Lecture Notes for Database Systems

Patrick E. O'Neil and Elizabeth O'Neil

Class 14. Exam 1

Class 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 class 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, Class_rooms, Courses, Course_sections (different offerings of a single course, generally at different times by different instructors), Class_periods.

Note that we Capitalize entity names.

Class_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).

Class_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 classroom during a class period, so time interval is treated just like classroom.

Normally, an entity such as Class_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 classroom, 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, $E_2, \ldots, E_m$, (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.

Class 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 class, 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.

Class 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.

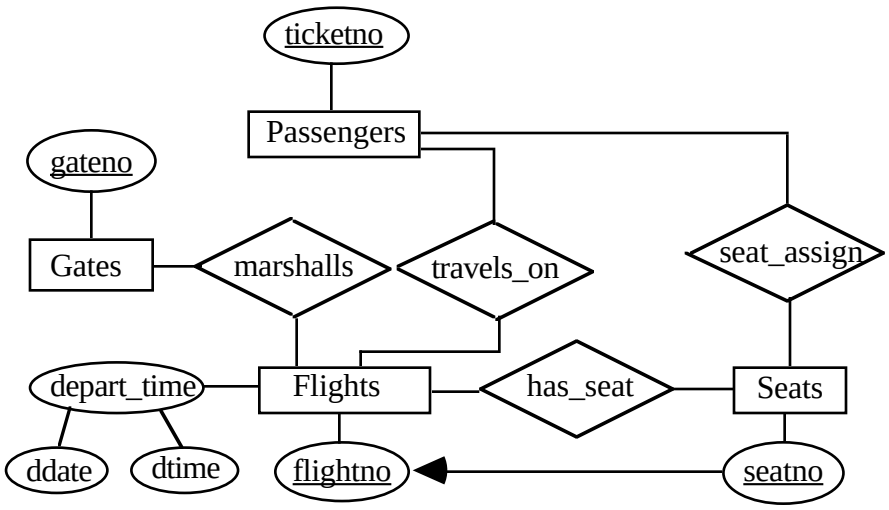


Fig. 6.13, pg. 351 (but leave out marshalls and travels_on), and note Gates off by itself. But clearly passengers go to a gate to meet a flight.

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 ca-porders 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.

Class 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 ≠ 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 –> A B C

(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 not quite minimal, 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.

Class 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 $\prod$ 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 $\prod$ 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] $\subset$ X[I]) so X[I-1] $\neq$ 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] $\subset$ 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. So can leave (4) out.

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 $\wp$ 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.)

IF WE DID REDUCE 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.

Class 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 Class: 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.

Class 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, class?

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?

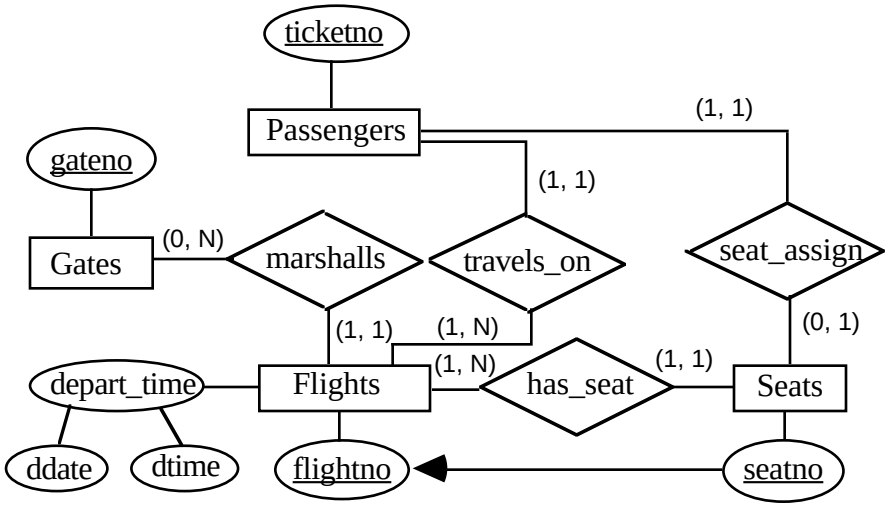
Class 21.

Last Class.

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.