724 — Then the insert statement will not work. Ex 3.7.7:

```
insert into customers (cid, cname, city)
   values ('c007', 'Windix', 'Dallas');
```

The discnt value is not specified, so will be placed as null value. Note that it is NOT usually possible with current products to specify null as value for discnt (OK in SQL-92 standard).

A null value has IMPORTANT implications. Two following are different:

```
select count(*) from customers;

select count(*) from customers where (discnt < 8 or discnt >= 8);
```

Why? Because null values for discnt will not be selected in second statement, even though the condition seems to be exhaustive.

A null value in any comparison expression evaluates to UNKNOWN, rather than TRUE or FALSE. In a Select statement, only rows for which the where condition is TRUE are retrieved. (See pg. 143 for reason for UNKNOWN.)

This means that the null value of an integer type variable cannot be kept simply as some value pattern, because all patterns are taken up by real integer values. Need special FLAG byte for a column to see when it is null.

Some older products didn't have null, represented unknown numeric values by zero and unknown char values by '' (null string). Obviously that doesn't have the right properties because of set functions.

Note another important property: the set functions IGNORE null values. If we write:

```
select avg(dollars) from orders where aid = 'a01';
```
or
```
select sum(dollars)/count(dollars) from orders where aid = 'a02';
```

and some rows have UNKNOWN dollars values, then the count, sum, and avg functions will all skip over those values. If the values were zero instead of null, clearly the avg would be lower.

Class 11.

Homework due Class 13, Monday, Oct. 20. Solutions online that evening. Exam 1, Wednesday, Oct. 22.

Section 3.8. Groups of rows.

SQL allows a select statement to serve a kind of report function. Groups rows of a table based on a common values and performs aggregate functions on rows grouped. E.g.

```
select pid, sum(qty) from orders group by pid;
```

New GROUP BY clause. Print out as if following logic was followed:

```
FOR EACH DISTINCT VALUE v OF pid IN orders;
    select pid, sum(qty) from orders where pid = v;
END FOR;
```

See pg. 129 for table printed out. Note, in Select can't include anything in target-list that is not single-valued for the groups created in the GROUP BY clause.

```
select pid, aid, sum(qty) from orders group by pid;  ** ILLEGAL **
```

This is because a single pid in orders may be associated with multiple aid values. However, we can have more finely aggregated groups: we can GROUP BY aid and pid both:

```
select pid, aid, sum(qty) as total from orders group by pid, aid;
```

This has the effect of the loop:

```
FOR EACH DISTINCT pair  of values (v, w) equal to (pid, aid) in orders
    select pid, aid, sum(qty) from orders where pid = v and aid = w;
END FOR;
```

See table retrieved in Example 3.8.1, pg. 130 in text.

Now a surprise! If say:

```
select p.pid, pname, sum(qty) from orders o, products p
    where o.pid = p.pid group by p.pid;
```

It won't work! **WHY?!** (Class.) Even though pname is uniquely valued for each group (unique pid value), SQL doesn't recognize that.

This is true even though we define in Create Table that pid is primary key. Must say:

```
select p.pid, pname, sum(qty) from orders o, products p
    where o.pid = p.pid group by p.pid, pname;
```

Note that we can have a WHERE clause at the same time as a GROUP BY:

```
select pid, sum(qty) from orders where aid = 'a03'
    group by pid;
```

ORDER: (1) Take Cartesian product; (2) cross out rows not selected in WHERE clause; (3) Group remaining rows in accordance with GROUP BY clause; (4) evaluate expressions in target list (aggregate function values depend on groups).

Would see contributions from four rows of the table on pg. 130, where aid = a03, but two of these are both for product p03 and would be combined in this result.

Now. What if want to eliminate rows where sum(qty) is too small in:

```
select pid, sum(qty) from orders group by pid;
```

Can't do it by saying:

```
select pid, sum(qty) from orders
    where sum(qty) >= 1000 group by pid;  ** ILLEGAL **
```

Because not allowed to use set function in WHERE clause. Also, Where acts before grouping occurs and long before target-list expressions evaluated.

Need to invent a new clause to act as Where after grouped quantities have been calculated: called a Having clause:

```
select pid, sum(qty) from orders
```

```
group by pid having sum(qty) >= 1000 ;
```

So now: (5) Eliminate rows in target-list that do not obey the HAVING clause requirements.

(*) Note this is a kluge: Very much not recursive: can't GROUP BY the result of a GROUP BY after a HAVING. But might want to.

Note that ANSI SQL-92 allows us to Select FROM a Subquery, thus can GROUP BY again on a second pass. But can't do this yet on andy product.

A HAVING restriction need not refer to a set function actually in the target list. A HAVING clause is just used to restrict after a GROUP BY.

Example 3.8.4 (variant). Get ids of all products purchased by more than two customers.

```
select pid from orders group by pid having count(distinct cid) > 2;
```

It certainly feels risky to mention cid in group by pid, but count() makes it single-valued. This is an even more efficient Select than 3.7.6.

```
select pid from products p where 2 <= (select count(distinct cid)
    from orders where pid = p.pid);
```

New method with HAVING clause retrieves from only one table, but old method retrieves from two and correlated subquery does a join.

Section 3.9. Now complete description of Select statement, pg. 135. Remember: idea is to give you confidence — everything has been covered.

But you must make special effort now to gain that confidence for yourself. VERY close to text here, probably no need for own notes.

Subquery General Form, see pg. 135.

```
SELECT [ALL|DISTINCT] expr [[AS] c_alias] {, expr [[AS] c_alias]}
  FROM tableref {, tabletrf}
  [WHERE search_condition]
  [GROUP BY column {, column}]
  [HAVING search_condition]
  | subquery [UNION [ALL] | INTERSECT [ALL] | EXCEPT [ALL]]
```

[CORRESPONDING [BY] (colname {, colname . . .})] subquery;

Recall tableref in FROM clause was defined in Section 3.6, Fig. 3.11, pg. 117. (BUT NOT ALL SYNTAX IS PRESENT IN ANY PRODUCT.) The most basic SQL form (supported by all databases) is just

Tableref := tablename [corr_name]

Square brackets means don't have to have the item. WHERE, GROUP BY, etc, all optional.

Thus [ALL|DISTINCT]: the phrase is optional; if we specify, we must choose exactly one form, ALL (duplicates allowed in target-list) or DISTINCT (no duplicates. But since ALL is underlined, it is the default, so if the phrase is not specified, it will be as if ALL was specified.

search_condition in the WHERE clause is a complex object; most of what follows gives details on this, all the various Subqueries and predicates, with a few new ones.

OK, this is a **Subquery** form: means can occur in subquery (part of search_condition), also part of full select as follows.

Full Select General Form

subquery
[ORDER BY result_col [ASC|DESC] {, result_col [ASC|DESC]}]

The ORDER BY clause is new, and allows us to order the rows output by a succession of result-column values, leftmost first.

Explain dictionary order. Note that result_col can be a number, referring to numbered col in the target list.

In a UNION, etc. of different Subqueries, we do not have to assume that corresponding columns have the same qualified names throughout.

(Although we could use column aliases for all retrieved rows, same for all subqueries.)

Therefore, the result_col in this case can be one of the column numbers 1 through n, where n columns occur in the result. (Breaks rel rule.)

Note that in the [ASC|DESC] choice (ascending order, or descending order), ASC is the default.

Now everything in a Subquery comes before ORDER BY and the order of clauses in a Subquery carries over to the conceptual order of evaluation.

Reasonable, since an ordering of rows by column values is clearly a final step before display.

See Figure 3.15. Conceptual order of evaluation of a Select statement.

- First the Cartesian product of all tables in the FROM clause is formed.
- From this, rows not satisfying the WHERE condition are eliminated.
- The remaining rows are grouped in accordance with the GROUP BY clause.
- Groups not satisfying the HAVING clause are then eliminated.
- The expressions of the Select clause target list are evaluated.
- If the key word DISTINCT is present, duplicate rows are now eliminated.
- The UNION, INTERSECT, EXCEPT is taken after each subquery is evaluated.
- Finally, the set of all selected rows is sorted if an ORDER BY is present.

Example 3.9.1. List all customers, agents, and the dollar sales for pairs of customers and agents, and order the result from largest to smallest sales totals. Retain only those pairs for which the dollar amount is at least 900.00.

```
select c.cname, c.cid, a.aname, a.aid,  sum(o.dollars)
   from customers c, orders o, agents a  where c.cid = o.cid and o.aid = a.aid
      group by  c.cname, c.cid, a.aname, a.aid
      having sum(o.dollars)  >= 900.00
      order by  5  desc;
```

Example 3.11.6 (from later). Create table called employees (see pg. 159 of text).

```
  create table employees (eid char(4) not null, ename varchar(16),
     mgrid  char(4));
```

Draw tree. Want to select everyone below some node, e.g. employee with eid = 'e001', in tree.

```
     select e.eid from employees e where e.mgrid chain
          goes up to 'e001';
```

Can't, can only retrieve employees one level down:

```
select e.eid from employees where e.mgrid = 'e001';
```

or two levels,

```
select e.eid from employees where e.mgrid in
    (select f.eid from employees f where f.mgrid = 'e001'); (typo)
```

or . . .. Can't leave number of levels flexible. In program, of course, can walk the tree.

This capability is spec'd in SQL99 (called Recursive Queries). (**New**)

Have this capability in Oracle now! But not with new standard syntax--it predates that standard.

Create employees table as on page 159. Now to perform a depth first search of the reports to tree starting with the President, Jacqueline:

```
select ename, eid, mgrid from employees
  start with eid = 'e001'
  connect by prior eid = mgrid;
```

Added to next homework: Try this on Oracle.

Now expressions, predicates and search_condition. Start with expressions: numeric value expressions, string value expression, datetime expression, interval value expression, and conditional expression.

```
expr = numexpr | strvexpr | datexpr | intvexpr | condexpr
```

numexpr, arithmetic stuff, Figure 3.16a, pg 137. Defined very much like a programming language expression, except allow columnname (like variable) and set_function(numexpr).

An numexpr can be a constant, or columname, or qualifier.columname or (recursively): numexpr arith_op numexpr, (numexpr), function(numexpr), set_function(numexpr)

Similarly for strvexpr, Fig. 3.13b. Concatenate two strings with a || (some products use + like string concatenation in Java and Basic).

I am going to skip other types for now. Note data types for columns defined in Appendix A.3.

Arith functions in most products, Fig 3.17. abs(n), mod(n,b), sqrt(n); NOT standardized in SQL-99.

Char functions ARE standardized, but not all products yet follow the standard, see Figure 3.18:

CHAR_LENGTH(str)
SUBSTRING(strval FROM start [FOR length])
TRIM({LEADING|TRAILING|BOTH} char FROM strval)
POSITION(str1 IN str2)
UPPER(strval), LOWER(strval)

Note that companies sometimes have unusual capitalization: is it TipTop or Tiptop? It is common to store only uppercase, retrieve in that form.

If sometimes caps are important, store column c1 with Upper/lower, retrieve asking for upper(c1) = 'TIPTOP' (but this can be inefficient since can't use index easily — could also create second Uppercase column).

Now, Predicates: Evaluate to TRUE, FALSE, or UNKNOWN (not yet in notes)

comparison predicate:	expr1 q expr2 | expr1 q (Subquery)
between predicate:	c.discnt between 10 and 12
in predicate:	expr [not] in (subquery)
Quantified predicate:	expr q[all|any|some] (subquery)
exists predicate;	exists(subquery)
is null predicate:	column is [not] null
like predicate	columname [not] like 'pattern'

Can we have more than one predicate involving a subquery. x not in (subquery) and y <some (subquery). Answer: No problem.

Class 12.

Homework 3 due next time (one week). Exam on Class 13 (Monday, 10/21). Returned in one week. Hand out Practice Exam.

We are going through a list of Predicates allowed for SQL. Subqueries next. Read these carefully, will ask Exam Questions about this.

Comparison predicate: e.g., expr1 < (expr2 | subquery)

Subquery must return at most ONE value (maybe ZERO), or must use quantified predicate. If empty set, result of comparison is U, UNKNOWN

Question. In comparison predicate is it necessary to always place comparison predicate first and predicate involving subquery last? Yes.

Can we have more than one predicate involving a subquery? I thought there was a restriction for some time. x not in (subquery) and y <some (subquery). Answer: No problem.

Note that for not equal, we commonly use (<>) — this is the most standard — also can be indicated by (!=) or (^=) on many database systems.

TRUTH VALUES: T, F, and U

A row is ruled out in a WHERE clause if the search_condition is not T, i.e., if it is F or U. However, U is not equivalent to F.

Look at AND, OR, NOT operator definitions on pg 144, Fig. 3.21.

AND	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

OR	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

NOT	
T	F
F	T
U	U

U just acts like there's doubt, might be T or F, and remains U in result if doubt remains. But of course F AND U is not in doubt: it results in F, since this is the case whether U is T or U is F.

But now why do we need U? Why can't we get along with T and F? Consider the query to retrieve orders made by customers whose city come after 'M' in the alphabet.

```
select * from orders o where 'M' < (select city from customers c
    where c.cid = o.cid);
```

This would seem to have the same result (by our rules of what not means) as:

```
select * from orders o where not( 'M' >= (select city from customers c
    where c.cid = o.cid));
```

But look at the first query above. What if we have an orders row in the outer select so we are retrieving a customer city with c.cid = o.cid where the city is not filled in (is null)? What do we want to do in the query? Retrieve the order or not? (Class?)

No, because we only want to retrieve the order if the customer city name OBEYS some property.

OK, but does that mean that the row for orders in the first query above doesn't get retrieved because 'M' < (select city from customers c where c.cid = o.cid) is FALSE?

But then isn't it also false that 'M' >= (select city from customers c where c.cid = o.cid)? There is no city, so the property is not OBEYED.

But if we assume F in the case with "'M' >=. . .", then in the second query (not('M' >=. . .), not(F) is T, and we DO retrieve the orders row.

Something wrong here. Answer is, as we've said, that 'M' >= (subquery), when the subquery is empty, is U, UNKNOWN, not F. And not(U) is U. Don't retrieve orders row unless get T, so U is a more sticky non T value than F.

The Between Predicate.

```
expr1 [not] between expr2 and expr 3
```

Conceptually equivalent to (without not):

```
expr1 >= expr2 and expr1 <= expr3
```

No good reason for existing; orginally just more efficient to evaluate (didn't trust query optimizer to notice the second was a range between).

Quantified Predicate.

expr q[some | all | any] (subquery)

Seen before. Only tricky part is that expr: qall (subquery) is TRUE if subquery is empty, and expr qsome (subquery) (or qany (subquery))is FALSE if subquery is empty.

Idea is that qall is TRUE if there is no COUNTER-EXAMPLE, but for μany to be true, need at least one EXAMPLE.

Ex. 3.9.3. Retrieve maximum discount of all customers.

```
select discnt from customers c where discnt >=all
    (select discnt from customers d where d.cid <> c.cid);
```

Never mind if you'd do it that way. It's valid in most cases. But if there were only a single customer row? Empty set. Yes, still OK. But only if >=all is true for empty set in subquery.

Skip In predicate (identical to =any), Exists predicate.

Is null Predicate.

columname is [not] null

NOT valid to say: columname = null or <> null. SQL won't get it.

Like predicate.

columname [not] like *'pattern'*

Build up patterns using normal characters and wildcards. Like UNIX file wildcards: ls *.c (* stands for any string)

Underscore (_)	any single character
Percent (%)	zero or more characters of any form (UNIX *)
Plus (+)	is an "escape character"
All other chars	represent themselves

```
select * from customers c where city like 'N_w%k';
```

'New York' and 'Newark' are OK, but not 'Nome' or 'Novow Sibirsk'.

In Oracle, if use % or _ twice in a row, it means REALLY '%' or "_'. Thus,

```
select * from customers where city like 'New_ _%';
```

Will retrieve New_Rochelle, or New_Caledonia, but not New York.

Escape character in INGRES (and UNIX) is backslash (\); to say we REALLY MEAN %, we write . . .\%. . . In DB2, '+%' means REALLY the character %. To type REALLY + , need to type '++'.

Section 3.10.

Insert, Update, and Delete statements perform data modifications to existing tables. Called as a group the "update statements". Need to tell difference between an "updata statement" and the Update statement.

Need update privilege on a table to update it, different than read privilege. You always have all privileges if you created the table.

Section 3.10 The Insert, Update and Delete Statements.

```
insert into tablename [(column {, column...})]
    {values (expr | null {expr | null...})] | subquery};
```

One of the two forms must be used, values ... or subquery. Note that null itself does not qualify as an expression, so it has to be explicitly allowed as an alternative here. Columns named and expressions must match in number and type.

Example 3.10.1.

```
insert into orders (ordno, month, cid, aid, pid)
    values('1107', 'aug', 'c006', 'a04', 'p01');
```

Notice, no qty or dollars, so on this new row they are null.

But if we are inserting ALL the columns in a table, can leave out column list (note it's optional?)

See Example 3.10.2 for other form. I can create new table swcusts, like customers, then write:

```
insert into swcusts select * from customers where
    city in ('Dallas', 'Austin');
```

I can insert a LOT at ONCE, build up arbitrarily from existing tables using select. This is an important capability!

The Update statement. Basic SQL form is this.

```
update tablename [corr_name]
    set colname = {expr | null | (subquery)} {, {column = expr |null |
(subquery)}...}
        [where search_condition];
```

The search_condition in the Where clause determines rows in the updated table to change. The expressions used can reference only column values on the specific row of the table currently being updated.

```
update agents set percent = 1.1*percent where city = 'New York';
```

With the (subquery) option we can reference values from other rows, as we did with a subquery in insert. However here it must be enclosed in parentheses.

```
update swcusts set discnt = (select discnt from customers c
    where c.cid = swcusts.cid);
```

Delete statement.

```
delete from tablename
    [where search_condition];
```

E.g., Fire all agents in New York.

```
delete from agents where city = 'New York';
```

Once again SQL-92 allows search condition to reference other tables.

Ex. Delete all agents with total orders less than $600.00

```
delete from agents where aid in (select aid from orders group by
```

```
aid having sum(dollars) < 600);
```