Chapters 1 and 2

1.

Hand out: Syllabus. Go over Syllabus, prerequisites. Grader, DBA.
Homework: Read Chapter 1 (quickly), and Chapter 2 through Section 2.4.
Homework in two weeks: All undotted exercises at end of Chapter 2.
2.1(b, d), 2.2b, 2.3, 2.4(b,d,f), 2.5(b, d, etc.), 2.6a, 2.8b, 2.9b, 2.10, 2.15a,c, the following hard ones to try: 2.12, 2.14. NOTE SOLVED PROBLEM IN TEXT CAN HELP YOU.
Want you to enter solution online, print it out, submit hardcopy.
Apply for a course account. Using Oracle Database.

GET BUDDY PHONE #, IF MISS — CATCH UP.

Now Quickly through ideas of Chapter 1, what we will learn. Just try to get the idea for now — helps later when will cover again.

DEF. A database management system, or DBMS, is a program product for keeping computerized records (on disk)

What we will be learning in this course is underlying concepts you need to make a database work: not how to build one but how to set one up.
Compiler or Operating System is also a program; not a trivial thing.

To begin with we will be dealing with the Relational Model, so all our data will look like tables. E.g., see database, Figures 2.1 & 2.2, pgs. 27, 28.

We will be using these tables throughout the course. Draw tables & two rows on board.

Nomenclature: We have: tables (or relations), columns (define attributes of the data), rows (or tuples) to represent individual students or courses (entities) or enrollments (relationships).

Go over column names and meaning. Note aid unique, aname might duplicate, month values in ORDERS. This is not realistic, e.g., anames are too short, would want more columns, time more accurate.

Can answer questions about the data with queries. SQL Example: List agent ID, agent name, and percent commission of all agents in New York.

```
select aid, aname, percent
   from agents where city = 'New York';
```

Now list the agent ID and agent name of all agents who placed an order in January. **How would you do this?** Need to connect TWO TABLES. In SQL:

```
select a.aid, a.aname from agents a, orders o
      where a.aid = o.aid and o.month = 'jan';
```

Chapter 2. RULES of Relational Model. Tables have followed rules so all commercial products are the same, like typewriter keyboards.

For years these rules were as in Sect. 2.3. E.g., no column of a table should have a complex column value (a record), only a simple type (an integer). Similarly, can't have multiple values for a column of a row.

But there is coming to be a new model, known as the Object-Relational model, that **does** allow complex column values (and collection types as well: Can have a Set, Multiset, or List of values in a single row column).

Something to bear in mind: When mathematics as a field was the age that database is now, it had just come up with Plane Geometry, and was still doing arithmetic in Roman Numerals.

There are going to be a lot of changes in database systems before everything settles down.

Relational Algebra. A non-machine query language that gives a good idea of power of query languages: make new tables out of old tables by simple operations. E.g., List cno, cname for courses meeting MW2.

Rel. Alg. (AGENTS where city = 'New York) [aid, aname, percent]

Gives us simplest possible approach understanding what SQL is aiming at. Relational Algebra in Chapter 3, SQL in Chapter 3.

Chapter 3. SQL queries, already seen. Turns out to have more power than Rel. Alg. Also covers how to Insert new row in table, Delete old rows from table, Update column values of existing rows in table.

SQL is also in the midst of change, changing from Relational model (SQL-92, Oracle R7) to Object-Relational model (SQL-3, DB2 V2, Oracle R8).

There are still data applications that don't fit the Relational Model: store 200 word abstracts of 2M papers, find abstracts of all papers with abstract words containing as many as possible of phrases such as the following: Chemical, Biosphere, Environmental, Lead poisoning, . . .

Chapter 4. Object-Relational SQL (Informix, Oracle). Now we CAN perform the query just named, by creating a table with one column (abstract_word) permitting MULTIPLE values.

We will cover chapter 6 before chapter 5.

Chapter 6. Some of what DBAs do. Design database. Lots of complex commands to construct a database. Begin with Logical Design.

Logical design: break down all the data into tables so tables have good properties. E.g., how handle employee with unknown number of dependents? Can get EXTREMELY complicated.

E.g., relatively simple school application: Departments contain teachers and are responsible for offering subjects, made up of multiple es.

The subjects have prerequisites. The es are offered in given periods in specific rooms. Students have schedules (each term), drop and add es, get grades, have library records, tuition payments, health insurance, etc. How break all these up?

One basic idea is that entities (real world objects: students, rooms, even -periods) each deserve their own table. They have attributes (student name and id, room number and floor, period length and how grouped: 3/week, 2/week).

There are relationships between entities: above, enrollment is a relationship between student and course. But in a more complex scheme, need to relate teachers with periods, rooms, and subjects to form a offering, then the offering and student are related in a student schedule and a distinct schedule.

Chapter 5. C programs with embedded SQL statements: Embedded SQL. Idea is to present a Menu to naive users so they never have to understand SQL. (SQL of Chapter 3 is called ad-hoc, or interactive SQL).

Reason: complex job to write SQL, bank tellers and airline reservation clerks don't want to have to do complex stuff like this. DANGEROUS to let people in a hurry try to update data using SQL.

Brings up problems of avoiding problems of concurrency and making data durable: Update transactions needed.

Term "User Friendly" is too vague; several different types of DBMS users.

-End Users:
  -Naive Users (Use Menu interface)
  -Casual Users (Compose SQL)
-Application Programmers (Write Menu applications)
-Database Administrators (DBAs)

Chapter 7. Commands for creating databases, creating and loading tables, (performance related issues, such as indexes, come later).

Integrity. Set up rules for how data can change. Rules keep errors from occurring because of inexperienced application writers. Rules can be treated as data, so high level designers know what is happening, not buried in millions of lines of applications.

Creating Views. Views are virtual tables, pretend tables created out of real base tables. DBA creates these so specific application programmers have an easier time (fewer fields to learn), easy to change DB without breaking old applications.

Security. What users can look at/change salary values for employees. Tax accountants can look at, can't change. Programmers have very low data security.

Chapter 8 and following. Skip for now. Physical design: how to place tables on disk so minimize access time; hard problem, because trade-offs. Index creation. Design for shared data.

All performance related stuff is put off to the second half of the book. But note idea of data sharing during update, something we run into in programming Embedded SQL. Problem where one person looks at data while another person is changing it. Sum up two account balances (rows of different tables), might get sum that is false if other application has just subtracted from one to transfer money. Need something clever to handle this, call it a "Transaction".

Next time start on Chapter 2 in detail.

2.

Assignment 1: GET BUDDY PHONE #, IF MISS — CATCH UP.
Read through end of Chapter 2.
Homework in two weeks: All undotted exercises at end of Chapter 2.

OK, Now start Chapter 2. Relational concepts & Relational Algebra.

Section 2.1. Look at pg. 28. CUSTOMERS, AGENTS, PRODUCTS, ORDERS. CAP database, collection of computerized records maintained for common purpose.
- Possible to have two databases on same machine used by same people (student records and University library contents)

Put on board first two lines of CUSTOMERS. Go over meanings of column-names, p 27.
- Note that cid is unique for customers, cname is NOT.
- See how calculate dollars value in orders: qty of orders row times price
in products row for that pid and take discnt for customer row for cid. But we don't want to do that every time we ask for the dollars value so we carry the value in the orders table.
- Note too that table is unrealistically small: more columns (name, straddr, contact, timestamp for orders), more rows, more tables (keep track of billing, salaries, taxes)

Section 2.2. Terminology.
Table (relation) (Old: file of records).
Column names (attributes) (Old: field names of records)
Rows (tuples) (Old: records of a file).
Table heading (Schema) (Old: set of attributes).
Set of columns is normally relatively constant, part of mental model used for queries, rows are changeable and not kept track of by user (queried).
- Note how we say Contents at a given moment on pg. 28, CAP. This turns out to be important when asked to make up a query to answer a question, query must still answer the question even if all the data changes.

This rule is sometimes called Program-Data Independence — mentioned in Chapter 1; used to be file of records, but this is better: level of abstraction (E. g. indexing transparent)
- Students once implemented DB in Software Engineering course, called back later because wasn't working well in job where they ported it. Didn't have Program-Data Independence!

Heading (Schema) of table. Head(CUSTOMERS) = {cid, cname, city, discnt}.
Typically name a table T, S, R with subscripted attributes.
Head(T) = A_1 A_2 A_3 A_4 Note notation: set of attributes is just a LIST.

VERY IMPORTANT CONCEPT. The number of rows changes frequently and rows are not remembered by users; the columns usually DON'T change in number, many names are remembered, and USED TO POSE QUERIES.

Column type (**Column Domain**) A table is DECLARED in SQL. Columns have certain TYPES as in SQL: float4, integer, char(13).

Idea of Domain in Relational Algebra, is like an enumerated type.
Domain of City: all the city names in the U.S.
Domain of DIscnt: all float #s between 0.00 and 20.00.

This concept is not implemented in commercial systems today. Only have types such as char(13) and float4. But assume it in Relational Algebra.

Say CID = Domain(cid), CNAME = Domain(cname), CITY and DISCNT are the domains for CUSTOMERS table columns, then consider:

CID x CNAME x CITY x DISCNT (Cartesian Product)

consisting of all tuples: (w, x, y, z), w in CID, x in CNAME, . . .

ALL POSSIBLE tuples: (c999, Beowulf, Saskatoon, 0.01)

A mathematical relation between these four domains is a subset of the Cartesian product. E.g., if have 4 attributes, A_1, A_2, A_3, and A_4 , and T is a relation such that Head(T) = A_1 A_2 A_3 A_4, then:

T □ Domain(A_1) x Domain(A_2) x Domain(A_3) x Domain(A_4)

T is said to relate a data item in Domain(A_1) with one in Domain(A_2), . . .

Section 2.3. Relational Rules. Idea is to make all products have same characteristics. But a bit too mathematical, overlooking important implementation aspects, so some rules are broken by most products.

Rule 1. First Normal Form rule. Can't have multi-valued fields. (Repeating fields). See pg. 37. All pure *relational* products obey this rule.
- But new object-relational products break this rule
- Thus can't have employees table with column "dependents" which contains multiple dependent's names (Figure 2.3)
- Could create one table with duplicates on different rows (e.g., employees-dependents table join of employees and dependents), but this is bad for other reasons. (Figure 2.4)
- Ends up meaning we have to create two tables and *join* them in later queries. (Figure 2.5)
— in OR model, Figure 2.3 is OK, but won't handle this for awhile so as not to confuse you; assume relational -- no multi-valued fields.

Rule 2. Access rows by content only. Can't say: the third row down from the top. No order to the rows. (Also: No order to the columns.)
- Disallows "pointers" to rows, e.g. Row IDs (RIDs) or "refs".

- Most relational products break this rule by allowing users to get at rows by RIDs; new object-relational products have refs as part of syntax.

Rule 3. Unique rows. Two tuples cannot be identical in all column values at once. A relation is an unordered SET of tuples (2 rows can't be same in all attributes). But many products allow this for efficiency of load.
- There are even some tables where it is a good thing (temperature readings in table, might repeat).

But in the current Chapter, Chapter 2, we will assume that all these rules hold perfectly.

Section 2.4. Keys and Superkeys.

Idea is that by intention of DBA, some set of columns in a table distinguishes rows. E.g., cid, ordno.

In commercial product, DBA declares which set of columns has this property and it becomes impossible for two columns to have the same values in all these columns.

It is USEFUL to have such a key for a table, so other tables can refer to a row, e.g.: employee number, bank account ID.

In CAP database, keys are columns: cid, aid, pid, and ordno.

Consider a new table, *morders*, sums up qty and dollars of orders table for each tuple of month, cid, aid, and pid. Key ordno of orders table must go away because several rows with different ordno values are added together - Key for morders is: month, cid, aid, pid.

A superkey is a set of columns that has the uniqueness property, and a key is a minimal superkey: no subset of columns also has uniqueness property.

A superkey for CUSTOMERS is: cid, cname; A key is: cid (alone)

Rigorous definitions on pg. 37, Def 2.4.1. (This is a rephrasing.)
Let $A_1, A_2, \ldots, A_n$ be the attributes of a table T (subset of Head(T)). Let S be a subset of these attributes, $A_{i1}, A_{i2}, \ldots, A_{ik}$. Then S is a SUPERKEY for T if the set S distinguishes rows of T **by designer intention**, i.e. if u and v are two rows then for some A_{im} in S, $u[A_{im}] ? v[A_{im}]$ (new notation).

Consider the set of all columns in T. Is this a superkey? (Yes, by relational rule 3. Designer intention must accept this rule.)

A set of attributes K is a KEY (a CANDIDATE KEY) if it is a minimal superkey, i.e., no proper subset of K will also act as a superkey.
- A singleton column superkey (cid) is always a key. Go to definition.

Question. Look at CAP, pg. 28. **Is pname a key for PRODUCTS?**

If it is a superkey, it is a key, because no subset except null set. (No, because not intention of designer. A superkey must remain one under all possible legal inserts of new rows.)

-There may be more than one candidate key. Consider table of agents with column for SSN added. Designer intention that SSN is unique?

Yes, this is a fact, and designer must accept this in intentions. (Product wouldn't allow duplicate SSN for two employees.) May also have full name and address, and need to get mail implies that set of columns is unique.

Example 2.4.1. Consider the table

T

A	B	C	D
a1	b1	c1	d1
a1	b2	c2	d1
a2	b2	c1	d1
a2	b1	c2	d1

Assume intent of designer is that this table will remain with the same contents. Then can determine keys from content alone (this is a VERY UNUSUAL situation).

(1) No single column can be a key. (2) No set of columns can be a key if it contains D. (3) Pairs: AB, AC, BC. Each of these pairs distinguishes all rows. Therefore all are keys. (4) All other sets (must have 3 or more elements) contain one of these or else D as a proper subset. (Show subset lattice). Therefore no more keys.

Various keys specified by intent of DBA are called Candidate Keys. A Primary Key is defined to be the candidate key chosen by the designer to identifies rows of T used in references by other tables.

As ORDERS references CUSTOMERS by cid. Probably keep that way of referencing even if cname city was another candidate key.

3.

Assignment Read to end of Chapter 2. Homework: All undotted problems through end of Chapter 2, due at 6.

Theorem 2.4.2. Every table T has at least one key.

Idea of proof based on fact that a key is a minimal set of columns in T that distinguishes all rows: if u and v are different rows and K is a key, then designer intention: u[K] (u restricted to K) is not identical to v[K].

Since the set of all columns has that property (is a superkey for T — note always relative to table T), seems that there MUST be a MINIMAL set of columns that does. (NOTE PARTICULARLY: dealing with SETS of columns.)

But many with no mathematical training can't show this. Like you wanted to write a program to find such a set, and argue why the program terminates.

Proof. NAME ITEMS. Given a table T with Head(T) = $A_1 \ldots A_n$.
SET UP TO LOOP: Let attribute set S_1 be this entire set. (In what follows, we assume we know the intentions of the table designer to decide which sets of columns will always distinguish all rows, but can't answer the questions until the set of columns is named.)

LOOP: Now S_1 is a superkey for T; either S_1 is a Key for T or it has a proper subset S_2, $S_2 \subset S_1$, such that S_2 is also a superkey for T. Now, either S_2 is a Key or it has a proper subset S_3, $S_3 \subset S_2$, such that S_3 is also a superkey for T. Can this go on forever? (OK if you say can't go on forever. Mathematician: why not?)

But each successive element in the sequence $S_1, S_2, S_3, \ldots$ has a smaller number of columns (it is a PROPER subset), and we can never go to 0 (0 columns don't have unique values) or below. Thus this must be a finite sequence with a smallest set at the end S_n. That must be the key. QED. You should see why it wasn't enough to stop without giving the loop and explaining why it terminates: proof explains as much as possible.

Def. Primary Key. A Primary Key of a table T is the candidate key chosen by the DBA to uniquely identify rows in T (usually used in references by other tables in "Foreign Key"). Thus on pg. 28, aid is the Primary key for agents and used as a foreign key in orders.

Note that this Def implies that there MIGHT be a situation where a table doesn't have to have a primary key, as when there is only one table in the database. Most books don't allow this idea, but commercial databases do.

Null values Insert a new agent (a12, Beowulf, unknown, unknown)

Agent hasn't been assigned a percent commission or city yet (still in training, but want to have a record of him).

A null value is placed in a field of a table when a specific value is either unknown or inappropriate. Here it is unknown. A slightly different meaning would be if warehouse manager also had a percent (commission) field, but since warehouse managers don't get commissions, would get null value.

A null value can be used for either a numeric or character type. BUT IT HAS A DIFFERENT VALUE FROM ANY REAL FIELD. In particular, it is not zero (0) or the null string (''). It is handled specially by commercial databases.

If we were to ask for the agent with the minimal percent or the shortest city name, we wouldn't want to get an agent with no percent or city yet assigned.

Similarly, if we were to ask for the average percent, wouldn't want to average in zero for a null.

In fact, if we were to ask for all agents with percent > 6, and then all agents with percent <= 6, we wouldn't get the new agent Beowulf in either answer. Just not a meaningful question.

2.5 Relational Algebra. Operations on tables to get other tables. Codd. Interesting thing is can get answer to many useful Qs (first order predicate logic). But abstract language: we can't use a machine to get answer.

Two types of operations: Set Theoretic (depend on fact that table is a set of rows), and Native operations (depend on structure of table). Given two tables R and S, Head(R) = $A_1 \ldots A_n$, (S often the same), define 8 operations. See pg. 41-42. Put on board.

Note: symbols are not portable between versions in Microsoft Word, so these notes use the keyboard forms.

SET THEORETIC OPERATIONS

NAME	SYMBOL	KEYBOARD FORM	EXAMPLE
UNION		UNION	R UNION S
INTERSECTION		INTERSECT	R INTERSECT S
DIFFERENCE		- or MINUS	R - S, or R MINUS S
PRODUCT		x orTIMES	R x S, or R TIMES S

SPECIAL OPERATIONS

NAME	SYMBOL	KEYBOARD FORM	EXAMPLE

PROJECT	R □ □ □ □ □ □	R [ ]	R [A_{i_1}. . .A_{i_k}]
SELECT	R where C	R where C	R where A_1 = 5
JOIN		JOIN	R S, or R JOIN S
DIVISION		DIVIDEBY	R DIVIDEBY S

Idea of Keyboard form: name to use for an operation on a typewriter that doesn't have special operation symbol.

Set Theoretic Operations

Def. 2.6.1. Two tables are said to be compatible iff they have the same schema. (iff means "if and only if", or "means exactly the same thing".)

R

A	B	C
a_1	b_1	c_1
a_1	b_2	c_3
a_2	b_1	c_2

S

A	B	C
a_1	b_1	c_1
a_1	b_1	c_2
a_1	b_2	c_3
a_3	b_2	c_3

Illustrate R UNION S. Union of two tables considering table as set of rows. 5 rows in result. Clearly must be compatible tables, because rows of different schemas wouldn't fit in same table.

Illustrate R INTERSECT S. two rows.

Illustrate R - S. only one row.

illustrate S - R. two rows.

Before we go on to product (Cartesian product), consider idea of assignment or alias.

Def 2.6.3 Let R be a table with Head(R) = A_1. . . , A_n, and assume we wish to create a table S with Head(S) = B_1, . . ., B_n, B_i attributes such that Dom(B_i) = Dom(A_i) for all i, 1 = i = n, which has the SAME CONTENT as A. We can define the table S by writing the assignment

$S(B_1,. . .,B_n) := R(A_1,. . .,A_n)$.

The content of the new table S is exactly the same as the content of the old table R, that is, a row u is in S if and only if a row t exists in R such that u[B_i] = t[A_i] for i, 1 = i = n. The symbol := used in this assignment is called the assignment operator.

If don't need to rename columns, just want a different table names with same attributes, we call this a table alias, and write:

S := R.

Use alias operator to save intermediate results of operations. Can write:

T := (R INTERSECT S) - (R □ S),

or could write instead:

T1 := (R INTERSECT S)
T2 := (R □ S)
T := T1 - T2 (Draw picture of this result)

NOTE THAT we often do not absolutely NEED to get intermediate results. Can usually use subexpression in place of named alias. One example below where absolutely need it.

OK, now PRODUCT or Cartesian product. Example of R □ S (from above, different than book): (calculate it). NOTE Tables NEED NOT BE COMPATIBLE for a product to be calculated. (Write following Def for terminology.)

Definition 2.6.4 Product. The *product* (or *Cartesian product*) of the tables R and S is the table T whose heading is Head(T) = $R.A_1 \ldots R.A_n\ S.B_1 \ldots S.B_n$. For every pair of rows u, v in R and S, respectively, there is a row t in T such that $t(R.A_i) = u(A_i)$ for $1 = i = n$ and $t(S.B_j) = v(B_j)$ for $1 = j = m$. No rows exist in T that do not arise in this way. The product T of R and S is denoted by R TIMES S.

Columns of product table are qualified names. Used to distinguish identically named attributes. See Example 2.6.4 for example. Difference between Cartesian product of sets and Relational product.

Special problem if try to take product of table with itself. R TIMES R would have identically named columns. Therefore say S := R and take R TIMES S.

2..7. Native Relational Operations

Projection. Limit columns and cast out duplicate rows. (Example, not Def.)

PROJECTION. Given table R where Head(R) = $A_1, \ldots, A_n$, project R on subset of columns T := $R[A_{i_1}, \ldots, A_{i_k}]$, where list in brackets is a subset of the complete set of attributes. Cross out all columns and column values in R not in set, then eliminate duplicate rows. (See Def 2.7.1 in text.)

Example 2.7.1. List all customer names from the CUSTOMERS table of Figure 2.2. Answer to Query: CUSTOMERS[cname]. Result is table with cname heading, TIpTop, Basics, Allied, ACME. (DRAW IT) (Note cast out duplicate rows.)

List city and percent commissions for all agents. Any duplicates go away? (New York, 6; but note duplicates in one of two columns is OK.)

SELECTION. R where Condition. The Condition is a logical condition that can be determined from the values of a single row of C. Very simple kind of conditions. Definition 2.7.2.

$A_i \theta A_j$ or $A_i \theta$ a, where A_i and A_j are attributes of R, a is a constant.
θ is any of: <, >, =, <=, >=, <>

If C, C', also get C and C', C or C', and finally: not C.

Example: Give the cid and cname of all customers living in Dallas with discount greater than 8.

(CUSTOMERS where city = 'Dallas' and discnt > 8) [cid, cname]

4.

Assignment
Homework (Hardcopy, Due in one week.): all non-dotted problems in Ch 2.

Display on board R TIMESS (Cartesian Product) (remember qualified names) for R & S below.

R

A	B
a_1	b_1
a_1	b_2
a_2	b_1

S

B	C
b1	c_1
b2	c2
b3	c_3

But now show what it is for R JOIN S.

JOIN. Start by taking R TIMES S.
Cross out rows where equivalently named attributes with qualifiers R or S in R TIMES S do **not** have equal values (do it above, rewrite smaller table).
Do away with one column with duplicate name; drop qualification names (don't need, since now all columns have different names)

Draw lines between matching rows in R & S. Note b3 in table S has no matching row in table R, does not appear in join. Write out R JOIN S.

Example: Look at: ORDERS JOIN CUSTOMERS (pg. 28,). For each row in orders, exactly one in customers with same cid value (only common name). Draw one row. extends orders row with more information about customer.

Most Relops defined now. Things can get kind of complex in forming relational algebra expressions to answer queries; take things one step at a time.

Example: Give the aid values of agents who take orders in dollar volume greater than 500 for customers living in Kyoto.

In following, assume have: C := CUSTOMERS, A := AGENTS, P := PRODUCTS,
O := ORDERS.

Note will need info from ORDERS and from CUSTOMERS, since don't know where customers live in ORDERS, don't know anything about orders in CUSTOMERS. Join ORDERS and CUSTOMERS, and have necessary info.

ORDERS JOIN CUSTOMERS or O JOIN C

But now need to select, then project.

((O JOIN C) where city = 'Kyoto' and dollars > 500.00) [aid]

Would the following be OK?

O JOIN C where city = 'Kyoto' and dollars > 500.00 [aid]

See precedence of operations table, pg. 53. Strongest binding is projection. [aid] would try to bind to C (no such column). What about

(O JOIN C) where city = 'Kyoto' and dollars > 500.00 [aid]

Projection first, then where clause doesn't work. Instead:

((O JOIN C) where city = 'Kyoto' and dollars > 500.00) [aid]

(Slow.) Note could have done this differently)

((C where city = 'Kyoto') JOIN (O where dollars > 500.00))[aid]

Example. Give all (cname, aname) pairs where the customer places an order through the agent. Need to involve three tables. Does this work?

(C JOIN O JOIN A) [cname, aname]

Look at pg. 28. No, but why not? (Slow: look at definition of Join). OK that haven't defined double join: turns out that (R JOIN S) JOIN T = R JOIN (S JOIN T), so can define R JOIN S JOIN T as either one of these.

Problem is: build up attribute names going left to right. Results in all (cname, aname) pairs where the customer places an order through the agent and the customer and agent are in the same city. To leave that out, need:

((C[cid, cname] JOIN O) JOIN A) [cname, aname]

Cuts out city in first join. Why need two attributes?

Example: give (cid, cid') pairs of customers who live in the same city.
Look at pg. 28. What do we want? (c001, c004) and (c002, c003). Will the following do this? (Assume K := CUSTOMERS)

(C JOIN K) [C.cid, K.cid]

No. No qualified names in join result, no duplicated attribute names. In fact C JOIN K (same attribute names) looks like what? (Look to def of Join.) OK, don't always use Join, try:

((C JOIN K) where C.city = K.city)[C.cid, K.cid]

Look at pg. 28. Draw multiplication table, 5 rows (C cid values), 5 columns (K cid values), put X in when equal city. Problem, will get (c002, c002). Will also get both (c001, c004) and (c004, c001). To solve this, write:

((C x K) where C.city = K.city and C.cid < K.cid)[C.cid, K.cid]

Like choosing lower triangle below the diagonal.

Now, Relational Division. Define it in terms of Cartesian product. The idea is, as with integers, given two integers X and Y, define Z = X/Y, so that Z*Y = X. Can't always quite do this, even with integers, so instead define Z to be the largest whole number smaller than X/Y: e.g., if X = 17 and Y = 5, then 3.4 = 17/5 and Z = 3. Same with table division.

Definition 2.7.5. Division. Consider two tables R and S, where the schema of S is a subset of the schema of R. Tables R and S:

Head(R) = $A_1 \ldots A_n B_1 \ldots B_m$, and Head(S) = $B_1 \ldots B_m$.

The table T is the result of the *division* R ÷ S (which is read as "R DIVIDEBY S") if Head(T) = $A_1 \ldots A_n$ and T consists of those rows t such that for *every* row s in S, the row resulting from concatenating t and s can be found in table R, and there is no larger set of rows for which this is true.□

Note there are no columns in common between S (Head(S) = $B_1 \ldots B_m$) and T = R ÷ S (Head(T) = $A_1 \ldots A_n$), but since Head(R) = $A_1 \ldots A_n B_1 \ldots B_m$, we see that S and T have the proper schemas so that we might be able to say: T x S = R.)
(This is the division operation we want to define, the inverse of product.) If R = T JOIN S, then we will be able to say that T = R ÷ S.

Might start with R and S so R ? T JOIN S, for any possible choice of T.

Still, when T = R ÷ S, the table T contains the <u>largest possible</u> set of rows such that T □ S is contained in R. (Exercise 3.) **<u>Make Analogy to Integer Division</u>**.

Example 2.7.9. (In book) Start with the table R given by:

R

A	B	C
a1	b1	c1
a2	b1	c1
a1	b2	c1
a1	b2	c2
a2	b1	c2
a1	b2	c3
a1	b2	c4
a1	b1	c5

We list a number of possible tables S and the resulting table T := R ÷ S.

S

C
c1

T

A	B
a1	b1
a2	b1
a1	b2

Note that all of the rows in S JOIN T are in R, and there couldn't be any larger a set of rows in T for which that is true because there are only three rows in R with a c1 value in column C.

S

C
c1
c2

T

A	B
a1	b2
a2	b1

All of the rows in S JOIN T are in R, and there couldn't be any larger a set of rows in T with this true: look at rows in R with C values c2.

S

C
c1
c2
c3
c4

T

A	B
a1	b2

S x T in R, and by looking at the rows in table R with C value c3 and c4, we see why T has the maximal content with this property.

S

B	C
b1	c1

T

A
a1
a2

Example of dividing R with three columns by a table S with two columns, resulting in a table T with a single column. T is maximal.

S

B	C
b1	c1
b2	c1

T

A
a1

Why is T maximal?

Next we given an example to see why we use division: what kind of question does it answer about the data.

Recall C:= CUSTOMERS, A := AGENTS, P := PRODUCTS, O := ORDERS

Example 2. Get names of customers who order all products. When see "all" think "division". Let's get cids of such customers.

O[cid, pid] ÷ P[pid]

Clear why must project P on pid: columns of divisor must be subset of columns of dividend.

Why project O on cid, pid? Otherwise will try to find other column values in ORDERS (e.g. qty) the same for all pid, but just want cid the same for all pid.

To get names of customers, join with C.

((O[cid, pid] ÷ P[pid]) JOIN C)[cid]

Example 3. Get cids of customers who order all products priced at $0.50. Build new table to take place of P above.

((O[cid, pid] ÷ (P where price = 0.50)[pid]) JOIN C)[cid]

Example 4. Get cids of customers who order all products that anybody orders. (Maybe few new products noone orders yet.)

O[cid, pid] ÷ O[pid]

OK. Some of the 8 operations of relational algebra: UNION, INTERSECTION, -, TIMES, PROJECT, SELECT, JOIN, DIVISION are unnecessary. Just carry them for ease of use.

5.

Hand in hw 1 on 6.
Assignment: reading through Section 3.5. Assignment to do exercise 3.2.1 (start up database), (try for next time), then UNDOTTED exercises at end of Chapter 3 through 3.7 (some in 3.7 are VERY HARD, OK if can't do — Ph.D. qual level question.)

OK. Some of the 8 operations of relational algebra: UNION, INTERSECTION, -, TIMES, PROJECT, SELECT, JOIN, DIVISION are unnecessary. Just carry them for ease of use.

Th. 2.8.1, don't need both INTERSECT and MINUS.

A □ B = A - (A - B). Show this with a picture.

Th. 2.8.2. Can define JOIN of R and S in terms of TIMES, PROJECT and ASSIGNMENT.

JOIN. Start by taking R X S.
Cross out rows where equivalently named attributes in R X S (such as cid in customers and cid in orders) don't have same value.
Do away with columns with duplicate values; drop qualification names (don't need, since now all columns have different names)

Say Head(R) = AB and Head(S) = BC.
Take T1 := R TIMES S (attributes are R.A, R.B, S.B, S.C).
Then take T2 := (T1 where R.B = S.B) [R.A, R.B, S.C]. (equate and project)
Finally let T3(A, B, C) := T2(R.A, R.B, S.C) (rename the attributes, get rid of S.B).
T3 is equivalent to R S.

DIVISION. (**Hard***) If Head(R) = $A_1 \ldots A_n\ B_1 \ldots B_m$ and Head(S) = $B_1 \ldots B_m$:

$R \div S = R[A_1 \ldots A_n] - ((R[A_1 \ldots A_n] \text{ X } S) - R)\ [A_1 \ldots A_n]$

Example: See Example 2.7.9 on pg. 58, the second S there. Work out.

$R[A_1 \ldots A_n]$ is R[A B], with three rows, (a1, b1), (a2, b1), and (a1, b2).

Calculate X S (6 rows), and subtract R. Only one we keep is (a1, b1, c2) and project on [A B] get (a1, b1), subtract from R[A B], result is two rows.

First term on left ($R[A_1 \ldots A_n]$) is certainly maximum possible answer. It will be the answer (T) if $R[A_1 \ldots A_n]$ X S = R: Then T X S = R, definition.

But if it's too large, then $R[A_1 \ldots A_n]$ X S contains more than R.

Consider reducing $R[A_1 \ldots A_n]$ by subtracting out the part that gives rows not in R, i.e., subtract out: $(R[A_1 \ldots A_n] \text{ X } S - R)\ [A_1 \ldots A_n]$

Claim now we have:

$(R[A_1 \ldots A_n] - ((R[A_1 \ldots A_n] \text{ X } S) - R)\ [A_1 \ldots A_n]) \text{ X } S \subseteq R$, and this is max.

Now, we seem to have an awful lot of capability with these operations, to create **queries** that answer requests for data. Describe any columns you want as a result (do with project), any way in which different tables are interconnected (product then select is more general than join).

CAN DO THIS IN GENERAL, as long as connections can be described with the conditions we have specified. (<, <=, =, <>, >=, >)

Could think of conditions not so described (city contains a character pattern 'ew'; New York or Newark), but could add new conditions: trivial.

Even FOR ALL type queries can be done without division really, not easy intuition.

BAG OF TRICKS FOLLOWS (EXAM Qs): PAY CAREFUL ATTENTION!!!

Example 1. Get aids of agents who do not supply product p02.

A[aid] - (O where pid = 'p02')[aid]

Example 2. Get aids of agents who supply only product p02.

O[aid] - (O where pid <> 'p02')[aid]

The question seems to imply the these agents DO supply product p02, so no good to say:

A[aid] - (O where pid <> 'p02')[aid]

(English is a bit ambiguous, and we may want to clarify.) Will this relational algebra expression have any agents who do not place an order for product p02 either? Do we need to write:

(O where pid = 'p02')[aid] - (O where pid <> 'p02')[aid]

(No, for has to be an order and it can't be for any product other than p02.)

Example 3. Get aids of agents who take orders on at least that set of products ordered by c004. (How?) (Rephrase: take orders on ALL products ordered by c004.)

O[aid, pid] ÷ (O where cid = 'c004')[pid]

Example 4. Get cids of customers who order p01 and p07. (How?) (Wrong to write: (O where pid = 'p01' and pid = 'p07')[cid])

(O where pid = 'p01')[cid] INTERSECT (O where pid = 'p07')[cid]

Example 5. Get cids of customers who order p01 **or** p07. (Can use or.)

(O where pid = 'p01' or pid = 'p07')[cid])

Example 6. List all cities inhabited by customers who order product p02 or agents who place an order for p02. (Can't use **or** in selection condition Why?)

((O where pid = 'p02') JOIN C)[city] □ ((O where pid = 'p02') JOIN A)[city]

Example 7. Get aids of agents who place an order for at least one customer that uses product p01. **NOTE**: might be that an agent retrieved does NOT place an order for p01. Consider diagram that follows.

```
                        /--     Agents
   /  --  Customers ---         who
p01 --- who order  ---         place orders
   \  --   p01  ---      for those
                        \--     customers
```

We are retrieving cloud on right. This is one of the most missed problems on exams. Do it inside-out. Customers who order product p01.

(O where pid = 'p01')[cid]

Agents who place orders for those customers.

((O where pid = 'p01')[cid] O) [aid] (Note OK: Join not Qualified.)

Example 8. Retrieve product ids for all products that are not ordered by any customers living in a city beginning with the letter "D". OK, start with the sense of "products not ordered".

P[pid] - (O (C where . . .))[pid]

This works if we come up with a condition for . . . that means living in city beginning with the letter "D". Right? Any ideas? Only have conditions: <, <=, =, <>, >=, >. With alpha strings, means earlier in alphabetical order.

C where C.city >= 'D' and C.city < 'E'

In the solved Exercise, in 2.5 (i), see how it is difficult to find customers who have the largest discount. (Good Exam question.)

Example 9. Retrieve cids of customers with the largest discounts.

CY := C

T1(cyid, cid) := ((CY □ C) where CY.discnt >= C.discnt)[CY.cid,C.cid]

Now have pairs (cyid, cid) where the ones with larger or equal discount are on the left. The cids with maximum discount will be paired with all other cid values on the right.

T2 := T1 ÷ CY[cid]

6. Hand in hw 1.
Assignment: reading through Section 3.5. UNDOTTED exercises at end of Chapter 3 through 3.7. Due 9. Tried Ex 3.2.1 (start up database)? How did it go?

FAST: Idea of outer join. Given two tables R and S, Head(R) = AB, Head(S) = BC,

R

A	B
a_1	b_1
a_2	b_2
a_3	b_5

S

B	C
b1	c_1
b2	c2
b3	c_3
b4	c4

What is the join, R JOIN S?

R JOIN S

A	B	C
a_1	b1	c_1
a_2	b2	c2

But we're losing a lot of information. Outer join KEEPS all the information of the two tables, filling in nulls where necessary.

R OUTER JOIN S

A	B	C
a_1	b1	c_1
a_2	b2	c2
a3	b5	null
null	b3	c3
null	b4	c4

You use this sort of thing when you don't want to lose information. Example given in the text is that even though an agent in AGENTS table has made no sales in SALES table, we DO want to see AGENT name in report.

Most commercial product versions of SQL (Including Oracle) already have outer join

Theta join. Recall in equijoin, of R and S, Head(R) = AB, Head(S) = BC:
I.e., (R x S) where R.B = S.B

What if wanted some other condition (allow common columns),

(R x S) where R.B □ S.B, where □ is <, <=, <>, >, >=, anything but =.

No big deal. In SQL, use product, no special name for this.

Chapter 3 Idea of SQL. Computerized query language.

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

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

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

```
%sqlplus
```

Any problems, contact DBA for course.

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

Create table. Result is empty table.

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

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

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

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

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

```
c001,TipTop,Duluth,10.00          <-- comma separated, no spaces
c002,Basics,Dallas,12.00
```

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

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

```
select * from customers;\g
```

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

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

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

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

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

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

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

The FROM clause names tables to take Cartesian product.

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

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

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

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

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

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

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

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

select distinct pid from orders;

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

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

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

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

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

In SQL:

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

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

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

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

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

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

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

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

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

Can perform calculations in select.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

to:

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

Obviously saves effort, since less looping.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
FOR c FROM ROW 1 TO LAST OF customers
    discard c if c.discnt < 10
    create L as empty list
    FOR x FROM ROW 1 TO LAST OF orders
        Add x.pid to list L if x.cid = c.cid   /* value set outside loop */
  END FOR x
  discard c if 'p05' not in list L
  otherwise place c.cname in ANSWER list
END FOR c
print out ANSWER list
```

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

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

Can evaluate this with precalculation of subquery.

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

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

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

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

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

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

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

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

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

Quantified Comparison Predicates. List:

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

If comparison operator □ stands for operator in the set {<, <=. =, <>, >, >=},

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

□ some and □ any mean same thing: □ some is easier to remember.

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

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

8.

Exam 1 on 13 Responsible for most of Chapter 3.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The Exists predicate.

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

Use not exists most often to handle FOR ALL conditions.

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

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

OK, that works, but also:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Is not exists more powerful than not in?

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

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

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

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

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

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

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

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

Harder in SQL. Formulate as follows.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9.

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

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

Section 3.6

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

These operators are NOT uniformly available in the different products.

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 3.10 Advanced SQL Subquery Form

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

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

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

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

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

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

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

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

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

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

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

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

With use of INTERSECT, this becomes:

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

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

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

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

Implementation of these operators.

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

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

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

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

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

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

Figure 3.11 SQL-99 Recursive tableref Definition

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

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

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

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

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

Foreshadowing in this query form of something called Views.

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

Can say: tableref1 CROSS JOIN tableref2

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

Same effect as

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

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

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

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

OK, next is NATURAL JOIN.

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

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

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

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

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

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

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

Correct this with following form:

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

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

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

or here's a condition join:

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

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

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

OK, next is **OUTER JOIN.**

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

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

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

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

S

C	A
c1	a1
c3	a3
c4	a4

T

A	B
a1	b1
a2	b2
a3	b3

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

select * from S FULL OUTER JOIN T

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

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

select * from S FULL OUTER JOIN T

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

Join Forms in operational Database Systems

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

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

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

10.

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

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

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

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

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

COL1
1400.00

Don't get confused, not like:

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

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

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

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

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

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

ORACLE has a few more, stddev and variance:

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

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

Note max and min return char max and min.

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

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

Note that these three MIGHT NOT give same values:

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

Surprising? Doesn't insist on <u>different</u> city values. But:

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

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

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

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

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

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

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

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

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

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

<u>REMEMBER THIS: MUST USE SUBQUERY IF SET FUNCTION IN WHERE!</u>

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

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

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

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

Handling Null Values.

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

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

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

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

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

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

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

```
select count(*) from customers;

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

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

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

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

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

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

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

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

11.

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

Section 3.8. Groups of rows.

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

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

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

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

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

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

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

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

This has the effect of the loop:

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

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

Now a surprise! If say:

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

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

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

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

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

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

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

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

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

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

Can't do it by saying:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Subquery General Form, see pg. 135.

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

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

Tableref := tablename [corr_name]

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

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

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

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

Full Select General Form

subquery

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

mgrid char(4));

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

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

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

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

or two levels,

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

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

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

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

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

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

Added to next homework: Try this on Oracle.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

12.

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

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

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

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

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

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

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

TRUTH VALUES: T, F, and U

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

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

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

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

NOT	
T	F
F	T
U	U

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

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

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

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

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

where c.cid = o.cid));

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

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

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

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

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

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

The Between Predicate.

expr1 [not] between expr2 and expr 3

Conceptually equivalent to (without not):

expr1 >= expr2 and expr1 <= expr3

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

Quantified Predicate.

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

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

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

Ex. 3.9.3. Retrieve maximum discount of all customers.

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

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

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

Is null Predicate.

columname is [not] null

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

Like predicate.

columname [not] like *'pattern'*

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

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

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

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

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

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

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

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

Section 3.10.

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

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

Section 3.10 The Insert, Update and Delete Statements.

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

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

Example 3.10.1.

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

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

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

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

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

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

The Update statement. Basic SQL form is this.

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

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

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

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

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

Delete statement.

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

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

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

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

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

```
delete from agents where aid in (select aid from orders group by
      aid having sum(dollars) < 600);
```

Chapter 4 Object-Relational Databases

Introduction

In the sense that we will use it, an object holds together a bundle of related information. An employee has a name, id, address, etc., so we can have an employee object for each employee. A document has a title, author, length, and date, so we can have a set of document objects in a library object.

All the employee objects are of a defined **object type**. Associated with an object type are the data attributes it has and the functions (methods) defined for it.

Here we use "object" in a loose sense, **without encapsulation**, even though some object-oriented experts would say that encapsulation is so essential to objects that we are mis-using the word object.

Encapsulation is also known as data-hiding, and is a software technology that builds walls around a combination of code and data and only allows interaction with other parts of the code via methods or messages. In Java, a with only private data members is encapsulated.

The object types we will be using are like Java es with all public data members, or structs in C; the data inside can be accessed easily. The object can be thought to *organize* the data without *policing* it.

Relational databases have always had the outlook that all data is accessible (except for secret data) so that you don't have to know ahead of time how you might use it. The data is strongly organized into neat tables of rows and columns. The object types give us a new way to organize data items.

But it does break RR1 by allowing combination data items. The collection types break RR1 another way by allowing whole sets of data items as column values. Look at Figure 4.1. Also read Chapter 1.

We will cover:
Section 4.2.1--ORACLE object types.
Section 4.3.1--ORACLE collections

We don't have time to cover how to write object methods (Section 4.4.1) in PL/SQL, but encourage you to read it, and also read the last section on packaged software.

Look at Example 4.2.1. Like a C struct declaration or Java declaration. Like these, it provides a **new type,** an **object type,** out of a combination of other types, called members in C/Java, **attributes** in Oracle object types.

This new object type can be used wherever ordinary SQL data types can be used, for ex., as a column type.

Ex. 4.2.2 create table teachers ... Note that teachers is a normal "relational table" that has a column type that is an object type. After this, have an empty table.

Next, insert some rows. Now finally have "objects", not just object types. Objects are instances of object types, that is, real data held together as prescribed by the type. Here the name_t objects are held as column values of the tname column.

select * from teachers;

This shows all the column values, including the tname column of object type name_t:

```
tid       tname                          room
----      -------                       ------
1234   name_t('Einstein', 'Albert', 'E')   120
1235   name-t(...)                          144
...
```

Note the display of the name_t objects. Since the name_t objects are column values, they are called **column objects**.

select t.tname.fname from teachers t;

This prints out just the fname. Note the dotted expression that digs down from the table alias all the way to the fname.

NOTE: The table alias is needed here and anywhere more than one dot is used, in Oracle.

```
select room from teachers
  where t.tname.lname like '%Neil';
```

Example 4.2.3--here we have an object type in an object type.

Table teachers was still a relational table, but table people is an **object table**, a new idea. Each row of people is of type person_t, and thus is called a **row object.**

We see that rows (with their columns) and object types (with their attributes) both provide ways of grouping several typed values together.

Table people has row objects of type person_t, and each of these has attributes ssno, pname, and age. These attribute names map naturally to column names. So we can say table people has columns ssno, pname, and age derived from the row object attributes.

Object table
-- each full row is of an object type
-- each row is a row object with certain attributes
-- the attributes of the row object provide effective column names
-- the table provides a natural home for objects of that object type

Relational table
-- each full row has one or more columns each of some data type, but no type describes the whole row.
-- some columns may have object types, providing column objects.

See Figure 4.2 for a picture of this.

There is a "duality" here:
an object table is, at the same time:
--a table of row objects
--a table of rows and columns where the column names are the top-level attribute names of the row object.

Look at Examples after Figure 4.2.

The whole row object can be evaluated by the expression "value(p)" where p is the table alias for the object table.

select value(p) vs. select *: select row object vs all-the-columns--see Figures 4.3 and 4.4. Note how the whole object is represented by

name_t('Delaney', 'Patrick', 'X').

This is also the form of the object constructor, shown in action in Example 4.2.5 to create a name to match in the query.

Dot notation

Just as in C and Java, we use dots to drill down into structured types. For example p.pname.fname specifies the fname attribute of the pname name_t object within the person_t row object referenced by the table alias p. Look at Example 4.2.6 to see this in use in a query.

Oracle is very picky about the use of the table alias in dotted expressions. The easy rule is simply to always use a table alias with dotted paths. In fact, Oracle does support simple cases without table aliases, to be consistent with the relational case. Look at the examples on p. 183.

To insert into an object table, we can view it as having various columns and providing values for them just as in the relational case, as shown in Example 4.2.7.

If we view the same object table in terms of row objects, it would seem natural to be able to insert a whole person_t object in all at once. However, this doesn't fit the Insert statement syntax, which allows either a values clause over the columns or a subquery, and Oracle has not stretched the syntax out to cover this possibility (as of version 8.1.5, anyway.)

However, Oracle has found a way to do a full-object Update of a row by letting the table alias symbolize the whole row, as shown on p. 184.

```
update scientists s set s = name_t(…) where …;
```

As you can see from Appendix C, p. 773, the use of the table alias in the Update statement is an extension of the standard form.

Object References

The row objects of object tables are somewhat special in that they have unique identifiers (this is related to the fact that they are full-fledged database rows), whereas the column objects are known only by their values. We can utilize this top-level identification information by defining REFs that, in effect, point directly to a row object.

The REF feature provides new built-in datatypes, of type "ref object_type", so we can have table columns that point to row objects in an appropriate object table.

For example in a new version of the CAP database, we can have a "ref product_t" column in orders that points to a whole row object representing a product.

Look at Example 4.2.9, where the details of this new version of CAP are laid out. Each order object has REFs to its customer, agent, and product objects. We still have the old relational identifiers as well, to help with loading the data. But we will utilize the REFs in queries wherever possible.

Note the scope clauses in the create table for orders. These are not required, but allow shorter (thus faster) REFs when they all (of one type) point into one table.

We will gloss over the issue of loading the table now and jump into doing queries on a fully loaded table. The dotted expressions (or "paths") can now follow a REF across to another table. For example

```
o.ordcust.city
```

(where o is a table alias for orders) accesses the city attribute of the pointed-to customer object via the REF o.ordcust housed in the order object.

In relational CAP, we needed a join on cid to make this association--here it is just a dot away. Further, we know that it is based on a unique identifier that is a "locator." It has information on how to find the row, not just a unique pattern of bits to search for.

Any relational query that does joins on orders with customers on cid, with agents on aid, and/or with products with pid is simplified by using REFs here.

For example, to list all cname, pname pairs where the customer ordered the product, we can just think of the cname and pname "hanging off" the order object:

```
select distinct o.ordcust.cname, o.ordprod.pname from orders o;
```

Two joins are saved here over the relational query.

Admittedly, only the easiest joins are saved. Many harder queries are not simplified. In Example 4.2.13, the notorious division query is rewritten with REFs. It does make a little clearer how the x row bridges from c to a, but the hard stuff is still all there.

We have to be aware of the possibility of "dangling" REFs. If an order has a REF pointing to a certain customer object, and that row is deleted from the customers table, then the REF in the order object is left "dangling", pointing nowhere, but not NULL.

Oracle has added a new predicate "is dangling" so that these REFs can be found and weeded out. Look at Example 4.2.14.

REF Dependencies

When you execute a Create Type that defines REF attributes, like order_t, the REF-target types (like customer_t, etc.) need to be already defined. We say that type order_t has a "REF type dependency" on customer_t, agent_t, and product_t. Similarly, we need to drop them in the opposite order, order_t first, then the others (in any order.)

This would seem to preclude circular REF type dependencies, but there is a loophole in the system to allow them, since they are so useful in certain circumstances.

For example, and employee_t type could have a REF department_t attribute for the employee's department, while the department_t type could have a REF employee_t attribute for the department's manager.

See pg. 190 for the trick to set this up--we can half-way define employee_t, then fully define department_t, and then finish defining employee_t. To drop these types, a special FORCE keyword needs to be used.

Loading Tables with REFs

Since we are only using REFs as extra columns in the CAP example, we can do the old relational load and then use an Update statement to set all the REFs, as shown in Example 4.2.17. We see it is using matching on the still-existent relational ids.

In general we need a way to specify the target of a REF uniquely by one or more columns, which of course means specifying a key in the target object table. And this key value must be obtainable from the source object, to do the match-up. Thus the simplest approach is to use REF columns in parallel with ordinary relational keys

Collection Types in Oracle

We have talked about the idea of multi-valued attributes before in Chapter 2, and how they break Relational Rule 1. See pg. 32. Relational techniques get rid of multi-valued attributes by setting up additional tables. Then we are constantly joining them back together to get our work done. Now, instead, we'll allow multiple values inside one column value by using collection types.

Look at Figure 4.11 to see collections of dependents for each employee.

Oracle has "nested tables", meaning a whole table (of dependents, for example) in a column value of another table. To make this work, new datatypes are made available, "table types" such as "table of person_t" or "table of int" that can be used as column datatypes.

As with ordinary tables, each nested table is a set of rows, with no order in the rows. Nested tables can be object tables or tables of a single column of a built-in type such as int.

If order needs to be maintained in a collection inside a column value, Oracle provides VARRAYs. We'll look into them after nested tables.

Oracle (version 8.1.5) allows only one level of nesting for nested tables. Employees can have a nested table of dependent_t objects, but these dependents can't have a nested table of hobbies. The employee can have a nested table of hobbies as well as dependents, since that still counts as single-level nesting.

To create a table that contains a nested table within it, we are first expected to define a table type and then come up with two tablenames. One tablename is for the whole table (the "parent" table) and the other is a helper (or "child") table to hold the rows of all the nested tables of one column. If we have two nested table columns, we have to come up with three tablenames.

See Example 4.3.1 for the employees and dependents example.

The syntax for the Create Table statement with nested tables is like the previous Create Table statements with an additional clause or clauses of the form "nested table colname store as tabname". Here the colname matches the columnname of a column of table type.

Although we can see the child tables as database objects (in say "select table_name from user_tables;"--see pg. 448,) we can't use them in queries directly. We always need to access the data through the parent table.

Each nested table is itself a column value of some table type. It is easy to display whole nested tables from SQL, as in Examples 4.3.2 and 4.3.3:

```
select dependents from employees where eid = 101;   -- one nested table
select dependents from employees;  -- one nested table per row
```

Figure 4.14 shows the output format for these retrieved nested tables. Following the object type display form, we see the type name and then a parenthesized list of the component parts, in this case the row objects of the nested table. As with object types, we can use the same form for constructing nested table objects.

Now the challenging part is retrieving just the desired information from nested tables rather than the whole tables. We can use TABLE(e.dependents) to convert the nested table column value (a single database item value) e.dependents into a query-compatible table of its contents, but note that there *is one of these for each row of the employees table.*

We can use TABLE(e.dependents) in a subquery, such as

```
select * from table(e.dependents)  -- subquery using TABLE
```

but this cannot stand as a top-level select because the table alias e is not defined. We can give e a definition by embedding this subquery in an outer query on employess:

```
select eid from employees e
        where exists (select * from table(e.dependents));
```

This query finds all employees with dependents. Many queries on nested tables have this format: one select runs over the parent table and the other over one or more child tables. See Examples 4.3.4 and 4.3.5.

We do have to be careful about using TABLE() in the right spots. Example 4.3.6 shows a query that produces different results based on whether or not TABLE() is applied to e.dependents. With it, we are accessing the rows of the table, whereas without it, we are accessing the whole table object, a single thing with count of 1.

In Example 4.3.7, we query one dependents table:

```
select d.ssno from table(select e.dependents from employees where e.eid = 101) d;
```

What happens if we remove the "where e.eid = 101" that specified a single nested table? Then TABLE() refuses to work with the multiple nested tables trying to flow into it. TABLE() requires a single nested table as an argument.

OK, but on a more logical level, how can we remove this specification of eid = 101 and retrieve *all* the ssno's for *all* the dependents? This is the "table of tables" data retrieval case. We are asking for data items from all the rows of all the nested tables in all the rows of the parent table.

It turns out there are two ways to do this in Oracle. The first and often easier method is by using a table product between the parent table and the child tables, for example:

```
select e.eid, d.ssno from employees e, table(e.dependents) d;
```

This table product generates a result row for each dependent. The result row contains one child table element value (here a dependent_t object) plus all the column values of its row in the parent table, even the column value containing the full nested table object.

Thus an employee with 3 dependents would get 3 rows in this result set, with triplicated employee column values. But an employee with no dependents doesn't show up at all!

To include the dependent-less employees, with null of course where the dependent_t object should show up, we put a (+) in the right spot, as follows:

```
select e.eid, d.ssno from employees e, table(e.dependents) (+) d;
```

Thus this table product has neatly filled out a big table with all (or nearly all, if you leave out the (+)) of the information of all the tables. This big table can be queried like any other.

For example, to find all the possibilities in groups of dependents of exactly the same age, where we are only interested in groups of 3 or more (for planning play groups, say), we could put:

```
select d.age, count(*) from employees e, table(e.dependents) d
        group by d.age having count(*)>= 3 order by d.age;
```

But exactly who are these dependents? We have their ages from this query, and that can be used to find out their identities:

```
select d1.ssno, d1.age from employees e1, table(e1.dependents) d1
        where d1.age in (select d.age from employees e, table(e.dependents) d
                group by d.age having count(*)>= 3)
        order by d1.age;
```

Note that table products turn a "table of tables" into an ordinary table, so it maps us back into the normal relational world. That's great for lots of cases, but sometimes we would like to maintain the hierarchical relationship of parent to child all the way through.

The Oracle CURSOR facility allows hierarchical presentation of query results (even from purely relational tables.) Unfortunately, its printout from SqlPlus is pretty ugly. See Figure 4.17. Perhaps this will improve in later releases.

The idea of the hierarchical output is simple: output the parent information once, and then print a table of all the relevant child rows for that parent, then go on to the next parent, and so on.

For our current example, employee 101 has two dependents and employee 102 has one, so we see the 101, then the table of two dependent rows, then 102, and finally a table of one row.

The first thing to note is that this result set is not a relational table! A relational table has rows that all have the same column types, very rectangular.

We see that the CURSOR provides another way of running a loop over rows, thus competing with SELECT for this privilege. But it runs many smaller loops, one for each row of the outer SELECT, and each loop execution produces a little table for that parent row.

We see that a CURSOR runs inside a SELECT. In fact, it can only be used in a select-list of a top-level SELECT, as in select expr, expr, …, cursor (…). See Examples 4.3.9 and 4.3.10 for more examples.

VARRAYs

VARRAYs are collection objects, like nested tables in many ways. But they differ in three important ways:

- VARRAYs maintain the order of their elements
- VARRAYs have a maximum number of elements, set at type definition time
- VARRAYs are held in the main table (usually), not in a separate storage table

In version 8.1.5 or later, VARRAYs work very much like nested tables in queries. You can use TABLE() to convert a VARRAY column value into a query-compatible table. In earlier versions, VARRAYs had to be cast to nested tables first, a terrible inconvenience.

Study the examples for details--there are no real new ideas here.

Inserts and Updates

Collection constructors can be used in the expected way to specify values for nested columns. See Example 4.3.15. The Update there should be:

```
update phonebook pb set extensions = extensions_t(345,999)
        where pb.phperson.ssno = 123897766;
```

The more interesting capability is inserting and updating the individual elements in nested tables or VARRAYs. We can use TABLE() to turn a single collection-value into a table that can be updated. We use a subquery to select the target collection-value, such as

```
TABLE(select e.dependents from employees e where e.eperson.ssno = 123897766)
```

and then embed this in an insert statement, as in Example 4.3.16:

```
insert into
table (select e.dependents from employees e where e.eperson.ssno = 123897766)
values (344559112, name_t('Smith', 'Diane', null), 0);
```

Chapter 5: Embedded SQL Programs.

Embedded SQL means SQL statements embedded in *host language* (C in our case). The original idea was for end-users to access a database through SQL. Called *casual users*.

But this is not a good idea. Takes too much concentration. Can you picture airline reservation clerk doing job with SQL? Customers waiting. Booking flight (update of seat table). Billing?

- Need to know all tables & columns, create complex syntax (FOR ALL)
- too much risk of mistakes, especially with updates

Instead, we have an *Application Programmers* create menu applications, perform selects and updates programmatically.

Programmers can spend a lot of time making sure the right SQL statement is used; programmers are temperamentally suited to this kind of work. Of course nice to have interactive SQL for some situations.

Aim for this chapter is to be able to implement ANY CONCEIVABLE ALGORITHM using a C program that can get at data through SQL statements.

Gets pretty complicated, but don't get lost -- there's a good reason for each new piece of complexity if you just understand it.

5.1 SQL statements in C (general host language) have slightly different syntax than the SQL we've seen so far.

```
exec sql select count(*) into :host_var from customers;
```

Statement starts with exec sql. Variable (host_var) needed to receive answer, so new Into clause; colon shows DBMS this is a program variable.

C compiler doesn't recognize SQL statements. The "exec sql" phrase is a signal to an explicit SQL precompiler that turns SQL into real C calls.

(Already have precompiler acting first when give gcc command for things like #include, #define; this is a different one).

In Oracle, start with pgm.pc (meaning has embedded SQL statements: exec sql . . .). Then give command (See Appendix B)

```
proc iname=pgm.pc
```

Creates new source file, pgm.c, with exec sql statements turned into pure C: they are now calls to ORACLE monitor functions.

Next do true compilation and linkage (with prompt.c, for example), create executable program pgm.ox. Have makefile. WIll create pgm.o:

```
gcc -g -c try.c
```

And so on. The provided makefile knows all the details. You can do everything in the homework using the makefile included. Just create your pgm.pc and type:

```
make E=pgm   (NO spaces on either side of "="; pgm.pc must exist)
```

5.1.1. A simple program.

Declare section. Need to declare C variables so they are understandable to ORACLE (ORACLE needs to know type of variable). Here is how we would set a variable used in a search condition and select into two others.

```
cust_id  =  "c001";        /* or prompt user for cust id */
exec sql select cname, discnt into :cust_name, :cust_discnt
        from customers where cid = :cust_id;
```

Note use of colon when variable appears in various positions in select statement. Important in Where clause so can tell :cust_id isn't a constant.

At runtime, cust_name and cust_city will be filled in with values "TipTop" and "Duluth". Could now write in program:

```
printf("Customer name is %s, and customer city is %s\n",
        cust_name, cust_city);
```

Note don't use colons here (used in exec sql as hint to dumb precompiler). In order to use C variables in SQL, must put in Declare Section:

```
exec sql begin declare section;
        char cust_id[5] = "c001", cust_name[14];
        float cust_discnt;
exec sql end declare section;
```

The character arrays need to contain enough characters to hold string values of columns PLUS ONE for terminal '\0' character. E.g., cid was declared as column char(4), with values like 'c001' in SQL.

In C when we specify a string such as "c001" (note double quotes), this includes a terminal zero value ('\0') at the end: this serves as a signal to functions such a printf that the string has ended.

Conversion can also occur if declare

```
        int cust_discnt;
```

in Declare section. Then float value 10.00 will become 10, but 10.50 will also become 10 — lose fractional significance.

SQL connect and disconnect. Can use sqlplus Scott to attach,

```
strcpy(username, "Scott");          /* set up username and password    */
strcpy(password, "Tiger");          /* for Oracle login                */
exec sql connect :username identified by :password;
```

This is what is used in ORACLE (Note: can't use literal names, must be in character string. Note too that COUNTED string (in text) is NOT needed!!!!)

The command to disconnect is represented by:

```
exec sql commit release;  /* ORACLE disconnect */
```

Now look at Example 5.1.1, Figure 5.1, pg 207. Task is to prompt repeatedly for customer cid and print out the customer name and city. Halt when user inputs a null line (just hits CR).

```
#include <stdio.h>

exec sql include sqlca;

exec sql whenever sqlerror stop;   --covered later
```

Connect, Loop, prompt, select, commit, print. Commit releases locks. Loop is while prompt( ) > 0, which is length of token in string input by user (white space doesn't count). When user types CR, end loop.

How is ORACLE program 5.1 different from SQL-92 Standard?

In ORACLE can't use literal name in "exec sql connect". must be in a character string. Also must include password.

See prompt() function explained on page 269 and also in Appendix B. Expected to use in homework (available online, handled by makefile).

Put out prompt[ ] string to terminal to insert any number K of tokens (separated one from another by whitespace: '\n', ' ' (SP), '\t'). Format:

```
int prompt(char prompt_str[],int ntokens, ...);
```

Variable-length argument list represented by ellipsis (...) (See K&R). Argument ntokens shows number of tokens to input.

Ellipsis (. . .) represents a variable number of argument pairs: buf1, len1, buf2, len2, . . ., bufN, lenN.

Here, bufK, K = 1 to N, is a character array to contain the Kth token (always char str), and lenK is the maximum allowed length of the Kth token.

All tokens will be read into character arrays by the prompt()

If some of these should be interpreted as int or float numbers, use ANSI stanard C library function sscanf(), with appropriate conversion string:

E.g.: %d for decimal integer and %f for double or float.

Can look at code: uses fgets, sscanf. Could use scanf but multiple tokens on a line (input by confused user) will cause things to get out of synch.

fgets brings keyboard input into array line; sscanf parses K tokens into x. (Any other tokens are lost.) On pg, 216 is prompt2 if need to input two values on one line.

OK, up to now, retrieved only a single row into a variable:

```
exec sql select cname, discnt into :cust_name, :cust_discnt
        from customers where cid = :cust_id;
```

Selecting multiple rows with a Cursor. (pg. 270, program in **Fig 5.2, pg. 273**)

The Select statement used above can only retrieve a SINGLE value. To perform a select of multiple rows, need to create a CURSOR.

It is an Error to use array to retrieve all rows.

ONE-ROW-AT-A-TIME PRINCIPAL: Need to retrieve one row at a time: cursor keeps track of where we are in the selected rows to retrieve.

COMMON BEGINNER'S MISTAKE, to try to get all the data FIXED in an array. Find the max of an input sequence of integers (don't know how many).

Here: Declare a cursor, open the cursor, fetch rows from the cursor, close the cursor. E.g., declare a cursor to retrieve aids of agents who place an order for a given cid, and the total dollars of orders for each such agent.

```
exec sql declare agent_dollars cursor for
        select aid, sum(dollars) from orders
        where cid = :cust_id group by aid;
```

But this is only a declaration. First Runtime statement is Open Cursor:

```
exec sql open agent_dollars;
```

VERY IMPORTANT: when open cursor, variables used are evaluated at that moment and rows to be retrieved are determined.

If change variables right after Open Cursor statement is executed, this will not affect rows active in cursor.

Now fetch rows one after another:

```
exec sql fetch agent_dollars into :agent_id, :dollar_sum;
```

The Into clause now associated with fetch rather than open, to give maximum flexibility. (Might want to fetch into different variables under certain conditions.)

A cursor can be thought of as always pointing to a CURRENT ROW. When just opened, points to position just before first row. When fetch, increment row and fetch new row values.

If no new row, get runtime warning and no returned value (as with getchar( ) in C, value of EOF).

Note "point before row to be retrieved" better behavior than "read values on row, then increment".

This way, if runtime warning (advanced past end of rows in cursor) then there are no new values to process, can leave loop immediately.

How do we notice runtime warning? SQL Communication Area, SQLCA (now deprecated by SQL-92, but stick with). Recall in pgm. 5.1.1 had:

```
exec sql include sqlca;
```

This creates a memory struct in the C program that is filled in with new values as a result of every exec sql runtime call; tells status of call.

25.

See **Fig 5.2, pg. 273**.

Review: Declare a cursor, open the cursor, fetch rows from the cursor, close the cursor.

E.g., declare a cursor to retrieve aids of agents who place an order for a given cid, and the total dollars of orders for each such agent.

```
exec sql declare agent_dollars cursor for
        select aid, sum(dollars) from orders
        where cid = :cust_id group by aid;
```

But this is only a declaration. First Runtime statement is Open Cursor:

```
exec sql open agent_dollars;
```

When open cursor, variables used are evaluated at that moment and rows to be retrieved are determined.

Now fetch rows one after another:

```
exec sql fetch agent_dollars into :agent_id, :dollar_sum;
```

A cursor can be thought of as always pointing the CURRENT ROW (if any), the row last fetched. When fetch and get NO DATA RETURNED, means end loop right away.

NOTE, in general, we cannot pass a cursor as an argument. But can make it external to any function in a file, so all functions can get at it.

How do we tell that the most recent fetch did not retrieve any data? Remember exec sql include sqlca? Communication area

Decprecated, but sqlca.sqlcode is still an important variable.

Still works to test the value of sqlca.sqlcode after delete, update, insert, etc. to figure out what errors occurred. **MAY** be replaced by SQLSTATE.

ORACLE has implemented SQLSTATE, but still uses sqlca.sqlcode because gives more error codes. Can use SQLSTATE in ANSI version of Oracle.

For portability best to use different method for most things: WHENEVER statement, below.

NOTE that sqlca must be declared where all functions that need to can access it: usually external to any function. Go through logic.

Starting to talk about 5.2 Error handling.

EXEC SQL INCLUDE SQLCA creates a C struct that's rather complicated, and denegrated, but we deal mainly with a single component, sqlca.sqlcode.

This tells whether

sqlca.sqlcode == 0, successful call
< 0, error, e.g., from connect, database does not exist -16
(listed as if positive)
> 0, warning, e.g., no rows retrieved from fetch
(saw this already, tell when cursor exhausted)

Error is often a conceptual error in the program code, so important to print out error message in programs you're debugging. Come to this.

See Example 5.2.4 in text.

In general, there are a number of conditions that a Whenever statement can test for and actions it can take. General form of Whenever statement:

exec sql whenever <condition> <action>

Conditions.

o SQLERROR	Tests if sqlca.sqlcode < 0
o NOT FOUND	Tests if no data affected by Fetch, Select, Update, etc.
o SQLWARNING	Tests if sqlca.sqlcode > 0 (different than not found)

Actions

o CONTINUE	Do nothing, default action
o GOTO label	Go to labeled statement in program
o STOP	In ORACLE, Prints out error msg and aborts program
o DO func_call	(IN ORACLE ONLY)Call named function; very useful

Look at exercise due: 5.3. Know how to test for cust_id value that doesn't exist.

The reason the call action is so handy is that WHENEVER statements don't respect function divisions. If we write a single whenever at the beginning of a program with a lot of functions:

```
whenever sqlerror goto handle_error

main( )  {
. . . }
func1( )  {
. . .}
func2( )  {
```

```
    . . .
```

Using the above trick, the effect of the WHENEVER statement will span the scope of the different functions. However, DO is not portable, and the portable way requires a GOTO target in each function. This GOTO-target code of course can have a function call.

The WHENEVER statement is handled by the precompiler, which puts in tests after all runtime calls (exec sql select… or whatever) and doesn't care about what function it's in (it doesn't even KNOW). For example:

```
    whenever sqlerror goto handle_error;  /* below this, will do this GOTO on error */
```

Continues doing this until its actions are overridden by a WHENEVER with a different action for the same condition:

```
    whenever sqlerror continue;      /* below this, will do nothing */
```

But note in example above, there must be a label handle_error in all these functions. A call to a handle_error function would save a lot of code.

Question. What happens here, where only two whenever statements are listed explicitly?

```
    main( )
    {
        exec sql whenever sqlerror stop;
        . . .
        goto label1;
        . . .
        exec sql whenever sqlerror continue;
label1: exec sql update agents set percent = percent+1;
```

If we arrive at label1 by means of the goto statement, which whenever statement will be in force for condition sqlerror? E.g., what if mispell columname percent? (Answer is: continue.)

What if did:

```
    main( )
    {
        exec sql whenever sqlerror goto handle_error;
        exec sql create table customers
            (cid char[4] not null, cname varchar(13), . . . )
        . . .
handle_error:
        exec sql drop customers;
        exec sql disconnect;
        exit(1);
```

What's the problem? Possible infinite loop because goto action still in force when drop table (may be error). Need to put in:

```
exec sql whenever sqlerror continue;
```

Right at beginning of handle_error. Overrides goto action (won't do anything). (Default effect if don't say anything about whenever.)

Also need to take default non-action to perform specific error checking. Maybe have alternative action in case insufficient disk space when try to create table:

```
exec sql whenever sqlerror goto handle_error;
exec sql create table customers ( . . .)
if (sqlca.sqlcode == -8112)              /* if insufficient disk space */
    <handle this problem>
```

But this won't work, because esqlc places test

```
if(sqlca.sqlcode < 0)
    <call stopfn>
```

IMMEDIATELY after create table, so it gets there first and test for sqlca.sqlcode = -8112 never gets entered (< 0 gets there first).

So have to do whenever sqlerror continue before testing for specific error.

Book tells you how to get a specific error msg. Good use in case don't want to just print out error msg. (Discuss application environment for naive user: never leave application environment, write errors to log file).

Recall that if discnt in customers has a null value for a row, do not want to retrieve this row with:

```
select * from customers where discnt <= 10 or discnt >= 10;
```

But now say we performed this test in program logic:

```
exec sql select discnt, <other cols> into :cdiscnt, <other vars>
    where cid = :custid;
```

Now we decide to print out these col values in situation:

```
if(cdiscnt <= 10 || cdiscnt > 10) printf (. . .).
```

Now look! Host variable cdiscnt is a float var, AND ALL MEANINGFUL BIT COMBINATIONS ARE TAKEN UP WITH REAL VALUES. There's no way that cdisnt will fail the if test above.

Need to add some way to test if the variable cdisnt has been filled in with a null value (then include AND NOT NULL in if test). Can do this with indicator variable, retrieve along with discnt:

```
short int cdiscnt_ind;     /* form of declaration */
```

Then change select:

```
exec sql select discnt, <other cols> into :cdiscnt:cdiscnt_ind,  /* ORACLE syntax */
      <other vars> where cid = :custid;
```

Now if cdiscnt_ind == -1, value of cdiscnt is really null. Add test that cdiscnt is not null by:

```
if((cdiscnt <= 10 || cdiscnt > 10) && cdiscnt_ind <> -1)
      printf (. . .).
```

Note can also STORE a null value by setting a null indicator cdiscnt_ind and writing

```
exec sql update customers set discnt = :cdiscnt:cdiscnt_ind where . . .
```

One other common use for indicator variable for char column being read in to array, is to notifiy truncation by value > 0, usually length of column string.

26.
Indicator Variables

Section 5.3. Complete descriptions of SQL Statements.

Select. See Section 5.3, pg 281, Figure 5.3. CAN ONLY RETRIEVE ZERO OR ONE ROW.

Look at general form: No UNION or ORDER BY or GROUP BY in Basic SQL form . like Subquery form on pg. 144, add into clause. (How use GROUP BY with only one row retrieved? Still part of full standard)

See Figure 5.4, corresponding (analogouus) C types for column type. Conversion done if integer col type, float C var.

Note that in general a variable can be used to build up an expression in a search_condition:

```
select cname into :cust_name where cid = :custid and city = :cname;
```

But we **cannot use character string to contain part of statement requiring parsing**:

```
char cond[ ] = "where cid = 'c003' and city = 'Detroit'";
exec sql select cname into :custname where :cond;
```

NOT LEGAL. The ability to do this is called "Dynamic SQL", covered later.

Declare Cursor statement, pg. 283. Used when retrieving more than one row in select, so basically an extension of interactive Select.

Has GROUP BY, HAVING, UNION, ORDER BY. Adds clause: for update of, need later.

Cursor can only move FORWARD through a set of rows. Can close and reopen cursor to go through a second time.

Two forms of delete, pg. 283, Searched Delete and Positioned Delete:

```
exec sql delete from tablename [corr_name]
        [where search_condition | where current of cursor_name];
```

After Searched Delete, used to be would examine sqlca.sqlerrd[2] to determine number of rows affected. New way exists (Don't know yet).

After Positioned delete, cursor will point to empty slot in cursor row sequence, like right after open or when have run through cursor. Ready to advance to next row on fetch. Works just right to delete everything after look at it in loop:

```
LOOP
    exec sql fetch delcust into :cust_id;
    <work>
    exec sql delete from customers where current of delcust;
END LOOP
```

If cursor moved forward to next row (say) after delete, would only be deleting every OTHER row.

Could create cursor to select all customers in Detroit to delete them, or could select all customers and then check if city is Detroit. First is more efficient. PRINCIPAL: do all search_conditions before retrieving.

But probably a Searched Delete is most efficient of all for Detroit customers -- save HAVING to switch threads, do all in query optimizer.

Second paragraph on p 284: In order for positioned delete to work, the cursor must (1) be already opened and pointing to a real row, (2) be an updatable curson (not READ ONLY), (3) FROM clauses of delete must refer to same table as FROM clause of cursor select.

Two forms of Update statement, pg. 285: Searched Update and Positioned Update. Same ideas as with Delete (sqlca.sqlerrd[ ] will contain count of rows affected).

First printing: the most general Update syntax is on pg. 773, Figures C.33 and C.34.

Same Update form as earlier, pg. 149: each column value can be determined by a scalar expression, an explicit null, or a value given by a scalar subquery.

Insert, pg. 286. No positioned insert, because position determined in other ways.

Open, Fetch, Close, pg. 286. Open evaluates expressions, sets up row list to retrieve from, unchanging even if expression values change.

Create table, drop table, connect, disconnect. Not create database, because that is not an SQL command. SEE APPENDIX C for more info.

Section 5.4 Idea of concurrency, bank tellers have simultaneous access to data. Problem: Example 5.4.1. Inconsistent view of data. Table A of accounts, A.balance where A.aid = A2 will be called A2. balance. One process wants to move money from one account to another. Say $400.00.

S1: A1.balance == $900.00 A2.balance == $100.00

S2: A1.balance == $500.00 A2.balance == $100.00

S3: A1.balance == $500.00 A2.balance == $500.00

If another process wants to do a credit check, adds A1.balance and A2.balance, shouldn't see S2 or may fail credit check. See Example 5.4.1, pg. 245.

Create an idea called transactions. Programming with transactions makes guarantees to programmer, one of which is Isolation. Means every transaction acts as if all data accesses it makes come in serial order with no intervening accesses from others. E.g.:

T1: R1(A1) W1(A1) R1(A2) W1(A2) (explain notation)
T2: R2(A1) R2(A2)

Turn Update in to R then W. Turn Select into R only. Problem of S2 comes because of schedule:

R1(A1) W1(A1) R2(A1) R2(A2) R1(A2) W1(A2)

But this is not allowed to happen with transactions, must act like:

T1 then T2 (T2 sees state S3) or T2 then T1 (T2 sees S1).

Don't care which. Called Serializability.

27.

Review idea last time of concurrency.

R1(A1) W1(A1) R2(A1) R2(A2) R1(A2) W1(A2)

But this is not allowed to happen with transactions, must act like:

T1 then T2 (T2 sees state S3) or T2 then T1 (T2 sees S1).

Don't care which. Called Serializability.

(Why do we want to have real concurrency, ? Why not true serial execution? Idea of keeping CPU busy. More next term.)

Two new SQL statements to cause transactions to occur.

exec sql commit work; Successful commit, rows updated, become concurrently visible.

exec sql rollback; Unsuccessful abort, row value updates rolled back and become concurrently visible.

Transactions START when first access is made to table (select, update, etc.) after connect or prior commit or abort. Ends with next commit work or rollback statement or system abort for other reason.

Recall idea that we hold locks, user has SQL statements to say: logical task is complete: you can make updates permanent and drop locks now.

Typically, applications loop around in logic, with user interaction, transaction extends from one interaction to the next. But might have multiple interactions between interactions. DON't typically hold transaction during user interaction (livelock).

User interaction may set off a lot of account balancing: each Tx subtracts money from n-1 acct balances, adds money to an end one. Say should abort if one acct is under balance.

Clearly system can't guess when one set of accts has been balanced off and start immediately on another set. User must tell WHEN to release locks.

Value of rollback is so don't have to write code to reverse all prior changes.

- **Atomicity.** The set of updates performed by a Tx are atomic, that is, indivisible. If something bad happens (terminal goes down) will abort.

- **Consistency.** E.g., money is neither created nor destroyed by logic, then won't happen.

- **Isolation.** As if serial set of Txs.

- **Durability.** Even if lose power & memory (crash), lose place in what doing, will RECOVER from the crash and guarantee atomicity.

LOCKING. Simple version here. (a) When a Tx accesses a row R, first must get lock. (b) Locks are held until Tx ends (Commit). (3 & 4) Locks are exclusive, so second locker will wait if it can.

Example 5.4.2. Recall problem of inconsistent view.

R1(A1) W1(A1) R2(A1) R2(A2) C2 R1(A2) W1(A2) C1

where add Ci for commit. Now add locks.

L1(A1) R1(A1) W1(A1) L2(A1) (conflict, must wait) L1(A2) R1(A2) W1(A2) C1 (Releases T1 locks including L1(A); Now lock request L2(A1) can succeed) R2(A1) R2(A2) C2

Result is T2 sees State S3. No matter what arrangement of attempts, T2 will see either S1 or S3. But New idea: DEADLOCK. What if rearrange:

R1(A1) W1(A1) R2(A2) R2(A1) C2 R1(A2) W1(A2) C1

Now locking proceeds as follows:

L1(A1) R1(A1) W1(A1) L2(A2) R2(A2) L2(A1) (prior L1(A1), so T2 must wait) L1(A2) (prior L2(A2) so T1 must wait: DEADLOCK! illustrate)

(must choose one Tx to abort, say T2, L1(A2) is successful) R1(A2) W1(A2) C2 (Now T2 audit attempt retrys as T3) L3(A2) R3(A2) L3(A1) R3(A1) C3

And state seen by T2 (retried as T3) is S3. If aborted other one, state seen by T2 would be S1. Note that in general, locks held are much more sophisticated than these exclusive locks.

This means that programmer must watch out for deadlock error return, attempt a retry. See Example 5.4.4, pg 296.

Note can't run into a deadlock abort until already HOLD a lock and try to get another one. Of course this could happen in a single instruction in (e.g.) Searched Update with multiple rows. But if only single rows accessed, no

No user interaction during transaction or Livelock. For example, if retrieve data of quantity available for an order and then confer with customer. See Example 5.4.5, program pg. 300But problem as a result.

To avoid this, typically commit transaction after retrieving data, then confer with customer, the start new transaction for update based on customer wishes. But new problem. See Example 5.4.6, program pg 301

14. Exam 1

15

Chapter 6, Database Design. We now tackle the following:

Q: how do we analyze an enterprise and list the data items for a database, then decide how to place these data items columns in relational tables.

Up to now, we've had this done for us, but now we have to decide. For example should we just put them all in the same table?

Answer is no, because (as we'll see) the data doesn't behave well. Consider the CAP database. Could we just have a single table, CAPORDERS,

CAPORDERS := C X A X P X O where C.cid = O.cid and A.aid = O.aid
and P.pid = O.pid

See any problems with that? One problem is redundency — Every row of CAPORDERS must duplicate all product information, and this happens lots of times. Ditto for customers and agents.

If you look at the number of columns and assume lots of orders for each C, A, P, see BIG waste of disk space compared to separate tables. Bad because cname, city, pname, etc., are long compared to ORDERS columns.

Further, note there is in PRODUCTS a quantity column, meaning quantity on hand. Assume needs updating every time a new order is placed.

But then in CAPORDERS with popular product, number or rows grows with number of orders (thousands!). ALL of them have to be ordered each time. BIG inefficiency compared to separate table for products.

And what do happens when we place a new order -- do we ask the application programmer to get the user (key entry clerk) to enter all the information about customer, agent, product in CAPORDERS every time?

Doesn't make sense -- waste of time -- clerk might get it wrong.

**** So the program looks up data (say for product) and enters it in CAPORDERS? Where does it look it up? Find another row with the same product and copy data?

Note that when have separate table for products, just enter pid (which can be looked up in products table if start with product name/description).

What if some product is out of stock and takes a long time to reorder so all the orders for that product have been filled from CAPORDERS (we don't keep these orders around forever).

But now where do we find product information? Have we forgotten about this product because we haven't had orders for awhile? Note solved if we have distinct products table.

There are also a number of "business rules" that it would be nice to have the database system guarantee -- simple things, but quite subject to error in data entry.

For example, we say that the pid column is a unique identifier for a product. That's a business rule. No two products have the same pid.

But review now the idea of CAPORDERS -- everything joined. For each orders row, have all info on customer, agent, and product. Put up picture.

It's a little hard to figure out what that means in the CAPORDERS table where pid will be duplicated on every row where a given product is ordered, but let's try.

In the database design where PRODUCTS is a separate table, we say that that pid is a "key" for the table, i.e. it is unique.

If we think in terms of rows in PRODUCTS being duplicated in CAPORDERS (for every order that deals with the same product), then how would we characterize pid?

Business rule: every unique pid value is associated with a unique quantity (it would be a bad thing if in CAPORDERS two rows with the same pid had different quantity values). Is the reverse true?

We write this: pid -> quantity, and say pid **functionally determines** quantity, or quantity **is functionally determined by** pid.

Is it also true that pid -> pname? pid -> city? pid -> price? Rules like this are called FDs. These are what we refer to as Interrelationships between data items in the text.

Writing down a set of rules like this is the beginning of a process called "normalization", a relatively mechanical process which leads to a well-behaved breakdown of data items into tables.

Once these tables have been created, all FDs are maintained by constraints defined with Create Table statement, candidate key (unique value constraint) and primary key constraint (see Figure 6.1, pg. 331).

A different design approach, called Entity-Relationship modelling, is more intuitive, less mechanical, but basically leads to the same end design.

Intuitively, we think that there is a real-world object for the set of products with identical properties we want to track.

We create a table where we have one row for each type. The cid column is the "identifier" of the row.

This intuitive concept is the beginning of what we call Entity-Relationship modelling — products are a single Entity (or Entity set).

Note that products are somewhat different from customers and agents. There is one row in AGENTS for each distinct agent and ditto customers.

But distinct products are not thought worth tracking — we deal with a row for each Category of products.

Section 6.1. Introduction to E-R

Def. 6.1.1 Entity. An entity is a collection of distinguishable real-world objects with common properties.

E.g., college registration database: Students, Instructors, _rooms, Courses, Course_sections (different offerings of a single course, generally at different times by different instructors), _periods.

Note that we Capitalize entity names.

_rooms is a good example of an entity. Distinguishable (by location). Have common properties, such as seating capacity, which will turn into a column of a table (different values of these common properties).

_periods is an entity? An interval of time is a real-world object? A bit weird, but basically think of assigning a student to a room during a period, so time interval is treated just like room.

Normally, an entity such as _rooms or Customers is mapped to a relational table, and each row is an entity occurrence, or entity instance, representing a particular object.

In the case of Products, an entity instance is a category of objects sold by our wholesale company.

It is unusual in the field to use a plural for entity and table, Customers instead of Customer. We do this to emphasize that the entity represents a Set of objects.

Def. 6.1.2 Attribute. An attribute is a data item that describes a property of an entity or a relationship (to follow).

Note that we have special terminology for special kinds of attributes, see pg. 306.

An identifier is an attribute or set of attributes that uniquely identifies an entity instance. Like cid for Customers. Can have primary identifier.

A descriptor is a non-key attribute, descriptive. E.g., city a customer is in, color of a car, seating capacity of a room, qty of an order.

A multi-valued attribute is one which can take on several values simultaneously for an entity instance. E.g., keyword for Journal_articles or hobby for Employees. Disallowed by relational rule 1, but in ORDBMS.

In E-R, we can have a composite attribute (like a nested struct inside the row), e.g., cname can have fname, lname, midinit as three parts.

Note that the relational model, rule 1, which disallows multi-valued columns, also disallows composite columns. OK in ORDBMS, but must map composite attribute to multiple columns in relational model.

Note that term attribute is also used in relations, but ideas correspond in the mapping of entities and relations into relational tables.

Note that while entity instances within an entity are said to be distinct, but this is only a mathematical idea until we have identier attributes.

We write E is an entity, with entity instances $\{e_1, e_2, \ldots, e_n\}$. Need an identifier attribute defined, unique for each occurrence e_i.

Put up diagram from pg. 333, Figure 6.2.

Transforming Entities and Attributes to Relations is pretty obvious, as mentioned previously.

Transformation Rule 1. An entity is mapped to a single table. The single-valued attributes of the Entity are mapped to columns (composite attributes are mapped to multiple simple columns). Entity occurrences become rows of the table.

Transformation Rule 2. A multi-valued attribute must be mapped to its own table. See bottom of pg. 334. (No longer true in ORDBMS.)

Not too much power so far, but relationship adds real modeling power.

Def. 6.1.3. Relationship (pg. 335). Given an ordered list of m entities, E_1, E_2, . . . , E_m, (where the same entity may occur more than once in the list), a relationship R defines a rule of correspondence between the instances of these entities. Specifically, R represents a set of m-tuples, a subset of the Cartesian product of entity instances.

Instructors teaches Course_sections
Employees works_on Projects (attribute, percent (of time))
Employees manages Employees (ring, or recursive relationship)

See top of pg 336 for diagram. Note "roles" of labeled connecting lines in case of recursive relationship.

Example 6.1.3. The orders table in the CAP database does NOT represent a relationship. Reason is that orders rows do not correspond to a subset of the entities involved. Multiple orders can exist with same cid, aid, pid.

The orders table is really an entity, with identifier ordno.

Of course, there is a relationship between Orders and each of (pg 391) Customers requests Orders, Agents places Orders, Orders ships Products.

Note labels try to describe left to right, top down order. Could change to Orders placed_by Agents.

Could have a ternary relationship, see Example 6.1.4, defined in terms of orders table.

```
create table yearlies(cid char(4), aid char(3), pid char(3),
        totqty integer, totqty float);
insert into yearlies select cid, aid, pid, sum(qty), sum(dollars)
        from orders group by cid, aid, pid;
```

Transformation rules are more difficult for relationships. The yearlies relationship is transformed into a yearlies table. However, no separate table for manages relationship from employees to employees.

Instead, put mgrid column in employees table (since every employee has at most one manager, this works. See top of page 338, Fig. 6.4.

The idea of common properties is that all we need to know can be listed in labeled columns. Standard business form.

16.

Remember the E-R diagram on pg 336 with the relationship that says Employees works_on Projects, where works_on is a relationship. works_on has the connected attribute percent. Draw it.

Note: percent, associated with relationship, i.e., a value with each relationship instance.

The relationship instance represents a specific pairing of an Employees instance with a Projects instance; percent represents the percent of time an employee instance works on that project.

Clearly have the business rule that an employee can work on more than one project. Also have rule that more than one employee can work on each project. This binary relationship is said to be Many-to-Many.

Now it's going to turn out that for relationships that are Many-to-Many, a table is needed in the relational model to represent the relationship. But this is NOT always true if the relationship is not Many-to-Many.

Consider the relationship (also on pg. 336): Instructors teaches Course_sections.

Say we have the rule that an Instructor can teach more than one course section (usually does, unfortunately for me), but we make the rule that only one instructor is associated with every course section.

This means that if there are two instructors teaching a , one of the two is actually responsible for the course, and the team approach is unofficial.

Now we know from transformation rule 1 that both entities Instructors and Course_sections map to relational tables.
(Draw this, with some attributes: iid for instructors, csid for course_sections -- assume no multi-valued attributes so these are only two tables).

Now the question is, do we need another table for the relationship teaches.

Answer: No. We can put a column in the course_sections table that uniquely identifies the instructor teaching each row (instance). This is done with an iid column.

Note that the iid column in the course_sections table is NOT an attribute of the Course_sections entity. The iid column instead represents the teaches relationship.

In relational terminology, this column is known as a foreign key in the course_sections table (not a key for course_sections but one for the foreign table instructors).

OK, what's the difference? Why did one relationship, Employees works_on Projects require a table for works_on, and the other, Instructors teaches Course_sections, require no new table?

Because one relationship is Many-to-Many and the other is Many-to-One! The Many-to-One relationship can be done with a foreign key because we only need to identify (at most) ONE connecting instance on one side.

Note these ideas are all BUSINESS RULES. They are imposed by the DBA for all time by the definition of the tables. We shall see how shortly.

Look at Figure 6.6. Entities E and F, relationship R. Lines between dots. Dots are entity instances. Lines are relationship instances.

If all dots in the entity E have AT MOST one line coming out, we say:
max-card(E, R) = 1.

If more than one line out is possible, we say max-card(E, R) = N.

If all dots in the entity E have AT LEAST one line coming out, we say:
min-card(E, R) = 1.

If some dots might not have a line coming out, we say min-card(E, R) = 0.

We combine these, by saying card(E, R) = (x, y) if min-card(E, R) = x and max-card(E, R) = y. (x is either 0 or 1 and y is either 1 or N.)

Go over Figure 6.7 on pg 341. Note that for recursive relationship manages, include role: card(employees(reports_to), manages) = (0, 1)

Note that saying min-card(E, R) = 0 is really NOT MAKING A RESTRICTION. There might be no lines leaving a dot, there might be one, or more (min-card NEVER says anything about maximum number).

Saying max-card(E, R) = N is also not making a restriction. There don't have to be a lot of lines leaving — N might be zero — just saying we are not restricting the maximum to one.

The most restrictive thing we can say is that card(E, R) = (1, 1). Then comes (1, N) and (0, 1). Then (0, N), which means no restrictions.

We can only note from a specific example (content of a given moment) of a relationship R with regard to an entity E if a restriction is BROKEN. How would we notice card(E, R) = (0, N)? (Draw it.)

If we had a situation that seemed to say card(E, R) = (1, 1), can't know this will continue to hold in future -- must know designer's intention.

Another term used, Def 6.2.2, if max-card(E, R) = 1 then E is said to have single-valued participation in R. If N, then multi-valued participation.

Def. 6.2.3. If min-card(E, R) = 1, E is said to have mandatory participation in R, if 0, then optional participation.

One-to-One (1-1) relationship if both entities are single-valued in the relationship (max-card concept only). Many-to-Many (N-N) if both entities are multi-valued. Many-to-One (N-1) if one entity is multi-valued and one is single valued. Draw.

Do not usually say which side is Many and which One by saying relationship is Many-to-One, N-1. Might actually be One-to-Many, but don't use that term.

HOWEVER -- the MANY side in a Many-to-One relationship is the one with single-valued participation. (there are Many of these entities possibly connected to One of the entities on the other side.)

OK, now a bunch of Transformation rules and examples, starting on pg. 310.

N-N relationship always transforms to a table of its own. Many Instructors teach Many Course_sections. Direct flights relate cities in Europe to cities in the US. Can't represent simply: rich complex structure.

N-1 relationships, can represent with foreign key in entity with single valued participation (the Many side).

1-1. Is it optional on one side or is it mandatory on both sides?

Optional on one side. Postmen carry Mailbags. Every postman carries one and only one mailbag, and every mailbag is carried by at most one postman, but there might be some spares in stock that are carried by none.

Represent as two tables, foreign key column in one with mandatory participation: column defined to be NOT NULL. Can faithfully represent mandatory participation. Clearly representing single-valued participation.

(Idea of faithful representation: programmer can't break the rule even if writes program with bug. Note can NOT faithfully represent single-value participation for both mail-bags AND postmen.)

1-1 and Mandatory on both sides: never can break apart. It's appropriate to think of this as two entities in a single table. E.g. couples on a dance floor -- no-one EVER is considered to be without a partner. Avoids foreign keys.

(Really? But might change partners and some info might be specific to individuals of partners - his height, age, weight - her height, age, weight.

This logical design is more concerned with not being able to end up with a mistake than making a transformation easy.)

Section 6.3, Additional E-R concepts.

Attributes can use idea of cardinality as well. See Figure 6.10.

(0, y) means don't have to say not null, (1, y) means do.

(x, 1) most common, single valued attribute, (x, N) multi-valued.

Weak entities. An entity that can't exist unless another entity exists. Depends for its existence and identification on another entity.

E.g., Line-items on an Order. Customer places an order, orders several products at once. Order has multiple line items. Line_items is a weak entity dependent on the entity Orders. See Figure 6.11.

There is an N-1 relationship, has_item, that relates one Orders instance to many Line_items instances.

Therefore, by transformation rules, Line_items sent to table, Orders sent to table, foreign key in Many side, line_items table.

Note that the identifier for a Line_items, lineno, is enough in E-R model to identify the weak entity, since can go back through has_item relationship to find what order it belongs to.

In relational model, must be identified by column value. Note ordno is not an attribute of line_items but a foreign key, and lineno and ordno must be used together as a key for line_items!!!

OK, think. What's the difference between the two situations:
Orders has_item Line_Items and Employees with multi-value attribute hobbies? Map the same way into relational tables!

Possibly very little difference. One man's attribute is another man's entity. If I cared about tracking all hobbies in the company so I could provide well thought out relaxation rooms, might say hobbies are entities.

In case of line number, there are usually several attributes involved, lineno, product ordered, quantity of product, cost, so seems reasonable to say Line_items is an entity, albeit a weak one.

17.

Skip Generalization Hierarchies for now to go on to Case Study.

Simple airline reservation database, data items we have to track: passangers, flights, departure gates, seat assignments.

Could get much more complex: a flight brings together a flight crew, a ground crew, an airplane, a set of passengers, a gate. The gates have to be cleaned and serviced, etc.

For simplicity, say represent situation with a few simple entities:

Entity Flights, primary identifier flightno, descriptive attribute depart_time (e.g., Nov 19, 9:32 PM). (Note the airplane is assumed given.)

Entity Passengers, primary identifier ticketno.

Entity Seats, identified by seatno, valid only for a specific flight. Hint, Seats is a weak entity depending on Flights. We say the Many-to-One relationship here is: Flights has_seat Seats.

Entity Gates, with primary identifier gateno.

Passengers must be assigned to Seats, and this is by a relationship seat_assign. Draw below without relating Gates, Flights, Passengers.

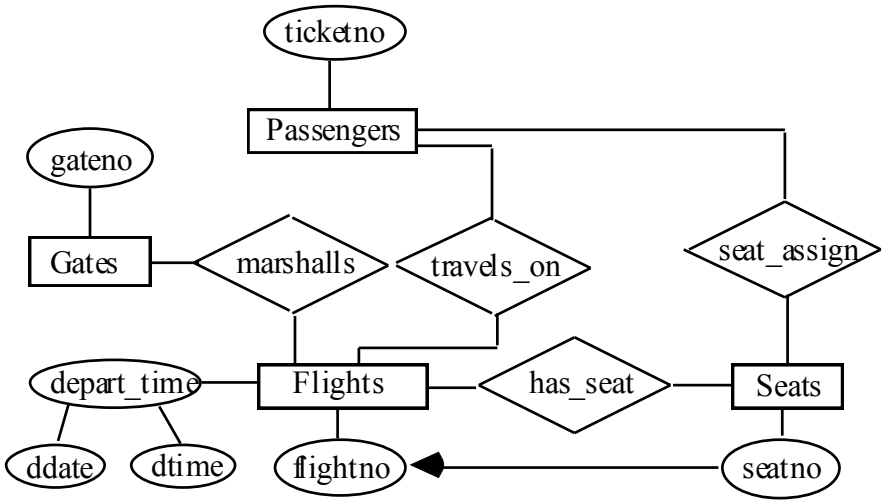


Fig. 6.13, pg. 351 (but leave out marshalls and travels_on), and note Gates off by itself. But clearly passengers go to a gate to meet a flight.

Is this a ternary relationship, then, relating these three? We say it is not, because it is possible to set up two N-1 binary relationships that more faithfully represent the situation.

I wouldn't want to assign two gates to one flight, for example, or two flights to one passenger. Therefore:

See Figure 6.13, pg. 351. marshalls and travels on. Ternery would be N-N-N.

Now work out cardinalities of relationships.

A gate might not be used or it might be used for multiple flights, so have (0, N) participation in marshalls. A flight must have a gate and can have only one gate, so (1, 1).

A passenger must have a flight and only one, so (1, 1) participation in travels_on. A flight must have a passenger (or it will certainly be cancelled) and may (probably will) have many, so (1, N).

Others are clear. Each passenger must have one and only one seat, a seat may or may not be used. Each seat is on some flight, but each flight has multiple seats.

Now transform into relational tables. Map entities, see bottom of pg. 321. Draw on board.

Since seats is weak entity (single-valued participation in has_seat), add flightno to seats table. Now that relationship is taken care of.

Passengers has single-valued participation in two relationships, so add seatno and flightno to passengers (seatno and flightno are needed for seat_assignment, and flightno is needed for travels_on, so one counted twice). Now those relationships are taken care of.

Only one left is Gates marshalls Flights, and put gateno as foreign key in flights table. Done.

See how three (1, 1) participations can be faithfully represented with not-null for seatno and flightno in passengers, not null for flightno in seats, not null for gateno in flights.

Section 6.5. Preliminaries for Normalization.

Idea in normalization, start with data item names (to be columns in some tables) together with a list of rules of relatedness. Then start with all data items in one table (universal table).

Rules of relatedness and a desire to avoid certain types of bad behavior (anomalies) causes us to factor this big table into smaller tables, achieving more and more restrictive forms (Normal Forms). 1NF, 2NF, 3NF, BCNF.

Will not cover 4NF, 5NF (frequently not considered in commercial use).

The idea is that from the factored tables can always get back all the original data by joins; this is called a "lossless decomposition".

A second desire we have is that the database system will be able to check the rules of relatedness as simply as possible (generally by simply enforcing uniqueness of a column in a table).

Point of Normalization is that reach the same design as with E-R, but it is more cut and dried, uses intuition less (given the set of rules — a big given!). Both E-R and Normalization have their points.

To start. 1NF means no repeating fields in tables; no relational products allow such repeating rules (Montage, now Illustra, does, showever).

OK. Running Example: Employee Information. See pg. 354.

Explain each: emp_id, emp_name, emp_phone, dept_name, dept_phone, dept_mgrname skill_id, skill_name, skill_date, skill_lvl

Design from an E-R standpoint is easy. Entities Emps, Depts, Skills, and relationship between Emps and Skills, has_skill, N-N so own table.

But let's take normalization approach. Start with Universal table; see Fig 6.16, emp_info, on pg 355.

Bad design. Turns out key is emp_id skill_id. How do we know that? By rules of relatedness. What's the problem with that? See following.

Anomalies. There is replication of employee data on different rows of emp_info, and this seems unnatural. But why is that bad?

Consider what would happen if skills had to be renewed with tests (giving skill_level) and some employee was dilatory, lost last skill by failing to take a test. Would forget about all employee details when deleted last skill. Called DELETE ANOMALY.

The other face of this problem, cannot add a new employee, say a trainee, until a skill for that employee exists. INSERT ANOMALY.

Also have UPDATE ANOMALY, if dept_name changes for an employee, might have to change a lot of rows. No problem if set-oriented update

```
update emp_info set dept_name = :newdept where emp_id = :eid;
```

except that it might be inefficient to update a lot of rows. Recall in caporders table when quantity for products changed.

[But there may be a serious problem if we are in the middle of a cursor fetch and notice that the dept_name value should change. We don't want to change some but not all of the dept_name values for this employee.]

The fact that we will have to update one conceptual fact in many spots is known as the UPDATE ANOMALY.

We jump ahead to a solution: create emp_info database with two tables, emps and skills. (Factor tables). Figure 6.17, pg. 358.

emps table: emp_id, emp_name, emp_phone, dept_name, dept_phone, dept_mgrname

skills table: skill_id, skill_name, skill_date, skill_lvl

emps table has key emp_id (can't lose info about employee if no skill)

skills table has key emp_id skill_id (skill_lvl determined by both).

Lossless join. Get back in join of skills and emps exactly what we had in emp_info. Not done though.

How can we prove all this? Need a LOT of machinery, very mathematical.

18.

Section 6.6. Functional Dependencies.

Remember what it meant to say that the identifier pid is a key (identifier) for product information in the CAPORDERS table?

The pid value is repeated many times, so not unique as a key. But still, pid uniquely determines quantity and pname and pcity &c. say pid -> quantity.

Def 6.6.1. Given a table T with at least two attributes A and B, we say that A -> B (A functionally determines B, or B is functionally dependent on A) iff it is the intent of the designer that for any set of rows that might exist in the table, two rows in T cannot agree on A and disagree on B.

More formally, given two rows r_1 and r_2 in T, if $r_1(A) = r_2(A)$ then $r_1(B) = r_2(B)$.

Idea of function in calculus — graphs. See it's the same, domain -> range.

Ex. 6.6.1. in emp_info table: emp_id -> emp_name, emp_id -> emp_phone, emp_id -> dept_name.

Think what this means in E-R terms. Consider your intution - if two rows with same emp_id and different emp_phone, assume data corrupted, but if same emp_phone and different emp_id say "Oh, so employees can share a phone." Then we say, emp_phone -/-> emp_id

Def. 6.6.2. OK, have when one attribute A functionally determines another B. Now sets of attributes: $X = A_1\ A_2 \ldots A_k$, and $Y = B_1\ B_2 \ldots B_m$. Say X -> Y iff it is the intention of the designer that two rows cannot simultaneously agree on X and disagree on Y.

Same wording as Def. 6.6.1, but note that for a set X, agrees if and only if agrees on ALL column value, disagrees if disagrees on ANY column value.

Ex. 6.6.3. We claim what follows is all FDs of emp_info. Interpret by E-R.

(1) emp_id -> emp_name emp_phone dept_name
(2) dept_name -> dept_phone dept_mgrname
(3) skill_id -> skill_name
(4) emp_id skill_id -> skill_date skill_lvl

Note that if we know emp_id -> emp_name, emp_id -> emp_phone, and emp_id -> dept_name, then know emp_id -> emp_name emp_phone dept_name. Easy to see by definition, but DOES require thinking about definition. Three facts about singleton attribute FDs lead to X -> Y fact.

Note that we can conclude from above that designer does not intend that skill_name should be unique for a particular skill. skill_name -> skill_id is not there, nor is it implied by this set (no skill_name on left).

Note that a set of FDs has logical implications to derive OTHER FDs. E.g., emp_id -> emp_name emp_phone dept_name above. There are RULES for how to derive some FDs from others. The simplest one fillows.

Theorem 6.6.3. Inclusion rule. Given T with Head(T). If X and Y are sets in Head(T) and $Y \subseteq X$, then X -> Y (for ANY content for T). (Venn Diagram.)

Proof. By def, need only demonstrate that if two rows u and v agree on X they must agree on Y. But Y is a subset of X, so seems obvious.

Def. 6.6.4. A trivial dependency is an FD of the form X -> Y that holds for any table T where X UNION $Y \subseteq$ Head(T). (Assume ANY content for T.)

Theorem 6.6.5. Given a trivial dependency X -> Y in T, it must be the case that $Y \subseteq X$. (Venn Diagram, X, Y disjoint, show A.)

Proof. Create a table T with Head(T) = X UNION Y and consider the attributes in Y - X. Assume it is non-empty (so it is false that $Y \subseteq X$) and we find a contradiction. Let A be an attribute in Y - X. A trivial dependency must hold for any possible content of T. But since A is not in X, it is possible to construct two rows u and v in T alike in all values for X but having different values in A. Then X -> Y does not hold for this constructed contents, in contradiction to its triviality.

Def. 6.6.6. Armstrong's Axioms. From the following small set of basic rules of implication among FDs, we can derive all others that are true.

[1] Inclusion rule: if $Y \subseteq X$, then X -> Y
[2] Transitivity rule: if X -> Y and Y -> Z, then X -> Z
[3] Augmentation rule: if X -> Y, then X Z -> Y Z
(**LEAVE UP**: Note X Z for sets is the same as X UNION Z.)

Most of these rules are like rules about how to operate in algebra:

if $xy = xz$ and $x \neq 0$, then $y = z$.

Or Geometry: if two triangles ABC and XYZ have side AB equal to side XY, side BC equal to YZ and angle B equal to angle Y, then the two triangles are congruent.

Remember, from Euclid's 11 axioms can derive all other true facts.

Armstrong's axioms are like that. These axioms are said to be COMPLETE, meaning that no other axioms can be added to increase their effectiveness.

Everything we can say about FDs is implied by these axioms.

Let's prove one of these axioms. We will be appealing to definitions of Functional Dependency here to prove an Armstrong Axiom.

Prove Augmentation Rule. If X -> Y then X Z -> Y Z. Consider two rows u and v that agree on X Z, that is they agree on all attributes in X UNION Z. Then since u and v agree on all attributes of X they agree on all attributes of Y (since X -> Y). Similarly, since it is given that they agree on all attributes of Z and we have just shown they agree on all attributes of Y, they agree on all attributes of Y UNION Z. Thus, from the fact that u and v agree on all attributes of X UNION Z we can conclude that they also agree on all attributes of Y UNION Z, and by definition we have X Z -> Y Z.

Some implications of Armstrong's Axioms. pg. 336. Look at! The idea here is that we stop appealing to the definitions and use only the Axioms.

(1) **Union Rule:** If X □ □ Y and X □ □ Z then X □ □ Y Z
(2) **Decomposition Rule:** If X □ □ Y Z then X □ □ Y and X □ □ Z
(3) **Pseudotransitivity Rule:** If X □ □ Y and W Y □ □ Z then X W □ □ Z
(4) **Accumulation Rule:** If X □ □ Y Z and Z □ □ B W then X □ □ Y Z B

NOTE: If you have an old enough text, proof of Accumulation Rule is wrong! Let's build up to it. One Example:

[1] Union Rule: If X -> Y and X -> Z, then X -> Y Z.

Proof: We have (a) X -> Y and (b) X -> Z. By Armstrong's Augmentation rule and (a), we have (c) X X -> X Y. But X X is X UNION X = X, so (c) can be rewritten (d) X -> X Y. Now by (b) and augmentation, we have (e) X Y -> Y Z. And by (d) and (e) and transitivity, we have X -> Y Z, the desired result. [2] & [4] Proved in text.

The idea is that we can use these new Rules as if they were axioms, as facts to prove yet other rules or special cases. Like proving theorem in Geometry, can then use to prove other theorems.

In what follows, when we list a set of FDs, we normally try to list a MINIMAL set, so that a smaller set doesn't exist that will imply these. It will turn out that finding a minimal set of FDs is very important in finding the right relational design by Normalization.

Example 6.6.4. Pg. 338. Consider the table T below fixed in content for all time so the intended FDs can be read off (VERY UNUSUAL). Let's try to list a minimal set of FDs.

Table T

row #	A	B	C	D
1	a1	b1	c1	d1

2	a1	b1	c2	d2
3	a2	b1	c1	d3
4	a2	b1	c3	d4

Analysis. Start by considering FDs with a single attribute on the left.

Always have the trivial FDs, A □ □ A, B □ □ B, C □ □ C, and D □ □ D, but don't list trivial FDs in a minimal set.

(a) All values of the B attribute are the same, so it can never happen for any other attribute P (i.e., where P represents A, C, or D) that $r_1(P) = r_2(P)$ while $r_1(B) \neq r_2(B)$; thus we see that A □ □ B, C □ □ B, and D □ □ B.

At the same time no other attribute P is functionally dependent on B since they all have at least two distinct values, and so there are always two rows r_1 and r_2 such that $r_1(P) \neq r_2(P)$ while $r_1(B) = r_2(B)$; thus: B -/-> A, B -/-> C, and B -/-> D.

(b) Because the D values are all different, in addition to D □ □ B of part (a), we also have D □ □ A and D □ □ C. We state: a KEY (D) functionally determines everything else, which will turn out to be the point.

At the same time D is not functionally dependent on anything else since all other attributes have at least two duplicate values, so in addition to B -/-> D of part (a), we have A -/-> D, and C -/-> D.

List all below without A to C and C to A and show how we know.

(c) We have A -/-> C (because of rows 1 and 2) and C -/-> A (because of rows 1 and 3). Therefore, we can list all FDs (and failed FDs) with a single attribute on the left (we provide a letter in parentheses keyed to the paragraph above that give us each fact).

(a) A □ □ B	(a) B -/-> A	(c) C -/-> A	(b) D □ □ A
(c) A -/-> C	(a) B -/-> C	(a) C □ □ B	(a) D □ □ B
(b) A -/-> D	(a) B -/-> D	(b) C -/-> D	(b) D □ □ C

By the union rule, whenever a single attribute on the left functionally determines several other attributes, as with D above, we can combine the attributes on the right: D □ □ A B C. From the analysis so far, we have the following set of FDs (which we believe to be minimal):

(1) A □ □ B, (2) C □ □ B, (3) D □ □ BAC

(What is the key for this table? Note, really D -> A B C D)

Now consider FDs with *pairs* of attributes on the left. (d) Any pair containing D determines all other attributes, by FD (3) above and the augmentation rule, so there is no new FD with D on the left that is not already implied.

(e) The attribute B combined with any other attribute P on the left, still functionally determines only those attributes already determined by P, as we see by the following argument. If P -/-> Q this means there are rows r_1 and r_2 such that $r_1(Q) \neq r_2(Q)$ while $r_1(P) = r_2(P)$. But because B has equal values on all rows, we know that $r_1(B\ P) = r_2(B\ P)$ as well, so B P -/-> Q. Thus we get no new FDs with B on the left.

(f) Now the only pair of attributes that does not contain B or D is A C, and since A C has distinct values on each row (examine table T again!), we know that A C □ □ A B C D. This is new, but is it minimal?

It is trivial that A C □ □ A, and AC □ □ C, and we already knew that A □ □ B, so it is easy to show that A C □ □ B. (A C □ □ B C by augmentation, B C □ □ B by inclusion, so A C □ □ B by transitivity.) Thus the only new fact we get from seeing that A C □ □ A B C D is that A C □ □ D.

Now consider looking for FDs with triples of attributes on the left. Any triple that does not contain D (which would assure that it functionally determine all other attributes) must contain A C (and therefore functionally determines all other attributes). So everything with three attributes on the left is already handled. Clearly the same holds for any set of four attributes on the left.

The complete set of FDs implicit in the table T is therefore the following:

(1) A □ □ B, (2) C □ □ B, (3) D □ □ A B C, (4) A C □ □ D.

The first three FDs come from the earlier list of FDs with single attributes on the left, while the last FD, A C □ □ D, is the new one generated with two attributes listed on the left. It will turn out that this set of FDs is <u>not quite minimal</u>, despite all our efforts to derive a minimal set. We will see this after we have had a chance to better define what we mean by a minimal set of FDs. This is why we need rigor.

19.

Review end of last time, finished Example 6.6.4, pg 365. Got FDs:

(1) A □ □ B, (2) C □ □ B, (3) D □ □ A B C, (4) A C □ □ D.

Rather startling fact that this is NOT a minimal set, although we tried very hard. Anybody see why? Solved Exercise 6.14 (a), get (reduced set):

(1) A □ □ B, (2) C □ □ B, (3) D □ □ A C, (4) A C □ □ D

Everybody see why it's a reduced set? (Less is said.)

Anybody see why the reduced set implies the original set? (3) D □ □ A C is equivalent (by decomposition and union) to (a) D □ □ A and (b) D □ □ C. Furthermore, since (a) D □ □ A and (1) A □ □ B, by transitivity (c) D □ □ B. Now by (3) and union, D □ □ A B C.

But how did we arrive at this reduced set? Do we always have to think super hard about it? No. There is an algorithm to find the minimal set of FDs for Normalization. Need more definitions.

Def 6.6.9. Closure of a set of FDs. Given a set F of FDs on attributes of a table T, we define the CLOSURE of F, symbolized by F^+, to be the set of all FDs implied by F.

E.g., in above, D □ □ A B C was in the closure of the reduced set of FDs. The MINIMAL set of FDs is what we want, so take out implied ones!

There can be a LOT of FDs in a closure. Ex. 6.6.5, From F = {A -> B, B -> C, C -> D, D -> E, E -> F, F -> G, G -> H}, we can add trivial dependencies A -> A, B -> B, etc., and by transitivity and union get A -> A B, A -> A B C, . . .

In fact any letter -> any subset of letters equal and to it's right in the alphabet. C -> C D E F , C -> D F, B -> C D F G.

In fact, if we name any two sets of letters, one of them has an earliest letter in the alphabet and -> the other (maybe they both have that letter and -> each other).

So the number of implied FDs is the number of subsets of letters squared, about $(2^n)^2 = 2^{2n}$. E.g., from A -> B, B -> C, . . . , J -> K, 10 FDs, get 2^{20} FDs, about a million. If go up to . . . , X □ □ Y, Y □ □ Z, get 2^{52} FDs, about 4,000,000,000,000,000 FDs (four quadrillion).

Lots. Exponential explosion in number of attributes. Even if start with a manageable number with commonsense rules, can get problem.

Really only want to arrive at a MINIMAL set of FDs, so we try to avoid this kind of explosion. Still we need all these concepts.

Def. 6.6.10. FD Set Cover. A set F of FDs on a table T is said to COVER another set G of FDs on T if the set G can be derived by implication rules from the set F, i.e., if G □ F^+. If F covers G and G covers F, we say the two sets of FDs are equivalent, F ∫ G.

Ex. 6.6.6. F = {B -> C D, A D -> E, B -> A} and

G = {B -> C D E, B -> A B C, A D -> E}.

Demostrate in book how F covers G. But also G covers F. See why? B -> C D E implies B -> C D and B -> E by decomposition rule. So have first FD in F.

A better behaved definition from a standpoint of explosion to characterize a set of FDs is the following.

Def. 6.6.11. Closure of a set of attributes. Given a set X of attributes in a table T and a set F of FDs on T, we define the CLOSURE of the set X (under F), denoted by X^+, as the largest set of attributes Y such that X -> Y is in F^+.

We will study closure of set of ***ATTRIBUTES***, not closure of set of ***FDs***.

Algorithm to determine set closure, pg. 342. Pretty intuitive: Start with X = X^+ and just keep looping through the (hopefully small set) of FDs as long as new attributes can be added to X^+.

Pg. 341-42. **Algorithm 6.6.12. Set Closure.** Algorithm to determine X^+, the closure of a given set of attributes X, under a given set F of FDs.

```
I = 0;  X[0] = X;                    /* integer I, attr. set X[0]                */
REPEAT                               /* loop to find larger X[I]                 */
   I = I + 1;                        /* new I                                    */
   X[I] = X[I-1];                    /* initialize new X[I]                      */
   FOR ALL Z □ □ W in F              /* loop on all FDs Z □ □ W in F             */
      IF Z ⊆ X[I]                    /* if Z contained in X[I]                   */
         THEN X[I] = X[I] □ W;       /* add attributes in W to X[I]              */
   END FOR                           /* end loop on FDs                          */
UNTIL X[I] = X[I-1];                 /* loop till no new attributes              */
RETURN X+ = X[I];                    /* return closure of X                      */
```

Note that the step in Algorithm 5 6.12 that adds attributes to X[I] is based on a simple inference rule that we call the *Set Accumulation Rule*, stated thus: If X □ □ Y Z and Z □ □ W then X □ □ Y Z W.

In our algorithm we are saying that since X □ □ X[I] (our inductive assumption) and X[I] can be represented as Y Z (because Z □ X[I]), we can write X □ □ X[I] as X □ □ Y Z, and since F contains the FD Z □ □ W, we conclude by the set accumulation rule that X □ □ Y Z W or in other words X □ □ X[I] UNION W.

Example 6.6.7. In Example 6.6.6, we were given the set F of FDs:

F = {B □ □ C D, A D □ □ E, B □ □ A} (Get B+ = B C D (1) A (3) E (2))
(Shorthand (1), (2): ORDER IMPORTANT)

Given X = B, we determine that X^+ = A B C D E. In terms of Algorithm 6.6.12, we start with X[0] = B. Then X[1] = B, and we begin to loop through the FDs. Because of B □ □ C D, X[1] = B C D.

The next FD, A D □ □ E, does not apply at this time, since A D is not a subset of X[1]. Next because of B □ □ A, we get X[1] □ □ A B C D.

Now X[0] is strictly contained in X[1] (i.e., X[I-1] ⊂ X[I]) so X[I-1] ≠ X[I]. Thus we have made progress in this last pass of the loop and go on to a new pass, setting X[2] = X[1] = A B C D.

Looping through the FDs again, we see that all of them can be applied (we could skip the ones that have been applied before since they will have no new effect), with the only new FD, A D □ □ E, giving us X[2] = A B C D E.

At the end of this loop, the algorithm notes that X[1] ⊂ X[2], progress has been made, so we go on to create X[3] and loop though the FDs again, ending up this pass with X[3] = X[2].

Since all of the FDs had been applied already, we could omit this pass by noting this fact. Note that a different ORDERING of the FDs in F can change the details of execution for this algorithm.

Finding a Minimal Cover of a set of FDs.

Algorithm 6.6.13, pg. 369. **BASIC TO NORMALIZATION!** Given a set F of FDs, construct a set M of FDs that is minimal and covers F.

Let's apply this to the (non-reduced) set of FDs above.

F: (1) A □ □ B, (2) C □ □ B, (3) D □ □ A B C, (4) A C □ □ D
^ Remember, this can come out!!

Step 1, From the set F of FDs, create an equivalent set H with only single FDs on the right. Use decomposition rule. See step 1, pg. 343.

H: (1) A □ □ B, (2) C □ □ B, (3) D □ □ A, (4) D □ □ B, (5) D □ □ C, (6) A C □ □ D

Step 2. Remove inessential FDs from the set H to get the set J. Determine inessential X □ □ A if A is in X^+ under FDs without X -> A.

Try removing (1) A □ □ B, leaving only (2) C □ □ B, (3) D □ □ A, (4) D □ □ B, (5) D □ □ C, (6) A C □ □ D. Take X = A in closure algorithm, clearly get only X^+ = A, because no other FD has its left side contained in X. So need (1).

Try removing others. (2) stays, since no other C on left. Therefore if set X = C couldn't get X+ to contain more than C. (That reasoning is OK.)

How about (3)? Would be left with only: (1) A □ □ B, (2) C □ □ B, (4) D □ □ B, (5) D □ □ C, (6) A C □ □ D. Consider X = D. Get X^+ = D B (4) C (5). Then stop. In fact A not on right of any FDs if take out (3), so (3) needed.

Now try removing (4). Keep: (1) A □ □ B, (2) C □ □ B, (3) D □ □ A, (5) D □ □ C, (6) A C □ □ D. Can we derive D -> B? X = D. Get: D A (3) C (5) B (2). Therefore, got D -> B. <u>So can leave (4) out</u>.

Can't leave out (5) because C not on right of any other FD. Can't leave out (6) because D not on right of any other. Therefore can only reduce set to:

H = (1) A □ □ B, (2) C □ □ B, (3) D □ □ A, (4) D □ □ C, (5) A C □ □ D (Renumber)

Step 3. Successively replace FDs in H with FDs that have a smaller number of FDs on the left-hand side so long as H^+ remains the same.

Test this by successively removing single attributes from multi-attribute left hand sides of FDs, changing X □ □ A to Y □ □ A, then checking if Y^+ under new FD set is unchanged.

(Clearly if we assume Y □ □ A, and Y □ X, can derive everything used to be able to: still true that X □ □ A. ONLY RISK is that Y □ □ A might imply TOO MUCH. I.e., might have Y^+ is LARGER than before!)

Only one to try is (5). If we change this to A □ □ D, does D change? used to be A^+ = A B, now, A^+ = A B D C. No good. How about changing A C □ □ D to C □ □ D? Does C^+ change? Used to be C^+ = C B. Now C^+ = C B D A. So no good, can't reduce. (NO NEED TO TRY STEP 2 AGAIN.)

<u>IF WE DID REDUCE</u> and created a new FD, let us say A -> D to replace A C -> D, we would need to apply Step 2 again to test if A -> D could be removed!!!

Step 4. Apply Union rules to bring things back together on the right for common sets of attributes on the left of FDs, renamed M.

H: (1) A □ □ B, (2) C □ □ B, (3) D □ □ A, (4) D □ □ C, (5) A C □ □ D

M: (1) A □ □ B, (2) C □ □ B, (3) D □ □ A C, (4) A C □ □ D

This is the reduced set, above.

OK, now have algorithm to find a minimal cover from any set of FDs. Almost ready to do Normalization. But need one more concept.

Section 6.7. Lossy and Lossless decomposition. We're going to be factoring tables into smaller tables (projecting onto two subsets of columns that cover all columns and have some columns in common), but it doesn't always work when join back that keep all information of original table.

Always get ALL rows back, but might get MORE. Lose Information. See Example 6.7.1 in text, Pg. 374, a Lossy decomposition.

Ex 6.7.1. A Lossy Decomposition. Consider table, ABC:

Table ABC

A	B	C
a1	100	c1
a2	200	c2
a3	300	c3
a4	200	c4

If we factor this table into two parts, AB and BC, we get the following table contents:

Table AB

A	B
a1	100
a2	200
a3	300
a4	200

Table BC

B	C
100	c1
200	c2
300	c3
200	c4

However, the result of joining these two tables is

AB JOIN BC

A	B	C
a1	100	c1
a2	200	c2
a2	200	c4
a3	300	c3
a4	200	c2
a4	200	c4

This is NOT the original table content for ABC! Note that the same decomposed tables AB and BC would have arisen if the table we had started with was ABCX, with content equal to AB JOIN BC above, or either of two other tables, ABCY or ABCZ:

ABCY

A	B	C
a1	100	c1
a2	200	c2
a2	200	c4
a3	300	c3
a4	200	c4

ABCZ

A	B	C
a1	100	c1
a2	200	c2
a3	300	c3
a4	200	c2
a4	200	c4

Since we can't tell what table content we started from, information has been lost by this decomposition and the subsequent join.

This is known as a *Lossy Decomposition*, or sometimes a *Lossy-Join Decomposition*.

Reason we lose information is many to many matching on join columns. Lose which one on the left was with which one on the right. E.g. a2, 200 on left and a4, 200 on left match with 200, c2 and 200, c4 on right.

Would be OK if always had N-1 relationships on join columns (or 1-1). E.g., CAP, orders can have multiple rows with same pid, but pid is unique for products. Must ALWAYS be unique on one side, not an accident, so need join columns to be SUPERKEY on at least ONE SIDE of join.

Theorem 6.7.3. Given a table T with a set of FDs F and a set of attributes X in Head(T) then X is a superkey of T iff X functionally determines all attributes in T, i.e., X -> Head(T) is in F^+.

Theorem 6.4.7. Given a table T with a set F of FDs valid on T, then a decomposition of T into two tables $\{T_1, T_2\}$ is a lossless decomposition if one of the following functional dependencies is implied by F:

(1) $Head(T_1) \cap Head(T_2) \rightarrow Head(T_2)$, or

(2) $Head(T_1) \cap Head(T_2) \rightarrow Head(T_2)$

This is proved in the text, will leave understanding to you.

NORMAL FORMS NEXT TIME, READ AHEAD. Through 6.8 on Exam 2.

20.

Section 6.8. Normal Forms

OK, what is the point of normalization? We start with a Universal table T and a set of FDs that hold on T. We then create a lossless decompostion:

$T = T_1$ JOIN T_2 JOIN . . . JOIN T_n

so that in each of the tables T_i the anomalies we studied earlier not present. Consider the following Universal table called emp_info:

emp_id, emp_name, emp_phone, dept_name, dept_phone, dept_mgrname, skill_id, skill_name, skill_date, skill_lvl

(1) emp_id -> emp_name emp_phone dept_name
(2) dept_name -> dept_phone dept_mgrname
(3) skill_id -> skill_name
(4) emp_id skill_id -> skill_date skill_lvl

In what follows, we will factor emp_info into smaller tables which form a lossless join. A set of table headings (decomposition tables) together with a set of FDs on those heading attributes is called a Database Schema.

Note that the FDs (1)-(4) above are the kind that define a key for a table. If the attributes on the left and right hand side of FD (1) are the only ones in the table, then emp_id is a key for that table. (Unique, so primary key.)

It is important that the FDs in our list form a minimal cover. E.g., if in above changed (1) to include dept_phone on the right, would have a problem with approach to come.

OK, now want to factor T into smaller tables to avoid anomalies of representation.

(Can concentrate on delete anomaly — if delete last row in some table, don't want to lose descriptive attribute values describing an entity instance.)

Example, remember CAPORDERS, if delete last order for some product, don't want to lose quantity on hand for that product.

In emp_info, if delete last employee with given skill, lose information about that skill (in this case, the name of the skill). So have to have separate table for skills.

What we are really saying here is that emp_id skill_id is the key for emp_info, but a SUBSET of this key functionally determines some skills attributes. This is to be avoided because of the delete anomaly.

There's a cute way of saying what FDs should appear in a normalized table: every attribute must be determined by "the key, the whole key, and nothing but the key." Need separate table for skills because of "the whole key".

Similarly, bunch of attributes are functionally determined by emp_id (including a bunch of dept information by transitivity). These attributes also don't belong in emp_info with key of two attributes. Does this lead to a delete anomaly?

Yes. If delete last skill for an employee, lose information about employee. Therefore have to have separate table for employees. (Include dept info.)

Now have (pg. 385): emps table (emp_id, emp_name, emp_phone, dept_name, dept_phone, dept_mgrname), skills table (skill_id, skill_name), and emp_skills table (emp_id, skill_id, skill_date, skill_lvl).

Now in every one of these tables, all attributes are functionally determined by the key (would not be true if put skill_date in emps table, say), and the whole key (even dept info determined by emp_id).

Any problems? Yes, if delete last employee in a department (during reorganization), lose info about the department, such as departmental phone number.

What's the problem in terms of FDs? Remember, every attribute is dependent on "the key, the whole key, and nothing but the key"? Problem here is that dept_phone, for example, is dependent on dept_name. So fail on "nothing but the key".

This is what is known as a "transitive dependency" or "transitive FD", and cannot exist in a well-behave table with no delete anomaly.

OK, now Figure 6.26, pg. 386: emps table, depts table, emp_skills table, skills table. Three entities and a relationship. Note that there is one table for each FD. Not an accident.

Is it a lossless join? Yes, because consider factoring. OK to factor skills from emp_skills because intersection of Headings contains key for one.
Similarly for emp_skills and emps. And for emps and depts.

(Just to step back to E-R model for a moment, note that the intersections of headings we have just named are foreign keys in one table, keys in the other.

In E-R model, have emp_skills table represents a relationship between Emps and Skills entities. There is also a relationship member_of between Emps and Depts that is N-1 and represented by a foreign key. Picture.)

OK, now need to define Normal Forms (in reverse order to their strength).

BCNF, 3NF, 2NF

2NF is too weak, no DBA would stop there. Essentially, means stopping with depts in emps table. Define near end of section as note.

BCNF definition is more intuitive than 3NF, but 3NF and BCNF are equivalent for the schema we have just been dealing with. Need special situation for difference to arise.

BCNF is in general somewhat more restrictive than 3NF where the situation arises, but has a problem. Often stop with 3NF in real design.

Def. 6.8.4. Boyce-Codd Normal Form, or BCNF. A table T in a database schema with FD set F is in BCNF iff, for any FD X -> A in F^+ that lies in T (all attributes of X and A in T), A is a single attribute not in X, then X must be a superkey for T.

OR: For a non-trivial FD X -> Y where X is minimal, X must be the key for T.

The minimal cover for the FDs whose attributes lie in F are all of the form X -> A, where X is a key for T. Note NOT saying it is a PRIMARY key — might be more than one key.

OK, this seems to be what we were trying to do: Every attribute in T is determined by a key (definition), the whole key (no subset of it) and nothing but the key (any FD determining A turns out to contain a key).

All of the tables in the final Employee Information database schema resulting above are in BCNF. When all the tables of a database schema are in BCNF (3NF, 2NF, etc) we say the database schema is in BCNF (3NF, etc.)

OK, now assume add attributes to emps table attributes to keep track of the street address (emp_straddr), the city and state (emp_cityst) and ZIPCODE (emp_zip). These are all determined by emp_id as now:

(1) emp_id -> emp_name emp_phone dept_name emp_straddr emp_cityst emp_zip (every employee has only one mailing address).

Now the way the PO has assigned ZIPCODES, turn out have new FDs:

(5) emp_cityst emp_straddr -> emp_zip
(6) emp_zip -> emp_cityst (Draw Picture)

Consider the extended emps table with these added attributes:

emps table: emp_id emp_name emp_phone dept_name emp_straddr emp_cityst emp_zip (Leave this up)

This still has key emp_id, but now there are FDs that are NOT determined solely by the key.

Thus emps must factor (database schema 1), while all other tables same:

emps table: emp_id emp_name emp_phone dept_name emp_straddr emp_cityst (now no extraneous FDs in this table.)

empadds table: emp_straddr emp_cityst emp_zip (LOSSLESS)

Note emp_cityst, emp_straddr is key for empadds and in intersection of factored tables (it is a foreign key in emps that determines the address held in empadds), so lossless.

But FD (6) still isn't derived from key in empadds. So for BCNF, empadds must factor further (database schema 2, emps same)

empadds table: emp_zip emp_straddr

zip table: emp_zip emp_cityst

Now empadds has a key consisting of both columns (there is no FD whose left-hand side is in empadds). zip table has emp_zip as key (FD (6)).

(Ask : Why BCNF? Why Lossless?)

PROBLEM: FD 5 doesn't lie in any table. But we always try to "preserve" each FD X -> Y, meaning that X union Y lies in the head of some T_i.

Allows us (for example) to update the street address, city, & state for an employee and test in one table that ZIP was entered correctly.

Thus BCNF has a bad property, and we want to fall back to design of database schema 1. That table doesn't fit BCNF definition because there is a FD X -> A where X is not a superkey of the table. (X is emp_zip, A is emp_cityst.) Need a new normal form.

Def. 6.8.5. A PRIME ATTRIBUTE of a table T is any attribute that is part of a key for that table (not necessarily a primary key).

Def. 6.8.6. Third Normal Form (3NF). A table T in a database schema with FD set F is in 3NF iff, for any FD X -> A implied by F that lies in T, if A is a single non-prime attribute not in X, then X must be a superkey for T.

(In book, makes it clear that this is the same def as BCNF with an escape clause, that can allow the implied condition, X must be a superkey of T, to fail if A is a prime attribute.)

OK, now with weakened 3NF and two FDs:

(5) emp_cityst emp_straddr -> emp_zip

(6) emp_zip -> emp_cityst

Can have all three attributes in a single empadds table, since emp_cityst emp_straddr is the key, and FD (6) doesn't break the rule for X -> A because A = emp_cityst is a prime attribute.

Now have: every non-prime attribute is determined by the key, the whole key, and nothing but the key.

OK, Algorithm 6.8.8 to achieved well behaved 3NF decomposition, pg 393

Given a universal table T and a set of FDs F, generate a set S of headings for a database schema in 3NF that is a lossless decomposition and preserves all FDs in F.

Look at pg. 393. Start by finding minimal cover. Then loop on all FDs, and for every FD, if there is not already a table heading that contains the FD, create a new heading. Finally, if there is a key K for T that is not in any table, create a new table that contains it.

Def. 6.8.8. Second Normal Form (2NF). A table T with FD set F is in 2NF iff: for any X -> A implied by F that lies in T, where A is a single non-prime attribute not in X, X is not **properly** contained in any key of T.

A database schema is 2NF iff all of its tables are 2NF.

21.
Note on Minimal cover Algorithm, pg 369

Note importance of performing Step 2 again after reducing lhs in Step 3. Consider Universal table with attributes A B C. FD set

F = {A B -> C, C -> B, A -> B}

Step 1. Decomposition on the right. No change.

Step 2. Take out single FDs. Can we take out A B -> C? No, since nothing else -> C, and thus if X = A B, then X^+ = A B. Take out C -> B? No, since now C -/-> anything, and if X = C, then X^+ = C. Take out A -> B? But now if X = A, X^+ = A..

Step 3. Reduce sets on the left. A B reduced to A? Now have A -> C. Does this increase X^+ when X = A Before had A -> B so X^+ = A B and then A B -> C so X^+ = A B C. So reducing to A -> C, unchanged.

Do you believe that? Can also test all other sets X for set of X^+, see unchanged. To change, it would have to involve the new FD A -> C and not be implied by A B -> C. But since A -> B, don't see how that is possible.

Step 4, Union. No effect. Therefore Minimal cover is:

A -> C, C -> B, A -> B.

Ridiculous, since FDs 1 and 2 give 3 by transitivity.

Work out Exercise 6.18.

(6.18) (a) Assume we wish to construct a database from a set of data items, {A, B, C, D, E, F, G, H} (which will become attributes in tables), and a set F of FDs given by:

F: (1) A □ □ B C , (2) A B E □ □ C D G H, (3) C □ □ G D, (4) D □ □ G, (5) E □ □ F

(a) Find the minimal cover for this set of FDs, and name this set G.

Step 1.
H = (1) A□ □ B, (2) A□ □ C, (3) ABE□ □ C, (4) ABE□ □ D, (5) ABE□ □ G, (6) ABE□ □ H, (7) C□ □ G, (8) C□ □ D, (9) D□ □ G, (10) E□ □ F

Step 2. (1) A□ □ B: J = rest, X = AC(2)G(5)D(8), not containing B, so keep.
(2) A□ □ C, J = rest, X^+ = A, keep.
(3) ABE□ □ C, X = ABEC(2)D(4)G(5)F(6), containing C, drop.
New H: (1) A□ □ B, (2) A□ □ C, (4) ABE□ □ D, (5) ABE□ □ G, (6) ABE□ □ H, (7) C□ □ G, (8) C□ □ D, (9) D□ □ G, (10) E□ □ F

(4) ABE□ □ D: X⁺ = ABEC(2)G(5)D(8)F(10), containing D, drop.
New H: (1) A□ □ B, (2) A□ □ C, (5) ABE□ □ G, (6) ABE□ □ H, (7) C□ □ G, (8) C□ □ D, (9) D□ □ G, (10) E□ □ F
(5) ABE□ □ G: X⁺ = ABEC(2)G(7)D(8)F(10)G, containing G, drop.
New H: (1) A□ □ B, (2) A□ □ C, (6) ABE□ □ H, (7) C□ □ G, (8) C□ □ D, (9) D□ □ G, (10) E□ □ F
(6) ABE□ □ H: X⁺ = ABEC(2)G(7)D(8)F(10), not containing H, keep.
(7) C□ □ G: X⁺ = CD(8)G(9) containing G, drop.
New H: (1) A□ □ B, (2) A□ □ C, (6) ABE□ □ H, (8) C□ □ D, (9) D□ □ G, (10) E□ □ F
C□ □ D: X⁺ = C, keep.
D□ □ G: X⁺ = D, keep.
E□ □ F: X⁺ = E, keep.

Result H: (1) A□ □ B, (2) A□ □ C, (3) ABE□ □ H, (4) C□ □ D, (5) D□ □ G, (6) E□ □ F

Step 3. Try to drop something from lhs of (3) ABE□ □ H:
AB□ □ H instead in H': AB⁺ under H' = ABC(2)H(3)D(4)G(5)
AB under H = ABC(2)D(4)G(5) Don't get H. Not the same, so can't use.
AE□ □ H instead in H': AE⁺ under H' = AEB(1)C(2)H(3)D(4)G(5)F(6),
AE⁺ under H = AEB(1)C(2)H(3)D(4)G(5)F(10). Same. Drop B from lhs.

New H: (1) A□ □ B, (2) A□ □ C, (3) AE□ □ H, (4) C□ □ D, (5) D□ □ G, (6) E□ □ F.

Must repeat Step 2.

(1) A□ □ B. Without (1) no B on rhs.
(2) A□ □ C. Without (2) no C on rhs.
(3) AE□ □ H. Without (3) no H on rhs.
(4) C□ □ D. Without (4) no D on rhs.
(5) D□ □ G. Without (5) no G on rhs.
(6) E□ □ F. Without (6) no F on rhs.

Step 4. G: (1) A□ □ BC, (2) AE□ □ H, (3) C□ □ D, (4) D□ □ G, (6) E□ □ F

(b) Start with the table T containing all these attributes, and perform a lossless decomposition into a 2NF but not 3NF schema. List carefully the keys for each table (T, T_1, and T_2), and the FDs that lie in each table. Justify the fact that the decomposition is lossless. Explain why it is 2NF but not 3NF.

Def. 6.8.8. Second Normal Form (2NF). A table T with FD set F is in 2NF iff: for any X □ □ A implied by F that lies in T, where A is a single non-prime attribute not in X, X is not **properly** contained in any key of T.

(Note that the **properly** was left out of the text.) A database schema is 2NF iff all of its tables are 2NF.

Remember 3NF says, for each table, FDs are determined by "the key, the whole key, and nothing but the key". 2NF gives us "the whole key". Still allow transitive dependencies, so don't have "nothing but the key". Have.

G: (1) A□ □ BC, (2) AE□ □ H, (3) C□ □ D, (4) D□ □ G, (5) E□ □ F

T = (A, B, C, D, E, F, G, H). Since A and E only occur on lhs of FDs in G, AE must be in any key, and AE^+ = AEBC(1)H(2)D(3)G(4)F(5) = ABCDEFGH, so AE is the key for T.

FD (5), E□ □ F, constitutes a key-proper-subset dependency, and thus needs decomposition for 2NF. E^+ = EF(5), and that's it. Let T1 = (E,F) and T2 = (A,B,C,D,E,G,H) and Then E is the key for T1 and AE is the key for T2.

Note we leave Key in old table for intersection, but take out dependent attribute that would cause bad FD to exist, E□ □ F.

Similarly (1) A□ □ BC is also a key-proper-subset dependency, and A^+ = ABC(1)D(3)G(4), so we further decompose: T1= (E,F), T2 = (A,E,H), T3 = (A,B,C,D,G).

Note H remains within T2 because we haven't found any other home for it: It is not true that A->H or E->H. Just pull out things of that kind.

Now FD E□ □ F lies in T1, FD AE□ □ H lies in T2, and FDs A□ □ BC, C□ □ D and D□ □ G lie in T3. C□ □ D and D□ □ G are transitive dependencies, allowed in 2NF, so we go no further here.

These are lossless decompositions by Theorem 6.7.4. In the first, we note that head(T1) INTERSECT head(T2) = E, the key for T1. In the second, head(T2) INTERSECT head(T3) = A, the key for T3.

The idea here is that A and E are "dimensions" on which other attributes are dependent: need AE, A, and E all separate if there are dependencies in all cases.

E.g., OLAP view of data on dimensions City, Month, and Fuel: $Cost, $Usage dependent on all three; Mean_temp dependent on City and Month; Yearly_usage dependent on City and Fuel; FOB_cost dependent on Fuel and Month; Latitude dependent on City only; BTU_per_unit dependent on Fuel only; Fuel_days dependent on Month only.

(c) Continue decomposition to bring this database to 3NF. Is this table also BCNF?

To complete the decomposition to 3NF, we need to factor out the transitive dependencies. Have T3 = (A,B,C,D,G), with FDs: (3) C□ □ D and (4) D□ □ G.

Change T3 to (A,B,C) and create new tables T4 = (C,D) and T5=(D,G). Can you say why join is now lossless, ?

Now we have one FD contained in each table. In each case the lhs is a key for its table, so the schema is in BCNF. (Don't have to worry about FDs with non-prime lhs.)

(d) Use Algorithm 6.8.8 and the set G of FDs to achieve a lossless 3NF decomposition that preserves FDs of G. Is this the same as the decomposition in part (c)?

S = nullset. Loop through all the FDs of G:
G = { A□ □ BC, AE□ □ H, C□ □ D, D□ □ G, E□ □ F}
(Loop through FDs, and if not already contained in table of S, create new)
A□ □ BC: S = {ABC}.
AE□ □ H: AEH not contained in ABC, so add it: S={ABC,AEH}
C□ □ D: CD not contained in either yet, so add it: S= {ABC,AEH,CD}
D□ □ G: DG not contained in any yet, so add it: S = {ABC,AEH,CD,DG}
E□ □ F: EF not contained in any yet, so add it: S = {ABC,AEH,CD,DG,EF}
AE is the only candidate key, and it is contained in AEH, so done.
Is this is the same decomposition as in c?

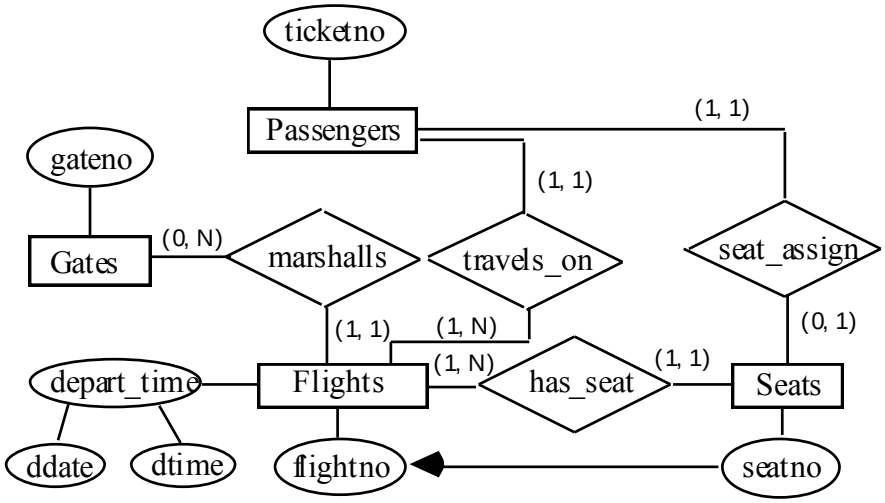
21.

Last .

Reconsider Airline Reservation design of Figure 6.14, pg. 351

What is the Universal table?

reservations: ticketno gateno flightno ddate dtime seatno



What are FDs? Have entity Flights with identifier flightno, and fact that each Flight has a unique Gate assigned:

(1) flightno □ □ ddate dtime gateno (or fn □ □ dd dt ga)

Have entity Passengers with identifier ticketno and fact that every passenger has a particular seat assigned and travels on a specific flight:

(2) ticketno □ □ seatno flightno (or tn □ □ sn fn)

We've been treating weak entity Seats as if all seatno values are different for different flights (each belongs to (1, 1) Flights) so:

(3) seatno □ □ flightno (or sn □ □ fn)

For this universal table, key would be: ticketno

Minimal cover? Step 1. Write out. Step 3, no multi-attribute sets on left to reduce. Step 2: Can get rid of ticketno □ □ flightno. Step 4, Union.

Final: (1) fn -> dd dt ga, (2) tn -> sn, (3) sn -> fn

2NF is: tn dd dt ga sn fn, but two transitive dependencies.

The FDs would give us tables (see pg. 322): passengers (ticketno, seatno), flights (flightno, gateno, ddate, dtime), and seats (seatno, flightno).

Have done away with relationahip travels_on and resulting foreign key, which is indeed unnecessary.

But for query efficiency, might want to keep it. Fewer joins that way when ask what gate passenger should go to. This is called "Denormalization".

You see, it helps to get minimal cover and do normalization in thinking about difficult E-R intuition.

No gates table. Normalization is saying that gates isn't an entity because gateno doesn't functionally determine anything. If had: gateno □ □ gatecapacity, then would have a table.

Recall that in emps table, the multi-valued attribute hobby did not qualify as an entity. We might disagree, but only if someone wanted to look up all hobbys that ever existed for some reason. Like gateno.

Without having gateno remembered in gates table, will lose track of gate numbers (delete anomaly). Might be inappropriate (might be used in programs to assign gates to flights, don't want to lose track).

But this fact doesn't come out of normalization as it stands. Comes out of intuition of E-R. So both approaches valuable.

OK, now Section 6.9. Consider CAP database, E-R design of Figure 7.2, pg. 451. And consider the following FD that holds in the original Universal CAPORDERS table:

(1) qty price discnt □ □ dollars

This is true because dollars is calculated for insert statement by:

```
exec sql insert into orders
    values (:ordno, :month, :cid, :aid, :pid, :qty,
        :qty*:price - .01*:discnt*:qty*:price);
```

Where :price has been determined from products for this pid and :discnt from customers for this cid.

But now does this mean we left a FD out of our normalization? How would we change the design of 2.2, customers, agents, products, orders?

Need to add a new table which I will call ddollars (for "determined dollars") with the heading: qty price discnt dollars, and remove dollars from orders. The ddollars table has key: qty price discnt

Now what is the value of this? When insert a new order, the key will guarantee that the value dollars is unique. But of lots of different qty, price, discnt triples, many such rows might be unique anyway.

Strange approach to validating. I would be happier to depend on the insert statement (which also has to calculate :discnt and :price).

Problem here is that a very common query for an order with dollars total will now require a join of orders, products, customers, and ddollars, instead of just being read off a row of orders.

I would leave FD out of the design. Improves queries at the expense of a validation of questionable value.

Design tools. Most of them are based on E-R. I brought one, the CHEN Workbench Methodology Guide. Tool will then do things like automatically translate into Create Table definitions, with appropriate constraints.

Microsoft Access will take a few hundred rows of data in a universal table and determine all FDs that seem to hold (only statistical). Often user would miss these.

Lecture Notes for Database Systems, part 2

by Patrick E. O'Neil

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.

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.

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 **can reference** a primary key **in the same table**.

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

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.)

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. 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;
```

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,), 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             YESNOTE: can update only columns in table
AID             NO          with single-valued participation, i.e..
PID             YESorders, except columns involved in join.
DOLLARS         YES
ANAME           NO
PERCENT         NO
ORDNO           YES
7 rows selected.
```

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. [: 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!

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 placed in Alphabetical order by lookup key in "B-tree" (usually), explained below. Also might be hashed.

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, recordpointer) 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 agnamcit on agents (aname, city);

and had (aname, city) for two rows eual 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.

NOTE: LATER CHANGES to a table are immediately reflected in the 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).

Probably 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.

7.

Recall X/OPEN form of Create Index:

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

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
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 in the index structure in the place of entries pointing to rows.

Clustered index means rows in same order as index entries (volumes on shalves 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.

Model 100 MIPS computer. Takes .00000001 seconds to acess 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 (2 KB or 4KB: data block, in ORACLE) (Transfer time). Rough idea of time:

Seek time:	.008 seconds
Rotational latency:	.004 seconds (analyze)
Transfer time:	.0005 seconds (few million bytes/sec: ew K bytes)

Total is .0125 seconds = 1/80 second. Huge compared to memory access.

Typically a disk unit addresses ten Gigabytes and costs 1 thousand dollars (new figure) with disk arm attached; not just pack which is much cheaper.

512 bytes per sector, 200 sectors per track, so 100,000 bytes per track. 10 surfaces so 1,000,000 bytes per cylinder. 10,000 cylinders per disk pack.

Total is 10 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 word can't spell. Send letter to St Petersberg, six weeks to get response (1780). Can't do work until response (work on other projects.)

From this see why read in what are called pages, 2K bytes on ORACLE, 4K bytes in 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.

Buffer Lookaside

Similarly, idea of buffer lookaside. Read pages into memory buffer so can access them. Once to right place on disk, transfer time is cheap.

Everytime 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. 473)

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 at least 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 $4000 per gigabyte.
Disk storage (with disk arms) costs about 100 dollars a Gigabyte.

So could buy enough memory so bring all data into buffers (no problem about fast access with millions of memory buffers; very efficient access.)
Coming close to this; probably will do it soon enough. Right now, being limited by 4 GBytes of addressing on a 32 bit machine.

8. Create Tablespace

OK, now DBA and disk resource allocation in ORACLE.

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

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

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

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.

DBA plan. Construct Tablespace from operating system files (or disk partitions). Can specify tablespace crosses disks, stays on one disk, etc.

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. Now see Fig 8.4, pg. 476. [**Leave Up**]

```
CREATE TABLESPACE tblspname
    DATAFILE 'filename' [SIZE n [K|M]] [REUSE] [AUTOEXTEND OFF
    | AUTOEXTEND ON [NEXT n [K|M] [MAXSIZE {UNLIMITED | n [K|M]}]
    {, 'filename' . . .}
    -- the following optional clauses can come in any order
    [ONLINE | OFFLINE]
    [DEFAULT STORAGE ([INITIAL n] [NEXT n] [MINEXTENTS n]
        [MAXEXTENTS {n|UNLIMITED}] [PCTINCREASE n])
        (additional DEFAULT STORAGE options not covered)]
    [MINIMUM EXTENT n [K|M]]
   [other optional clauses not covered];
```

Operating systems files named in datafile clause. ORACLE is capable of creating them itself; then DBA loses ability to specify particular disks.

The idea here is that ORACLE (or any database system) CAN create its own "files" for a tablespace.

If SIZE keyword omitted, data files must already exist, ORACLE will use. If SIZE is defined,ORACLE will normally create file. REUSE means use existing files named even when SIZE is defined; then ORACLE will check size is right.

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 alter offline/online later for recovery purposes, reorganization, etc., without bringing down whole database.

SYSTEM tablespace, created with Create Database, never offline.

When table first created, given an initial disk space allocation. Called an initial *extent*. When load or insert runs out of room, additional allocations are provided, each called a "next extent" and 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: default 10240
NEXT n: size in bytes of next extent 1. (same) May grow.
MAXEXTENTS n: maximum number of extents segment can get
MINEXTENTS n: start at creation with this number of extents
PCTINCREASE n: increase from one extent to next. default 50.

Minimum possible extent size is 2048 bytes, max is 4095 Megabytes, all extents are rounded to nearest block (page) size.

The 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. (**Leave Up**)

```
CREATE TABLE [schema.]tablename
   ({colname datatype [DEFAULT  {constant|NULL}] [col_constr] {, col_constr…}
         | table_constr}
   {, colname datatype etc. . . .}
   [ORGANIZATION HEAP | ORGANIZATION INDEX (with clauses not covered)]
   [TABLESPACE tblspname]
   [STORAGE ([INITIAL n [K|M]] [NEXT n [K|M]] [MINEXTENTS n]
         [MAXEXTENTS n] [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 delete's, 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). There can be a default storage clause with a tablespace, inherited by 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, then stop inserrts when >90% full, start when <90% full. If 20, 90, ill-defined between 80 and 90.

Data Storage Pages and Row Pointers: Fig. 8.6

Typically, after table is created, extents are allocated, rows are placed one after another on successive disk pages (in INGRES, call 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 names the kind of page (data segment, index segment), and 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.

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.

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

Disk Pointer. RID (DB2), ROWID (ORACLE), TID (INGRES). 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 tableas 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 CROWID
---------- ----------
Basics 00000EF3.0000.0001
Allied 00000EF3.0001.0001

The alternative "extended ROWID" form 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 + 5, 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 find a row (through an index or slower means) 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

In ORACLE, allow rows that are larger than any single page (2 KB).

If a row gets too large to fit into it's home block, it is split to a second block. Second part of row has new ROWID, not available to any external method, only to chainer.

Means that random access requires two or more page reads (Voltaire problem). 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.

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 . . . ]  (see pg. 478, par 4 ff, 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 really used. ORACLE B-tree is usable in both directions (leaf sibling pointers left and right).

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

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

Now sort these on disk, so in order by kevalues. Explain NOSORT clause (telling ORACLE rows are in right order, so don't have to sort, but error returned if ORACLE notices this is false).

Now idea of binary search, Example 8.3.1, pg. 486 ff. Assume N = 7, searching array of structs: arr[K].keyval, arr[K].rowid (list of pairs 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)
   {
      intprobe = 3,                    /* first probe position at subscript K =  3          */
            diff = 2;

      while (diff > 0) {               /* loop until K to return                            */
         if (probe <= 6 && x >  arr[probe].keyval)
            probe = probe + diff;
         else
            probe = probe - diff;
         diff = diff/2;
      }                                /* 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?) return 1. If x = 5, return -1. More?

Given duplicate values in the array, binsearch will always return the *smallest* subscript K such that x == arr[K].keyval (exercise).

The binsearch function generalizes easily: given N entries rather than 7, choose m so that $2^{m-1} < N \leq 2^m$; 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 14 (maybe).

That's 13 I/Os!! A year of letters to Voltaire. We can make it 9 weeks!

10.

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 pg has a range of keyvalues, and want to create optimal directory to get to proper leaf node. Node ptrs np and separators.

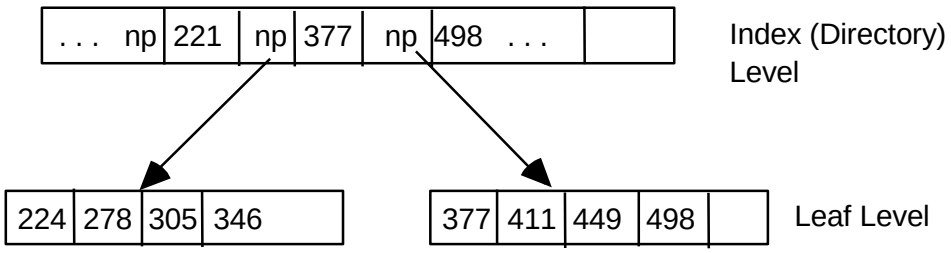


Figure 8.10 Directory structure to leaf level nodes

Search for entry keyvalue 305. Bin search in index (directory) for largest key smaller than x. Then following separator points to proper node.

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.

Say have gotten to right index directory node; do binary search on index node for np to follow, then follow it to proper leaf page.

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

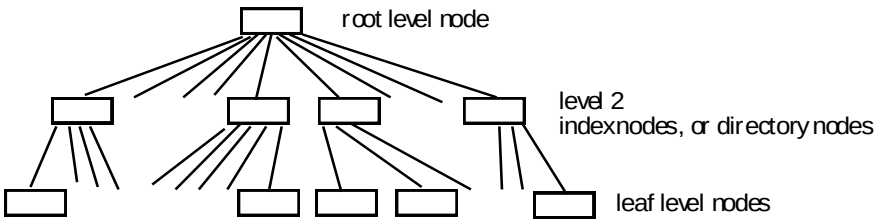


Figure 8.11 Schematic Picture of a three level B-tree

Above *leaf nodes*, have *index nodes* or *directory nodes*. Depth of 3.

Search algorithm is to start at root, find separators surrounding, follow np down, until leaf. Then search for entry with keyval == x if exists.

Only 3 I/Os. 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 fits, but not 4000 leaf nodes: only ONE I/O.

Get same savings in binary search. First several levels have1 + 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.)

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 in an even, balanced way. Consider a sequence of inserts, **SEE 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.

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.

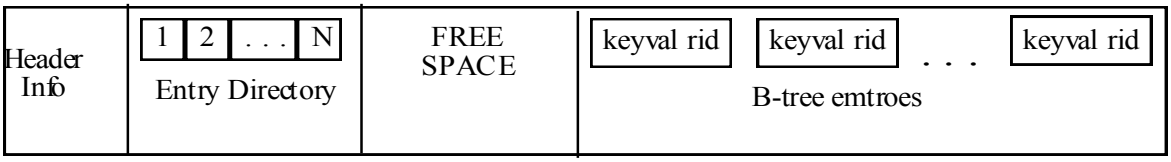
Properties of the B-tree, pg. 499 ff, Talk through. (Actually, B+tree.)

- 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. 500), 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.

ROUGH CALCULATIONS LIKE THIS ARE FINE. 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 makes 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.

11.

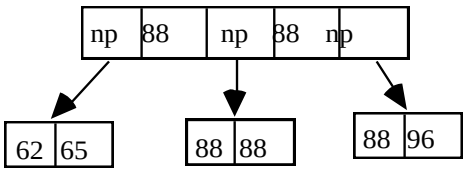
Missed a . Exam 1 is Wed, March 22. (After Vac.) Hw 2 due by Monday, March 20, solutions put up on Tuesday to allow you to study.

If you are late, your hw 2 doesn't count. **NO EXCEPTIONS. TURN IN WHAT YOU HAVE READY!**

Duplicate Keyvalues in an Index.

What happens if have duplicate keyvalues in an index? Theoretically OK, but differs from one product to another.

Look at Fig 8.12, pg 498, right-hand, and assume add we a bunch of new rows with duplicate keyvalues 88. Get e.g. Fig 8.16 following on pg 504.



(Work out slowly how get it). Seems kind of strange to call 88 a separator value, since 88 on both nodes below. Now consider Select statement:

select * from tbl where keyval = 88;

Have to find way down to list of 88s. Done in binsearch, remember? Leftmost one GE 88. Must follow np to left of sepkey = 88. (No 88 at that leaf level, but there could be one since dups allowed)

Then will go right on leaf level while keyval <= 88, following sibling pointers, access ROWIDs and read in rows. Point of this is that everything works fine with multiple duplicates: you should think about how.

Consider the process of deleting a row: must delete its index entries as well (or index entries point to non-existent row; could just keep slot empty, but then waste time later in retrievals).

Thus when insert row, must find way to leaf level of ALL indexes and insert entries; when delete row must do the same. Overhead.

Treat update of column value involved in index key as delete then insert.

Point is that when there are multiple duplicate values in index, find it HARD to delete specific entry matching a particular row. Look up entry with given value, search among ALL duplicates for right RID.

Ought to be easy: keep entries ordered by keyvalue||RID (unique). But most commercial database B-trees don't do that. (Bitmap index solves this.)

Said before, there exists an algorithm for efficient self-modifying B-tree when delete, opposite of split nodes, merge nodes. But not many products use it: only DB2 UDB seems to. (Picture B-tree shrinking from 2 levels to one in Figure 8.12! After put in 39, take out 41, e.g., return to root only.)

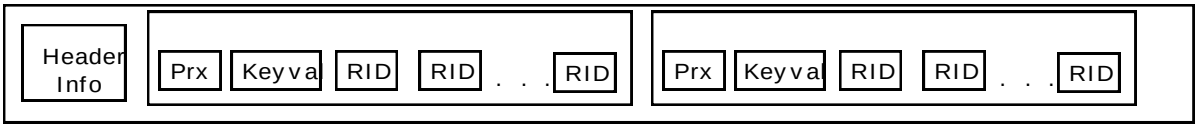
Lot of I/O: argues against index use except for table with no updates. But in fact most updates of tables occur to increment-decrement fields.

Example. Credit card rows. Indexed by location (state || city || staddress), credit-card number, socsecno, name (lname || fname || midinit), and possibly many others.

But not by balance. Why would we want to look up all customers who have a balance due of exactly 1007.51? Might want all customers who have attempted to overdraw their balance but this would use an (indexed) flag.

Anyway, most common change to record is balance. Can hold off most other updates (change address, name) until overnight processing, and reorganize indexes. Probably more efficient.

DB2 has nice compression for multiple duplicate keyvalues in an index.
See Fig. 8.17 on pg 505 for data page layout with duplicate keyvalues.



Each block has keyval once and then a list of RID values, up to 255. Prx contains pointer to prior block (2 bytes), next block (2 bytes) and count of RIDs in this block (1 bytes) plus 1 byte unused. [TEXT: now 1 block/page?]

Since Keyval can be quite long (lname || fname || midint, say 30 chars or bytes), get great space saving (if multiple duplicates). With 255 RIDs for each keyval, down nearly to 4 bytes/entry instead of 34 bytes/entry.

Oracle Bitmap Index

A bitmap (or bit vector) can be used in an ADT to represent a set, and this leads to very fast union, intersection, and member operations.

A bitmap index uses ONE bitmap for each distinct keyval, like DB2 with RID list. A bitmap *takes the place of a RID list*, specifying a set of rows.

See example, pg. 505-506, emps table with various columns. Low card columns are good candidates for bitmap indexes. Easy to build them.

```
create table emps (eid char(5) not null primary key,
    ename varchar(16), mgrid char(5) references emps,(ERROR IN TXT)
    gender char(1), salarycat smallint, dept char(5));

create bitmap index genderx on usemps(gender); (2 values, 'M' &'F')
create bitmap index salx on usemps(salrycat);  (10 values, 1-10)
create bitmap index deptx on usemps(dept);  (12 vals, 5 char: 'ACCNT')
```

These columns are said to have *low cardinality* (cardinality is number of values in a column).

For structure of a bitmap index, think of assigning ordinal numbers to N rows, 0, 1, 2, 3, . . . Keep bits in load order for rows (order as they lie on disk?). Require a function to go from ordinal number to ROWID and back.

Then bitmap for a property such as gender = 'F' is a sequence of N bits (8 bits/byte) so bit in position j is 1 iff row j has property, 0 otherwise.

OK, have bitmaps for each column *value*. Put one for each keyvalue in B-tree to make an index?

Bitmap index, Fig. 8.18, pg. 506. Note that 100,000 rows gives 12,500 bytes of bitmap; Bitmap broken into *Segments* on successive leaf pages.

Same idea as Blocks of RID-lists in DB2 UDB.

Idea of bitmap density: number of 1-bits divided by N (total # bits)

In DB2, RID needs = 4 bytes. Thus RID-list can represent one row with 32 bits. At density = 1/32 , bitmap can represent in same space.

But if density is a lot lower than 1/32, say 1/1000 (out of 100,000 rows) need 1000 N-bit bitmaps. Total RID-list stays the same length with 1000 column values, but (Verbatim) bitmap index requires a lot more space.

ORACLE uses compression for low-density bitmaps, so don't waste space. Call bitmap "verbatim" if not compressed (means moderately high density).

Fast AND and OR of verbatim bitmaps speeds queries. Idea is: overlay unsigned int array on bitmap, loop through two arrays ANDing array (& in C), and producing result of AND of predicates. Parallelism speeds things.

But for updates, bitmaps can cause a slowdown when the bitmaps are compressed (need to be decompressed, may recompress differently).

Don't use bitmap indexes if have frequent updates (OLTP situation).

8.4 Clustered and Non-Clustered Indexes

The idea of a clustered index is that the rows of the table are in the same order as the index entries — by keyvalue.

In library, order fiction on shelves by Author (lname || fname || midinit || title). If look up novels by Dickens, go to shelves, they're all together.

In most database products, default placemen of rows on data pages on disk is in order by load or by insertion (heap). If we did that with books in a library, would be hard to collect up all Dickens.

Look at Figure 8.19, pg 510. Advantage of clustering in range search is that rows are not on random pages but in appropriate order to read in successive rows (don't need to perform new page: multiple rows/page).

Example 8.4.1. Advantage of Clustering. Large department store with hundreds of branch stores, has records for 10 million customers.

Assume each row is 100 bytes. 1 Gbyte of data. Probably only small fraction of data is in memory buffers (most systems have less than 200 MBytes of buffer available for a single table).

Boston has 1/50 of the 10M customers, or 200,000. Do a mailing,

```
select name, staddress, city, state, zip from customers
    where city = 'Boston' and age between 18 and 50 and hobby in
  ('racket sports', 'jogging', 'hiking', 'bodybuilding', 'outdoor sports');
```

OK, so each row is on a random page in non-clustered case, not found in buffer. Say read all rows in Boston (200,000 or 1/50 of 10 M).

If each row is a random page, that's 200,000 real I/Os. But 200,000 I/Os divided by 80 (I/Os/(I/Os/sec)) = secs) = 2500 seconds, about 40 minutes.

In this calculation we assumed only one disk arm moving at a time. Even if more in motion, will answer THIS query more quickly, but important point is resource use: might have multiple users contending for disk arms.

In clustered case, fit 100 byte rows twenty to a page. 20 rows/page (2KBytes/page). Since clustered, right next to one another, place 200,000 rows on 200,000/20 = 10,000 pages. (pages/(rows/page) = pages).

Now reading all rows in Boston takes (10,000/80) = 125 secs. Two minutes. 20 times faster.

That's reading all the rows for Boston. If we have indexes on age and hobby we don't need to read all these rows. Suppose these predicates eliminate 80% of the rows, leaving 1/5 of them to read, or 40,000 rows.

In the non-clustered case, 40,000 reads take 40,000/80 = 500 secs.

In the clustered case, we might still have to read the 10,000 pages: depends if hobby or age cluster city = 'Boston' further; probably not.

So still need 125 seconds (assume every page still contains at least one row, when 1/5 are hit). But still 5 times faster than non-clustered.

Clustered Index in DB2

Now how do we arrange clustered index? In DB2 UDB, have CLUSTER clause in Create Index statement (Fig 8.20, pg 512).

At most one cluster index. Create index with empty table, then LOAD table with sorted rows (or REORG after load with unsorted rows).

This is NOT a primary key index: DB2 UDB places all rows in data pages rather than on leaf of B-tree. No guarantee that newly inserted rows will be placed in clustered order!

DB2 UDB leaves space on data pages using PCTFREE spec, so newly inserted row can be *guided* to a slot in the right order. When run out of extra space get inserted rows far away on disk from their cluster ordering position.

DB2 uses multipage reads to speed things up: "prefetch I/O"; more on this in next chapter.

ORACLE Index-Organized Tables

This can provide a *primary key* clustered index, with table rows on leaf level of a B-tree, kept in order by B-tree splitting even when there are lots of changes to table. See pg. 513-514: ORGANIZATION INDEX

create table schema.tablename ({columnname datatype . . .
[ORGANIZATION HEAP | ORGANIZATION INDEX (with clauses not covered)]

(Back before ORACLE 8.i, used not to be able to have secondary indexes at the same time. But that's no longer a problem.)

ORGANIZATION INDEX *does require* a primary key index. The primary key is used automatically.

12.

ORACLE [Table] Clusters: (Not *Clustering*), Fig. 8.21, pg. 514

CREATE CLUSTER [schema.]clustername -- this is BEFORE any table!
(colname datatype {, . . .}
[cluster_clause {. . .}];

The cluster_clauses are chosen from the following:

[PCTFREE n] [PCTUSED n]
[STORAGE ([INITIAL n [K|M]] [NEXT n [K|M]] [MINEXTENTS n] [MAXEXTENTS n]
[PCTINCREASE n])]
[SIZE n [K|M]] -- disk space for one keyval: default to 1 disk block
[TABLESPACE tblspacename]
[INDEX | HASHKEYS n [HASH is expr]]
[other clauses not covered];

DROP CLUSTER statement:

DROP CLUSTER [schema.]clustername
[INCLUDING TABLES [CASCADE CONSTRAINTS]];

Consider example of employees and departments, where the two tables are very commonly joined by deptno. Example 8.4.2, pg. 516.

```
create cluster deptemp
    (deptno int)
    size 2000;

create table emps
(   empno int primary key,
    ename varchar(10) not null,
    mgr int references emps,
    deptno int not null)
    cluster deptemp (deptno);

create table depts
(   deptno int primary key,
    dname varchar(9),
    address varchar(20))
    cluster deptemp (deptno);
```

When we use deptno as a cluster key for these two tables, then all data for each deptno, including the departments row and all employee rows, will be clustered together on disk.

Steps:

1. create a cluster (an index cluster, the default type)
2. create the tables in the cluster
3. create an index on the cluster
4. load the data, treating the tables normally.
5. Possibly, add more (secondary) indexes.

Step 1. includes all the physical-level specifications (STORAGE clauses), so these are not done in step 2.

```
CREATE TABLE [schema.]tablename
   (column definitions as in Basic SQL, see Figure 8.5)
   CLUSTER clustername (columnname {, columnname})
      -- table columns listed here that map to cluster key
   [AS subquery];
```

The cluster owns the table data. It does not own the index data, so step 3 can specify tablespace or STORAGE specs.

```
   create index deptempx on cluster deptemp;
```

Note this does NOT give us the clustered index features mentioned earlier. ROWS ARE NOT PUT IN ORDER BY KEYVALUE WHEN LOADED IN CLUSTER.

All rows with single keyvalue are stored together, but there is no ordering attempted; index on keyvalyue is separate idea from cluster.

8.5 Hash Primary Key Index

OK, now for a complete departure. See pg 517, ORACLE Cluster with use of optional clause

HASHKEYS n [HASH is expr]

Rows in a table located in hash cluster are placed in pseudo-random data page slots using a hash function, and looked up the same way, often with only one I/O. THERE IS NO DIRECTORY ABOVE ROW LEVEL.

Also, no order by keyvalue. Successive keyvals are not close together, probably on entirely different pages (depends on hash function).

Can't retrieve, "next row up in keyvalue," need to know exact value. Serious limitation, but 1 I/O lookup is important for certain applications.

Idea of Primary Index. Determines placement of rows of a table. Cluster index does this, but implies that rows close in keyvalue are close on disk.
More correct to say Primary Index when hash organization.

In hash cluster, HASHKEYS n is the proposed number of slots S for rows to go to (like darts thrown at random slots, but repeatable when search).

ORACLE equates the value n with "HASHKEYS" in it's references.

It will actually choose the number of slots S to be the first prime number >= HASHKEYS, called PRIMEROOF(HASHKEYS). Example 8.5.1 has this:

```
create cluster acctclust (acctid int)
    size 80
    hashkeys 100000;
```

Can figure out how many slots actually allocated:

```
select hashkeys from user_clusters
    where cluster_name = 'ACCTCLUST';
```

Find out it is 100003. Note we can create our own hash function (not normally advisable, add to Create Cluster above):

```
create cluster acctclust (acctid int)
    size 80
    hashkeys 100000 hash is mod(acctid, 100003);
```

Now in Example, create single table in this cluster with unique hashkey:

```
create table accounts
(   acctid integer primary key,
    . . .)  -- list of other columns here
cluster acctclust (acctid);
```

Note that in accounts table, B-tree secondary index created on acctid because it's a primary key. Access normally through hash though.

HASHKEYS and SIZE determine how many disk blocks are initially allocated to cluster. First determine how many slots S = PRIMEROOF(HASHKEYS).

Figure how many slots B will fit on a disk block: [TEXT] [WAS ERROR]

```
B = MAX(FLOOR(Number of usable bytes per disk block/SIZE), 1)
```

The number of usable byte in a disk block on UMB machine db is 1962. Will change. Can use 2000 for calculations on hw, Exam.

Next, calculate how many Disk blocks needed for S slots when B to a block.

```
D = CEIL(S/B)
```

(So SIZE and HASHKEYS determine number of disk blocks in hash cluster; can't change HASHKEYS later; SIZE only advisory.)

Assign one or more tables to hash cluster with (common) hashkey (i.e., clusterkey for hash cluster). I will tend to speak of one table, unique key.

Example given in Ex. 8.5.1 is single accounts table with accountid hashkey.

Key values will be hashed to some slot and row sent to that slot.

sn1 = h(keyval1)

Could have two distinct key values hashed to same slot (collision).

(When multiple tables involved, could clearly have multiple rows with SAME keyvalue go to same slot, not a collision but must be handled.)

(Even if unique, if number of distinct keys grows beyond number of slots, must have a lot of collisions. Will have collisions before that.)

The way ORACLE handles this is to expand its slots on a page as long as space remains, and after that overflow to new pages. See Fig. 8.23, pg 516.

(Draw a picture of Root pages originally allocated and overflow pages.)

In hashing, try not to overflow. If lot of rows would hash to same slot, not appropriate for hashing. E.g., depts-emps when lot of emps per deptno.

In accounts case when lot of collisions, while most rows on root block, Query plan uses hashed access; after a lot of overflow, tries to use secondary index on acctid.

Usually try to create enough slots of sufficient size so collision relatively rare, disk block pages only half full.

In Unique hashkey case, make enough slots so on average only half-full.

No Incremental Changes in Size of Hash Table

How create function h(keyvalue) for 0 to S-1. Have more generic function r(x) (random number based on x), so r(keyvalue) is in range (0.0 to 1.0).

(Easy — just generate random mantissa, characteristic is 2^0). Now set:

h(keyvalue) = INTEGER_PART_OF(S*r(keyvalue))

But note that we can't incrementally increase size of hash table based on this. If have different number of slots S', new h' won't give same slots for old values. If r(keyvalue) = 0.49, INT(4*0.49) = 1, int(5*0.49) = 2.

Therefore can't enlarge number of slots incrementally to reduce number of collisions. (There is a known algorithm allowing dynamic hash table growth, used for example in Sybase IQ).

8.6 Throwing Darts at Random Slots

Conceptual experiment of throwing darts at slots from a LONG way away. Hash into slots as in ORACLE (random numbers), how many slots empty?

The case of ORACLE is the experiment: Unlimited Slot Occupancy, question of how many slots are occupied. Number of darts N, number of slots M.

Question is: how many slots occupied. Not just M - N because some slots will be occupied twice, some three times.

Pr(slot s gets hit by dart d) = 1/M

Pr(slot s does not get hit by dart d) = (1 – 1/M)

Pr(slot s does not get hit by N darts thrown in succession) = $(1 – 1/M)^N$

Pr(slot s does get hit by one or more of N darts) = $1 - (1 – 1/M)^N$

E(S) = Expected number of slots hit by N darts = $M(1 - (1 – 1/M)^N)$

Show that (approximately) **[8.6.3]** $e^{-1} = (1 – 1/M)^M$

[8.6.4] $E(S) = M (1 – e^{–N/M})$

[8.6.5] $E(\text{Slots not hit}) = M e^{–N/M}$

Then solve question when slot occupancy of 1: how many retries to get last dart in slot (this is question of Retry Chain, will use later).

Easy to figure out that as number of darts gets closer and closer to number of slots, number of retries becomes longer and longer.

If (N-1)/M close to 1, Length of Retry Chain E(L) to place Nth dart approaches infinity. If N-1 = M, can't do it. Call (N-1)/M the FULLNESS, F.

Then **[8.6.11]** E(L) = 1/(1 - F)

Finally: When Do Hash Pages Fill Up. Say disk page can contain 20 rows, gets average of 10 rows hashed to it.

When will it get more than it can hold (hash overflow, both cases above)?

Say we have 1,000,000 rows to hash. Hash them to 100,000 pages. This is a binary distribution, approximated by Normal Distribution.

Each row has 1/100,000 probability P of hitting a particular page.

Expected number of rows that hit page is P*1,000,000 = 10.

Standard Deviation is: SQRT(P*(1-P)*1,000,000) = SQRT(10) = 3.162.

Probility of 20 or more rows going to one page is PHI(10/3.162) = .000831, but that's not unlikely with 100,000 pages: 83 of them.

Chapter 9. Query Optimization

Query optimizer creates a procedural access plan for a query. After submit a Select statement, three steps.

```
select * from customers where city = 'Boston'
    and discnt between 10 and 12;
```

(1) Parse (as in BNF), look up tables, columns, views (query modification), check privileges, verify integrity constraints.

(2) Query Optimization. Look up statistics in System tables. Figure out what to do. Basically consider competing plans, choose the cheapest.
(What do we mean by cheapest? What are we minimizing? See shortly.)

(3) Now Create the "program" (code generation).
Then Query Execution.

We will be talking about WHY a particular plan is chosen, not HOW all the plan search space is considered. Question of how compiler chooses best plan is beyond scope of course. Assume QOPT follows human reasoning.

For now, considering only queries, not updates.

Section 9.1. Now what do we mean by cheapest alternative plan? What resources are we trying to minimize? CPU and I/O. Given a PLAN, talk about $COST_{CPU}(PLAN)$ and $COST_{I/O}(PLAN)$.

But two plans $PLAN_1$ and $PLAN_2$ can have incomparable resource use with this pair of measures. See Figure 9.1. (pg. 535)

	$COST_{CPU}(PLAN)$	$COST_{I/O}(PLAN)$
$PLAN_1$	9.2 CPU seconds	103 reads
$PLAN_2$	1.7 CPU seconds	890 reads

*** Which is cheaper? Depends on which resources have a bottleneck.
DB2 takes a weighted sum of these for the total cost, COST(PLAN):

$$COST(PLAN) = W_1 \cdot COST_{CPU}(PLAN) + W_2 \cdot COST_{I/O}(PLAN)$$

A DBA should try to avoid bottlenecks by understanding the *WORKLOAD* for a system. Estimated total CPU and I/O for all users at peak work time.

Figure out various queries and rate of submission per second. Q_k, $RATE(Q_k)$ as in Figure 9.2. (pg. 537.)

Query Type	RATE(Query) in Submissions/second
Q1	40.0
Q2	20.0

Figure 9.2 Simple Workload with two Queries

Then work out $COST_{CPU}(Q_k)$ and $COST_{CPU}(Q_k)$ for plan that database system will choose. Now can figure out total average CPU or I/O needs per second:

$$RATE(CPU) = \sum_K \; RATE(Q_k) \cdot COST_{CPU}(Q_k)$$

Similarly for I/O. Tells us how many disks to buy, how powerful a CPU.
(Costs go up approximately linearly with CPU power within small ranges).

Ideally, the weights W_1 and W_2 used to calculate COST(PLAN) from I/O and CPU costs will reflect actual costs of equipment

Good DBA tries to estimate workload in advance to make purchases, have equipment ready for a new application.

*** Note that one other major expense is response time. Could have poor response time (1-2 minutes) even on lightly loaded inexpensive system.

This is because many queries need to perform a LOT of I/O, and some commercial database systems have no parallelism: all I/Os in sequence.

But long response time also has a cost, in that it wastes user time. Pay workers for wasted time. Employees quit from frustration and others must be trained. Vendors are trying hard to reduce response times.

Usually there exist parallel versions of these systems as well; worth it if extremely heavy I/O and response time is a problem, while number of queries running at once is small compared to number of disks.

Note that in plans that follow, we will not try to estimate CPU. Too hard. Assume choose plan with best I/O and try to estimate that. Often total cost is proportional to I/O since each I/O entails extra CPU cost as well.

Statistics. Need to gather, put in System tables. System does not automatically gather them with table load, index create, updates to table.

In DB2, use Utility called RUNSTATS. Fig. 9.3, pg. 538.

```
RUNSTATS ON TABLE username.tablename
  [WITH DISTRIBUTION [AND DETAILED] {INDEXES ALL | INDEX indexname}]
   [other clauses not covered or deferred]
```

e.g.: runstats on table poneil.customers;

System learns how many rows, how many data pages, stuff about indexes, etc., placed in catalog tables.

ORACLE uses ANALYZE command. Fig. 9.4, pg. 538.

```
ANALYZE {INDEX | TABLE | CLUSTER}
  [schema.] {indexname | tablename | clustername}
  {COMPUTE STATISTICS | other alternatives not covered}
  {FOR TABLE | FOR ALL [INDEXED] COLUMNS [SIZE n]
   | other alternatives not covered}
```

Will see how statistics kept in catalog tables in DB2 a bit later.

Retrieving the Query Plans. In DB2 and ORACLE, perform SQL statement. In DB2:

```
  EXPLAIN PLAN [SET QUERYNO = n] [SET QUERYTAG = 'string'] FOR
    explainable-sql-statement;
```

For example:

```
  explain plan set queryno = 1000 for
    select * from customers
      where city = 'Boston' and discnt between 12 and 14;
```

The Explain Plan statement puts rows in a "plan_table" to represent individual procedural steps in a query plan. Can get rows back by:

```
select * from plan_table where queryno = 1000;
```

Recall that a Query Plan is a sequence of procedural access steps that carry out a program to answer the query. Steps are peculiar to the DBMS.

From one DBMS to another, difference in steps used is like difference between programming languages. Can't learn two languages at once.

We will stick to a specific DBMS in what follows, MVS DB2, so we can end up with an informative benchmark we had in the first edition.

But we will have occasional references to ORACLE, DB2 UDB. Here is the ORACLE Explain Plan syntax.

```
EXPLAIN PLAN [SET STATEMENT_ID = 'text-identifier'] [INTO
  [schema.]tablename]
    FOR explainable-sql-statement;
```

This inserts a sequence of statements into a user created DB2/ORACLE table known as PLAN_TABLE one row for each access step. To learn more about this, see ORACLE8 documentation named in text.

Need to understand what basic procedural access steps ARE in the particular product you're working with.

The set of steps allowed is the "bag of tricks" the query optimizer can use. Think of these procedural steps as the "instructions" a compiler can use to create "object code" in compiling a higher-level request.

A system that has a smaller bag of tricks is likely to have less efficient access plans for some queries.

MVS DB2 (and the architecturally allied DB2 UDB) have a wide range of tricks, but not bitmap indexing or hashing capability.

Still, very nice capabilities for range search queries, and probably the most sophisticated query optimizer.

Basic procedrual steps covered in the next few Sections (thumbnail):

Table Scan	Look through all rows of table
Unique Index Scan	Retrieve row through unique index
Unclustered Matching Index Scan	Retrieve multiple rows through a non-unique index, rows not same order
Clustered Matching Index Scan	Retrieve multiple rows through a non-unique clustered index

Index-Only Scan	Query answered in index, not rows

Note that the steps we have listed access all the rows restricted by the WHERE clause in some single table query.

Need two tables in the FROM clause to require two steps of this kind and Joins come later. A multi-step plan for a single table query will also be covered later.

Such a multi-step plan on a single table is one that combines multiple indexes to retrieve data. Up to then, only one index per table can be used.

9.2 Table Space Scans and I/O

Single step. The plan table (plan_table) will have a column ACCESSTYPE with value R (ACCESSTYPE = R for short).

Example 9.2.1. Table Space Scan Step. Look through all rows in table to answer query, maybe because there is no index that will help.

Assume in DB2 an employees table with 200,000 rows, each row of 200 bytes, each 4 KByte page just 70% full. Thus 2800 usable pages, 14 rows/pg. Need CEIL(200,000/14) = 14,286 pages.

Consider the query:

```
select eid, ename from employees where socsecno = 113353179;
```

If there is no index on socsecno, only way to answer query is by reading in all rows of table. (Stupid not to have an index if this query occurs with any frequency at all!)

In Table Space Scan, might not stop when find proper row, since statistics for a non-indexed column might not know socsecno is unique.

Therefore have to read all pages in table in from disk, and $COST_{I/O}$(PLAN)
= 14286 I/Os. Does this mean 14286 random I/Os? Maybe not.

But if we assume random I/O, then at 80 I/Os per second, need about 14286/80 = 178.6 seconds, a bit under three minutes. This dominates CPU by a large factor and would predict elapsed time quite well.

Homework 4, non-dotted Exercises through Exercise 9.9. This is due when we finish Section 9.6..

Assumptions about I/O

We are now going to talk about I/O assumptions a bit. First, there might be PARALLELISM in performing random I/Os from disk.

The pages of a table might be *striped* across several different disks, with the database system making requests in parallel for a single query to keep all the disk arms busy. See Figure 9.5, pg. 542

When the first edition came out, it was rare for most DBMS systems to make multiple requests at once (a form of parallelism), now it's common.

DB2 has a special form of sequential prefetch now where it stripes 32 pages at a time on multiple disks, requests them all at once.

While parallelism speeds up TOTAL I/O per second (expecially if there's ony one user process running), it doesn't really save any RESOURCE COST.

If it takes 12.5 ms (0.0125 seconds) to do a random I/O, doesn't save resources to do 10 random I/Os at once on 10 different disks.

Still have to make all the same disk arm movements, cost to rent the disk arms is the same if there is parallelism: just spend more per second.

Will speed things up if there are few queries running, fewer than the number of disks, and there is extra CPU not utilized. Then can use more of the disk arms and CPUs with this sort of parallelism.

Parallelism shows up best when there is only one query running!

But if there are lots of queries compared to number of disks and accessed pages are randomly placed on disks, probably keep all disk arms busy already.

But there's another factor operating. Two disk pages that are close to each other on one disk can be read faster because there's a shorter seek time.

Recall that the system tries to make extents contiguous on disk, so I/Os in sequence are faster. Thus, a table that is made up of a sequence of (mainly) contiguous pages, one after another within a track, will take much less time to read in.

In fact it seems we should be able to read in successive pages at full transfer speed would take about .00125 secs per page.

Used to be that by the time the disk controller has read in the page to a memory buffer and looked to see what the next page request is, the page immediately following has already passed by under the head.

But now with multiple requests to the disk outstanding, we really COULD get the disk arm to read in the next disk page in sequence without a miss.

Another factor supports this speedup: the typical disk controller buffers an entire track in it's memory whenever a disk page is requested.

Reads in whole track containing the disk page, returns the page requested, then if later request is for page in track doesn't have to access disk again.

So when we're reading in pages one after another on disk, it's like we're reading from the disk an entire track at a time.

I/O is about TEN TIMES faster for disk pages in sequence compared to randomly place I/O. (Accurate enough for rule of thumb.)

PUT ON BOARD: We can do 800 I/Os per second when pages in sequence (S) instead of 80 for randomly placed pages (R). Sequential I/O takes 0.00125 secs instead of 0.0125 secs for random I/O.

DB2 Sequential Prefetch makes this possible even if turn off buffering on disk (which actually hurts performance of random I/O, since reads whole track it doesn't need: adds 0.008 sec to random I/O of 0.0125 sec)

IBM puts a lot of effort into making I/O requests sequentially in a query plan to gain this I/O advantage!

Example 9.2.2. Table Space Scan with Sequential Advantage. The 14286R of Example 9.2.1 becomes 14286S (S for Sequential Prefetch I/O instead of Random I/O). And 14286S requires 14286/800 = 17.86 seconds instead of the 178.6 seconds of 142286R. Note that this is a REAL COST SAVINGS, that we are actually using the disk arm for a smaller period. Striping reduces elapsed time but not COST.

Cover idea of List Prefetch. 32 pages, not in perfect sequence, but relatively close together. Difficult to predict time.

We use the rule of thumb that List Prefetch reads in 200 pages per second.

See Figure 9.10, page 546, for table.

Plan table row for an access step will have PREFETCH = S for sequential prefetch, PREFETCH = L for list prefetch, PREFETCH = blank if random I/O. See Figure 9.10. And of course ACCESSTYPE = R when really Random I/O.

Note that sequential prefetch is just becoming available on UNIX database systems. Often just put a lot of requests out in parallel and depend on smart I/O system to use arm efficiently

14.

Exam 1.

15.

9.3 Simple Indexed Access in DB2.

Index helps efficiency of query plan. There is a gread deal of complexity here. Remember, we are not yet covering queries with joins: only one table in FROM clause and NO subquery in WHERE clause.

Examples with tables: T1, T2, . . , columns C1, C2, C3, . .

Example 9.3.1. Assume index C1X exists on column C1 of table T1 (always a B-tree secondary index in DB2). Consider:

```
select * from T1 where C1 = 10;
```

This is a *Matching Index Scan*. In plan table: ACCESSTYPE = I, ACCESSNAME = C1X, MATCHCOLS = 1. (MATCHCOLS might be >1 in multiple column index.)

Perform matching index scan by walking down B-tree to LEFTMOST entry of C1X with C1 = 10. Retrieve row pointed to.

Loop through entries at leaf level from left to right until run out of entries with C1 = 10. For each such entry, retrieve row pointed to. No assumption about clustering or non-clustering of rows here.

In Example 9.3.2, assume other restrictions in WHERE clause, but matching index scan used on C1X. Then other restrictions are validated as rows are accessed (row is ***qualified:*** look at row, check if matches restrictions).

Not all predicates are *indexable*. In DB2, indexable predicate is one that can be used in a *matching index scan*, i.e. a lookup that uses a contiguous section of an index. Covered in full in Section 9.5.

For example, looking up words in the dictionary that start with the letters 'pre' is a matching index scan. Looking up words ending with 'tion' is not.

DB2 considers the predicate C1 <> 10 to be non-indexable. It is not impossible that an index will be used in a query with this predicate:

```
select * from T1 where C1 <> 10;
```

But the statistics usually weigh against it's use and so the query will be performed by a table space scan. More on indexable predicates later.

OK, now what about query:

```
select * from T1 where C1 = 10 and C2 between 100 and 200
    and C3 like 'A%';
```

These three predicates are all indexable. If have only C1X, will be like previous example with retrieved rows restricted by tests on other two predicates.

If have index combinx, created by:

```
create index combinx on T1 (C1, C2, C3) . . .
```

Will be able to limit (filter) RIDs of rows to retrieve much more completely before going to data. Like books in a card catalog, looking up

```
authorlname = 'James' (c1 = 10) and authorfname between 'H' and 'K'
    and title begins with letter 'A'
```

Finally, we will cover the question of how to filter the RIDs of rows to retrieve if we have three indexes, C1X, C2X, and C3X. This is not simple.

See how to do this by taking out cards for each index, ordering by RID, then merge-intersecting.

It is an interesting query optimization problem whether this is worth it.

OK, now some examples of simple index scans.

Example 9.3.3. Index Scan Step, Unique Match. Continuing with Example 9.2.1, employees table with 200,000 rows of 200 bytes and pctfree = 30, so 14 rows/pg and CEIL(200,000/14) =14,286 data pages. Assume in index on eid, also have pctfree = 30, and eid||RID takes up 10 bytes, so 280 entries per pg, and CEIL(200,000/280) = 715 leaf level pages. Next level up CEIL(715/280) = 3. Root next level up. Write on board:

employees table: 14,286 data pages
index on eid, eidx: 715 leaf nodes, 3 level 2 nodes, 1 root node.

Now query: select ename from employees where eid = '12901A';

Root, on level 2 node, 1 leaf node, 1 data page. Seems like 4R. But what about buffered pages? **Five minute rule** says should purchase enough memory so pages referenced more frequently than about once every 120 seconds (popular pages) should stay in memory. Assume we have done this. If workload assumes 1 query per second of this on ename with eid = predicate (no others on this table), then leaf nodes and data pages not buffered, but upper nodes of eidx are. So really 2R is cost of query.

This Query Plan is a single step, with ACCESSTYPE = I, ACCESSNAME = eidx, MATCHCOLS = 1.

16.

OK now we introduce a new table called prospects. Based on direct mail applications (junk mail). People fill out warrenty cards, name hobbies, salary range, address, etc.

50M rows of 400 bytes each. FULL data pages (pctfree = 0) and on all indexes: 10 rows on 4 KByte page, so 5M data pages.

prospects table: 5M data pages

Now:create index addrx on prospects (zipcode, city, straddr) cluster . . .;

zipcode is integer or 4 bytes, city requires 12 bytes, straddr 20 bytes, RID 4 bytes, and assume NO duplicate values so no compression.

Thus each entry requires 40 bytes, and we can fit 100 on a 4 KByte page. With 50M total entries, that means 500,000 leaf pages. 5000 directory nodes at level 2. 50 level 3 node pages. Then root page. Four levels.

Also assume a nonclustering hobbyx index on hobbies, 100 distinct hobbies (. . . cards, chess, coin collecting, . . .). We say CARD(hobby) = 100.

(Like we say CARD(zipcode) = 100,000. Not all possible integer zipcodes can be used, but for simplicity say they are.)

Duplicate compression on hobbyx, each key (8 bytes?) amortized over 255 RIDS (more?), so can fit 984 RIDs (or more) per 4 KByte page, call it 1000.

Thus 1000 entries per leaf page. With 50M entries, have 50,000 leaf pages. Then 50 nodes at level 2. Then root.

index on eid, eidx: 715 leaf nodes, 3 level 2 nodes, 1 root node.

prospects table	**addrx index**	**hobbyx index**
50,000,000 rows	500,000 leaf pages	50,000 leaf pages
5,000,000 data pages	5,000 level 3 nodes	151 level 2 nodes
(10 rows per page)	50 level 2 nodes	1 root node
	1 root node	(1000 entries/leaf)
	CARD(zipcode)= 100,000	CARD(hobby)=100

Figure 9.12. Some statistics for the prospects table, page 552

Example 9.3.4. Matching Index Scan Step, Unclustered Match. Consider the following query:

select name, straddr from prospects where hobby = 'chess';

Query optimizer assumes each of 100 hobbies equally likely (knows there are 100 from RUNSTATS), so restriction cuts 50M rows down to 500,000.

Walk down hobby index (2R for directory nodes) and across 500,000 entries (1000 per page so 500 leaf pages, sequential prefeth so 500S).

For every entry, read in row -- non clustered so all random choices out of 5M data pages, 500,000 distinct I/Os (not in order, so R), 500,000R.

Total I/O is 500S + 500,002R. Time is 500/800 + 500,002/80, about 500,000/80 = 6250 seconds. Or about 1.75 hours (2 hrs = 7200 secs).

Really only picking up 500,000 distinct pages, will lie on less than 500,000 pages (out of 5 M). Would this mean less than 500,000 R because buffering keeps some pages around for double/triple hits?

VERY TRIVIAL EFFECT! Hours of access, 120 seconds pages stay in buffer.

Can generally assume that upper level index pages are buffer resident (skip 2R) but leaf level pages and maybe one level up are not. Should calculate index time and can then ignore it if insignificant.

If we used a table space scan for Example 9.3.4, qualifying rows to ensure hobby = 'chess, how would time compare to what we just calculated?

Simple: 5M pages using sequential prefetch, 5,000,000/800 = 625 seconds. (Yes, CPU is still ignored — in fact is relatively insignificant.)

<u>But this is the same elapsed time as for indexed access of 1/100 of rows!!</u>

Yes, surprising. But 10 rows per page so about 1/10 as many pages hit, and S is 10 times as fast as R.

Query optimizer compares these two approaches and chooses the faster one. Would probably select Table Space Scan here But minor variation in CARD(hobby) could make either plan a better choice.

Example 9.3.5. Matching Index Scan Step, Clustered Match. Consider the following query:

```
select name, straddr from prospects
    where zipcode between 02159 and 03158;
```

Recall CARD(zipcode) = 100,000. Range of zipcodes is 1000. Therefore, cut number of rows down by a factor of 1/100. SAME AS 9.3.4.

Bigger index entries. Walk down to leaf level and walk across 1/100 of leaf level: 500,000 leaf pages, so 5000 pages traversed. I/O of 5000S.

And data is clustered by index, so walk across 1/100 of 5M data pages, 50,000 data pages, and they're in sequence on disk, so 50,000S.

Compared to Nonmatching index scan of Example 9.3.4, walk across 1/10 as many pages and do it with S I/O instead of R. Ignore directory walk.

Then I/O cost is 55,000S, with elapsed time 55,000/500 = 137.5 seconds, a bit over 2 minutes, compared with 1.75 hrs for unclustered index scan.

The difference between Examples 9.3.4 and 9.3.5 doesn't show up in the PLAN table. Have to look at ACCESSNAME = addrx and note that this index is clustered, (clusterratio) whereas ACCESSNAME = hobbyx is not.

(1) Clusterratio determines if index still clustered in case rows exist that don't follow clustering rule. (Inserted when no space left on page.)

(2) Note that entries in addrx are 40 bytes, rows of prospects are 400 bytes. Seems natural that 5000S for index, 50,000S for rows.

Properties of index:
1. Index has directory structure, can retrieve range of values
2. Index entries are ALWAYS clustered by values
3. Index entries are smaller than the rows.

Example 9.3.6. **Concatenated Index, Index-Only Scan**. Assume (just for this example) a new index, naddrx:

```
create index naddrx on prospects (zipcode, city, straddr, name)
    . . . cluster . . .;
```

Now same query as before:

```
select name, straddr from prospects where zipcode
    between 02159 and  03158;
```

Can be answered in INDEX ONLY (because find range of zipcodes and read name and straddr off components of index: Show components:

naddrx keyvalue: zipcodeval.cityval.straddrval.nameval

This is called an Index Only scan, and with EXPLAIN plan table gets new column: INDEXONLY = Y (ACCESSTYPE = I, ACCESSNAME = naddrx). Previous plans had INDEXONLY = N.
(All these columns always reported; I just mention them when relevant.)

Time? Assume naddrx takes 60 bytes instead of 40 bytes, then amount read in index, instead of 5000S is 7500S, elapsed time 7500/800 = 9.4 seconds. Compare to 62.5 seconds with Example 9.3.5.

Valuable idea, Index Only. Select count(*) from . . . is always index only if index can do in a single step at all, since count entries.

But can't build index on the spur of the moment. If don't have needed one already, out of luck. E.g., consider query:

```
select name, straddr, age from prospects where zipcode
    between 02159 and 02258;
```

Now naddrx doesn't have all needed components. Out of luck.

If try to foresee all needed components in an index, essentially duplicating the rows, and lose performance boost from size.

Indexes cost something. Disk media cost (not commonly crucial). With inserts or updates of indexed rows, lot of extra I/O (not common).

With read-only, like prospects table, load time increases. Still often have every col of a read-only table indexed.

Chapter 9.4. Filter Factors and Statistics

Recall, estimated probability that a random row made some predicate true. By statistics, determine the fraction (FF(pred)) of rows retrieved.

E.g., hobby column has 100 values. Generally assume uniform distribution, and get: FF(hobby = const) = 1/100 = .01.

And zipcode column has 100,000 values, FF(zipcode = const) = 1/100,000. FF(zipcode between 02159 and 03158) = 1000·(1/100,000) = 1/100.

How does the DB2 query optimizer make these estimates?

DB2 statistics.

See Figure 9.13, pg. 558. After use RUNSTATS, these statistics are up to date. (Next pg. of these notes) Other statistics as well, not covered.

DON'T WRITE THIS ON BOARD -- SEE IN BOOK

Catalog Name	Statistic Name	Default Value	Description

SYSTABLES	CARD	10,000	Number of rows in the table
	NPAGES	CEIL(1+CARD/20)	Number of data pages containing rows
SYSCOLUMNS	COLCARD	25	Number of distinct values in this column
	HIGH2KEY	n/a	Second highest value in this column
	LOW2KEY	n/a	Second lowest value in this column
SYSINDEXES	NLEVELS	0	Number of Levels of the Index B-tree
	NLEAF	CARD/300	Number of leaf pages in the Index B-tree
	FIRSTKEY-CARD	25	Number of distinct values in the first column, C1, of this key
	FULLKEY-CARD	25	Number of distinct values in the full key, all components: e.g. C1.C2.C3
	CLUSTER-RATIO	0% if CLUSTERED = 'N' 95% if CLUSTERED = 'Y'	Percentage of rows of the table that are clustered by these index values

Figure 9.13. Some Statistics gathered by RUNSTATS used for access plan determination

Statistics gathered into DB2 Catalog Tables named. Assume that index might be composite, (C1, C2, C3)

Go over table. CARD, NPAGES for table. For column, COLCARD, HIGH2KEY, LOW2KEY. For Indexes, NLEVELS, NLEAF, FIRSTKEYCARD, FULLKEYCARD, CLUSTERRATIO. E.g., from Figure 9.12, statistics for prospects table (given on pp. 552-3). Write these on Board.

SYSTABLES

NAME	CARD	NPAGES
. . .	. . .	. . .
prospects	50,000,000	5,000,000
. . .	. . .	. . .

SYSCOLUMNS

NAME	TBNAME	COLCARD	HIGH2KEY	LOW2KEY
. . .	. . .	. . .	. . .	. . .
hobby	prospects	100	Wines	Bicycling
zipcode	prospects	100000	99998	00001
. . .	. . .	. . .	. . .	. . .

SYSINDEXES

NAME	TBNAME	NLEVELS	NLEAF	FIRSTKEY CARD	FULLKEY CARD	CLUSTER RATIO
. . .	. . .	. . .	. . .	. . .	. . .	. . .
addrx	prospects	4	500,000	100,000	50,000,000	100
hobbyx	prospects	3	50,000	100	100	0
. . .	. . .	. . .	. . .	. . .	. . .	. . .

CLUSTERRATIO is a measure of how well the clustering property holds for an index. *With 80 or more*, will use Sequential Prefetch in retrieving rows.

Indexable Predicates in DB2 and their Filter Factors

Look at Figure 9.14, pg. 560. QOPT guesses at Filter Factor. Product rule assumes independent distributions of columns. Still no subquery predicate.

Predicate Type	Filter Factor	Notes
Col = const	1/COLCARD	"Col <> const" same as "not (Col = const)"
Col □ const	Interpolation formula	"□" is any comparison predicate other than equality; an example follows
Col < const or Col <= const	$\frac{(\text{const} - \text{LOW2KEY})}{(\text{HIGH2KEY}-\text{LOW2KEY})}$	LOW2KEY and HIGH2KEY are estimates for extreme points of the range of Col values
Col between const1 and const2	$\frac{(\text{const2} - \text{const1})}{(\text{HIGH2KEY}-\text{LOW2KEY})}$	"Col not between const1 and const2" same as "not (Col between const1 and const2)"
Col in list	(list size)/COLCARD	"Col not in list" same as "not (Col in list)"
Col is null	1/COLCARD	"Col is not null" same as "not(Col is null)"
Col like 'pattern'	Interpolation Formula	Based on the alphabet
Pred1 and Pred2	FF(Pred1)·FF(Pred2)	As in probability
Pred1 or Pred2	FF(Pred1)+FF(Pred2) -FF(Pred1)·FF(Pred2)	As in probability
not Pred1	1 - FF(Pred1)	As in probability

Figure 9.20. Filter Factor formulas for various predicate types 17.

Matching Index Scans with Composite Indexes

(Finish -> 9.6 19, homework due 20 (Wed, April 12)

Assume new index mailx:

create index mailx on prospects (zipcode, hobby, income, age);

NOT clustered. column income has 10 distinct values, age has 50.

FULLKEYCARD(mailx) could be as much as

CARD(zipcode)·CARD(hobby)·CARD(income)·CARD(age) =
100,000·100·10·50 = 1,000,000,000.

Can't be that much, only 50,000,000 rows, so assume FULLKEYCARD is 50,000,000, with no duplicate rows. (Actually, 50M darts in 5G slots. About 1/100 of slots hit, so only about 1% duplicate keyvalues.)

Entries for mailx have length: 4 (integer zipcode) + 8 (hobby) + 2 (income) + 2 (age) + 4 (RID) = 20 bytes. So 200 entries per page.

NLEAF = 50,000,000/100 = 500,000 pages. Next level up has 5,000 nodes, next level 50, next is root, so NLEVELS = 4.

SYSINDEXES

NAME	TBNAME	NLEVELS	NLEAF	FIRSTKEY CARD	FULLKEY CARD	CLUSTER RATIO
. . .	. . .	. . .	. . .	. . .	. . .	. . .
mailx	prospects	4	250,000	100,000	50,000,000	0
. . .	. . .	. . .	. . .	. . .	. . .	. . .

Example 9.5.1. Concatenated Index, Matching Index Scan.

```
select name straddr from prospects where
    zipcode = 02159 and hobby = 'chess' and income = 10;
```

Matching Index Scan here means that the three predicates in the WHERE clause match the INITIAL column in concatenated mailx index.

Argue that *matching* means all entries to be retrieved are contiguous in the index.

Full filter factor for three predicates given is 1/100,000 ·1/100·1/10 = 1/100M, with 50M rows, so only 0.5 rows selected. 0.5R

Interpret this probabilistically, and expected time for retrieving rows is only 1/80 second. Have to add index I/O of course. 2R, .05 sec.

Example 9.5.2. **Concatenated Index, Matching index scan**.

```
select name straddr from prospects
    where  zipcode between 02159 and 04158
    and hobby = 'chess' and income = 10;
```

Now, important. Not one contiguous interval in index. There is one interval for: z = 02159 and h = 'c' and inc = 10, and then another for z = 02160 and h = 'c' and inc = 10, and . . . But there is stuff between them.

Analogy in telephone directory: last name between 'Sma' and 'Smz' and first name 'John'. Lot of directory to look through, not all matches.

Query optimizer here traverses from leftmost z = 02159 to rightmost z = 04158 and uses h = 'c' and inc = 10 as screening predicates.

We say the first predicate is a MATCHING predicate (used for cutting down interval of index considered) and other two are SCREENING predicates.

(This MATCHING predicate is what we mean by Matching Index Scan.)

So index traversed is: (2000/100,000) (filter factor) of 500,000 leaf pages, = 10,000 leaf pages. Query optimizer actually calculates FF as

```
(04158-02159)(HIGH2KEY-LOW2KEY) = 2000/(99998-00001)
= 200/99997 or approximately 2000/100,000 = 1/50
```

Have to look through 1/50·NLEAF = 5000 pages, I/O cost is 5,000S with elapsed time: 5,000/400 = 12.5 seconds.

How many rows retrieved? (1/50)(1/100)(1/10) = (1/50,000) with 50M rows, so 1000 rows retrieved. Sequential, ?

No. 1000R, with elapsed time 1000/40 = 25 seconds. Total elapsed time is 37.5 secs.

Example 9.5.3. Concatenated Index, Non-Matching Index Scan.

```
select name straddr from prospects where
    hobby = 'chess' and income = 10 and age = 40;
```

Like saying First name = 'John' and City = 'Waltham' and street = 'Main'.
Have to look through whole index, no matching column, only screening predicates.

Still get small number of rows back, but have to look through whole index. 250,000S. Elapsed time 250,000/400 = 625 seconds, about 10.5 minutes.

Number of rows retrieved: (1/100)(1/10)(1/50)(50,000,000) = 1000. 1000R = 25 seconds.

In PLAN TABLE, for Example 9.5.2, have ACCESSTYPE = I, ACCESSNAME = mailx, MATCHCOLS = 1; In Example 9.5.3, have MATCHCOLS = 0.

Definition 9.5.1. Matching Index Scan. A plan to execute a query where at least one indexable predicate must match the first column of an index (known as matching predicate, matching index). May be more.

What is an indexable predicate? Equal match predicate is one: Col = const See Definition 9.5.3. Pg. 565

Say have index C1234X on table T, composite index on columns (C1, C2, C3, C4). Consider following compound predicates.

C1 = 10 and C2 = 5 and C3 = 20 and C4 = 25 (matches all columns)

C2 = 5 and C3 = 20 and C1 = 10 (matches first three: needn't be in order)

C2 = 5 and C4 = 22 and C1 = 10 and C6 = 35 (matches first two)

C2 = 5 and C3 = 20 and C4 = 25 (NOT a matching index scan)

Screening predicates are ones that match non-leading columns in index. E.g., in first example all are matching, in second all are matching, in third, two are matching, one is screening, and one is not in index, in fourth all three are screening.

Finish through Section 9.6 by next . Homework due next (Wednesday after Patriot's day). NEXT homework is rest of Chapter 9 non-dotted exercises if you want to work ahead.

Definition 9.5.2. Basic Rules of Matching Predicates

(1) A *matching predicate* must be an *indexable* predicate. See pg. 560, Table 9.14 for a list of indexable predicates.

(2) Matching predicates must match successive columns, C1, C2, . . . of an index. Procedure: Look at index columns from left-to right. If find a matching predicate for this column, then this is a matching column. As soon as column fails to be matching terminate the search.

Idea is that sequence of matching predicates cuts down index search to smaller contiguous range. (One exception: In-list predicate, covered shortly).

(3) A non-matching predicate in an index scan can still be a screening predicate.

Look at rule (1) again. This is actually a kind of circular definition. For a predicate to be *matching* it must be *indexable* and:

Definition 9.5.3: An *indexable* predicate is one that can be used to match a column in a matching index scan.

Calling such a predicate *indexable* is confusing. Even if a predicate is not indexable, the predicate can use the index for screening.

Would be much better to call such predicates *matchable*, but this nomenclature is embedded in the field for now.

When K leading columns of index C1234X are matching for a query, EXPLAIN into plan table, get ACCESSTYPE = I, ACCESSNAME = C1234X, MATCHCOLS = K. When non-matching index scan, MATCHCOLS = 0.

Recall Indexable Predicates in Figure 9.14, pg. 560, and relate to telephone directory. Does the predicate give you a contiguous range?

Col > const? between? In-list is special. like 'pattern' with no leading wild card? Col1 like Col2? (same middle name as street name) Predicate and? Predicate or? Predicate not?

OK, a few more rules on how to determine matching predicates. Page 566, **Def 9.5.4**. Match cols. in index left to right until run out of predicates. But
(3) Stop at first range predicate (between, <, >, <=, >=, like).
(4) At most one In-list predicate.

In-list is special because it is considered a sequence of equal matching predicates that the query optimizer agrees to bridge in the access plan

C1 in (6, 8, 10) and C2 = 5 and C3 = 20 is like C1 = 6 and . . .;

Plan for C1 = 8 and . . .; then C1 = 10 and . . .; etc.

But the following has only two matching columns since only one in-list can be used.

C1 in (6, 8, 10) and C2 = 5 and C3 in (20, 30, 40)

When In-list is used, say ACCESSTYPE = 'N'.

Example 9.5.4. In following examples, have indexes C1234X, C56X, Unique index C7X.

(1) select C1, C5, C8 from T where C1 = 5 and C2 = 7 and C3 <> 9;
ACCESSTYPE = I, ACCESSNAME = C1234X, MATCHCOLS = 2

(2) select C1, C5, C8 from T where C1 = 5 and C2 >= 7 and C3 = 9;
C3 predicate is indexable but stop at range predicate.
ACCESSTYPE = I, ACCESSNAME = C1234X, MATCHCOLS = 2

(3) select C1, C5, C8 from T
where C1 = 5 and C2 = 7 and C5 = 8 and C6 = 13;
We ignore for now the possibility of combining multiple indexes
Note, we don't know what QOPT will choose until we see plan table row
ACCESSTYPE = I, ACCESSNAME = C56X, MATCHCOLS = 2

(4) select C1, C4 from T
where C1 = 10 and C2 in (5, 6) and (C3 = 10 or C4 = 11);
C1 and C2 predicates are matching. The "or" operator doesn't give indexable predicate, but would be used as screening predicate (not mentioned in plan table, but all predicates used to filter, ones that exist in index certainly would be used when possible). ACCESSTYPE = 'N', ACCESSNAME = C1234X, MATCHCOLS = 2 Also: INDEXONLY = 'Y'

(5) select C1, C5, C8 from T where C1 = 5 and C2 = 7 and C7 = 101;
ACCESSTYPE = I, ACCESSNAME = C7X, MATCHCOLS = 1
(Because unique match, but nothing said about this in plan table)

(6) select C1, C5, C8 from T
where C2 = 7 and C3 = 10 and C4 = 12 and C5 = 16;
Will see can't be multiple index. Either non-matching in C1234X or matching on C56X. ACCESSTYPE = I, ACCESSNAME = C1234X, MATCHCOLS = 2

Some Special Predicates

Pattern Match Search. Leading wildcards not indexable.

"C1 like 'pattern'" with a leading '%' in pattern (or leading '_'?) like looking in dictionary for all word ending in 'tion'. Non-matching scan (non-indexable predicate).

Exist dictionaries that index by backward spelling, and DBA can use this trick: look for word with match: Backwards = 'noit%'

Expressions. Use in predicate makes not indexable.

```
select * from T where 2*C1 <= 56;
```

DB2 doesn't do algebra. You can re-write: where C1 <= 28;

Never indexable if two different columns are used in predicate: C1 = C2.

One-Fetch Access. Select min/max . . .

```
select min(C1) from T;
```

Look at index C1234X, leftmost value, read off value of C1. Say have index C12D3X on T (C1, C2 DESC, C3). Each of following qs has one-fetch access.

```
select min(C1) from T where C1 > 5;  (NOT obviously 5)
select min(C1) from T where C1 between 5 and 6;
select max(C2) from T where C1 = 5;
select max(C2) from T where C1 = 5 and C2 < 30;
select min(C3) from T where C1 = 6 and C2 = 20 and C3 between 6 and 9;
```

18.

9.6 Multiple Index Access

Assume index C1X on (C1), C2X on (C2), C345X on (C3, C4, C5), query:

(9.6.1) select * from T where C1 = 20 and C2 = 5 and C3 = 11;

By what we've seen up to now, would have to choose one of these indexes. Then only one of these predicates could be matched.

Other two predicates are not even screening predicates. Don't appear in index, so must retrieve rows and validate predicates from that.

BUT if only use FF of one predicate, terrible inefficiency may occur. Say T has 100,000,000 rows, FF for each of these predicates is 1/100.

Then after applying one predicate, get 1,000,000 rows retrieved. If none of indexes are clustered, will take elapsed time of: 1,000,000/40 = 25,000 seconds, or nearly seven hours.

If somehow we could combine the filter factors, get composite filter factor of (1/100)(1/100)(1/100) = 1/1,000,000, only retrieve 100 rows.

Trick. Multiple index access. For each predicate, matching on different index, extract RID list. Sort it by RID value.

(Draw a picture - three extracts from leaf of wedge into lists)

Intersect (AND) all RID lists (Picture) (easy in sorted order). Result is list of RIDs for answer rows. Use List prefetch to read in pages. (Picture)

This is our first multiple step access plan.
See Figure 9.15. Put on board.

V Note

TNAME	ACCESSTYPE	MATCHCOLS	ACCESSNAME	PREFETCH	MIXOPSEQ
T	M	0		L	0
T	MX	1	C1X	S	1
T	MX	1	C2X	S	2
T	MX	1	C345X	S	3
T	MI	0			4
T	MI	0			5

Figure 9.15 Plan table rows of a Multiple Index Access plan for Query (9.6.1)

Row M, start Multiple index access. Happens at the end, after Intersect Diagram steps in picture. RID lists placed in RID Pool in memory.

Note Plan acts like reverse polish calculation: push RID lists as created by MX step; with MI, pop two and intersect , push result back on stack.

MX steps require reads from index. MI steps require no I/O: already in memory, memory area known as RID Pool.

Final row access uses List prefetch, program disk arms for most efficient path to bring in 32 page blocks that are not contiguous, but sequentially listed. Most efficient disk arm movements.

(Remember that an RID consists of (page_number, slot_number), so if RIDs are placed in ascending order, can just read off successive page numbers).

Speed of I/O depends on distance apart. Rule of thumb for Random I/O is 40/sec, Seq. prefetch 400/sec, List prefetch is 100/sec.

Have mentioned ACCESSTYPEs = M, MX, MI. One other type for multiple index access, known as MU (to take the Union (OR) of two RID lists). E.g.

(9.6.2) select * from T where C1 = 20 and (C2 = 5 or C3 = 11);

Here is Query Plan (Figure 9.16):

TNAME	ACCESSTYPE	MATCHCOLS	ACCESSNAME	PREFETCH	MIXOPSEQ
T	M	0		L	0
T	MX	1	C1X	S	1
T	MX	1	C2X	S	2
T	MX	1	C345X	S	3
T	MU	0			4
T	MI	0			5

Figure 9.16 Plan table rows of a Multiple Index Access plan for Query (9.6.2)

Actually, query optimizer wouldn't generate three lists in a row if avoidable. Tries not to have > 2 in existence (not always possible).

Figure 9.17. 1. MX C2X, 2. MX C345X, 3. MU, 4. MX C1X, 5. MI

Example 9.6.1. **Multiple index access**. prospects table, addrx index, hobbyx index (see Figure 9.11, p 544), index on age (agex) and income (incomex). What is NLEAF for these two, ? (Like hobbyx: 50,000.)

```
select name, straddr from prospects
    where zipcode = 02159 and hobby = 'chess' and income = 10;
```

Same query as Example 9.5.1 when had zhobincage index, tiny cost, 2R index, only .5R for row. But now have three different indexes: PLAN.

TNAME	ACCESSTYPE	MATCHCOLS	ACCESSNAME	PREFETCH	MIXOPSEQ
T	M	0		L	0
T	MX	1	hobbyx	S	1
T	MX	1	addrx	S	2
T	MI	0			3
T	MX	1	incomex	S	4
T	MI	0			5

Figure 9.18 Plan table rows of Multiple Index Access plan for Example 9.6.1

Calculate I/O cost. FF(hobby = 'chess') = 1/100, on hobbyx (NLEAF = 50,000), 500S (and ignore directory walk).

For MIXOPSEQ = 2, FF(zipcode = 02159) = 1/100,000 on addrx (NLEAF = 500,000), so 5S.

Intersect steps MI requires no I/O.

FF(income = 10) = 1/10, on incomex (NLEAF = 50,000), so 5,000S.

Still end up with 0.5 rows at the end by taking product of FFs for three predicates: (1/100,000)(1/100)(1/10) = 1/100,000,000

Only 50,000,0000 rows; ignore list prefetch of .5L.

So 5,505, elapsed time 5,505/400 = 13.8 seconds.

List Prefetch and RID Pool

All the things we've been mentioning up to now have been relatively universal, but RID list rules are quite product specific. Probably won't see all this duplicated in ORACLE.

-> We need RID extraction to access the rows with List Prefetch, but since this is faster than Random I/O couldn't we always use RID lists, even in single index scan to get at rows with List Prefetch in the end?

YES, BUT: There are restrictive rules that make it difficult. Most of these rules arise because RID list space is a scarce resource in memory.

-> The RID Pool is a separate memory storage area that is 50% the size of all four Disk Buffer Pools, except cannot exceed 200 MBytes.

Thus if DBA sets aside 2000 Disk Buffer Pages, will have 1000 pages (about 4 MBytes) of RID Pool. (Different memory area, though.)

Def. 9.6.1. **Rules for RID List Use**.

(1) QOPT, when it constructs query plan, predicts size of RIDs active at any time. Cannot use > 50% of total capacity. If guess wrong, abort during runtime.

(2) No Screening predicates can be used in an Index scan that extracts a RID List.

(3) An In-list predicate cannot be used when extract a RID list.

Example 9.6.3. RID List Size Limit. addrx, hobbyx, incomex, add sexx on column sex. Two values, 'M' and 'F', uniformly distributed.

```
select name, straddr from prospects where zipcode between
    02159 and 02358 and income = 10 and sex = 'F';
```

Try multiple index access. Extract RID lists for each predicate and intersect the lists.

But consider, sexx has 50,000 leaf pages. (Same as hobbyx, from 50,000,000 RID size entries.) Therefore sex = 'F' extracts 25,000 page RID list. 4 Kbytes each page, nearly 100 MBytes.

Can only use half of RID Pool, so need 200 MByte RID Pool.

Not unreasonable. A proof that we should use large buffers.

Example 9.6.5. No Index Screening. Recall zhobincage index in Example 9.3.8.

```
select name, straddr from prospects where zipcode between 02159
    and 04158 and hobby = 'chess' and income = 10;
```

Two latter predicates were used as screening predicates. But screening predicates can't be used in List Prefetch. To do List Prefetch, must resolve two latter predicates after bring in rows.

The 'zipcode between' predicate has a filter factor of 2000/100,000 = 1/50, so with 50 M rows, get 1M rows, RID list size of 4 MBytes, 1000 buffer pages, at least 2000 in pool. Assume RID pool is large enough.

So can do List Prefetch, 1,000,000/100 = 10,000 seconds, nearly 3 hours.

If use screening predicates, compound FF is (1/50)(1/100)(1/10) = 1/50,000. End with 1000 rows. Can't use List Prefetch, so 1000R but
1000/40 is only 25 seconds. Same index I/O cost in both cases. Clearly better not to use RID list.

What is reason not to allow List Prefetch with Screening Predicates? Just because resource is scarce, don't want to tie up RID list space for a long time while add new RIDs slowly. If refuse screening, QOPT might use other plans without RIDs.

Point of Diminishing Returns in MX Access.

Want to use RID lists in MIX. >> Talk it

Def. 9.6.2. (1) In what follows, assume have multiple indexes, each with a disjoint set of matching predicates from a query. Want to use RID lists.

(2) Way QOPT sees it: List RID lists from smallest size up (by increasing Filter Factor). Thus start with smaller index I/O costs (matching only) and larger effect in saving data page I/O.

(3) Generate successive RID lists and calculate costs; stop when cost of generating new RID list doesn't save enough in eventual data page reads by reducing RID set.

Example 9.6.6.

```
select name straddr from prospects
```

where prospects between o2159 and 02659
and age = 40 and hobby = 'chess' and income = 10;

(1) FF(zipcode between 02159 and 02658) = 500/100,000 = 1/200.
(2) FF(hobby = 'chess') = 1/100
(3) FF(age = 40) = 1/50 (Typo in text)
(4) FF(income = 10) = 1/10

Apply predicate 1. (1/200) (50M rows) retrieved, 250,000 rows, all on separate pages from 5M, and 250,000L is 2500 seconds. Ignore the index cost.

Apply predicate 2 after 1. Leaf pages of hobbyx scanned are (1/100) (50,000) = 500S, taking 500/400 = 1.25 seconds. Reduce number of rows retrieved from 250,000 to about 2500, all on separate pages, and 2500L takes about 25 seconds. So at cost of 1.25 seconds of index I/O, reduce table page I/O from 2500 seconds to 25 seconds. Clearly worth it.

Apply predicate 3 after 1 and 2. Leaf pages of agex scanned are (1/50) (50,000) = 1000S, taking 1000/400 = 2.5 seconds. Rows retrieved down to (1/50) (2500) = 50, 50L is .5 seconds. Thus at cost of 2.5 seconds of index I/O reduce table page I/O from 25 seconds to 0.5 seconds. Worth it.

With predicate 4, need index I/O at leaf level of (1/10) (50,000) = 5000S or 12.5 seconds. Not enough table page I/O left (0.5 seconds) to pay for it. Don't use predicate 4. We have reached point of diminishing returns.

Chapter 10

20.

We have encountered the idea of a transaction before in Embedded SQL.

Def. 10.1 Transaction. A transaction is a means to package together a number of database operations performed by a process, so the database system can provide several guarantees, called the ACID properties.

Think of writing: BEGIN TRANSACTION op1 op2 . . . opN END TRANSACTION

Then all ops within the transaction are packaged together.

There is no actual BEGIN TRANSACTION statement in SQL. A transaction is begun by a system when there is none in progress and the application first performs an operation that accesses data: Select, Insert, Update, etc.

The application logic can end a transaction successfully by executing:

```
exec sql commit work;       /* called simply a Commit                    */
```

Then any updates performed by operations in the transaction are successfully completed and made permanent and all locks on data items are released. Alternatively:

```
exec sql rollback work;      /* called an Abort                    */
```

means that the transaction was unsuccessful: all updates are reversed, and locks on data items are released.

The ACID guarantees are extremely important -- This and SQL is what differentiates a database from a file system.

Imagine that you are trying to do banking applications on the UNIX file system (which has it's own buffers, but no transactions). There will be a number of problems, the kind that faced database practitioners in the 50s.

1. **Inconsistent result**. Our application is transferring money from one account to another (different pages). One account balance gets out to disk (run out of buffer space) and then the computer crashes.

When bring computer up again, have no idea what used to be in memory buffer, and on disk we have destroyed money.

2. **Errors of concurrent execution**. (One kind: Inconsistent Analysis.) Teller 1 transfers money from Acct A to Acct B of the same customer, while Teller 2 is performing a credit check by adding balances of A and B. Teller 2 can see A after transfer subtracted, B before transfer added.

3. **Uncertainty as to when changes become permanent**. At the very least, we want to know when it is safe to hand out money: don't want to forget we did it if system crashes, then only data on disk is safe.

Want this to happen at transaction commit. And don't want to have to write out all rows involved in transaction (teller cash balance -- very popular, we buffer it to save reads and want to save writes as well).

To solve these problems, systems analysts came up with idea of transaction (formalized in 1970s). Here are ACID guarantees:

Atomicity. The set of record updates that are part of a transaction are indivisible (either they all happen or none happen). This is true even in presence of a crash (see Durability, below).

Consistency. If all the individual processes follow certain rules (money is neither created nor destroyed) and use transactions right, then the rules won't be broken by any set of transactions acting together. Implied by Isolation, below.

Isolation. Means that operations of different transactions seem not to be interleaved in time -- as if ALL operations of one Tx before or after all operations of any other Tx.

Durability. When the system returns to the logic after a Commit Work statement, it guarantees that all Tx Updates are on disk. Now ATM machine can hand out money.

The system is kind of clever about Durability. It doesn't want to force all updated disk pages out of buffer onto disk with each Tx Commit.

So it writes a set of notes to itself on disk (called logs). After crash run *Recovery* (also called *Restart*) and makes sure notes translate into appropriate updates.

What about Read-Only Tx? (No data updates, only Selects.) Atomicity and Durability have no effect, but Isolation does.

Money spent on Transactional systems today is about SIX BILLION DOLLARS A YEAR. We're being rigorous about some of this for a BUSINESS reason.

10.1 Transactional Histories.

Reads and Writes of data items. A data item might be a row of a table or it might be an index entry or set of entries. For now talking about rows.

Read a data item when access it without changing it. Often a select.

```
select val into :pgmval1 from T1 where uniqueid = A;
```

We will write this as $R_i(A)$: transaction with identification number i reads data item A. Kind of rough — won't always have be retrieving by uniqueid = A. But it means that we are reading a row identified as A. Now:

```
update T1 set val = pgmval2 where uniqueid = B;
```

we will write this as $W_j(B)$; Tx j writes B; say Update results in Write.

Can get complicated. Really reading an index entry as well to write B. Consider:

```
update T1 set val = val + 2 where uniqueid = B;
```

Have to read an index entry, R_j (predicate: uniqueid = B), then a pair of row operations: $R_j(B)$ (have to read it first, then update it) $W_j(B)$. Have to read it in this case before can write it.

```
update T set val = val + 2 where uniqueid between :low and :high;
```

Will result in a lot of operations: R_j(predicate: uniqueid between :low and :high), then $R_j(B1)$ $W_j(B1)$ $R_j(B2)$ $W_j(B2)$ . . . $R_j(BN)$ $W_j(BN)$.

>>The reason for this notation is that often have to consider complex interleaved histories of concurrent transactions; Example history:

(10.1.2) . . . $R_2(A)$ $W_2(A)$ $R_1(A)$ $R_1(B)$ $R_2(B)$ $W_2(B)$ C_1 C_2 . . .

Note C_i means commit by Tx i. A sequence of operations like this is known as a *History* or sometimes a *Schedule*.

A history results from a series of operations submitted by users, translated into R & W operations at the level of the Scheduler. See Fig. 10.1.

It is the job of the scheduler to look at the history of operations as it comes in and provide the Isolation guarantee, by sometimes delaying some operations, and occasionally insisting that some transactions be aborted.

By this means it assures that the sequence of operations is equivalent in effect to some serial schedule (all ops of a Tx are performed in sequence with no interleaving with other transactions). See Figure 10.1, pg. 640.

In fact, (10.1.2) above is an ILLEGAL schedule. Because we can THINK of a situation where this sequence of operations gives an inconsistent result.

Example 10.1.1. Say that the two elements A and B in (10.1.2) are Acct records with each having balance 50 to begin with. Inconsistent Analysis.

T_1 is adding up balances of two accounts, T_2 is transferring 30 units from A to B.

. . . $R_2(A, 50)$ $W_2(A, 20)$ $R_1(A, 20)$ $R_1(B, 50)$ $R_2(B, 50)$ $W_2(B, 80)$ C_1 C_2 . . .

And T determines that the customer fails the credit check (because under balance total of 80, say).

But this could never have happened in a serial schedule, where all operation of T_2 occurred before or after all operations of T_2.

. . . $R_2(A, 50)$ $W_2(A, 20)$ $R_2(B, 50)$ $W_2(B, 80)$ C_2 $R_1(A, 20)$ $R_1(B, 80)$ C_2 . . .
or
. . . $R_1(A, 50)$ $R_1(B, 50)$ C_1 $R_2(A, 50)$ $W_2(A, 20)$ $R_2(B, 50)$ $W_2(B, 80)$ C_2 . . .

And in both cases, T_1 sees total of 100, a Consistent View.

Notice we INTERPRETED the Reads and Writes of (10.1.2) to create a model of what was being read and written to show there was an inconsistency.

This would not be done by the Scheduler. It simply follows a number of rules we explain shortly. We maintain that a serial history is always consistent under any interpretation.

10.2. Interleaved Read/Write Operations

Quick>>

If a serial history is always consistent, why don't we just enforce serial histories.

The scheduler could take the first operation that it encounters of a given transaction (T_2 in the above example) and delay all ops of other Txs (the Scheduler is allowed to do this) until all operations of T_2 are completed and the transaction commits (C_2).

Reason we don't do this? Performance. It turns out that an average Tx has relatively small CPU bursts and then I/O during which CPU has nothing to do. See Fig 10.3, pg. 644. When I/O is complete, CPU can start up again.

What do we want to do? Let another Tx run (call this another thread) during slack CPU time. (Interleave). Doesn't help much if have only one disk (disk is bottleneck). See Fig 10.4, pg. 644.

But if we have two disks in use all the time we get about twice the throughput. Fig 10.5, pg. 645.

And if we have many disks in use, we can keep the CPU 100% occupied. Fig 10.6, pg 646.

In actuality, everything doesn't work out perfectly evenly as in Fig 10.6. Have multiple threads and multiple disks, and like throwing darts at slots.

Try to have enough threads running to keep lots of disk occupied so CPU is 90% occupied. When one thread does an I/O, want to find another thread with completed I/O ready to run again.

Leave this to you -- covered in Homework.

10.3 Serializability and the Precedence Graph.

We want to come up with a set of rules for the Scheduler to allow operations by interleaved transactions and guarantee *Serializability*.

Serializability means the series of operations is EQUIVALENT to a Serial schedule, where operations of Tx are not interleaved.

How can we guarantee this? First notice that if two transactions never access the same data items, it doesn't matter that they're interleaved.

We can commute ops in the history of requests permitted by the scheduler until all ops of one Tx are together (serial history). The operations don't affect each other, and order doesn't matter.

We say that the Scheduler is reading a history H (order operations are submitted) and is going to create a serializable history S(H) (by delay, etc.) where operations can be commuted to a serial history.

OK, now if we have operations by two different transactions that do affect the same data item, what then?

There are only four possibilities: R or W by T_1 followed by R or W by T_2.
Consider history:

$$\ldots R_1(A) \ldots W_2(A) \ldots$$

Would it matter if the order were reversed? YES. Can easily imagine an interpretation where T_2 changes data T_1 reads: if T_1 reads it first, sees old version, if reads it after T_2 changes it, sees later version.

We use the notation:

$$R_1(A) <<_H W_2(A)$$

to mean that $R_1(A)$ comes before $W_2(A)$ in H, and what we have just noticed is that whenever we have the ordering in H we must also have:

$$R_1(A) <<_{S(H)} W_2(A)$$

That is, these ops must occur in the same order in the serializable schedule put out by the Scheduler. If $R_1(A) <<_H W_2(A)$ then $R_1(A) <<_H W_2(A)$.

Now these transaction numbers are just arbitrarily assigned labels, so it is clear we could have written the above as follows:

If $R_2(A) <<_H W_1(A)$ then $R_2(A) <<_H W_1(A)$.

Here Tx 1 and Tx 2 have exchanged labels. This is another one of the four cases. Now what can we say about the following?

$$R_1(A) <<_H R_2(A)$$

This can be commuted -- reads can come in any order since they don't affect each other. Note that if there is a third transaction, T_3, where:

$$R_1(A) <<_H W_3(A) <<_H R_2(A)$$

Then the reads cannot be commuted (because we cannot commute either one with $W_3(A)$), but this is because of application of the earlier rules, not depending on the reads as they affect each other.

Finally, we consider:

$$W_1(A) <<_H W_2(A)$$

And it should be clear that these two operations cannot commute. The ultimate outcome of the value of A would change. That is:

If $W_1(A) <<_H W_2(A)$ then $W_1(A) <<_{S(H)} W_2(A)$

To summarize our discussion, we have Definition 10.3.1, pg. 650.

Def. 10.3.1. Two operations $X_i(A)$ and $Y_j(B)$ in a history are said to *conflict* (i.e., the order matters) if and only if the following three conditions hold:

(1) $A \int B$. Operations on distinct data items never conflict.

(2) $i \neq j$. Operations conflict only if they are performed by different Txs.

(3) One of the two operations X or Y is a write, W. (Other can be R or W.)

Note in connection with (2) that two operations of the SAME transaction also cannot be commuted in a history, but not because they conflict. If the scheduler delays the first, the second one will not be submitted.

21.

We have just defined the idea of two conflicting operations. (Repeat?)

We shall now show how some histories can be shown not not to be serializable. Then we show that such histories can be characterized by an easily identified characteristic in terms of conflicting operations.

To show that a history is not serializable (SR), we use an *interpretation* of the history.

Def. 10.3.2. An interpretation of an arbitrary history H consists of 3 parts. (1) A description of the purpose of the logic being performed. (2) Specification of precise values for data items being read and written in the history. (3) A consistency rule, a property that is obviously preserved by isolated transactions of the logic defined in (1).

Ex. 10.3.1. Example 10.3.1. Here is a history, H1, we clain is not SR.

$H1 = R_2(A)\ W_2(A)\ R_1(A)\ R_1(B)\ R_2(B)\ W_2(B)\ C_1\ C_2$

Here is an interpretation. T_1 is doing a credit check, adding up the balances of A and B. T is transferring money from A to B. Here is the consistency rule: Neither transaction creates or destroys money. Values for H1 are:

$H1' = R_2(A,50)\ W_2(A,20)\ R_1(A,20)\ R_1(B,50)\ R_2(B,50)\ W_2(B,70)\ C_1\ C_2$

The schedule H1 is not SR because H1' shows an *inconsistent result:* sum of 70 for balances A and B, though no money was destroyed by T_2 in the transfer from A to B. This could not have occurred in a serial execution.

The concept of conflicting operations gives us a direct way to confirm that the history H1 is not SR. Note the second and third operations of H1,
$W_2(A)$ and $R_1(A)$. Since W(A) comes before R(A) in H1, written:

$W_2(A) <<_{H1} R_1(A)$

We know since these operations conflict that they must occur in the same order in any equivalent serial history, S(H1), i.e.: $W_2(A) <<_{S(H1)} R_1(A)$

Now in a serial history, all operations of a transaction occur together

Thus $W_2(A) <<_{S(H1)} R_1(A)$ means that $T_2 <<_{S(H1)} T_1$, i.e. T_2 occurs before T_1 in any serial history S(H1) (there might be more than one).

But now consider the fourth and sixth operations of H1. We have:

$R_1(B) <<_{H1} W_2(B)$

Since these operations conflict, we also have $R_1(B) <<_{S(H1)} W_2(B)$

But this implies that T_1 comes before T_2 , $T_1 <<_{S(H1)} T_2$, in any equivalent serial history H1. And this is at odds with our previous conclusion.

In any serial history S(H1), either $T_1 <<_{S(H1)} T_2$ or $T_2 <<_{S(H1)} T_1$, not both. Since we conclude from examining H1 that both occur, S(H1) must not really exist. Therefore, H1 is not SR.

We illustrate this technique a few more times, and then prove a general characterization of SR histories in terms of conflicting operations.

Ex. 10.3.2. Consider the history:

$H2 = R_1(A)\ R_2(A)\ W_1(A)\ W_2(A)\ C_1\ C_2$

We give a interpretation of this history as a paradigm called a *lost update*.

Assume that A is a bank balance starting with the value 100 and T1 tries to add 40 to the balance at the same time that T2 tries to add 50.

$H2' = R_1(A, 100)\ R_2(A, 100)\ W_1(A, 140)\ W_2(A, 150)\ C_1\ C_2$

Clearly the final result is 150, and we have lost the update where we added 40. This couldn't happen in a serial schedule, so H1 is non-SR.

In terms of conflicting operations, note that operations 1 and 4 imply that $T1 <<_{S(H2)} T2$. But operations 2 and 3 imply that $T2 <<_{S(H2)} T1$. No SR schedule could have both these properties, therefore H2 is non-SR.

By the way, this example illustrates that a conflicting pair of the form $R_1(A) \ldots W_2(A)$ does indeed impose an order on the transactions, T1 << T2, in any equivalent serial history.

H2 has no other types of pairs that COULD conflict and make H2 non-SR.

Ex 10.3.3. Consider the history:

$H3 = W_1(A)\ W_2(A)\ W_2(B)\ W_1(B)\ C_1\ C_2$

This example will illustrate that a conflicting pair $W_1(A) \ldots W_2(A)$ can impose an order on the transactions T1 << T2 in any equivalent SR history.

Understand that these are what are known as "Blind Writes": there are no Reads at all involved in the transactions.

Assume the logic of the program is that T1 and T2 are both meant to "top up" the two accounts A and B, setting the sum of the balances to 100.

T1 does this by setting A and B both to 50, T2 does it by setting A to 80 and B to 20. Here is the result for the interleaved history H3.

$$H3' = W_1(A, 50)\ W_2(A, 80)\ W_2(B, 80)\ W_1(B, 50)\ C_1\ C_2$$

Clearly in any serial execution, the result would have A + B = 100. But with H3' the end value for A is 80 and for B is 50.

To show that H3 is non-SR by using conflicting operations, note that operations 1 and 2 imply T1 << T2, and operations 3 and 4 that T2 << T1.

Seemingly, the argument that an interleaved history H is non-SR seems to reduce to looking at conflicting pairs of operations and keeping track of the order in which the transactions will occur in an equivalent S(H).

When there are two transactions, T1 and T2, we expect to find in a non-SR schedule that T1 $<<_{S(H)}$ T2 and T2 $<<_{S(H)}$ T1, an impossibility.

If we don't have such an impossibility arise from conflicting operations in a history H, does that mean that H is SR?

And what about histories with 3 or more transactions involved? WIll we ever see something impossible other than T1 $<<_{S(H)}$ T2 and T2 $<<_{S(H)}$ T1?

We start by defining a Precedence Graph. The idea here is that this allows us to track all conflicting pairs of operations in a history H.

Def. 10.3.3. The Precedence Graph. A precedence graph for a history H is a directed graph denoted by PG(H).

The vertices of PG(H) correspond to the transactions that have COMMITTED in H: that is, transaction Ti where C exists as an operation in H.

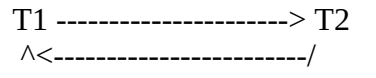
An edge Ti -> Tj exists in PG(H) whenever two conflicting operations X_i and Y_j occur in that order in H. Thus, Ti -> Tj should be interpreted to mean that Ti must precede Tj in any equivalent serial history S(H).

Whenever a pair of operations conflict in H for COMMITTED transactions, we can draw the corresponding direct arc in the Precedence Graph, PG(H).

The reason uncommitted transactions don't count is that we're trying to figure out what the scheduler can allow. Uncommitted transactions can always be aborted, and then it will be as if they didn't happen.

It would be unfair to hold uncommitted transactions against the scheduler by saying the history is non-SR because of them.

The examples we have given above of impossible conditions arising from conflicting operations look like this:



Of course this is what is called a circuit in a directed graph (a digraph). This suggests other problems that could arise with 3 or more Txs.

It should be clear that if PG(H) has a circuit, there is no way to put the transactions in serial order so Ti always comes befor Tj for all edges Ti -> Tj in the circuit.

There'll always be one edge "pointing backward" in time, and that's a contradiction, since Ti -> Tj means Ti should come BEFORE Tj in S(H).

How do we make this intuition rigorous? And if PG(H) doesn't have a circuit, does that mean the history is SR?

Thm. 10.3.4. **The Serializability Theorem**. A history H has an equivalent serial execution S(H) iff the precedence graph PG(H) contains no circuit.

Proof. Leave only if for exercises at end of chapter. I.e., will show there that a circuit in PG(H) implies there is no serial ordering of transactions.

Here we prove that if PG(H) contains no circuit, there is a serial ordering of the transactions so no edge of PG(H) ever points from a later to an earlier transaction.

Assume there are m transactions involved, and label them T1, T2, . . ., Tm.

We are trying to find a reordering of the integers 1 to m, i(1), i(2), . . ., i(m), so that Ti(1), Ti(2), . . ., Ti(m) is the desired serial schedule.

Assume a lemma to prove later: In any directed graph with no circuit there is always a vertex with no edge entering it.

OK, so we are assuming PG(H) has no circuit, and thus there is a vertex, or transaction, Tk, with no edge entering it. We choose Tk to be Ti(1).

Note that since Ti(1) has no edge entering it, there is no conflict in H that forces some other transaction to come earlier.

(This fits our intuition perfectly. All other transactions can be placed after it in time, and there won't be an edge going backward in time.)

Now remove this vertex, Ti(1), from PG(H) and all edges leaving it. Call the resulting graph $PG^1(H)$.

By the Lemma, there is now a vertex in $PG^1(H)$ with no edge entering it. Call that vertex Ti(2).

(Note that an edge from Ti(1) might enter Ti(2), but that edge doesn't count because it's been removed from $PG^1(H)$.)

Continue in this fashion, removing Ti(2) and all it's edges to form $PG^2(H)$, and so on, choosing Ti(3) from $PG^2(H)$, . . ., Ti(m) from $PG^{m-1}(H)$.

By construction, no edge of PG(H) will ever point backward in the sequence S(H), from Ti(m) to Ti(n), m > n.

The algorithm we have used to determine this sequence is known as a *topological sort*. This was a hiring question I saw asked at Microsoft.

The proof is complete, and we now know how to create an equivalen SR schedule from a history whose precedence graph has no circuit.

Lemma 10.3.5. In any finite directed acyclic graph G there is always a vertex with no edges entering it.

Proof. Choose any vertex v1 from G. Either this has the desired property, or there is an edge entering it from another vertex v2.

(There might be several edges entering v1, but choose one.)

Now v2 either has the desired property or there is an edge entering it from vertex v3. We continue in this way, and either the sequence stops at some vertex vm, or the sequence continues forever.

If the sequence stops at a vertex vm, that's because there is no edge entering vm, and we have found the desired vertex.

But if the sequence continues forever, since this is a finite graph, sooner or later in the sequence we will have to have a repeated vertex.

Say that when we add vertex vn, it is the same vertex as some previously mentioned vertex in the sequence, vi.

Then there is a path from vn -> v(n-1) -> . . . v(i+1) ->vi, where vi ∫ vn. But this is the definition of a circuit, which we said was impossible.

Therefore the sequence had to terminate with vm and that vertex was the one desired with no edges entering.

22.

10.4 Locking Ensures Serializabilty

See Fig. 10.8, pg. 609. TM passes on calls such as fetch, select, insert, delete, abort; Scheduler interprets them as: $R_i(A)$, $W_j(B)$.

It is the job of the scheduler to make sure that no non-SR schedules get past. This is normally done with *Two-Phase Locking*, or 2PL.

Def. 10.4.1. 2PL. Locks taken in released following three rules.

(1) Before Tx i can read a data item, $R_i(A)$, scheduler attempts to Read Lock the item on it's behalf, $RL_i(A)$; before $W_i(A)$, try Write Lock, $WL_i(A)$.

(2) If conflicting lock on item exists, requesting Tx must WAIT. (Conflicting locks corresponding to conflicting ops: two locks on a data item conflict if they are attempted by different Txs and at least one of them is a WL).

(3) There are two phases to locking, the growing phase and the shrinking phase (when locks are released: $RU_i(A)$; The scheduler must ensure that can't shrink (drop a lock) and then grow again (take a new lock).

Rule (3) implies can release locks before Commit; More usual to release all locks at once on Commit, and we shall assume this in what follows.

Note that a transaction can never conflict with its own locks! If Ti holds RL on A, can get WL so long as no other Tx holds a lock (must be RL).

A Tx with a WL doesn't need a RL (WL more powerful than RL).

Clearly locking is defined to guarantee that a circuit in the Precedence Graph can never occur. The first Tx to lock an item forces any other Tx that gets to it second to "come later" in any SG.

But what if other Tx already holds a lock the first one now needs? This would mean a circuit, but in the WAIT rules of locking it means NEITHER Tx CAN EVER GO FORWARD AGAIN. This is a DEADLOCK. Example shortly.

Side effect of 2PL is that Deadlocks can occur: When a deadlock occurs, scheduler will recognize it and force one of the Txs involved to Abort.
(Note, there might be more than 2 Txs involved in a Deadlock.)

Ex. Here is history not SR (Error in text: this is a variant of Ex. 10.4.1).

$$H4 = R_1(A)\ R_2(A)\ W_2(A)\ R_2(B)\ W_2(B)\ R_1(B)\ C_1\ C_2$$

Same idea as 10.3.1 why it is non-SR: T2 reads two balances that start out A=50 and B=50, T1 moves 30 from A to B. Non-SR history because T1 sees A=50 and B=80. Now try locking and releasing locks at commit.

$RL_1(A)$ $R_1(A)$ $RL_2(A)$ $R_2(A)$ $WL_2(A)$ (conflicting lock held by T1 so T2 must WAIT) $RL_1(B)$ $R_1(B)$ C_1 (now T2 can get $WL_2(A)$) $W_2(A)$ $RL_2(B)$ $R_2(B)$ $WL_2(B)$ $W_2(B)$ C_2

Works fine: T1 now sees A=50, B=50. Serial schedule, T1 then T2.

But what if allowed to Unlock and then acquire more locks later. Get non-SR schedule. Shows necessity of 2PL Rule (3).

$RL_1(A)$ $R_1(A)$ $RU_1(A)$ $RL_2(A)$ $R_2(A)$ $WL_2(A)$ $W_2(A)$ $WU_2(A)$ $RL_2(B)$ $R_2(B)$ $WL_2(B)$ $W_2(B)$ $WU_2(B)$ $RL_1(B)$ $R_1(B)$ C_1 C_2

So H4 above is possible. But only 2PL rule broken is that T1 and T2 unlock rows, then lock other rows later.

The Waits-For Graph. How scheduler checks if deadlock occurs. Vertices are currently active Txs, Directed edgs Ti -> Tj iff Ti is waiting for a lock held by Tj.

(Note, might be waiting for lock held by several other Txs. And possibly get in queue for W lock behind others who are also waiting. Draw picture.)

The scheduler performs lock operations and if Tx required to wait, draws new directed edges resulting, then checks for circuit.

Ex 10.4.2. Here is schedule like H4 above, where T2 reverses order it touches A and B (now touches B first), but same example shows non-SR.

H5 = $R_1(A)$ $R_2(B)$ $W_2(B)$ $R_2(A)$ $W_2(A)$ $R_1(B)$ C_1 C_2

Locking result:

$RL_1(A)$ $R_1(A)$ $RL_2(B)$ $R_2(B)$ $WL_2(B)$ $W_2(B)$ $RL_2(A)$ $R_2(A)$ $WL_2(A)$ (Fails: RL1(A) held, T2 must WAIT for T1 to complete and release locks) $RL_1(B)$ (Fails: WL2(B) held, T1 must wait for T2 to complete: But this is a deadlock! Choose T2 as victim (T1 chosen in text)) A2 (now $RL_1(B)$ will succeed) R1(B) C1 (start T2 over, retry, it gets Tx number 3) $RL_3(B)$ $R_3(B)$ $WL_3(B)$ $W_3(B)$ $RL_3(A)$ $R_3(A)$ $WL_3(A)$ W3(A) C3.

Locking serialized T1, then T2 (retried as T3).

Thm. 10.4.2. Locking Theorem. A history of transactional operations that follows the 2PL discipline is SR.

First, **Lemma 10.4.3**. If H is a Locking Extended History that is 2PL and the edge Ti -> Tj is in PG(H), then there must exist a data item D and two conflicting operations Xi(D) and Yj(D) such that XUi(D) <<H YLj(D).

Proof. Since Ti -> Tj in PG(H), there must be two conflicting ops Xi(D) and Yj(D) such that Xi(D) <<H Yj(D).

By the definition of 2PL, there must be locking and unlocking ops on either side of both ops, e.g.: XLi(D) <<H Xi(D) <<H XUi(D).

Now between the lock and unlock for Xi(D), the X lock is held by Ti and similarly for Yj(D) and Tj. Since X and Y conflict, the locks conflict and the intervals cannot overlap. Thus, since Xi(D) <<H Yj(D), we must have:

XLi(D) <<H Xi(D) <<H XUi(D) <<H YLj(D) <<H Yj(D) <<H YUj(D)

And in particular XUi(D) <<H YLj(D).

Proof of Thm. 10.4.2. We want to show that every 2PL history H is SR.

Assume in contradiction that there is a cycle T1 -> T2 -> . . . -> Tn -> T1 in PG(H). By the Lemma, for each pair Tk -> T(k+1), there is a data item Dk where XUk(Dk) <<H YL(k+1)(Dk). We write this out as follows:

1. XU1(D1) <<H YL2(D1)
2. XU2(D2) <<H YL3(D2)
. . .
n-1. XU(n-1)(D(n-1)) <<H YLn(D(n-1))
n. XUn(Dn) <<H YL1(Dn) (Note, T1 is T(n+1) too.)

But now have (in 1.) an unlock of a data item by T1 before (in n.) a lock of a data item. So not 2PL after all. Contradiction.

Therefore H is 2PL implies no circuit in the PG(H), and thus H is SR.

Note that not all SR schedules would obey 2PL. E.g., the following is SR:

H7 = R1(A) W1(A) R2(A) R1(B) W1(B) R2(B) C1 C2

But it is not 2PL (T2 breaks through locks held by T1). We can optimistically allow a Tx to break through locks in the hopes that a circuit won't
occur in PG(H). But most databases don't do that.

23.

10.5 Levels of Isolation

The idea of Isolation Levels, defined in ANSI SQL-92, is that people might want to gain more concurrency, even at the expense of imperfect isolation.

A paper by Tay showed that when there is serious loss of throughput due to locking, it is generally not because of deadlock aborts (having to retry) but simply because of transactions being *blocked* and having to wait.

Recall that the reason for interleaving transaction operations, rather than just insisting on serial schedules, was so we could keep the CPU busy.

We want there to always be a new transaction to run when the running transaction did an I/O wait.

But if we assume that a lot of transactions are waiting for locks, we lose this. There might be only one transaction running even if we have 20 trying to run. All but one of the transactions are in a wait chain!

So the idea is to be less strict about locking and let more transactions run. The problem is that dropping proper 2PL might cause SERIOUS errors in applications. But people STILL do it.

The idea behind ANSI SQL-99 Isolation Levels is to weaken how locks are held. Locks aren't always taken, and even when they are, many locks are released before EOT.

And more locks are taken after some locks are released in these schemes. Not Two-Phase, so not perfect Isolation.

(Note in passing, that ANSI SQL-92 was originally intended to define isolation levels that did not require locking, but it has been shown that the definitions failed to do this. Thus the locking interpretation is right.)

Define *short-term locks* to mean a lock is taken prior to the operation (R or W) and released IMMEDIATELY AFTERWARD. This is the only alternative to *long-term locks*, which are held until EOT.

Then ANSI SQL-92 Isolation levels are defined as follows (Fig. 10.9 -- some difference from the text):

	Write locks on rows of a table are long term	Read Locks on rows of a table are long term	Read locks on predicates are long term
Read Uncommitted (Dirty Reads)	NA (Read Only)	No Read Locks taken at all	No Read Locks taken at all

Read Committed	Yes	No	No
Repeatable Read	Yes	Yes	No
Serializable	Yes	Yes	Yes

Note that Write Predicate Locks are taken and held long-term in all isolation levels listed. What this means is explained later.

In Read Uncommitted (RU), no Read locks are taken, thus can read data on which Write lock exists (nothing to stop you if don't have to WAIT for RL).

Thus can read uncommitted data; it will be wrong if Tx that changed it later aborts. But RU is just to get a STATISTICAL idea of sales during the day (say). CEO wants to know ballpark figure -- OK if not exact.

In Read Committed (RC), we take Write locks and hold them to EOT, and Read Locks on rows read and predicates and release immediately. (Cover predicates below.)

Problem that can arise is serious one, Lost Update (Example 10.3.2):

... $R_1(A,100)$ $R_2(A,100)$ $W_1(A,140)$ $W_2(A,150)$ C_1 C_2 ...

Since R locks are released immediately, nothing stops the later Writes, and the increment of 40 is overwritten by an increment of 50, instead of the two increments adding to give 90.

Call this the Scholar's Lost Update Anomaly (since many people say Lost Update only happens at Read Uncommitted).

This is EXTREMELY serious, obviously, and an example of lost update in SQL is given in Figure 10.12 (pg. 666) for a slightly more restrictive level: Cursor Stability. Applications that use RC must avoid this kind of update.

In Figure 10.11, we see how to avoid this by doing the Update indivisibly in a single operation.

Not all Updates can be done this way, however, because of complex cases where the rows to bue updated cannot be determined by a Boolean search condition, or where the amount to update is not a simple function.

It turns out the IBM's Cursor Stability guarantees a special lock will be held on current row under cursor, and at first it was thought that ANSI Read Committed guaranteed that, but it does not.

Probably most products implement a lock on current of cursor row, but there is no guarantee. NEED TO TEST if going to depend on this.

In Repeatable Read Isolation, this is the isolation level that most people think is all that is meand by 2PL. All data items read and written have RLs and WLs taken and held long-term, until EOT.

So what's wrong? What can happen?

Example 10.5.3, pg. 666, Phantom Update Anomaly.

R_1(predicate: branch_id = 'SFBay') R_1(A1,100.00) R_1(A2, 100.00)
R_1(A3,100.00) I_2(A4, branch_id = 'SFBay', balance =100.00)
R_2(branch_totals, branch_id = SFBay, 300.00)
W_2(branch_totals, branch_id = SFBay ,400.00) C_2
R_1(branch_totals, branch_id = SFBay, 400) (Prints out error message) C_1

T1 is reading all the accounts with branch_id = SFBay and testing that the sum of balances equals the branch_total for that branch (accounts and branch_totals are in different tables)

After T1 has gone through the rows of accounts, T2 inserts another row into accounts with branch_id = SFBay (T1 will not see this as it's already scanned past the point where it is inserted), T2 then updates the branch_total for SFBay, and commits.

Now T1, having missed the new account, looks at the branch_total and sees an error.

There is no error really, just a new account row that T1 didn't see.

Note that nobody is breaking any rules about data item locks. The insert by T2 holds a write lock on a new account row that T1 never read. T2 locks the branch_total row, but then commits before T1 tries to read it.

No data item lock COULD help with this problem. But we have non-SR behavior nonetheless.

The solution is this: When T1 reads the predicate branch_id = SFBay on accounts, it takes a Read lock ON THAT PREDICATE, that is to say a Read lock on the SET of rows to be returned from that Select statement.

Now when T2 tries to Insert a new row in accounts that will change the set of rows to be returned for SFBay, it must take a Write lock on that predicate.

Clearly this Write lock and Read lock will conflict. Therefore T2 will have to wait until T1 reaches EOT and releases all locks.

So the history of Example 10.5.3 can't happen. (In reality, use a type of locking called Key-Range locking to guarantee predicate locks. Cover in Database Implementation course.)

ANSI Repeatable Read Isolation doesn't provide Predicate Locks, but ANSI Serializable does.

Note that Oracle doesn't ever perform predicate locks. Oracle's SERIALIZABLE isolation level uses a different approach, based on snapshot reads, that is beyond what we can explain in this course.

10.6 Transactional Recovery.

The idea of transactional recovery is this.

Memory is "Volatile", meaning that at unscheduled times we will lose memory contents (or become usure of the validity).

But a database transaction, in order to work on data from disk, must read it into memory buffers.

Remember that a transaction is "atomic", meaning that all update operationa a transaction performs must either ALL succeed or ALL fail.

If we read two pages into memory during a transaction and update them both, we might (because of buffering) have one of the pages go back out to disk before we commit.

What are we to do about this after a crash? An update has occurred to disk where the transaction did not commit. How do we put the old page back in place?

How do we even know what happened? That we didn't commit?

A similar problem arises if we have two pages in memory and after commit we manage to write one of the pages back to disk, but not the other.

(In fact, we always attempt to minimize disk writes from popular buffers, just as we minimize disk reads.)

How do we fix it so that the page that didn't get written out to disk gets out during recovery?

The answer is that as a transaction progresses we write notes to ourselves about what changes have been made to disk pages. We ensure that these notes get out to disk to allow us to correct any errors after a crash.

These notes are called "logs", or "log entries". The log entries contain "Before Images" and "After Images" of every update made by a Transaction.

In recovery, we can back up an update that shouldn't have gotten to disk (the transaction didn't commit) by applying a Before Image.

Similarly, we can apply After Images to correct for disk pages that should have gotten to disk (the transaction did commit) but never made it.

There is a "log buffer" in memory (quite long), and we write the log buffer out to the "log on disk" every time one of following events occur.

(1) The log buffer fills up. We write it to disk and meanwhile continue filling another log buffer with logs. This is known as "double buffering" and saves us from having to wait until the disk write completes.

(2) Some transaction commits. We write the log buffer, including all logs up to the present time, before we return from commit to the application (and the application hands out the money at the ATM). This way we're sure we won't forget what happened.

Everything else in the next few sections is details: what do the logs look like, how does recovery take place, how can we speed up recovery, etc.

10.7 Recovery in Detail: Log Formats.

Consider the following history H5 of operations as seen by the scheduler:

(10.7.1) H5 = $R_1(A,50)$ $W_1(A,20)$ $R_2(C,100)$ $W_2(C,50)$ C2 $R_1(B,50)$ $W_1(B,80)$ C_1

Because of buffering, some of the updates shown here might not get out to disk as of the second commit, C1. Assume the system crashes immediately after. How do we recover all these lost updates?

While the transaction was occurring, we wrote out the following logs as each operation occurred (Figure 10.13, pg. 673).

OPERATION	**LOG ENTRY**	***** LEAVE UP ON BOARD *****
$R_1(A,50)$	(S, 1) — Start transaction T_1 - no log entry is written for a Read operation, but this operation is the start of T_1	
$W_1(A,20)$	(W, 1, A, 50, 20) — T_1 Write log for update of A.balance. The value 50 is the **Before Image** (**BI**) for A.balance column in row A, 20 is the **After Image** (**AI**) for A.balance	
$R_2(C,100)$	(S, 2), another start transaction log entry.	
$W_2(C,50)$	(W, 2, C, 100, 50), another Write log entry.	
C2	(C, 2) — Commit T_2 log entry. (***Write Log Buffer to Log File***.)	
$R_1(B,50)$	No log entry.	
$W_1(B,80)$	(W, 1, B, 50, 80)	
C_1	(C, 1) Commit T_1 (***Write Log Buffer to Log File***.)	

Assume that a System Crash occurred immediately after the W_1(B,80) operation.

This means that the log entry (W, 1, B, 50, 80) has been placed in the log buffer, but the last point at which the log buffer was written out to disk was with the log entry (C, 2)

This is the final log entry we will find when we begin to recover from the crash. Assume that the values out on disk are A = 20 (the update to 20 drifted out to disk), B = 50 (update didn't get to disk), and C = 100 (same).

If you look carefully at the sequence, where T2 committed and T1 didn't, you will see that the values should be: A = 50, B = 50, C = 50.

After the crash, a commend is given by the system operator that initiates recovery. This is usually called the RESTART command.

The process of recovery takes place in two phases, *Roll Back* and *Roll Forward*.. The Roll Back phase backs out updates by uncommitted transactions and Roll Forward reapplies updates of committed transactions.

In Roll Back, the entries in the disk log are read backward to the beginning, System Startup, when A = 50, B = 50, and C = 100.

In Roll Back, the system makes a list of all transactions that did and did not commit. This is used to decide what gets backed out and reapplied.

LOG ENTRY	**ROLL BACK/ROLL FORWARD ACTION PERFORMED**
1. (C, 2)	Put T_2 into "Committed List"
2. (W, 2, C,100,50)	Since T_2 is on "Committed List", we do nothing.
3. (S, 2)	Make a note that T_2 is no longer "Active"
4. (W, 1, A, 50, 20)	Transaction T_1 has never committed (it's last operation was a Write). Therefore, the system performs UNDO of this update by Writing the Before Image value (50) into data item B. Put T_1 into "Uncommitted List"
5. (S, 1)	Make a note that T_1 is no longer "Active". Now that no transactions were active, we can end the ROLL BACK phase.
ROLL FORWARD	
6. (S, 1)	No action required

7. (W, 1, A, 50, 20) T_1 is Uncommitted — No action required

8. (S, 2) No action required

9. (W, 2, C,100,50) Since T_2 is on Committed List, we REDO this update by writing After Image value (50) into data item C

10 (C, 2) No action required

11 We note that we have rolled forward through all log entries and terminate Recovery.

Note that at this point, A = 50, B = 50, and C = 50.

Guarantees That Needed Log Entries are on Disk

How could a problem occur with our method of writing logs and recovering? Look at the history earlier again and think what would happen if we ended up with B = 80 because the final written value of B got out to disk.

Since we have been assuming that the log (W, 1, B, 50, 80) did NOT get out to disk, we wouldn't be able to UNDO this update (which should not occur, since T1 did not commit).

This is a problem that we solve with a policy that ties the database buffer writes to the Log. The policy is called *Write-Ahead Log (WAL)*.

It guarantees that no buffer dirty page gets written back to disk before the Log that would be able to UNDO it gets written to the disk Log file.

OK, this would solve the problem of UNDOs. Do we ever have a problem with REDOs? No, because we always write the Log buffer to the Log file as part of the Commit. So we're safe in doing REDO for committed Txs.

The text has a "proof" that recovery will work, given these log formats, the RESTART procedure of Roll Back/Roll Forward, and the WAL policy.

24.

10.8 Checkpoints

In the recovery process we just covered, we performed ROLLBACK to *System Startup Time*, when we assume that all data is valid.

We assume that System Startup occurs in the morning, and database processing continues during the day and everything completes at the end of day.

(This is a questionable assumption nowadays, with many companies needing to perform 24X7 processing, 24 hours a day, 7 days a week, with no time when transactions are guaranteed to not be active.)

Even if we have an 8-hour day of processing, however, we can run into real problems recovering a busy transactional system (heavy update throughput) with the approach we've outlined so far.

The problem is that it takes nearly as much processing time to RECOVER a transaction as it did to run it in the first place.

If our system is strained to the limit keeping up with updates from 9:00 AM to 5:00 PM, and the system crashes at 4:59, it will take nearly EIGHT HOURS TO RECOVER.

This is the reason for checkpointing. When a "Checkpoint" is taken at a given time (4:30 PM) this makes it possible for Recovery to limit the logs it needs to ROLLBACK and ROLLFORWARD.

A simple type of Checkpoint, a "Commit Consistent Checkpoint," merely duplicates the process of shutting down for the night, but then transactions start right up again.

The problem is that it might take minutes to take a Commit Consistent Checkpoint, and during that time NO NEW TRANSACTIONS CAN START UP.

For this reason, database systems programmers have devised two other major checkpointing schemes that reduce the "hiccup" in transaction processing that occurs while a checkpoint is being performed.

The Commit Consistent Checkpoint is improved on by using something called a "Cache Consistent Checkpoint". Then an even more complicated checkpoint called a "Fuzzy Checkpoint" improves the situation further.

So this is what we will cover now, in order (put on board):

Commit Consistent Checkpoint
Cache Consistent Checkpoint
Fuzzy Checkpoint

We define the Checkpoint Process. From time to time, a Checkpoint is triggered, probably by a time since last checkpoint system clock event.

Def 10.8.1. Commit Consistent Checkpoint steps. After the "performing checkpoint state" is entered, we have the following rules.

(1) No new transactions can start until the checkpoint is complete.

(2) Database operation processing continues until all existing transactions Commit, and all their log entries are written to disk. (Thus we are *Commit Consistent*).

(3) Then the current log buffer is written out to the log file, and after this the system ensures that all dirty pages in buffers have been written out to disk.

(4) When steps (1)-(3) have been performed, the system writes a special log entry, (CKPT), to disk, and the Checkpoint is complete. Ÿ

It should be clear that these steps are basically the same ones that would be performed to BRING THE SYSTEM DOWN for the evening.

We allow transactions in progress to finish, but don't allow new ones, and everything in volatile memory that reflects a disk state is put out to disk.

As a matter of fact, the Disk Log File can now be emptied. We needed it while we were performing the checkpoint in case we crashed in the middle, but now we don't need it any longer.

We will never need the Log File again to UNDO uncommitted transactions that have data on disk (there are no such uncommitted transactions) or REDO committed transactions that are missing updates on disk (all updates have gone out to disk already).

From this, it should be clear that we can modify the Recovery approach we have been talking about so that instead of a ROLLBACK to the Beginning of the Log File at System Startup, we ROLLBACK to the LAST CHECKPOINT!!!

If we take a Checkpoint every five minutes, we will never have to recover more than five minutes of logged updates, so recovery will be fast.

The problem is that the Checkpoint Process itself might not be very fast.

Note that we have to allow all transactions in progress to complete before we can perform successive steps. If all applications we have use very short transactions, there should be no problem.

Cache Consistent Checkpoint

But what if some transactions take more than five minutes to execute?

Then clearly we can't guarantee a Checkpoint every five minutes!!!

Worse, while the checkpoint is going on (and the last few transactions are winding up) nobody else can start any SHORT transactions to read an account balance or make a deposit!

We address this problem with something called the "Cache Consistent Checkpoint". With this scheme, transactions can continue active through the checkpoint. We don't have to wait for them all to finish and commit.

Definition 10.8.2. Cache Consistent Checkpoint procedure steps.

(1) No new transactions are permitted to start.

(2) Existing transactions are not permitted to start any new operations.

(3) The current log buffer is written out to disk, and after this the system ensures that all dirty pages in cache buffers have been written out to disk. (Thus, we are "Cache" (i.e., Buffer) Consistent on disk.)

(4) Finally, a special log entry, (CKPT, List) is written out to disk, and the Checkpoint is complete. NOTE: this (CKPT) log entry contains a list of active transactions at the time the Checkpoint occurs. Ÿ

The recovery procedure using Cache Consistent Checkpoints differs from Commit Consistent Checkpoint recovery in a number of ways.

Ex 10.8.1 Cache Consistent Checkpoint Recovery.

Consider the history H5:

H5: $R_1(A, 10)$ $W_1(A, 1)$ C_1 $R_2(A, 1)$ $R_3(B, 2)$ $W_2(A, 3)$ $R_4(C, 5)$ CKPT
$W_3(B, 4)$ C_3 $R_4(B, 4)$ $W_4(C, 6)$ C_4 CRASH

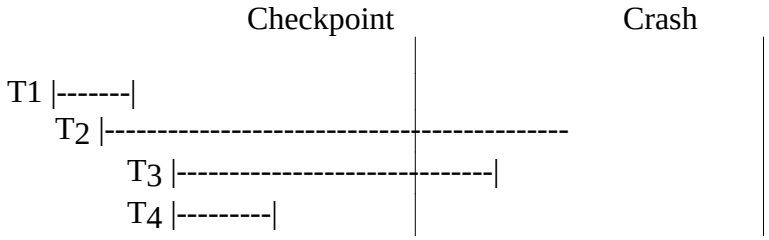
Here is the series of log entry events resulting from this history. The last one that gets out to disk is the (C, 3) log entry.

(S, 1) (W, 1, A, 10, 1) (C, 1) (S 2) (S, 3) (W, 2, A, 1, 3) (S, 4)
(CKPT, (LIST = T_2, T_4)) (W, 3, B, 2, 4) (C, 3) (W, 4, C, 5, 6) (C, 4)

At the time we take the Cache Consistent Checkpoint, we will have values out on disk: A = 3, B = 2 , C = 5. (The dirty page in cache containing A at checkpoint time is written to disk.)

Assume that no other updates make it out to disk before the crash, and so the data item values remain the same.

Here is a diagram of the time scale of the various events. Transaction T_k begins with the (S, k) log, and ends with (C, k). **LEAVE ON BOARD**



Next, we outline the actions taken in recovery, starting with ROLL BACK.

ROLL BACK

1. (C, 3) Note T_3 is a committed Tx in active list.
2. (W, 3, B, 2, 4) Committed transaction, wait for ROLL FORWARD.
3. (CKPT, (LIST = T_2, T_4)) Note active transactions T_2 and T_4; THESE HAVE NOT COMMITTED (no (C, 2) or (C, 4) logs have been encountered)
4. (S, 4) List of Active transactions now shorter: {T_2, T_3}
5. (W, 2, A, 1, 3) Not Committed. UNDO: A = 1
6. (S, 3) List of Active Transactions shorter.
7. (S, 2) List of Active Transactions empty. STOP ROLLBACK.

With a Cache Consistent Checkpoint, when ROLL BACK encounters the CKPT log entry the system takes note of the transactions that were active, even though we have never seen any operations in the log file.

We now take our list of active transactions, remove those that we have seen committed, and have a list of transactions whose updates we need to UNDO. Since Transactions can live through Checkpoints we have to go back PRIOR to the Checkpoint, while UNDO steps might remain.

We continue in the ROLL BACK phase until we complete all such UNDO actions. We can be sure when this happens because as we encounter (S, k) logs, rolling backward.

When all Active Uncommitted T_k have been removed, the ROLL BACK is complete, even though there may be more entries occurring earlier in the log file.

ROLL FORWARD

8. (CKPT, (LIST = T_2, T_3)) Skip forward in log file to this entry, start after this.

9. (W, 4, C, 5, 6)	Roll Forward: C = 6.
18. (C, 4)	No Action. Last entry: ROLL FORWARD is complete.

In starting the Roll Forward Phase, we merely need to REDO all updates by committed transactions that might not have gone out to disk.

We can jump forward to the first operation after the Checkpoint, since we know that all earlier updates were flushed from buffers.

Roll Forward continues to the end of the Buffer File. Recall that the values on disk at the time of the crash were: A = 3, B = 2 , C = 5. At the end of Recovery, we have set A = 1 (Step 5), and C = 6 (Step 9).

We still have B = 4. A glance at the time scale figure shows us that we want updates performed by T_4 to be applied, and those by T_2 and T_3 to be backed out. There were no writes performed by T_4 that got out to disk, so we have achieved what is necessary for recovery: A = 1, B = 4, C = 6.

Fuzzy Checkpoint.

A problem can still arise that makes the Cache Consistent Checkpoint a major hiccup in Transaction Processing.

Note in the procedure that we can't let any Active Transactions continue, or start any new ones, until all buffers are written to disk. What if there are a LOT of Buffers?

Some new machines have several GBytes of memory! That's probably Minutes of I/O, even if we have a lot of disks. DISK I/O is SLOW!!!

OK, with a Fuzzy Checkpoint, each checkpoint, when it completes, makes the PREVIOUS checkpoint a valid place to stop ROLLBACK.

Definition 10.8.3. Fuzzy Checkpoint procedure steps.

(1) Prior to Checkpoint start, the remaining pages that were dirty at the prior checkpoint will be forced out to disk (but the rate of writes should leave I/O capacity to support current transactions in progress; there is no critical hurry in doing this).

(2) No new transactions are permitted to start. Existing transactions are not permitted to start any new operations.

(3) The current log buffer is written out to disk with an appended log entry, ($CKPT_N$, List), as in the Cache Consistent Checkpoint procedure.

(4) The set of pages in buffer that have become dirty since the last checkpoint log, $CKPT_{N-1}$, is noted.

This will probably be accomplished by special flags on the Buffer directory. There is no need for this information to be made disk resident, since it will be used only to perform the next checkpoint, not in case of recovery. At this point the Checkpoint is complete. Ÿ

As explained above, the recovery procedure with Fuzzy Checkpoints differs from the procedure with Commit Consistent Checkpoints only that ROLL FORWARD must start with the first log entry following the SECOND to last checkpoint log. We have homeowork on this.

25.

Covered last time the various Checkpoints: Commit Consistent, Cache Consistent, and Fuzzy. Any questions?

What really happens with commercial databases. Used to be all Commit Consistent, now often Fuzzy.

Also used to be VERY physical. BI and AI meant physical copies of the entire PAGE. Still need to do this sometimes, but for long-term log can be more "logical".

Instead of "Here is the way the page looked after this update/before this update", have: this update was ADD 10 to Column A of row with RID 12345 with version number 1121.

Version number is important to keep updates idempotent.

Note that recovery is intertwined with the type of recovery. It doesn't do any good to have row-level locking if have page level recovery.

T1 changes Row 12345 column A from 123 to 124, and log gives PAGE BI with A = 123, T1 hasn't committed yet.

T2 changes Row 12347 (same page) column B from 321 to 333 and Commits, log gives Row 12345 with 124, Row 12347 column B with 333 on page AI.

Transaction T2 commits, T1 doesn't, then have crash. What do we do?

Put AI of T2 in place? Gives wrong value to A. Put BI of T1 in place? Wrong value of B.

Page level logging implies page level locking is all we can do.

Sybase SQL Server STILL doesn't have row-level locking.

10.9 Media Recovery

Problem is that Disk can fail. (Not just stop running: head can score disk surface.) How do we recover?

First, we write our Log to TWO disk backups. Try to make sure two disks have Independent Failure Modes (not same controller, same power supply).

We say that storage that has two duplicates is called *stable storage*, as compared to *nonvolatile storage* for a normal disk copy.

Before System Startup run BACKUP (bulk copy of disks/databases).

Then, when perform Recovery from Media failure, put backup disk in place and run ROLL FORWARD from that disk, as if from startup in the morning.

As if normal recovery on this disk except that all pages on this disk were VERY popular and never got out from disk. Don't need Roll Back except conceptually.

RAID Disks

Something they have nowadays is RAID disks. RAID stands for Redundant Arrays of Inexpensive Disks. Invented at Berkeley. The Inexpensive part rather got lost when this idea went commercial.

The simplest kind, RAID 1, mirrors Writes. Could have two disks that write everything to in two copies. Every time we Write, need 2 Writes.

So if one disk lost, just use other disk. Put another blank disk in for mirror, and while operating with normal Reads and Writes do BACKUP to new disk until have mirror.

This approach saves the time needed to do media recovery. And of course works for OS files, where there IS no media recovery.

You can buy these now. As complex systems get more and more disks, will eventually need RAID. The more units there are on a system, the more frequent the failures.

Note that mirrored Writes are handled by controller. Doesn't waste time of System to do 2 Writes.

But when Read, can Read EITHER COPY. Use disk arms independently.

So we take twice as much media, and if all we do is Writes, need twice as many disk arms to do the same work.

But if all we do is Reads, get independent disk arm movements, so get twice as many Reads too.

But in order for twice as many Reads to work, need warm data, where disk capacilty is not the bottleneck, but disk arm movement is.

Definitely lose the capacity in RAID 1. But if we were only going to use half the capacity of the disk because we have too Many Reads, RAID 1 is fine.

There is an alternative form of RAID, RAID 5, that uses less capacity than mirroring. Trick is to have 6 disks, 5 real copies of page, one checksum.

Use XOR for Checksum. CK = D1 XOR D2 XOR D3 XOR D4 XOR D5.

If (say) D1 disappears, can figure out what it was:

D1 = CK XOR D2 XOR D3 XOR D4 XOR D5

(Prove this: A = B XOR C, then B = A XOR C and C = A XOR B.
1 = 1 XOR 0 => 1 = 1 XOR 0 and 0 = 1 XOR 1
1 = 0 XOR 1 => 0 = 1 XOR 1 and 1 = 1 XOR 0
0 = 1 XOR 1 => etc.
0 = 0 XOR 0)

So if one disk drops out, keep accessing data on it using XOR of other 5. Recover all disk pages on disk in same way. This takes a lot of time to recover, but it DOES save disk media.

10.10 TPC-A Benchmark

The TPC-A Benchmark is now out of date. Newer TPC-C Benchmark: more complex.

See Fig 10.16, pg. 686. Size of tables determined by TPS.

See Fig 10.17. pg. 687. All threads do the same thing. Run into each other in concurrency control because of Branch table and History table.

Benchmark specifies how many threads there are, how often each thread runs a Tx, costs of terminals, etc.

On a good system, just add disks until use 95% of CPU. On a bad system, run into bottlenecks.

Ultimate measure is TPS and $/TPS