Daily Course Notes for CS634, Patrick E. O'Neil

Class 1.

Hand out: Syllabus. CS630 Exam practice. Turn in sheet (Phone #, etc.). Turn in even if you are an auditor; auditors are OK by me. Show text. **HAVE TEXT?** If not, get it for next class.

Go over Syllabus. Prerequisite: This is a continuation of CS630. CS630 covered Chapters 2-6; this term covers Chapters 7-11.

How many people took CS630 from this text? If not, need to check you know what you need. Practice Exam from CS630. **NEED: NOT KIDDING!** Chaps 1-6. PARTICULARLY **Embedded SQL** (Chap. 5), **But NOT** O-R SQL (Chap. 4). Note dotted Exercises at end of chapters are solved in the back of the text.

Apply for course CS634. (New at school? Then show ID.) **IMPORTANT!!!** Even if already have account must do this!!! Can login to **users**, rlogin to **dbs2** to use ORACLE. Need to get on ORACLE immediately.

PDF Notes on Course **Home Page**, http://www.cs.umb.edu/cs634. Give me $5 for handouts, $10 if you want to get printed notes handouts, Homework assignments in UNIX directory ~poneil/cs634/hw1, hw2, etc., file "assignment". Dotted problems at end of Test Chapters are solved, and you will basically do all the non-dotted problems. Problems like these will be given on Exams.

Take Roll. GET BUDDY PHONE #, IF MISS CLASS — CATCH UP.

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

7. DBA commands. We teach our own Basic SQL standard; other standards are X/Open, SQL-99, but our Basic SQL limits syntax to features that are on most major databases. Then we go into special extra features of products.

Start 7.1, Integrity Constraints specified as part of Create Table command.

Create Table command, of course, makes table we can then add rows to, but learn now to impose rules so that updates that break the rules won't work!

7.2, Views: virtual tables to make it easier for DBA to simplify and control user access. Like a permanent Select statement to use in a FROM Clause (Select in FROM possible in Advanced SQL Syntax, Fig 3.11 .pg 117, Ex 3.6.3.)

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 which are very slow compared to CPU -- like walking to library shelves to find books.

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

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

We will cover MVS (Mainframe) DB2 of 1990, topics such as sequential prefetch, join algorithms, how to read Query Execution Plans. The concepts are much the same in modern DB2 UDB (UNIX DB2)

We'll go through this VERY quickly, review it in understanding MVS DB2 Query performance measured in 1990 Set Query benchmark (SSBM) at end of Chapt 9.

NEW MATERIAL. Then we will present material NOT in the text: SSBM on MODERN query performance (2008) for DB2 UDB. We will combine this with study of a modern application known as Data Warehousing.

Data Warehousing places operational data into a set of "Star Schemas" (see http://en.wikipedia.org/wiki/Star_schema). Data in these Star Schemas are typically not subject to interactive update in any great volume.

Instead a data warehouse provides a format that is optimized for querying data derived from operations (i.e., sales) in order to answer questions about which products are losing money, how to place products in the store for best sales, what colors/styles customers want most during different seasons, etc.

Chapter 10. Update Transactions (see Chap 5). Transactional histories. Performance value of *concurrent execution*, but problem of updates stepping on reads & other updates.

E.g.: R1(A, 100) R1(B, 100) R2(B, 100) W1(A, 50) W1(B, 150) C1 R2(A, 50) C2 (T2 sees total balance of 150)

ACID properties, transactions make guarantees to programmer. Atomic, Consistent, Isolated, and Durable.

Isolation means all concurrent executions are as if transactions happened non-concurrently, finish one and start the next, in some order.

Durable = Transactional Recovery. Idea is that if the system crashes, want to save results and guarantee all-or-nothing. E.g.: transfer of money -- don't want to be chopped off in the middle.

Chapter 11, Parallel and Distributed databases. Nothing on this.

Class 2.

Notes up on http://www.cs.umb.edu/cs634/. If you don't yet have a text, sit next to someone who does so you can look on.

Chapter 7.1 Integrity Constraints

Covering idea of Create Table constraint clauses & faithful representation.
HW is online at ~poneil/cs634/hw1, due class 7. Chapter takes a long time.

CALL ROLL. Correct my pronunciation. New students? Handouts -- turn in even if auditor; new BUDDIES. **Make sure you Apply!**

Section 7.1: Integrity constraints. You should 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 doing 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). We call this:

faithful representation

The way to achieve this is with constraint clauses in a Create Table statement. Basic SQL form handled by ORACLE, DB2 UDB.

Recall CAP database, pg 28 (with explanations on pg 27). Now see pg 415, for the E-R diagram of our CAP database.

E-R 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 we will use (**PUT ON BOARD**):

```
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: (1) not null for cid, (2) primary key cid which implies not null, (3) check discnt <= 15.00 (value not allowed to go over 15.00).

Primary key clause implies unique AND not null (if multi-column primary key, no nulls among them). If say primary, **don't say** unique. Very old products required you to say "not null" with "primary key", but can skip this now.

Could also declare cid (cid char(4) not null unique) if it were just a 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: failed rows go in exception list.

E-R 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!

```
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);
```

LEAVE ON BOARD!

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, straddr, zip) references ziptst(cityst, straddr, 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. (Wait to look up in text.)

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 [schema.]tablename
  ({colname datatype [DEFAULT {default_constant | NULL}]
        [col_constr {col_constr. . .}]
    | table_constr}  -- choose either colname def. of table_constr
  {, <repeat clause above, colname once for each column in table>. . .});
```

Recall CAPS means literal -- use lower case when writing SQL. Cover typographical conventions, Chap 3, pg 85. [ ] means optional, | means OR (choice of forms), { } can include choices {choice1 | choice2) or signify repeat when ellipsis . . ., e.g.: clause1 {, clause2 . . .} means 1 or more clauses.

Start by naming the table [schema.]tablename. A schema is a subdivision of database by user names qualified to create tables -- poneil.orders in same database as eoneil.orders.

Follow this with list of column names OR table constraints.

The column names are given with datatype (list on pg 733). A DEFAULT clause provides a value the system will supply if 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, column only).

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. PUT ON BOARD

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]}
```

Note that col_constr (& table_constr below) have "constraintname" for later reference. Can DROP or ADD col_constr later, using later Alter Table.

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.

Constraint names can go before ANY SINGLE ONE of the following. (Use names later with Alter Table command, e.g., to Drop named constraint.)

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.

CHECK clause defines search_condition that must be true for new rows in the current table, e.g. qty >= 0. In most database products, CHECK clause can only reference constants and the value of the current column on the single row being inserted/updated to perform a check for a column. Here is invalid use:

```
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 in other table won't work, but nulls are OK.

Multi-column equivalent uses 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. Just like column constraints except handle multi-column keys.

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.

Just covered Create Table col constraints, table constraints. Must Internalize all these examples: not just lists -- learn to USE knowledge. Mention Feinman French Curve; hws will help make knowledge operational!

Call ROLL.

Note page 411, [schema.]tablename; subdivision of database by user names qualified to create tables -- poneil.orders in same database as eoneil.orders.

(3rd edition)
Example, employees works_on projects: see Fig 6.7 pg 341 for E-R diagram.

```
create table employees (eid char(3) not null, straddr 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 that here we have a table representing a relationship: works_on. There is at most one row in works_on for each employees-projects pair!

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 leave a table without a primary key.)

(3rd edition)
Another example, from figure F.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 can reference a primary key in the same table.

What do we need to do to achieve Normalization in Ch. 6? All FDs should be dependent on primary key for table (OK: rows have unique key columns).

(3rd edition)
Recall idea of Lossless Decomposition in Section 6.7, SEE pg 374: risk is that when factor one table into two, get back MORE rows when join them.

Later (Thm. 6.7.4 pg 378) learn that to be lossless, intersection of two tables (the columns on which they join) must be **unique** on one of them! That means columns in intersection in one table must contain candidate or primary key!

Foreign Key References constraint means that foreign key in table referencing must **duplicate** unique column combination in foreign table. Typically, primary key of one table will then be intersection of schemas of two tables.

E.g.: orders joins with customers, intersection of columns contains cid, primary key of customers, means each row of orders is uniquely extended in join with other column information of customers: clear no information lost.

Create Table Statement in ORACLE, see Figure 7.3 (pg 416).

Extra disk storage and other clauses not mentioned in our Basic SQL format. But all DB 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 **need** any column names, datatypes, etc.: inherit from Select.

```
create table secusts
    as select * from customers where city = 'Dallas';
```

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 [3rd Edition], 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.

create table boys(name varchar(25) primary key, partnername varchar(25) not null unique disable girlpartner references girls);

Problem: this is not portable, so in general we depend on ADDing such a constraint later with Alter Table (can't define it early if load by Insert). Still, we'd like to think about this only once, see constraint before load.

DB2 Create Table: 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, pg 421 for standard options in X/Open, Full SQL-99 is same as DB2.

NOTE: You are not responsible for INFORMIX information in text; IBM bought INFORMIX -- still selling it, but no new development.

Referential Integrity

See pg. 417, Definition 7.1.3. 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 candidate or primary key P on some row of T2.

Thus for "optional participation" (0,N) in a relationship from the referencing table F, must allow at least one column of F to be nullable. Note some theoretical definitions of foreign keys I've seen don't allow optional nulls.

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 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** as suggested in Chapter 2. (Note it takes more effort to check.)

Class 4.

Review: just talked more about referential integrity, how there can be a loop in table constraints and table can refer to itself; definition of ref integrity (how if columns of foreign key can be null, need not be a match); differences with Oracle and DB2. Any questions?

Call ROLL.

The Alter Table Statement: pg. 422 & ff.

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-99 is too general (few clauses). 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 colname | (colname {colname . . . })]}
   [MODIFY (colname 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. Can't ADD col_constrs for existing cols, but can ADD tbl_constrs that have same effect. Can DROP a column.

Can MODIFY column to new definition: data type change from varchar to varchar2, but **can't modify constraints for a column.**

Can DROP named constraint, unique constraint or primary key. In ORACLE have ENABLE and DISABLE clauses mentioned before in Create Table, but we don't use them because they are not portable.

TRY THIS: connect to http://www.cs.umb.edu/cs634, then connect to ORACLE 10.2 Documentation, and search for **Alter Table** (Look for Alter Table topic).

When will CHECK constraint allow ANY search condition? **NOT** possible in ORACLE or any other product.

DB2 UDB Alter Table, Fig. 7.8, pg. 423. What differences from ORACLE? Can't DROP a column.

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.

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 non-proc also lack power.

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

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

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.)

Idea of [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.

(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 about old-corr-names or new.)

ORACLE uses PL/SQL (BEGIN and END as in usual PL/SQL progrms, starting pg 224). 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)]
       pl-sql-block;
```

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

Can REPLACE old named trigger with a new one. Trigger action can occur INSTEAD OF or AFTER or BEFORE an 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;
/
show errors;      -- will display any compilation errors
```

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.

Example 7.1.6
We could use an ORACLE trigger to implement a referential integrity policy "on delete set null" in the customers-orders foreign key. (Not good idea though.)

```
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;
/
show errors;
```

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

More Complete CS634 Trigger Example

--Simple Auditing example: track changes to discnt via trigger: Also note that Oracle has an AUDIT command.

```
dbs2(3)% more cr_audit_tab.sql
drop table audit_discnt;

     create table audit_discnt (
        upd_user varchar(20),
        upd_time varchar(20),
        upd_cid char(4),
        old_discnt real,
        new_discnt real);
dbs2(4)% more audit.sql

drop trigger audit_discnt_trig;

--Note no comma between "old o" and "new n" (correct pg 427, Fig. 7.12)
create trigger audit_discnt_trig
 after update of discnt on customers
 referencing old o new n for each row
 begin
  insert into audit_discnt values (
     user,
     to_char(sysdate, 'MM-DD-YY HH24:MI:SS'),
     :n.cid,
     :o.discnt,
     :n.discnt);
 end;
/                       <-IMPORTANT: use / after create trigger
show errors;            <-IMPORTANT: to see compilation errors

select trigger_name from user_triggers where table_name = 'CUSTOMERS';
update customers set discnt = 12.0 where cid = 'c001';
select * from audit_discnt;
dbs2(5)% more audit.out

SQL*Plus: Release 8.1.7.0.0 - Production on Mon Feb 11 14:22:24 2002
(c) Copyright 2000 Oracle Corporation.  All rights reserved.

Connected to: Oracle8i Enterprise Edition Release 8.1.7.0.0 - Production
With the Partitioning option
JServer Release 8.1.7.0.0 - Production

SQL> SQL> SQL> Trigger dropped.

SQL> SQL> SQL>   2    3    4    5    6    7    8    9   10   11   12   13
Trigger created.

SQL> SQL> No errors.

TRIGGER_NAME
------------------------------
AUDIT_DISCNT_TRIG
DISCNT_MAX
FOREIGNCID
FOREIGNCIDNULL

SQL> SQL>
1 row updated.

SQL> SQL>
```

```
UPD_USER             UPD_TIME             UPD_ OLD_DISCNT NEW_DISCNT
-------------------- -------------------- ---- ---------- ----------
EONEIL               02-11-02 14:12:28    c001         10         10
EONEIL               02-11-02 14:12:52    c001         10         10
EONEIL               02-11-02 14:13:03    c001         10         10
EONEIL               02-11-02 14:13:46    c001         10         12
EONEIL               02-11-02 14:22:24    c001         12         12

SQL> SQL> Disconnected from Oracle8i Enterprise Edition Release 8.1.7.0.0 -
With the Partitioning option
JServer Release 8.1.7.0.0 - Production
dbs2(6)%
```

Pros and Cons of Non-Procedural and Procedural Constraints
Starting on 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 creates a 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 a hw 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 for program 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.

Constraint failure alternative. 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. Triggers are NON-STRUCTURED CODE!!!)

Transactional Consistency. 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 (pg. 347, See Fig. 6.11). 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 of text, 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, adds 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 Updating views.)

Here is example view, **Example 7.2.1: (LEAVE UP!!!)**

```
create view agentorders (ordno, month, cid, aid, pid, qty,
   charge, aname, acity, percent)  -- note "charge" new name for o.dollars
   as select ordno, month, o.cid as cid, o.aid as aid, o.pid as pid, qty,
   dollars as charge, a.name as aname, a.city as acity, a.percent as 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, so don't need colnames in Create View.

Recall that Basic SQL Subquery, defined in Figure 3.14, pg 135, DOES allow UNION, does not allow ORDER BY. (OK, rows not ordered in table.)

Mention materialized views: must change if tables updated.

The WITH CHECK OPTION will not permit an update or insert to base table, 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. Following fails 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, usually all constraints (or named constraints, anyway).

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 (Someone try not UC???)

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, pg. 437.)

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

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

Updatable and Read-Only Views.

THERE USED TO BE LIMITATIONS OF QUERYING A GROUPED VIEW; THIS IS NOW OUT OF DATE.

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 read-only), a view must have the following limitations in its view definition in the X/Open standard (Fig. 7.15, pg 439):

(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 in Notes 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 pg. 28 tables). Have:

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

Consider delete of second row. How achieve? Change city of c003 (to what?). Delete a06? Then delete first row of view as well. Delete c003? Maybe, but what if there were a second agent (a07) in the same city, so four rows above.

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

Problem of GROUP BY (2). Example 7.2.8. Consider 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.

However: 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 librarian access to student records) and maintain old obsolete tables so code would continue to work (pg. 441 ff. in Notes).

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 1 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 no columns 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.. . .   NOW YES??****
PID          YES     . . . orders, except columns involved in join.
DOLLARS      YES
ANAME        NO  \ update ANAME would update multiple rows;
PERCENT      NO  / set-oriented SQL is disallowed in Joined View
ORDNO        YES                                                ^
7 rows selected.                                                |
```

Note: Still would not be able to update aname for given aid, even though it's clear what is meant.

Class 6. HW due soon

A little review. Talked last time about Views -- WITH CHECK OPTION clause -- Updatability -- ORACLE allows some column updates of FK-PK Join View.

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

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

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 privileges.

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

```
grant select, update, insert on orders to eoneil;  (Note, no delete.)
grant all privileges on products to eoneil;
```

Then eoneil can write:

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

The poneil. prefix is called the schema name.

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

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.custview. [**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. [TEXT Exerc.?]

```
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 RESTRAINTS];
```

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

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

ORACLE adds role_name to TO list objects, e.g., librarians

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, 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). See pgs. 448, 449. **NOTE:** See Corrections to Text, online in http://www.cs.umb.edu/cs634.

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 in USER_COL_COMMENTS), and so on.

ORACLE Key for ALL_TABLES is owner.TABLE_NAME, for ALL_TAB_COLUMNS is owner.TABLE_NAME COLUMN_NAME. Describe keys in ALL_TAB_COLUMNS and ALL_COL_COMMENTS. Tables for privileges granted, constraints, indexes.

In Oracle, USER_TABLES (key is TABLE_NAME, no OWNER given) 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;
```

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
------------------------------ HW: EXPERIMENT WITH THESE!!!
USER_RESOURCE_LIMITS           E.G.: WHAT IS KEY OF THESE VIEWS?
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  <===
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 <===
USER_TS_QUOTAS
USER_TYPES
USER_TYPE_ATTRS
USER_TYPE_METHODS
USER_UPDATABLE_COLUMNS  <===
USER_USERS  <===
USER_VIEWS  <===
```

Explore around with these. E.g.: try "select view_name from user_views". Create a new view xxx of your own and see how select changes. Then try "describe xxx". Exam questions will assume you know commands mentioned here and in text.

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!

Chapter 8. Indexing.

8.1 Overview. Usually, an index is like a card catalog in a library. Each card (entry) has:

```
(keyvalue, row-pointer) (keyvalue is for lookup, call row-pointer ROWID)
                        (ROWID is enough to locate row on disk: one I/O)
```

Entries are usually placed in Alphabetical/Numerical (typed) order by lookup key in "B-tree", explained below. Might be hashed instead by request.

An index is a lot like memory resident structures you've seen for lookup: binary tree, 2-3-tree. But index is disk resident. Like the data itself, often won't all fit in memory at once.

X/OPEN Syntax, Fig. 8.1, pg. 467, is extremely basic (not much to standard).

```
CREATE [UNIQUE] INDEX indexname ON tablename (colname [ASC | DESC]
   {,colname [ASC | DESC] . . .);
```

Index key created is a concatenation of column values. Each index entry looks like (keyvalue, rowpointer) for a row in customers table.

An index entry looks like a small row of a table of its own. If created concatenated index:

```
create index anamcit on agents (aname, city);
```

and had (aname, city) for two rows equal to (SMITH, EATON) and (SMITHE, ATON), the system would be able to tell the difference. Which once comes earlier alphabetically?

After being created, index is sorted and placed on disk. Sort is by column value asc or desc, as in SORT BY description of Select statement.

Later changes to a table (inserts, deletes, updates) are immediately reflected in an index, don't have to create a new index.

Ex 8.1.1. Create an index on the city column of the customers table.

```
create index citiesx on customers (city);
```

Note that a single city value can correspond to many rows of customers table. Therefore term index key is quite different from relational concept of primary key or candidate key. LOTS OF CONFUSION ABOUT THIS.

With no index if you give the query:

```
select * from customers where city = 'Boston'
   and discnt between 12 and 14;
```

Need to look at every row of customers table on disk, check if predicates hold (TABLE SCAN or TABLESPACE SCAN).

Not such a bad thing with only a few rows as in our CAP database example, Fig 2.2. (But bad in DB2 from standpoint of concurrency.)

And if we have a million rows, tremendous task to look at all rows. Like in Library. With index on city value, can winnow down to (maybe 10,000) customers in Boston, like searching a small fraction of the volumes.

User doesn't have to say anything about using an Index; just gives Select statement above.

Query Optimizer figures out an index exists to help, creates a Query plan (Access plan) to take advantage of it -- an Access plan is like a PROGRAM that extracts needed data to answer the Select.

Class 7. Homework is due

Recall X/OPEN form of Create Index:

```
CREATE [UNIQUE] INDEX indexname ON tablename (colname [ASC | DESC]
  {,colname [ASC | DESC] . . .);
```

(See pg. 485 for Oracle Create Index)

The unique clause of create index can be used to guarantee uniqueness of a candidate table key. Example 8.1.2, instead of declaring cid a primary key in the Create Table statement:

```
create unique index cidx on customers (cid);
```

OK if multiple nulls, like Create Table with column UNIQUE.

When use Create Table with UNIQUE or PRIMARY KEY clause, this causes a unique index to be created under the covers. Can check this, querying:

USER_INDEXES -- index_name, table_name, column_name
USER_IND_COLUMNS

OK, now the X/OPEN standard doesn't say much about what's possible with indexes. Foreshadowings.

Different index types: B-tree, Hashed. Usually, B-tree (really B^+-tree).

Primary index, secondary index, clustered index. Primary index means rows are actually <u>in</u> the index structure in the place of entries pointing to rows.

Primary index NOT NECESSARILY THE SAME SET OF COLUMNS AS THE PRIMARY KEY; common misunderstanding (although it is for Microsoft SQL-Server).

Clustered index means rows in same order as index entries (volumes on shelves for Dickens all in one area): may be primary index, may not.

Difficult to appreciate the index structures without understanding disk access properties: disk I/O is excruciatingly slow, most of index design is directed at saving disk I/O: even binary search of list inappropriate on disk.

8.2 Disk storage.

Computer memory is very fast but Volatile storage. (Lose data if power interruption.) Disk storage is very slow but non-volatile (like, move data from one computer to another on a diskette) and very cheap.

Disk: 500GB for $115; Memory (not memory card): 1GB $80+ Factor of ~400

Model 100 MIPS computer. Takes .00000001 seconds to access memory and perform an instruction. Getting faster, like car for $9 to take you to the moon on a gallon of gas.

Disk on the other hand is a mechanical device — hasn't kept up.

(Draw picture). rotating platters, multiple surfaces. disk arm with head assemblies that can access any surface.

Disk arm moves in and out like old-style phonograph arm (top view).

When disk arm in one position, cylinder of access (picture?). On one surface is a circle called a track. Angular segment called a sector.

To access data, move disk arm in or out until reach right track (Seek time)

Wait for disk to rotate until right sector under head (rotational latency)

Bring head down on surface and transfer data from a DISK PAGE (8 KB: data block, in ORACLE) (Transfer time). Rough idea of time: (differs from text)

Seek time:	.008 seconds
Rotational latency:	.004 seconds (analyze -- 120 rotations/sec)
Transfer time:	.0005 seconds (few million bytes/sec)

Total is .0125 seconds = 1/80 second (right for cheap disk -- maybe 1/120 second for high-end disk)

Typically a disk unit addresses 500 Gigabytes and costs $115 (new figure) with disk arm attached; not just disk pack which is cheaper.

512 bytes per sector, 2,500 sectors per track, so 1,250,000 bytes per track. 10 surfaces so 5,000,000 bytes per cylinder. 40,000 cylinders per disk pack.

Total is 500 GB (i.e., gigabytes)

Now takes .0125 seconds to bring in data from disk, .000'000'01 seconds to access (byte, longword) of data in memory. How to picture this?

Analogy of Voltaire's secretary. Copying letters at one word a second. Run into illegible word. Send letter to St Petersberg, six weeks to get response (1780). Can't do work until get response (solution: work on other projects.)

From this see why read in what are called pages, 8K bytes on ORACLE, 4K bytes on DB2 UDB. Want to make sure Voltaire answers all our questions in one letter. Structure Indexes so take advantage of a lot of data together.

Modern Disk Access Time

From: http: //www.storagereview.com/map/lm.cgi/access

Let's compare a high-end, mainstream IDE/ATA drive, the Maxtor DiamondMax Plus 40, to a high-end, mainstream SCSI drive, the IBM Ultrastar 72ZX. (When I say "high end" I mean that the drives are good performers, but neither drive is the fastest in its interface class at the time I write this.) The Maxtor is a 7200 RPM drive with a seek time spec of "< 9.0 ms", which to me means 9 ms. Its sum of its seek time and latency is about 13.2 ms. The IBM is a 10,000 RPM drive with a seek time spec of 5.3 ms. It's sum of seek time and latency is about 8.3 ms. This difference of 5 ms represents an enormous performance difference between these two drives, one that would be readily apparent to any serious user of the two drives.

As you can see, the Cheetah beats the DiamondMax on both scores, seek time and latency. When comparing drives of a given class, say, IDE/ATA 7200 RPM drives, they will all have the same latency, which means, of course that the only number to differentiate them is seek time. Comparing the Maxtor above to say, the Seagate Barracuda ATA II with its 8.2 ms seek time shows a difference of 0.8 ms, or around 10%. But the proper comparison includes the other components of access time. So the theoretical access time of the Maxtor drive is about 13.7 ms (including 0.5 ms for command overhead) and that of the Seagate Barracuda drive 12.9. The difference now is about 6%. Is that significant? Only you can judge, but you also have to remember that even access time is only one portion of the overall performance picture.

Buffer Lookaside

Idea of buffer lookaside. Read pages into memory buffer so can access them faster. Read popular pages only once from right place on disk, save time.

Every time want a page from disk, hash on dkpgaddr, h(dkpgaddr) to entry in Hashlookaside table to see if that page is already in buffer. (Pg. 472)

If so, saved disk I/O. If not, drop some page from buffer to read in requested disk page. Try to fix it so popular pages remain in buffer.

Another point here is that we want to find something for CPU to do while waiting for I/O. Like having other tasks for Voltaire's secretary.

This is one of the advantages of multi-user timesharing. Can do CPU work for other users while waiting for this disk I/O.

If only have 0.5 ms of CPU work for one user, then wait 12.5 ms for another I/O. Can improve on by having more than 25 disks, trying to keep them all busy, switching to different users when disk access done, ready to run.

Why bother with all this? If memory is faster, why not just use it for database storage? Volatility, but solved. Cost no longer big factor:

Memory storage costs about a $80 per gigabyte.
Disk storage (with disk arms) costs about $0.2 per Gigabyte.

So could buy enough memory so bring all data into buffers (constant speed fast access with millions of memory buffers; very efficient access: hashing!) Coming close to this; probably will do it soon enough. Stilll ofen limited by 4 GBytes of addressing on a 32 bit machine, but some machines have 64 bits.

Class 8. Create Tablespace

OK, now DBA and disk resource allocation in ORACLE.

We have avoided disk resource considerations up to now because they're so hard to handle in any standard. All the commercial systems are quite different in detail. But there are a lot of common problems.

A tablespace is built out of OS files (or raw disk partition [TEXT]), and can cross files (disks) Look at Fig. 8.3, pg. 476. Segment can also cross files.

All the products use something like a tablespace. DB2 uses tablespace.

Tablespaces have just the right properties, no matter what OS really living on -- general layer insulates from OS specifics. Typically used to define a table that crosses disks.

See Fig. 8.3 again. When table created, it is given a data segment, Index an index segment. A segment is a unit of allocation from a tablespace.

Tablespace is the basic resource of disk storage, can grant user RESOURCE privilege in ORACLE to use tablespace in creating a table.

Any ORACLE database has at least one tablespace, named SYSTEM, created with Create Database: holds system tables. Look at Fig 8.4, pg. 476.

Operating systems "files" named in datafile clause. ORACLE can create them itself of given SIZE; then DBA loses ability to specify particular disks.
  DATAFILE 'filename' [SIZE n [K|M]] [REUSE]

If SIZE keyword omitted, data files ORACLE uses must already exist. If SIZE is given, ORACLE will normally create file, but REUSE means use existing files named even when SIZE is defined; then ORACLE checks size is valid
  [AUTOEXTEND OFF | AUTOEXTEND ON [NEXT n [K|M] [MAXSIZE . . .

If AUTOEXTEND ON, system can extend size of datafile. The NEXT n [K|M] clause gives size of expansion when new extent is created. MAXSIZE limit.

If tablespace created *offline*, cannot immediately use for table creation. Can later Alter Tablespace (not shown here) to make offline/online for recovery purposes, reorganization, etc., without bringing down whole database.

SYSTEM tablespace, created with Create Database, never offline.

When table first created, given initial disk space allocation. Called an initial *extent*, should be contiguous on disk. When load or insert runs out of room, additional extents are provided, each called a "next extent", numbered from 1.

Create Tablespace gives DEFAULT values for extent sizes and growth in STORAGE clause; Create Table can override these.

INITIAL n:	size in bytes of initial extent 0: default 10240
NEXT n:	size in bytes of next extent 1. (same) May grow.
MAXEXTENTS n:	maximum number of extents segment can get; can also make UNLIMITED (not good idea normally)
MINEXTENTS n:	start at creation with this number of extents; used when know initial use will be very large
PCTINCREASE n:	increase from one extent to next. default 50.

Minimum possible extent size is 8K (8192) bytes, max is 4095 Megabytes, all extents are rounded to nearest block (page) size.
[MINIMUM EXTENT n [K|M]]

Many Create Tablespace extent options can be overriden by later Create Table, but MINIMUM EXTENT clause guarantees that Create Table won't be able to override with too small an extent: extent below this size can't happen.

Next, Create Table in ORACLE, Figure 8.5, pg. 478. (<u>Leave Up</u>)

```
CREATE TABLE [schema.]tablename
   (colname datatype [DEFAULT {constant|null}] [col_constr] {, col_constr}
     | table_constr}
   {, colname . . .}
   [ORGANIZATION HEAP | ORGANIZATION INDEX (with clauses not covered)]
   [TABLESPACE tblspname]
   [STORAGE ([INITIAL n [K|M]] [NEXT n [K|M]] [MINEXTENTS n]
     [MAXEXTENTS n|UNLIMITED] [PCTINCREASE n] ) ]
   [PCTFREE n] [PCTUSED n]
   [other disk storage and update tx clauses not covered or deferred]
   [AS subquery]
```

ORGANIZATION HEAP is default: as insert new rows, normally placed left to right. Trace how new pages (data blocks) are allocated, extents. Note if space on old block from deletes, etc., might fill in those.

ORGANIZATION INDEX means place rows in A B-TREE INDEX (will see) in place of entries. ORDER BY primary key!

The STORAGE clause describes how initial and successive allocations of disk space occur to table (data segment). The default storage clause defined by Create Tablespace will be inherited by a newly created table unless overridden.

The PCTFREE n clause determines how much space on each page used for inserts before stop (leave space for varchar expand, Alter table new cols.)

PCTFREE n, n goes from 0 to 99, default 10.

The PCTUSED n clause specifies a condition where if page (block) gets empty enough, inserts will start again! Range n from 1 to 99, default 40.

Require PCTFREE + PCTUSED < 100, or invalid. Problem of hysteresis; don't always want to be switching from inserting to not inserting and back.

E.g., if PCTFREE 10 PCTUSED 90, DBMS stops inserts when >90% full, start when <90% full. If PCTFREE 20 PCTUSED 90, behavior ill-defined between 80 and 90. Commonly used is PCTFREE 10 PCTUSED 40

Data Storage Pages and Row Pointers: Fig. 8.6, pg 480.

Typically, after table is created, extents are allocated, rows are placed one after another on successive disk pages (this is called "heap" storage). Most products have very little capability to place rows in any other way.

A row on most architectures is a contiguous sequence of bytes. See Figure 8.6 (pg. 480): N rows placed on one page (called a *Block* in ORACLE).

Header info area names the kind of page (data segment, index segment) and numbered object of that type, the page number (in ORACLE the OS file and page number). Rows added right to left from right end on block.

Row Directory entries left to right on left after header. Give offsets of corresponding row beginnings. Provide number of row slot on page. Good for **Information Hiding!!!**

When new row added, tell immediately if we have free space for new directory entry and new row. Conceptually, all space in middle, implies when delete row and reclaim space must shift to fill in gap. (Can defer that.)

Also might have "Table Directory" in block when have CLUSTER. (Later.)

There's overhead for column values within a row (offsets), not shown here. Must move rows on pg if updates changes char varying field to larger size.

Class 9. Review Fig. 8.6 above (put on board, discuss)

Disk Pointer. RID (DB2), ROWID (ORACLE), TID (INGRES - OLD product, but good example). A row in a table can be uniquely specified with the page number (P) and slot number (S). In INGRES, have TID (Tuple ID), given by:

TID = 512*P + S

Pages in INGRES are numbered successively within the table allocation, from zero to 2^{23} - 1: all 1's in 23 bit positions. Slots are 0 to 511. all 1's in 9 bit positions: total container is 32 bits, unsigned int, 4 bytes.

So if rows are 500 bytes long (small variation), 2K byte pages in UNIX will contain only 4 rows, and TIDs go: 0, 1, 2, 3, 512, 513, 514, 515, 1024, 1025, . . .

Note value of information hiding. Give row ptr in terms of Page and Slot number; if gave byte offset instead of slot number would have to change RID/ROWID/TID when reorganized page.

Example 8.2.1, pg 481, (variant). INGRES DBA (or any user) can check space calculations are right (200 byte rows 10 to 2 KB disk page, make sure knows what's going on) by selecting TID from table for successive rows:

```
select tid from employees where tid <= 1024;
```

Note tid is a Virtual column associated with every table, so can select it. Will appear in order (tablespace scan goes to first pg, first row, then . . .)

A DB2 record pointer is called a RID, also 4 bytes, encoding page number within Tablespace and slot number, but DB2 RID structure is not public, and we can't Select a DB2 RID from a table as a virtual column.

In ORACLE, row pointer is called ROWID. and is normally **6 bytes long!!** Restricted ROWID display representation is made up of Block number (page) within OS file, Slot number in block, & file no. (why?):

BBBBBBBB.RRRR.FFFF (each hexadecimal, total of 8 bytes for ROWID)

(Block number, Row (Slot) number, File number)

The ROWID value for each row can be retrieved as a virtual column by an SQL Select statement on any table, as with the following query:

select cname, rowid from customers where city = 'Dallas';

which might return the following row information (if the ROWID retrieved is in restricted form):

```
CNAME        ROWID
------------ ------------------
Basics       00000EF3.0000.0001
Allied       00000EF3.0001.0001
```

The standard "extended ROWID" form, used to reference rows across tables, is displayed as a string of four components having the following layout, with letters in each component representing a base-64 encoding:

OOOOOOFFFBBBBBRRR

Here OOOOOO is the data object number, and represents the database segment (e.g., a table). The components FFF, BBBBB, and RRR represent the file number, block number, and row number (slot number).

Here is the "base-64" encoding, comparable to hexadecimal representation except that we have 64 digits. The digits are printable characters:

DIGITS	CHARACTERS
0 to 25	A to Z (Capital letters)
26 to 51	a to z (lower case letters)
52 to 61	0 to 9 (decimal digits)
62 and 63	+ and / respectively

For example, AAAAm5AABAAAEtMAAB represents object AAAAm5 = 38*64 + 57, file AAb = 1, block AAEtM = 4*642 + 44*64 + 13, and slot 1. The query:

select cname, rowid from customers where city = 'Dallas';

might return the row information (different values than restricted ROWID):

```
CNAME         ROWID
------------- ------------------
Basics        AAAAm5AABAAAEtMAAB
Allied        AAAAm5AABAAAEtMAAC
```

Since an index is for a specific table, don't need extended ROWID with object number of table segment. But Select statement commonly displays the extended form. Functions exist to convert (need special library).

We use ROWID nomenclature generically if database indeterminate.

Example 8.2.2. See Embedded SQL program on pg. 483. Idea is that after we find a row (through an index or slower means), we can retrieve it a second time through ROWID in ORACLE (not in DB2). This saves time.

Danger to use ROWID if someone might come and delete a row with a given ROWID, then reclaim space and new row gets inserted in that slot. If out of date ROWID could refer to wrong row.

But certainly safe to remember ROWID for length of a transaction that has a referenced row locked.

Rows Extending Over Several Pages: Product Variations (Important!)

In ORACLE, allow rows that are larger than any single page (8 KB).

If a row gets too large to fit into its home block, it is split to a second block. Second part of row has new ROWID, not available to any external method (such as indexing), only to chain from prior part(s) of row.

Means that random access may require two or more page reads (Voltaire forwarding). DB2 on other hand limits size of row to disk page (4005 bytes).

Still may need to move row when it grows, leave forwarding pointer at original RID position (forwarding pointer called "overflow record").

Why do you think DB2 does this?

But only ever need ONE redirection, update forwarding pointer in original position if row moves again, no chain of forwarding pointers. [TEXT]

8.3 B-tree Index

Most common index is B-tree form. Assumed by X/OPEN Create Index. We will see other types — particularly hashed, but B-tree is most flexible.

The B-tree is like the (2-3) tree in memory, except that nodes of the tree take up a full disk page and have a lot of *fanout*. (Picture on board.)

ORACLE Form of Create Index Figure 8.7, pg. 485 (leave on board):

```
CREATE [UNIQUE | BITMAP] INDEX [schema.]indexname ON tablename
   (colname [ASC | DESC] {,colname [ASC | DESC]])
   [TABLESPACE tblespace]
   [STORAGE . . . ]  (Override Tablespace default)
   [PCTFREE n]
   [other disk storage and update tx clauses not covered or deferred]
   [NOSORT]
```

We will discuss the concept of a BITMAP Index later. Non-Bitmap for now.

Note ASC | DESC is not necessary: ORACLE is just copying DB2 here.

What Create Index does: reads through all rows on disk (assume N), pulls out (keyvalue, rowid) *entries* for each row. Put following list out on disk.

(keyval1, rowid1) (keyval2, rowid2) . . . (keyvalN, rowidN)

Now sort these on disk order by keyvalues. NOSORT clause tells ORACLE rows are in right order, so need sort, error returned if ORACLE notices this is false.

Binary search Example 8.3.1, pg. 486. Assume N = 7, search array of structs: arr[K].keyval, arr[K].rowid (lpairs named above). Ordered by keyval K = 0 to 6.

```
/* binsearch:  return K so that arr[K].keyval == x, or -1 if no match;      */
/* arr is assumed to be external, size of 7 is wired in                     */
  int binsearch(int x)
  {
    int  probe = 3,             /* first probe position at subscript K =  3  */
         diff = 2;              /* difference to 2nd probe                   */

    while (diff > 0) {          /* loop until position K finds x or not      */
      if (probe <= 6 && x >  arr[probe].keyval)  /* if probe too low         */
        probe = probe + diff;   /* raise the probe position                  */
      else                      /* otherwise                                 */
        probe = probe - diff;   /* lower the probe position (even if = K)    */
      diff = diff/2;            /* home in on answer                         */
    }                           /* we have reached final K                   */
    if (probe <= 6 && x == arr[probe].keyval) /* have we found it?           */
      return probe;
    else if (probe+1 <= 6 && x == arr[probe+1].keyval) /* maybe next         */
      return probe + 1;
    else return -1;             /* otherwise, return failure                 */
  }
```

Figure 8.8 Function binsearch, with 7 entries wired in

Consider the sequence of keyval values {1, 7, 7, 8, 9, 9, 10} at subscript positions 0-6.

Work through if x = 1, binsearch will probe successive subscripts 3, 1, and 0 and will return 0. If x = 7, the successive subscript probes will be 3, 1, 0. (undershoot — see why?) then up, return 1. If x = 5, return -1. More?

Given duplicate values in the array, binsearch will always return the ***smallest*** subscript K with x == arr[K].keyval (exercise). That's why go low even if = x.

The binsearch function generalizes easily: given N entries rather than 7, choose m so that $2^{m-1} < N \le 2^m$; then initialize probe to $2^{m-1}-1$ and diff to 2^{m-2}. Tests if probe <= 6 or probe +1 <= 6 become <= N-1.

Optimal number of comparisons (if all different). $\log_2 N$. But not optimal in terms of disk accesses.

Example 8.3.2. Assume 1 million entries. First probe is to entry 524,287 (2^{19}-1). Second prove is 2^{18} = 262,144 entries away. Look at list pg. 488.

But with 2K byte pages, assume keyvalue is 4 bytes (simple int), ROWID is 4 bytes (will be either 6 or 7 in ORACLE), then 8 bytes for whole entry; 2000/8 is about 250 entries per page. (1M/250 = 4000 pgs.) Therefore successive probes are always to different pages until probe 13 (maybe).

That's 13 I/Os!! A year of letters to Voltaire, counting 3 weeks for each letter delivery in 1780. We can make it 9 weeks!

Class 10.

Exam 1, will be on Class 15, Wednesday, March 23, 2005 (Don't trust dates in general).

Last time, talked about binary search on 1M entries of 8 bytes each assuming search list is on disk, Example 8.3.2, pg. 487. Required 13 I/Os.

OK, now we can improve on that. Layout 1M entries on 4000 pgs. These are called leaf nodes of the B-tree.

Each leaf has a range of keyvalues; we want to create an optimal directory to get to proper leaf node. Directory entries have node ptrs np and separator

keys S between values in two nodes below (possibly equal to high of node key-values on left pointed to by np or or to low keyvalue of node on right).

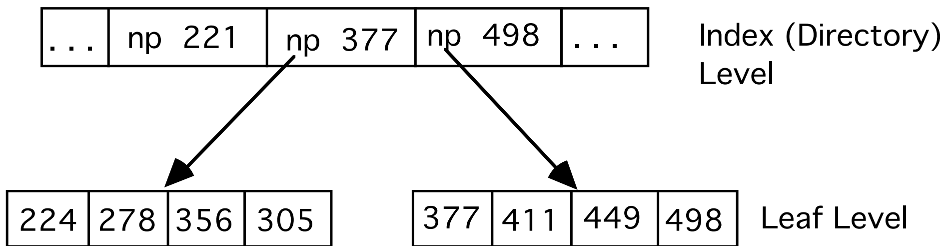


Figure 8.10 Directory structure to leaf level nodes

OK, now put directory on disk pages. Each entry is: (sepkeyval, np). About same size as B-tree entry, 8 bytes. Number of pages is 4000/250 = 16.

Now repeat trick with HIGHER level directory to index nodes just created. Same types of entries, only 16 pointers needed, all on one pg. Root page.

Seeking keyvalue x, use binary search on each index directory node to find smallest sepkeyvalue >= x, follow np down to leaf, then find x (if it exists).

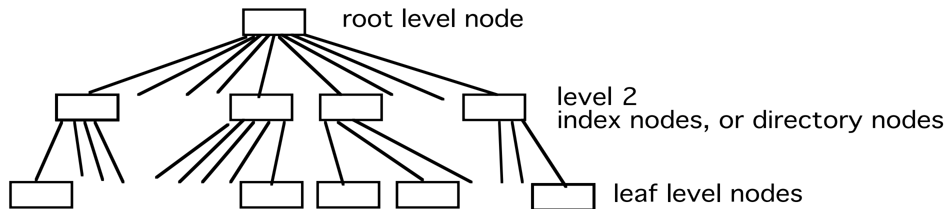


Figure 8.11 Schematic Picture of a three level B-tree

Nodes above *leaf nodes* are called*index nodes* or *directory nodes*. Note that the depth of the tree is 3, so will require at most 3 I/Os. (In fact, less, because of buffering of disk pages in memory.)

Get the most out of all information on (index node) page to locate appropriate subordinate page. 250 way fanout. Say this is a *bushy tree* rather than *sparse tree* of binary search, and therefore flat. Fanout f:

depth = $\log_f(N)$ — e.g., $\log_2(1{,}000{,}000) = 20$, $\log_f(1{,}000{,}000) = 3$ if $f > 100$

Actually, will have commonly accessed index pages resident in buffer. 1 + 16 in f = 250 case fit in buffer, but not 4000 leaf nodes: only ONE I/O.

Get same savings in binary search. First several levels have 1 + 2 + 4 + 8 +16 nodes, but to get down to one I/O would have to have 2000 pages in buffer. There's a limit on buffer, more likely save 6-7 I/Os out of 13.

(Of course 2000 pages fits in only 4 MB of buffer, not much nowadays. But in a commercial database we may have HUNDREDS of indexes for DOZENS of tables! Thus buffer space must be shared and becomes more limited. Assume you can contain 20 2KByte pages unless otherwise instructed.)

Dynamic changes in the B-tree

OK, now saw how to create an index when know all in advance. Sort entries on leaf nodes and created directory, and directory to directory.

But a B-tree also grows with random inserts in an even, balanced way. Consider the sequence of inserts, shown in **Figure 8.12, pg. 491.**

(2-3)-tree. All nodes below root contain 2 or 3 entries, split if go to 4. This is a 2-3 tree, B-tree is probably more like 100-200.

(Follow along in Text). Insert new entries at leaf level. Start with simple tree, root = leaf (don't show rowid values). Insert 7, 96, 41. Keep ordered. Insert 39, split. Etc. Correct bottom tree, first separator is 39.

Note separator key doesn't HAVE to be equal to some keyvalue below, so don't correct it if later delete entry with keyvalue 39.

Insert 88, 65, 55, 62 (double split).

Note: stays balanced because only way depth can increase is when root node splits. All leaf nodes stay same distance down from root. DOES THIS MEAN an extra I/O after root splits???

All nodes are between half full (right after split) and full (just before split) in growing tree. Average is .707 full: SQRT(2)/2.

Now explain what happens when an entry is deleted from a B-tree (because the corresponding row is deleted).

In a (2-3)-tree if number of entries falls below 2 (to 1), either BORROW from sibling entry or MERGE with sibling entry. Can REDUCE depth.

This doesn't actually get implemented in most B-tree products: nodes might have very small number of entries. Note B-tree needs malloc/free for disk pages, malloc new page if node splits, return free page when empty.

In cases where lots of pages become ALMOST empty, DBA will simply reorganize index.

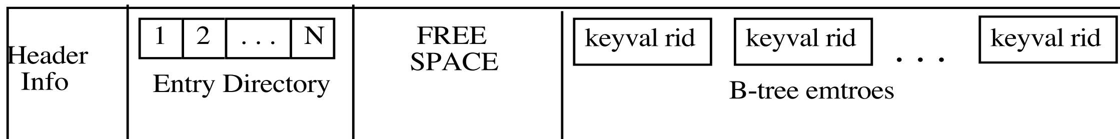
Properties of the B-tree, pg. 493 ff, Talk through. (Actually, B+tree.) Assumes fixed-size keyvalues -- not most general -- but intuitive.

- Every node is disk-page sized and resides in a well-defined location on the disk.
- Nodes above the leaf level contain directory entries, with n – 1 separator keys and n disk pointers to lower-level B-tree nodes.
- Nodes at the leaf level contain entries with (keyval, ROWID) pairs pointing to individual rows indexed.
- All nodes below the root are at least half full with entry information.
- The root node contains at least two entries (except when only one row is indexed and the root is a leaf node).

Index Node Layout and Free Space on a page

See Figure 8.13, pg. 494. Always know how much free space we have on a node. Usually use free space (not entry count) to decide when to split.

This is because we are assuming that keyval can be variable length, so entry is variable length; often ignored in research papers.



Note Figure looks like rows in block; Entry directory slots for entries give opportunity to perform binary search although entries are var. length.

In Load, leave some free space when create index so don't start with string of splits on inserts. See Figure 8.7, pg. 485, PCTFREE n.

Idea is, when load, stop when pctfree space left. Split to new node. Free space available for later inserts.

Example 8.3.4. Corrected B-tree structure for a million index entries.

Index entry of 8 bytes, page (node) of 2048 bytes, assume 48 bytes for header info (pg. 494), and node is 70% full.

Thus 1400 bytes of entries per node, 1400/8 = 175 entries. With 1,000,000 entries on leaf, this requires 1,000,000/175 = 5715 leaf node pages. Round up in division, CEIL(1,000,000/175), say why.

Next level up entries are almost the same in form, (sepkeyvalue, nptr), say 8 bytes, and 5715 entries, so 5715/175 = 33 index node pages.

On next level up, will fit 33 entries on one node (root) page. Thus have B-tree of depth 3.

<u>ROUGH CALCULATIONS LIKE THIS ARE FINE.</u> We will be using such calculations a lot! (It turns out, they're quite precise.)

Generally most efficient way to create indexes for tables, to first load the tables, then create the indexes (and REORG in DB2).

This is because it is efficient to sort and build left-to-right, since multiple entries are extracted from each page or rows in the table.

Create Index Statements in ORACLE and DB2. See Figures 8.14, and 8.15., pg. 496 & 497.

ORACLE has always made an effort to be compatible with DB2. Have mentioned PCTFREE before, leaves free space on a page for future expansion.

DB2 adds 2 clauses, INCLUDE columname-list to carry extra info in keyval and CLUSTER to speed up range retrieval when rows must be accessed. See how this works below.