Lecture Notes for Database Systems, part 2

by Patrick E. O'Neil

Class 1.

Last term covered Chapters 2-6; this term cover Chapters 7-11.

I will assume everything in text through Chapter 6. Ad-hoc SQL (Chap. 3), O-R SQL (Chap. 4), Embedded SQL (Chap. 5), Database Design (Chap. 6). **Note** solved and unsolved Exercises at end of chapters.

Starting on Chapter 7. Homework assignment online. Dotted problem solutions are at end of Chapter.

OUTLINE. Review what is to be covered this term. Chapter 7-11.

7. DBA commands. Use examples from commercial products and own Basic SQL standard, also X/Open, ANSI SQL-92, SQL-99.

Start 7.1, Integrity Constraints, generally as part of Create Table command. (Rules for an Enterprise from Logical Design: review Chapter 6.)

7.2, Views: virtual tables to make it easier for DBA to set up data for access by users. Base tables vs. View tables.

7.3, Security. Who gets to access data, how access is controlled by DBA.

7.4, System catalogs. If you come to a new system, new database, how figure out what tables are, what columns mean, etc. All info carried in catalogs, relational data that tells about data (sometimes called Metadata).

Chapter 8. Introduction to physical structure and Indexing. Records on disk are like books in a library, and indexes are like old style card catalogues.

Generally, what we are trying to save with indexing is disk accesses (very slow -- analogous to walking to library shelves to get books).

Lots of different types of indexing for different products, not covered at all by any standards. Somewhat complex but fun topic.

Chapter 9. Query Optimization. How the Query Optimizer actually creates a Query plan to access the data. Basic access steps it can use.

Just as an example, sequential prefetch. Join algorithms. How to read Query Execution Plans. *Important skill when we cover thisis learning to add quickly and make numeric estimates of how many I/Os are required*.

Query Optimization is a VERY complex topic. All examples in text from MVS DB2. Will try to add examples from other vendor products.

End of Chapter 9, see Query performance measured in Set Query benchmark.

Chapter 10. Update Transactions. Transactional histories. Performance value of *concurrent execution*. Concurrency theory.

ACID properties, transactions make guarantees to programmer. Atomic, Consistent, Isolated, and Durable. TPC-A benchmark.

Transactional Recovery. Idea is that if the system crashes, want to save results. Transactions are all-or-nothing, like transfer of money. Don't want to be chopped off in the middle.

Chapter 11, Parallel and Distributed databases. We’ll see how much we can do on this.

Chapter 7.1 Integrity Constraints

Section 7.1: Integrity constraints. Review Chapter 6, Database Design, from last term, since concepts are applied here in Chapter 7.

Idea in Chapter 6 was that DBA performs Logical database design, analyzing an enterprise to be computerized:

o listing the data items to be kept track of
o the rules of interrelatedness of these data items (FDs)
o apportioning the data items to different tables (E-R or Normalization)

After do this, Chapter 7 tells you how to actually construct the tables and load them. Build rules into table so SQL update statements *can't break the rules* of interrelatedness (and other rules we'll come up with). Call this a

faithful representation

The way to achieve this is with **constraint clauses** in a Create Table statement. Show Basic SQL to reflect ORACLE, DB2 UDB, and INFORMIX.

See pg 415, for the E-R diagram of our CAP database.

Entities: Customers, Agents, Products, and Orders. Attributes of entities; Concentrate on customers: look at card( ) designations:

o cid (primary key attribute, min-card 1 so must exist, max-card 1 so at most one cid per entity instance)
o discnt (min-card 0, so can enter a record with no discnt known, null value allowed). And so on . . .

Here is Create Table statement of last term (with primary key constraint):

```
create table customers (cid char(4) not null, cname varchar(13),
   city varchar(20), discnt real check(discnt <= 15.0), primary key(cid));
```

Three constraints here: not null for cid (also primary key cid, which implies not null), check discnt <= 15.00 (not allowed to go over 15.00).

Primary key clause implies unique AND not null. If say primary, don't say unique. Old products required not null with primary key, still the most robust approach.

Could also declare cid (cid char(4) not null unique) if it were just any old candidate key rather than a primary key.

IF **ANY RULE IS BROKEN** by new row in customers resulting from SQL update statement (Insert, Delete, and Update), update won't "take" — will fail and give error condition. Not true for load, however.

Class 2.

Covering idea of Create Table constraint clauses & faithful representation.

Again: Relationships in Figure 7.2, pg. 415. Each of C, A, P is related to O.

Customers requests Orders, max-card(Customers, requests) = N; can have Many links out of each Customer instance, but max-card(Orders, requests) = 1, only one into each order: *single-valued participation*. Many-One, N-1.

Which side is Many? Orders! Side that has single-valued participation! One customer places MANY orders, reverse not true. The MANY side is the side that can contain a foreign key in a relational table!

Representation in relational model? N-1 relationship represented by foreign key in orders referencing primary key in customers.

```
create table orders ( ordno integer not null, month char(3),
   cid char(4) not null, aid char(3) not null,
   pid char(3) not null, qty integer not null check(qty >= 0),
   dollars float default 0.0 check(dollars >= 0.0),
   primary key ( ordno ),
   foreign key (cid) references customers,
   foreign key (aid) references agents,
   foreign key (pid) references products);
```

In Create Table for orders, see ordno is a primary key, and at end says cid is foreign key references customers. The implication is that cid in orders must match a primary key value in customers (name needn't be cid).

If we wanted to match with a candidate key, colname2, could instead say:

```
foreign key (colname1) references tablename (colname2)
```

Can also have larger tuples matching:

```
Create table employees ( . . .
foreign key (cityst, staddr, zip) references ziptst(cityst, staddr, zip);
```

Now with this FOREIGN KEY . . . REFERENCES clause for cid in orders table, if try to insert an orders row with cid value that isn't in customers, insert will fail and give an error condition.

For general form of Create Table command, see pg 411, Figure 7.1. This is our Basic SQL standard. (See first page of Chapter 3 for definition of Basic SQL)

Put on board one section at a time. Here is Create Table specification block:

Figure 7.1. Clauses of the Create Table command.

```
CREATE TABLE tablename
   ((colname datatype [DEFAULT {default_constant | NULL}]
          [col_constr {col_constr. . .}]
      | table_constr
   {, {colname datatype [DEFAULT {default_constant | NULL}]
          [col_constr {col_constr. . .}]
      | table_constr}
   . . .});
```

Recall CAPS means literal. Cover typographical conventions, Chap 3, pg 85

Start by naming the table. Then list of column names OR table constraints.

The column names are given with datatype and possibly a DEFAULT clause: specifies a value the system will supply if an SQL Insert statement does not furnish a value. (Load command is not constrained to provide this value.)

Column Constraints can be thought of as shorthand for Table Constraints. Table Constraints are a bit more flexible (except for NOT NULL).

Note that both constraints have "constraintname" for later reference. Can DROP both using later Alter Table, can only ADD Table Constraint (not CC).

Can tell difference between the two: Column Constraints stay with colname datatype definition, no separating comma. See Create Table orders on board.

Table Constraint has separating comma (like comma definition). CC keeps constraint near object constrained, TC might be far away in large table.

Def 7.1.2. Column Constraints.

The col_constr form that constrains a single column value follows:

```
{NOT NULL |
[CONSTRAINT constraintname]
   UNIQUE
   | PRIMARY KEY
   | CHECK (search_cond)
   | REFERENCES tablename [(colname) ]
         [ON DELETE CASCADE]}
```

The NOT NULL condition has already been explained (means min-card = 1, mandatory participation of attribute. Doesn't get a constraintname If NOT NULL appears, default clause can't specify null.

A column constraint is either NOT NULL or one of the others (UNIQUE, etc.) optionally prefixed by a "CONSTRAINT constraintname" clause. The final "}" of the syntax form matches with the "{" before the NOT NULL and the first "|", although admittedly this is ambiguous, since the other "|"'s lie at the same relative position.

A more careful specification would be:

```
{NOT NULL |
[CONSTRAINT constraintname]
   {UNIQUE
   | PRIMARY KEY
   | CHECK (search_cond)
   | REFERENCES tablename [(colname) ]
         [ON DELETE CASCADE]}}
```

Constraint names can go before ANY SINGLE ONE of the following. (Repeated col_contr elements are allowed in Create Table specification block.)

Either UNIQUE or PRIMARY KEY can be specified, but not both. Not null unique means candidate key.

UNIQUE is possible without NOT NULL, then multiple nulls are possible but non-null values must all be unique. UNIQUE NOT NULL means candidate key.

CHECK clause defines search_condition that must be true for new rows in the current table, e.g. qty >= 0. (Only allowed to reference own column?)

Can only reference constants and the value of this column on the single row being inserted/updated to perform a check for a column.

```
   Create table orders ( . . . ,
      cid char(4) not null check (cid in (select cid from customers),  INVALID
```

The REFERENCES clause means values in this column must appear as one of the values in the tablename referenced, a column declared unique in that table. A non-unique column won't work, but nulls are OK.

Multi-column equivalent, use table_constr: FOREIGN KEY . . . REFERENCES. Column constraint "references" is just a shorthand for single column name.

The optional ON DELETE CASCADE clause says that when a row in the referenced table is deleted that is being referenced by rows in the referencing table, then those rows in the referencing table are deleted!

If missing, default "RESTRICTs" delete of a referenced row (disallows it).

Def 7.1.3. Table Constraints.

The table_constr form that constrains multiple columns at once follows:

```
   [CONSTRAINT constraintname]
      {UNIQUE (colname {, colname. . .})
      | PRIMARY KEY (colname {, colname. . .})
      | CHECK (search_condition)
      | FOREIGN KEY (colname {, colname. . .})
            REFERENCES tablename [(colname {, colname. . .})]
               [ON DELETE CASCADE]}
```

The UNIQUE clause is the same as UNIQUE for column, but can name a set of columns in combination. It is possible to have null values in some of the columns, but sets of rows with no nulls must be unique in combination.

UNIQUE multi-column, all columns declared NOT NULL is what we mean by a Candidate Key. If have multiple column candidate key, (c1, c2, . . ., cK), must define each column NOT NULL & table constr. unique (c1, c2, . . ., cK)

The PRIMARY KEY clause specifies a non-empty set of columns to be a primary key. Literally, this means that a FOREIGN KEY . . . REFERENCES clause will refer to this set of columns by default if no columns named.

Every column that participates in a primary key is implicitly defined NOT NULL. We can specify per column too. There can be at most one PRIMARY KEY clause in Create Table. UNIQUE cannot also be written for this combination.

The CHECK clause can only be a *restriction condition*: can only reference other column values ON THE SAME ROW in search_condition.

A FOREIGN KEY . . . REFERENCES clause. The foreign key CAN contain nulls in some of its columns, but if all non-null than must match referenced table column values in some row.

Optional ON DELETE CASCADE means same here as in Column Constraint.

Class 3.

Example, employees works_on projects from Figure 6.8

```
create table employees (eid char(3) not null, staddr varchar(20), . . .
   primary key (eid));

create table projects (prid char (3) not null, proj_name varchar(16),
   due_date char(8), primary key (prid);

create table works_on (eid char(3) not null, prid char(3) not null,
   percent real, foreign key (eid) references employees,
   foreign key (prid) references projects, unique (eid, prid) );
```

Note there is no primary key (only candidate key) for works_on; none needed since not target of referential integrity. Could make (eid, prid) primary key. (Some practitioners think it's awful to have table without primary key.)

Another example, from Figure 6.7, Employees manages Employees. (TEXT)

```
create table employees (eid char(3) not null, ename varchar(24),
   mgrid char(3), primary key (eid),
   foreign key (mgrid) references employees);
```

A foreign key in a table **<u>can reference</u>** a primary key **<u>in the same table</u>**.

What do we need to do for Normalization of Ch. 6? All FDs should be dependent on primary key for table (OK: rows have **<u>unique</u>** key columns).

Recall idea of Lossless Decomposition: this is what we need Referential Integrity (foreign key) for. To be lossless, intersection of two tables (on which join) must be **<u>unique</u>** on one of them! Must reference to primary key!

Create Table Statement in ORACLE, see Figure 7.3 (pg 423).

Extra disk storage and other clauses; all products have this: disk storage will be covered in next chapter.

Create Table AS SUBQUERY: table can be created as Subquery from existing table. Don't **<u>need</u>** any column names, datatypes, etc.: inherit from Select.

ORACLE has ENABLE and DISABLE clauses for constraints. Can define table with named constraint Disabled, later Enable it (after load table).

Sometimes can't load tables if have all constraints working: e.g. tables for boys and girls at a dance, each must have partner from other table. A referential integrity constraint that must fail when insert first row.

Therefore Create Table with constraint DISABLE'd. Later use Alter Table to ENABLE it. Nice idea, since Constraint stays in Catalog Table.

Problem that this is not portable, so we depend on ADDing such a constraint later with Alter Table (can't define it early if load by Insert).

INFORMIX and DB2 Create Table

INFORMIX has minor variation as of this version: all column definitions must precede table constraint definitions (old way), can't be mixed in any order.

DB2, **see Figure 7.6, pg 421**. Adds to FOREIGN KEY . . . REFERENCES.

```
FOREIGN KEY
   (colname {, colname}) REFERENCES
   tablename [ON DELETE [NO ACTION | CASCADE | SET NULL | RESTRICT]]
```

What if delete a row in customers that cid values in orders reference?

NO ACTION, RESTRICT means delete of customers row won't take while foreign keys reference it. (Subtle difference between the two.)
SET NULL means delete of customers row will work and referencing cid values in orders will be set to null.
CASCADE means delete of customers will work and will also delete referencing orders rows. Note that this might imply cascading (recursive) deletes, if other foreign key references column in orders.

Default is NO ACTION. Presumed by products that don't offer the choice. See Fig. 7.6 for standard options in X/Open, Full SQL-99 is same as DB2.

Referential Integrity

See pg. 419, Definition 7.1.4. We define an ordered set of columns F to make up a Foreign key in table T1 to match with an ordered set of columns P in table T2. (foreign key . . . references . . .)

A referential integrity constraint is in force if the columns of a foreign key F in any row of T1 must either (1) have a null value in at least one column that permits null values, or (2) have no null values and equate to the values of a primary key P on some row of T2.

Thus if we want "optional participation" in a relationship from the referencing table, must allow at least one column of F to be nullable.

Example 7.1.4. Use Referential Integrity to define an enumerated Domain.

```
create table cities(city varchar(20) not null,
    primary key (city) );
  create table customers (cid char(4) not null, cname varchar(13),
    city varchar(20), discnt real check(discnt <= 15.0),
    primary key (cid), foreign key city references cities);
```

Idea here is that cityname can't appear as city column of customers unless it also appears in cities. List all valid cities in first table, impose an enumerated domain. (Note it takes more effort to check.)

I will skip explaining rest of Referential Integrity subsection in class; read it on your own (nothing startling).

The Alter Table Statement

With Alter Table statement, can change the structure of an existing table. Must be owner of table or have Alter privileges (other privileges as well).

The earlier standards didn’t cover the Alter Table statement, and the Alter table statement in SQL-92 is too general.

So there are lots of differences between products in Alter Table. We cover the differences, and don’t attempt a Basic SQL form to bring them together.

ORACLE Alter Table, Fig. 7.7, pg. 423.

```
ALTER TABLE tblname
  [ADD ({colname datatype [DEFAULT {default_const|NULL}]
    [col_constr {col_constr. . .}]
    | table_constr}      -- choice of colname-def. or table_constr
    {, . . .})]       -- zero or more added colname-defs. or table_constrs.
   [DROP {COLUMN columnname | (columnname {, columnname…})}]
  [MODIFY (columnname data-type
       [DEFAULT {default_const|NULL}] [[NOT] NULL]
    {, . . .})]       -- zero or more added colname-defs.
  [DROP CONSTRAINT constr_name]
  [DROP PRIMARY KEY]
  [disk storage and other clauses (not covered, or deferred)]
  [any clause above can be repeated, in any order]
  [ENABLE and DISABLE clauses for constraints];
```

Figure 7.7 ORACLE Standard for Alter Table Syntax

Can ADD a column with column constraints, ADD named table constraints. (Can't ADD col constrs., but just shorthand for table constraints.)

Can MODIFY column to new definition: data type change from varchar to varchar2, give new set of column_constr's (Can't do in other products).

Can DROP column (in version 8.1.5), named constraint or primary key. ENABLE and DISABLE clauses as mentioned before in Create Table, but we don't use them (portability).

When will CHECK constraint allow ANY search condition? Not in Core SQL-99 but a named feature. No product has it. Can do referential integrity.

DB2 UDB Alter Table, Fig. 7.8, pg. 423. What differences from ORACLE? Can't DROP or MODIFY a column. This may be added, so check your current documentation.

Note both ADD arbitrary number of columns or table_constrs (LIMIT exists). But ORACLE requires parens around colname|table_constr list. DB2 UDB just repeats ADD phrase to set off different ones.

INFORMIX Alter Table, Fig. 7.9, pg. 423. Can't MODIFY column as in ORACLE. Informix does allow us to DROP column.

Non-Procedural and Procedural Integrity Constraints

Note that non-procedural constraints are "data-like"; easy to look up and understand, unlike code logic (arbitrarily complex), but also lack power.

For more flexibility, procedural constraints: Triggers. Provided by Sybase first (T-SQL), DB2 UDB, Oracle, Informix and Microsoft SQL Server.

Triggers weren't in SQL-92, but now in SQL-99. **Fig. 7.10, pg 425.**

```
CREATE TRIGGER trigger_name BEFORE | AFTER
   {INSERT | DELETE | UPDATE [OF colname {, colname...}]}
   ON tablename [REFERENCING corr_name_def {, corr_name_def...}]
   [FOR EACH ROW | FOR EACH STATEMENT]
      [WHEN (search_condition)]
      {statement                    --    action (single statement)
      | BEGIN ATOMIC statement; { statement;...} END} -- action (mult. stmts.)
```

The corr_name_def in the REFERENCING clause looks like:

```
   {OLD [ROW] [AS] old_row_corr_name
   | NEW [ROW] [AS] new_row_corr_name
   | OLD TABLE [AS] old_table_corr_name
   | NEW TABLE [AS] new_table_corr_name}
```

Can also use command: DROP TRIGGER trigger_name;

The trigger is *fired* (executed) either BEFORE or AFTER one of the events {INSERT | DELETE | UPDATE [of colname {, colname . . .}]} to the NAMED TABLE.

When trigger is fired, optional WHEN search_cond (if present) is executed to determine if the current subset of rows affected should execute the multi-statement block starting with BEGIN ATOMIC. (Example below for ORACLE.)

Jump to [FOR EACH ROW | FOR EACH STATEMENT]; Update statement might update multiple rows, so default fires only once for multiple row Update.

REFERENCING gives names to old row/new row (or old table/new table), so know which is referenced in program block.

Row_corr_name also used as table aliases in WHERE clause (in customers, old_row_corr_name x, then x.discnt is old value of discnt on row.)

CANNOT use row_corr_name on FOR EACH STATEMENT. (So FOR EACH ROW is more commonly used in practice.) Table_corr_name is used in FROM clause.

Note: you shouldn't try to put any other table into this WHEN search_cond – not even with a Subquery.

(E.g., if Update customers in Texas, fire trigger and search for customers with discnt > 5, only act on rows with AND of those two; if update adds 1 to discnt, clearly important if we are talking aboout old-corr-names or new.)

Either have single statement of a block of statements between BEGIN ATOMIC and END that executes when triggered; can refer to corr_names.

Can write procedural program using SQL-99 procedural language SQL/PSM (stands for "Persistent Stored Module").

ORACLE uses PL/SQL (BEGIN and END as in usual PL/SQL progrms, Chapter 4). DB2 has no full procedural language but invents simple one for triggers.

Now ORACLE Create Trigger statement. There will be homework on this.

```
CREATE [OR REPLACE] TRIGGER trigger_name {BEFORE | AFTER | INSTEAD OF}
   {INSERT | DELETE | UPDATE [OF columnname {, columnname...}]}
   ON tablename [REFERENCING corr_name_def {, corr_name_def...}]
   {FOR EACH ROW | FOR EACH STATEMENT}
      [WHEN (search_condition)]
      BEGIN statement {statement; . . .} END;
```

The corr_name_def that provides row correlation names follows:

```
   {OLD old_row_corr_name
   | NEW new_row_corr_name}
```

Figure 7.12 ORACLE Create Trigger Syntax

Note differences from SQL-99. Can REPLACE old named trigger with a new one. Trigger action can occur INSTEAD OF the INSERT, DELETE, or UPDATE.

Don't have TABLE corr_name options, only ROW. Therefore don't have access to old table in AFTER trigger.

ORACLE Examples using PL/SQL for procedures – see Chapter 4.

Example 7.1.5
Use an ORACLE trigger to CHECK the discnt value of a new customers row does not exceed 15.0.

```
create trigger discnt_max after insert on customers
```

```
   referencing new x
   for each row when (x.discnt > 15.0)
      begin
         raise_application_error(-20003, 'invalid discount on insert');
      end;
/
```

Enter this in SQL*Plus, need "/" after it to make it work. Note that Trigger is long-lived, but could save Create Trigger to .cmd file as with loadcaps.

The raise_application_error is special form in some products to return error; here –20003 is an application error number programmers can assign.

Note if want to guarantee constraint with Update statement as well, need to change AFTER clause of Example 7.1.5 "after insert or update".

Example 7.1.6
We use an ORACLE trigger to implement a referential integrity policy "on delete set null" in the customers-orders foreign key.

```
create trigger foreigncid after delete on customers
  referencing old ocust
  for each row                    -- no WHEN clause
  -- PL/SQL form starts here
  begin
    update orders set cid = null where cid = :ocust.cid;
  end;
```

This trigger works as a BEFORE trigger, but according to ORACLE documentation, AFTER triggers run faster in general.

OK now Pros and Cons of Non-Procedural and Procedural constraints starting pg. 437. What is the value of having these constraints?

Stops SQL update statement from making a mistake, that might corrupt the data. Referential integrity is a RULE of the business, and one error destroys the "integrity" of the data.

Greatest danger is ad-hoc update statement. Could make rule that ad-hoc updates are not allowed, leave integrity to programmers. But what if inexperienced programmer, subtle bug?

OR we could create LAYER of calls, everyone has to access data through call layer, and it makes necessary tests to guarantee RULES. E.g., logic checks referential integrity holds. Will have an exercise to do that. Talk about.

One problem is that this must be a CUSTOM LAYER for a shop; some shop's programming staff won't know enough to do this right.

There is another issue of control. Managers often afraid of what programmers might do, and want to see the reins of control in a separate place.

This "Fear, Uncertainty, and Doubt" argument about programmers was a main way in which constraints were originally sold to business managers.

But there is real value to having a limited number of non-procedural constraints, with a few parameters: UNIQUE, FOREIGN KEY . . . REFERENCES.

Acts like DATA! Can place all constraints in tables (System Catalogs), check that they're all there. Not just LOGIC (hard to grasp) but DATA.

But problem we've met before. Lots of RULEs we'd like that can't be put in terms of the non-procedural constraints we've listed: not enough POWER!

I point out two weaknesses of non-procedural constraints: (1) Specifying constraint failure alternative, (2) Guaranteeing transactional Consistency.

<u>Constraint failure alternative.</u> E.g., try to add new order and constraint fails - misspelling of cid. Now what do we do with the order, assuming we have the wrong zip code for a new prospect address? Throw it away?

Programmer has to handle this, and it's a pain to handle it on every return from application using SQL. (But we can handle this using Triggers now.)

<u>Transactional Consistency.</u> There is an accounting need to have Assets = Liabilities + Proprietorship (Owners Equity). Lots of RULES are implied.

RULE: in money transfer, money is neither created nor destroyed. In midst of transaction, may create or destroy money within single accounts, but must balance out at commit time.

But there is no constraint (even with triggers) to guarantee consistency of a transaction at commit time. And this can be a VERY hard problem.

So we **have** to trust programmers for these hard-to-constrain problems: Why try to second-guess them for the easy problems.

The answer probably is that non-procedural constraints can solve some easy things and thus simplify things for programmers. Most programmers would support the idea of “data-like constraints”.

Example 7.1.7. Here we assume each CAP with orders having a corresponding set of line-items in a table lineitems with foreign key ordnum. The order is originally inserted with n_items set to zero, and we keep the number current as new lineitem rows are inserted for a given order.

```
create trigger incordernitems after insert on lineitems
  referencing old oldli
    for each row
    begin          --for ORACLE, leave out for DB2 UDB
      update orders set n_items = n_items + 1 where ordno = :oldli.ordno;
    end;           --for ORACLE
create trigger decordernitems after delete on lineitems
  referencing old  oldli
    for each row
    begin
      update orders set n_items = n_items - 1 where ordno = :oldli.ordno;
    end;
```

Class 5.

7.2 Views Idea is that since a Select statement looks like a Virtual Table, want to be able to use this table in FROM clause of other Select.

(Can do this already in advanced SQL: allow Subquery in the FROM clause; check you can find this in Section 3.6, Figure 3.11, pg 117)

A "View table" or just "View" makes the Select a long-lived virtual table: no data storage in its own right, just window on data it selects from.

Value: simplify what programmer needs to know about, allow us to create virtual versions of obsolete tables so program logic doesn't have to change, add a field security capability.

Base table is created table with real rows on disk. Weakness of view is that it is not like a base table in every respect (limits to view Updates.)

Here is example view, **Example 7.2.1**:

```
create view agentorders (ordno, month, cid, aid, pid, qty,
   charge, aname, acity, percent)
   as select o.ordno, o.month, o.cid, o.aid, o.pid, o.qty,
      o.dollars, a.aname, a.city, a.percent
      from orders o, agents a where o.aid = a.aid;
```

Now can write query (**Example 7.2.2**):

```
select sum(charge) from agentorders where acity = 'Toledo';
```

System basically turns this query into

```
select sum(o.dollars) from orders o, agents a
   where o.aid = a.aid and a.city = 'Toledo';
```

Syntax in Figure 7.13, pg. 434.

```
CREATE VIEW view_name [(colname {,colname…})]
   AS subquery [WITH CHECK OPTION];
```

Can leave out list of colnames if target list of subquery has colnames that require no qualifiers. Need to give names for expressions or for ambiguous colnames (c.city vs a.city). Can rename column in any way at all.

Aliases for expressions will give names to expressions in DB2 UDB and ORACLE (not INFORMIX at present), so don't need colnames in Create View.

Recall that Subquery, defined at the beginning of Section 3.9, DOES allow UNION (check this is in your notes, class), does **not** allow ORDER BY.

The WITH CHECK OPTION will not permit an update or insert (through a legally updatable view) that would be invisible in the resulting view.

Example 7.2.4. Assume we had no CHECK clause on discnt when we created customers. Can create updatable view custs that has this effect.

```
  create view custs as select * from customers
     where discnt <= 15.0 with check option;
```

Now cannot insert/update a row into custs with discnt > 15.0 that will take on table customers. This will fail for customers cid = 'c002' (discnt = 12):

```
  update custs set discnt = discnt + 4.0;
```

Can nest views, create a view depending on other views. **Example 7.2.5**:

```
create view acorders (ordno, month, cid, aid, pid, qty,
     dollars, aname, cname)
  as select ordno, month, ao.cid as cid, aid, pid, qty,
        charge, aname, cname
     from agentorders ao, customers c where ao.cid = c.cid;
```

Listing views. All objects of a database are listed in the catalog tables (section 7.4). Can list all base tables, all views, all named columns, often all constraints (or named constraints, anyway).

In X/Open standard, would list views as follows:

```
  select tablename from info_schem.tables where table_type = 'VIEW';
```

In ORACLE: select view_name from user_views;

Once you have a view_name (or table_name from user_tables) can write in ORACLE:

```
   DESCRIBE {view_name | table_name};
```

To find out exactly how the view/table was defined in ORACLE:

```
 select text from user_views where view_name = 'AGENTORDERS';
   (NOTE: name must be in Upper Case)
```

If you find you don't get the whole view definition (it's chopped off), use the ORACLE statement: set long 1000, then try again.

In DB2 UDB, way to list views for user eoneil is:

```
   select viewname from syscat.views where definer = 'EONEIL';
```

To delete a view, or table (or later, index), standard statement is:

```
   DROP {TABLE tablename | VIEW viewname};
```

When table or view is dropped, what happens to other objects (views, constraints, triggers) whose definitions depend on the table or view?

By default in ORACLE, if drop table or view, other views and triggers that depend on it are marked invalid.

Later, if the table or view is recreated with the same relevant column names, those views and triggers can become usable again.

With [CASCADE CONSTRAINTS] in ORACLE, constraints referring to this table (e.g., foreign key references) are dropped. (See Figure 7.14.)

If there is no such clause, then the Drop fails (RESTRICT, effect named in INFORMIX and SQL-99).

In INFORMIX have [CASCADE | RESTRICT]; refers to constraints.and also nesting views.

In DB2, no options. Invalidates views & triggers depending on what dropped; retain definition in system catalog, but must be recreated later.

Updatable and Read-Only Views.

The problem is that a View is the result of Select, maybe a join. How do we translate updates on the View into changes on the base tables?

To be updatable (as opposed to a read-only), a view must have the following limitations in its view definition in the X/Open standard (Fig. 7.15, pg 445):

(1) The FROM clause must have only a single table (if view, updatable).
(2) Neither the GROUP BY nor HAVING clause is present.
(3) The DISTINCT keyword is not present.
(4) The WHERE clause has no Subquery referencing a table in FROM clause.
(5) Result columns are simple: no expressions, no col appears > once.

For example, if disobey (3), the distinct clause is present, will select one of two identical rows. Now if update that row in the view, which one gets updated in base tables? Both? Not completely clear what is wanted.

(Update customer Smith to have larger discnt. Two customers with identical names, discnts in View, but different cid (not in view). Inexperienced programmer might update through this view, not differentiate).

Limitations (1) means ONLY ONE TABLE INVOLVED in view. Example 7.2.6 shows why might want this, because of view colocated:

```
create view colocated as select cid, cname, aid, aname, a.city as acity
   from customers c, agents a, where c.city = a.city;
```

Example rows (from Chapt 2). Have:

```
c002  Basics   a06   Smith  Dallas
```
and
```
c003  Allied   a06   Smith  Dallas
```

Consider delete of second row. How achieve? Delete c003? change city of c003 (to what?). Delete a06? Then delete first row of view as well.

NOT CLEAR WHAT REQUESTOR WANTS!! SIDE-EFFECTS might be unexpected.

Problem of GROUP BY (2). Example 7.2.8. Recall agentsales view:

```
create view agentsales (aid, totsales) as select aid, sum(dollars)
```

```
      from orders group by aid;
```

Now consider update statement:

```
   update agents set totsales = totsales + 1000.0 where aid = 'a03';
```

How achieve this? Add new sale of $1000.0? To whom? What product? Change amounts of current orders? Which ones? By how much?

AGAIN NOT CLEAR WHAT USER WANTS.

But limitations are too severe as they stand. Assume that agentsales contained agent city (with aid) and we wanted to update the city for aid= 'a03'. Clear what user wants but NOT ALLOWED by rules of Fig. 7.15.

The Value of Views

Original idea was to provide a way to simplify queries for unsophisticated users (allow libararian access to student records) and maintain old obsolete tables so code would continue to work.

But if need to do updates, can't join tables. Not as useful as it should be.

Homework on when updates work on ORACLE joined tables. Also as SQL-99 special feature (not in Core) and in ODBC.

In ORACLE can update join views if join is N-1 and table on N side has primary key defined (lossless join).

Still no function operators (Set or Scalar) or other complex expression result columns, GROUP BY, or DISTINCT; Here's the idea in N-1 Join.

Consider view ordsxagents (assuming agents has primary key aid):

```
   create view ordsxagents as
      select ordno, cid, x.aid as aid, pid, dollars, aname, percent
         from orders x, agents a where x.aid = a.aid and dollars > 500.00;
```

Can update all columns from the orders table (the N side of the N-1 join) not in join condition (x.aid), but none from agents (1 side of the N-1 join).

If give the following command to list the updatable columns, will get:

```
   select column_name, updatable from user_updatable_columns
      where table_name = 'ORDSXAGENTS';

COLUMN_NAME          UPD
---------------      ---
CID                  YES NOTE: can update only columns in table
AID                  NO           with single-valued participation, i.e..
PID                  YES orders, except columns involved in join.
DOLLARS              YES
ANAME                NO
PERCENT              NO
ORDNO                YES
7 rows selected.
```

Class 6.

7.3 Security. Basic SQL (X/Open) Standard. pg 443

```
   GRANT {ALL PRIVILEGES | privilege {,privilege…}}
      on [TABLE] tablename | viewname
      TO {PUBLIC | user-name {, user-name…}} [WITH GRANT OPTION]
```

(Oracle adds role-name in TO list.)

privileges are: SELECT, DELETE, INSERT, UPDATE [(colname {, colname})], REFERENCES [(colname {, colname})] (this grants the right to reference the specified columns from a foreign key)

The WITH GRANT OPTION means user can grant other user these same priviliges.

The owner of a table automatically has all priviliges, and they cannot be revoked. Example 7.3.2 (that I might give):

```
   grant select, update, insert on orders to eoneil;
   grant all privileges on products to eoneil;
```

Then eoneil can write:

```
   select * from poneil.orders;
```

The poneil. prefix is called the schema name -- See pg. 453 in text.

You can access any user's tables in the same database (all of us are in the same database) if you have appropriate priviliges.

Can limit user select access to specific columns of a base table by creating a View and then granting access to the view but NOT the underlying base table. Example 7.3.3

```
   create view custview as select cid, cname, city from customers;
   grant select, delete, insert, update (cname, city) on custview to eoneil;
```

Now eoneil has Select access to cid, cname, city BUT NOT discnt on poneil.customers. [**CLASS**: Think how to create Exam Question out of this.]

IMPORTANT: User does NOT need privilege on a base table for privileges granted through a view to "take".

Could give student librarian access to other students' address, phone without giving access to more private info.

Can even give access to manager **only to** employees managed.

```
create view mgr_e003 as select * from employees where mgrid = e003;
grant all on mger_e003 to id_e003;
```

Standard is to drop privileges by command:

```
REVOKE {ALL PRIVILEGES | priv {, priv…} } on tablename | viewname
   FROM {PUBLIC | user {, user…} } [CASCADE | RESTRICT];
```

In X/Open, must specify either CASCADE or RESTRICT at end if privileges for other tables are based on the privileges being revoked.

But in ORACLE, default is RESTRICT. If include clause CASCADE RESTRAINTS then referential integrity constraints will be dropped in cascade fasion.

There are lots of other privileges supplied by DB2 and ORACLE, e.g.:

Create Database privileges (DBADM in ORACLE), privileges to look at or change query execution plans, perform LOAD operations on tables, etc.

In order to look at another person's tables, need not be in his/her database (actually usually all in same database with ORACLE). Grant gives you access.

7.4 System Catalogs.

Idea clear: all objects created by SQL commands are listed as objects in tables maintained by the system. Oracle calls this: data dictionary

The names of these tables are very non-standard, but at least there is a table for tables (ALL_TABLES, DBA_TABLES, and USER_TABLES in Oracle) and a table for columns (e.g., USER_TAB_COLUMNS).

DBA visiting another site would look at catalog tables (meta-data) to find out what tables exist, what columns in them, the significance of the columns (descriptive text string), and so on.

ORACLE Key for ALL_TABLES is TABLE_NAME, key for ALL_TAB_COLUMNS is TABLE_NAME COLUMN_NAME. Descriptions of keys in ALL_TAB_COLUMNS and ALL_COL_COMMENTS. Tables for priviliges granted, constraints, indexes.

In Oracle, USER_TABLES, for example, contains: number of rows, and disk space usage statistics, such as average row length in bytes, etc.

DESCRIBE command to find columnnames & columntypes of a user table.

```
     describe customers;
```

Could also do this through user_tab_columns, but more of a pain, not as nicely laid out. If want to describe, say, user_tab_columns, write:

```
     describe user_tab_columns;
```

It used to be, prior to ORACLE 8.i, that we had to write:

```
     describe "PUBLIC".user_tab_columns;
```

The double quotes around "PUBLIC" were necessary! What other data dictionary tables are there? Actually they are views! Try this:

```
    spool view.names
    select view_name from all_views where owner = 'SYS' and
        view_name like 'ALL_%' or view_name like 'USER_%';
```

Note that any single-quoted values like 'ALL_%' must be in CAPS. Here are view.names you might find. E.g.:

```
VIEW_NAME
------------------------------
USER_RESOURCE_LIMITS
USER_ROLE_PRIVS
USER_SEGMENTS
USER_SEQUENCES
USER_SNAPSHOTS
USER_SNAPSHOT_LOGS
USER_SNAPSHOT_REFRESH_TIMES
USER_SOURCE
USER_SYNONYMS
USER_SYS_PRIVS
USER_TABLES  <===
VIEW_NAME
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
USER_TABLESPACES <===
USER_TAB_COLUMNS <===
USER_TAB_COL_STATISTICS
USER_TAB_COMMENTS <===
USER_TAB_HISTOGRAMS
USER_TAB_PARTITIONS
USER_TAB_PRIVS <===
USER_TAB_PRIVS_MADE
USER_TAB_PRIVS_RECD
USER_TRIGGERS <===
USER_TRIGGER_COLS <===

VIEW_NAME
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
USER_TS_QUOTAS
USER_TYPES
USER_TYPE_ATTRS
USER_TYPE_METHODS
USER_UPDATABLE_COLUMNS  <===
USER_USERS  <===
USER_VIEWS  <===
```

Think about how in a program to explore new databases (Microsoft Explorer) would want to use dynamic SQL, find out what tables are around, find all columns in them, give Icon interface, list views, definitions, etc.

DB2 has provided the capability to change the Query Access Plan.

DB2 Catalog Tables are too complex to go over without an IBM manual.

I will skip over coverage of object-relational catalogs. Very important when you're actually using object-relational objects, of course!