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

Chapters 1 and 2

Class 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 CLASS — 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 classes.

The subjects have prerequisites. The classes are offered in given periods in specific rooms. Students have schedules (each term), drop and add classes, 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 class-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 class of-

fering, then the class offering and student are related in a student schedule and a distinct class 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.

Class 2.

Assignment 1: GET BUDDY PHONE #, IF MISS CLASS — 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
a 1	b 1	c 1	d 1

a 1	b 2	c 2	d 1
a 2	b 2	c 1	d 1
a 2	b 1	c 2	d 1

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.

Class 3.

Assignment Read to end of Chapter 2. Homework: All undotted problems through end of Chapter 2, due at Class 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 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 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, . . . 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 $\times$ S),

or could write instead:

T1 := (R INTERSECT S)
T2 := (R $\times$ 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) = $R.A_1 . . . R.A_n$ $S.B_1 . . . S.B_n$. 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) = $A_1, . . . , A_n$, project R on subset of columns T := $R[A_{i_1}, . . ., A_{i_k}]$, 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.

$A_i \propto A_j$ or $A_i \propto$ a, where A_i and A_j are attributes of R, a is a constant. $\propto$ 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]

Class 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
b 1	c_1
b 2	c 2
b 3	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.ϒ

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 largest possible set of rows such that T S is contained in R. (Exercise 3.) **Make Analogy to Integer Division**.

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

T

A	B

c1
c2
c3
c4

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.

Class 5.

Hand in hw 1 on class 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.

$A \cap B = A - (A - B)$. 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: $(R[A_1 \ldots A_n]$ X S - R) $[A_1 \ldots A_n]$

Claim now we have:

$(R[A_1 \ldots A_n] - ((R[A_1 \ldots A_n]$ X S) - R) $[A_1 \ldots A_n])$ 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]

Class 6. Hand in hw 1.
Assignment: reading through Section 3.5. UNDOTTED exercises at end of Chapter 3 through 3.7. Due Class 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
b 1	c_1
b 2	c 2
b 3	c_3
b 4	c 4

What is the join, R JOIN S?

R JOIN S

A	B	C
a_1	b 1	c_1
a_2	b 2	c 2

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	b 1	c_1
a_2	b 2	c 2
a 3	b 5	null
null	b 3	c 3
null	b 4	c 4

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 θ S.B, where θ is <, <=, <>, >, >=, anything but =.

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