CS352 Lecture - Conceptual Relational Database Design

last revised September 20, 2006

Objectives:

1. To define the concepts "functional dependency" and "multivalued dependency"
2. To show how to find the closure of a set of FD's and/or MVD's
3. To define the various normal forms and show why each is valuable
4. To show how to normalize a design
5. To discuss the "universal relation" and "ER diagram" approaches to database design.

Materials:

1. Projectable of Armstrong's Axioms and additional rules of inference (p. 279f)
2. Projectable of algorithm for computing F+ from F (Fig. 7.8 p. 280)
3. Projectable of algorithm for computing closure of an attribute (Fig 7.9 p. 281)
4. Projectable of 3NF algorithm (Fig 7.13 p. 291)
5. Projectable of BCNF algorithm (Fig. 7.12 p. 289)
6. Projectable of rules of inference for FD's and MVD's (C-2, 3)
7. Projectable of 4NF algorithm (Fig. 7.17 p. 297)

I. Introduction

A. We have already looked at some issues arising in connection with the design of relational databases. We now want to take the intuitive concepts and expand and formalize them.

B. We will base most of our examples in this series of lectures on a simplified library database similar to the one we used in our introduction to relational algebra and SQL lectures, with some modifications

1. We will deal with only book and borrower entities and the checked_out relationship between them (we will ignore the reserve_book and employee tables)

2. We will add a couple of attributes to book (which will prove useful in illustrating some concepts)

a) We will allow for the possibility of having multiple copies of a given book, so we include a copy_number attribute

b) We will include an accession_number attribute for books. (The accession_number is a unique number - almost like a serial number - assigned to a book when it is acquired)

C. There are two major kinds of problems that can arise when designing a relational database. We illustrate each with an example.

1. There are problems arising from including TOO MANY attributes in one relation scheme.

   Example: Suppose a naive user purchases a commercial database product and designs a database based on the following scheme. Note that it incorporates all of the attributes of the separate tables relating to borrowers and books from our SQL examples into a single table - plus two new ones just added)

```
Everything( borrower_id,  last_name,  first_name, //  from  borrower
            call_number,  copy_number,
            accession_number,  title,  author,     //  from  book
            date_due)                              //  from  checked_out
```

   (Don't laugh - people do this!)

   a) Obviously, this scheme is useful in the sense that a desk attendant could desire to see all of this information at one time.

   b) But this makes a poor relation scheme for the conceptual level of database design. (It might, however, be a desirable view to construct for the desk attendant at the view level, using joins on conceptual relations.)

   c) As we've discussed earlier, this scheme exhibits a number of anomalies. Let's identify some examples.

      ASK CLASS

      (1) Update anomalies:

         If a borrower has several books out, and the borrower's name changes (e.g. through marriage), failure to update all the tuples creates inconsistencies.

      (2) Insertion anomalies:

         We cannot store a book in the database that is not checked out to some borrower. (We could solve this one by storing a null for the borrower_id, though that's not a desirable solution.)

         We cannot store a new borrower in the database unless the borrower has a book checked out. (No good solution to this.)

(3) Deletion anomalies

When a book is returned, all record of it disappears from the database if we simply delete the tuple that shows it checked out to a certain borrower. (Could solve by storing a null in the borrower_id instead.)

If a borrower returns the last book he/she has checked out, all record of the borrower disappears from the database. (No good solution to this.)

d) Up until now, we have given intuitive arguments that designing the database around a single table like this is bad - though not something that a naive user is incapable of! What we want to do in this series of lectures is formalize that intuition into a more comprehensive, formal set of tests we can apply to a proposed database design.

2. Problems of the sort we have discussed can be solved by DECOMPOSITION: the original scheme is decomposed into two or more schemes, such that each attribute of the original scheme appears in at least one of the schemes in the decomposition (and some attributes appear in more than one).

However, decomposition must be done with care, or a new problem arises.

Example: Suppose our naive user overhears a couple of CS352 students talking at lunch and decides that, since decomposition is good, lots of decomposition is best - and so creates the following set of schemes:

Borrower(borrower_id, last_name, first_name)
Book(call_number, copy_number, accession_number, title, author)
Checked_out(date_due)

a) This eliminates all of the anomalies we listed above - so it must be good - right?

ASK CLASS for the problem

b) There is now no way to represent the fact that a certain borrower has a certain book out - or that a particular date_due pertains to a particular Borrower/Book combination.

c) This decomposition is an example of what is called a LOSSY-JOIN decomposition.

(1) To see where this term comes from, suppose we have two borrowers and two books in our database, each of which is checked out - i.e, using our original scheme, we would have the following single table:

```
20147  1  17  cat  charlene AB123.40  Karate   elephant 2002-11-15
89754  1  24  dog  donna    LM925.04  Cat  Cook dog      2002-11-10
```

(2) Now suppose we decompose this along the lines of the proposed decomposition. We get the following three tables.

```
20147  cat  charlene
89754  dog  donna

AB123.40  1  17  Karate  elephant
LM925.04  1  24  Cat  Cook  dog

2002-11-15
2002-11-10
```

(3) Finally, we attempt to reconstruct our original table, by doing a natural join of our decomposed tables.

Borrower |X| Book |X| Checked_Out

(Note that, in this case, the natural join is equivalent to cartesian join because the tables being joined have no attributes in common.)

What do we get?

ASK

8 rows: each consisting of one of the two borrowers, one of the two books, and one of the two due data

(4) We say that the result is one in which information has been <u>lost</u>. At first, that sounds strange - it appears that information has actually been gained, since the new table is 4 times as big as the original, with 6 extraneous rows. But we call this an <u>information</u> loss because

(a) Any table is a subset of the cartesian join of the domains of its attributes.

(b) The information in a table can be thought of as the knowledge that certain rows from the set of potential rows are / are not present.

(c) When we lose knowledge as to which rows from the cartesian join are actually present, we have lost information.

(5) We say that a decomposition of a relation scheme R into two or more schemes R1, R2 ... Rn (where R = R1 U R2 U .. U Rn) is a lossless-join decomposition if, for every legal instance r of R, decomposed into instances r1, r2 .. rn of R1, R2 .. Rn, it is always the case that

r = r1 |x| r2 |x| ... |x| rn

(Note: it will always be the case that r is a SUBSET of r1 |x| r2 |x| ... |x| rn. The relationship is lossy if the subset is a proper one.)

(6) Exercise: can you think of a lossless-join decomposition of Everything that also eliminates the anomalies?

ASK

If we kept Borrower and Book as above, and made Checked_out be on the scheme (borrower_id, call_number, date_due), the decomposition onto Borrower, Book and Checked_Out would be lossless join, as desired.

3. There is actually another problem that can result from over-doing decomposition; however, we cannot discuss it until we have introduced the notion of functional dependencies.

D. First, some notes on terminology that we will use in this lecture:

1. A relation scheme is the set of attributes for some relation - e.g. the scheme for Borrower is { borrower_id, last_name, first_name}.

   We will use upper-case letters, or Greek letters (perhaps followed by a digit), to denote either complete relation schemes or subsets. Typically, we will use something like "R" or "R1" to refer to the scheme for an entire relation, and a letter like "A" or "B" or α or β to refer to a subset.

2. A relation is the actual data stored in some scheme.

   We will use lower-case letters (perhaps followed by a digit) to denote actual relations - e.g. we might use "r" to denote the actual relation whose scheme is "R", and "r1" to denote the actual relation whose scheme is "R1".

3. A tuple is a single actual row in some relation.

   We will also use lower-case letters (perhaps followed by a digit) to denote individual tuples - often beginning with the letter "t" - e.g. "t1" or "t2".

II. Functional Dependencies

A. Though we have not said so formally, what was lurking in the background of our discussion of decompositions was the notion of FUNCTIONAL DEPENDENCIES. A functional dependency is a property of the UNDERLYING REALITY which we are modeling, and affects the way we model it.

1. Definition: for some relation-scheme R, we say that a set of attributes B (B a subset of R) is functionally dependent on a set of attributes A (A a subset of R) if, for any legal relation on R, if there are two tuples t1 and t2 such that t1[A] = t2[A], then it must be that t1[B] = t2[B].

   (This can be stated alternately as follows: there can be no two tuples t1 and t2 such that t1[A] = t2[A] but t1[B] <> t2[B].)

2. We denote such a functional dependency as follows:

   $A \rightarrow B$ (Read: A determines B)

   Example: We assume that a borrower_id uniquely determines a borrower (that's the whole reason for having it), and that any given borrower has exactly one last name and one first name. Thus, we have the functional dependency:

   borrower_id $\rightarrow$ last_name, first_name

   [Note: this does not necessarily have to hold. We could conceive of a design where, for example, a borrower_id could be assigned to a family, with several individuals able to use it However, in the scheme we are developing, we will assume that the FD above does hold.]

3. Let's list some functional dependencies for the reality underlying a simplified library database scheme, which includes the attributes listed below, organized into tables in some appropriate way.

   Reminder: we've added two attributes to the list used in previous examples. These will be important to allow us to illustrate some concepts.

   borrower_id
   last_name
   first_name
   call_number
   copy_number
   accession_number
   title
   author
   date_due

   ASK FOR FD'S

   borrower_id → last_name, first_name
   call_number → title
   call_number, copy_number → accession_number
   call_number, copy_number → borrower_id, date_due *
   * If a certain book is not checked out, then, of course, it has no borrower_id or date_due (they are null)
   accession_number → call_number, copy_number

   (Note: these FD's imply a lot of other FD's - we'll talk about this shortly)

4. What about the following - should it be a dependency?

   call_number → author

   ASK

   a) Obviously, this is not true in general - books can have multiple authors.

   b) At the same time, it is certainly not the case that there is NO relationship between call_number and author.

   c) The relationship that exists is one that we will introduce later, called a multi-valued dependency.

d) For now, we will make the simplifying assumption that each book has a single, principal author which is the only one listed in the database. Thus, we will assume that, for now:

call_number → author

holds. (Later, we will drop this assumption and this FD)

B. The first step in using functional dependencies to design a database is to LIST the functional dependencies that must be satisfied by any instance of the database.

1. We begin by looking at the reality being modeled, and make explicit the dependencies that are present in it. This is not always trivial.

a) Example: earlier, we considered the question about whether we should include

call_number → author

in our set of dependencies

b) Example: Should we include the dependency

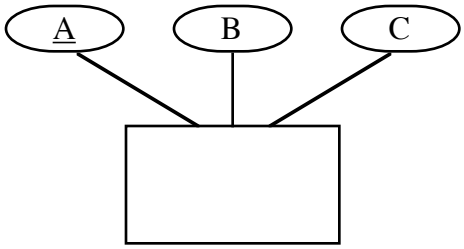
last_name, first_name → borrower_id

in our set of dependencies used for our design?

The answer depends on some assumptions about peoples' names, and on whether we intend to store a full name (last, first, mi, plus suffixes such as Sr, Jr, III etc.) For our examples, we will <u>not</u> include this dependency.

2. Note that there is a correspondence between FD's and symbols in an ER diagram - so if we start with an ER diagram, we can list the dependencies in it.

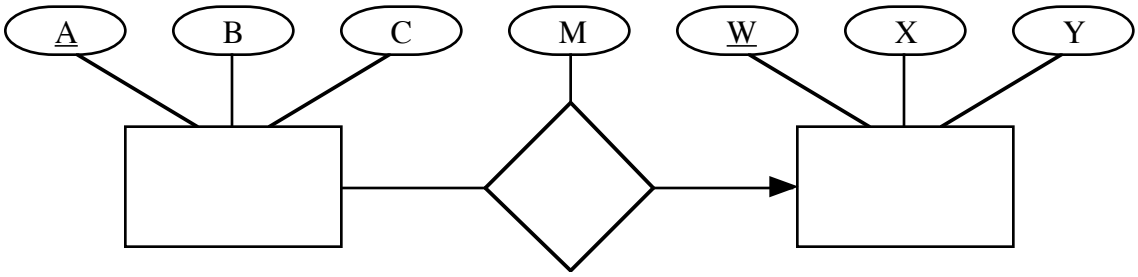
a) What does the following pattern in an ER diagram translate into in terms of FD's?



ASK

A → BC

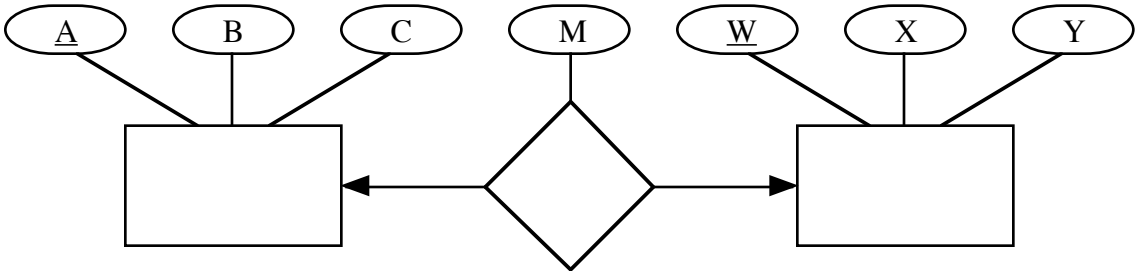
b) How about this?



ASK

A → BCMWXY
W → XY

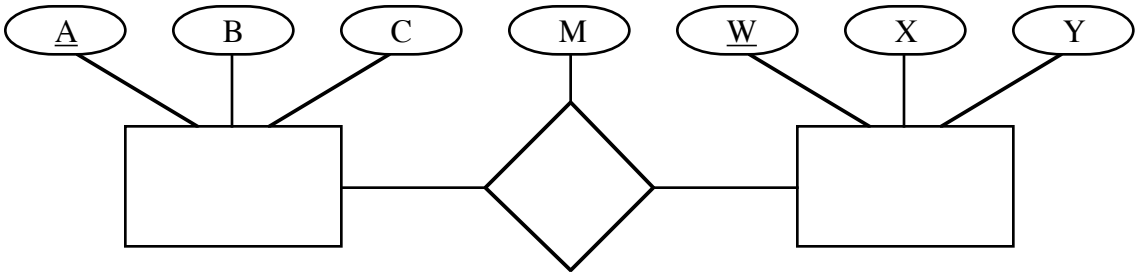
c) How about this?



ASK

A → BCMWXY
W → XYMABC

d) Or this?



ASK

A → BC
W → XY
AW → M

e) Thus, the same kind of thinking that goes into deciding on keys and one-to-one, one-to-many, or many-to-many relationships in ER diagrams goes into identifying dependencies in relational schemes.

C. We then generate from this initial listing of dependencies the set of functional dependencies that they IMPLY.

1. Example: given the dependencies

call_number, copy_number → borrower_id
borrower_id → last_name, first_name

a) The following dependency also must hold

call_number, copy_number → last_name, first_name

(The call number of a (checked out) book determines the name of the borrower who has it)

b) We can show this from the definition of functional dependencies by using proof by contradiction, as follows:

(1) We want to show that, given any two legal tuples t1 and t2 such that t1[call_number, copy_number] = t2[call_number, copy_number], it must be the case that t1[last_name, first_name] = t2[last_name, first_name].

(2) Suppose there are two tuples t1 and t2 such that this does not hold - e.g.

t1[call_number, copy_number] = t2[call_number, copy_number] and t1[last_name, first_name] ≠ t2[last_name, first_name]

(3) Now consider the borrower_id values of t1 and t2.

If it is the case that

t1[borrower_id] ≠ t2[borrower_id]

then the FD call_number, copy_number → borrower_id is not satisfied

But if it is the case that

t1[borrower_id] = t2[borrower_id]

then the FD borrower_id → last_name, first_name is violated.

(4) Either way, if t1 and t2 violate

call_number, copy_number → last_name, first_name

then they also violate one or the other of the given dependencies.

QED

2. Formally, if F is the set of functional dependencies we develop from the logic of the underlying reality, then F+ (the transitive closure of F) is the set consisting of all the dependencies of F, plus all the dependencies they imply.

To compute F+, we can use certain rules of inference for dependencies

a) A minimal set of such rules of inference is a set known as Armstrong's axioms [Armstrong, 1974]. These are listed in the text on page 279-280 (PROJECT)

(1) Example of reflexivity:

last_name, first_name → last_name

Note: this is the only rule that lets us "start with nothing" and still create FD's. Dependencies created using this rule are called "trivial dependencies" because they always hold, regardless of the underlying reality.

Definition: Any dependency of the form $\alpha \rightarrow \beta$, where $\alpha \supseteq \beta$ is called trivial.

(2) Example of augmentation:

Given:

borrower_id → last_name, first_name

It follows that

borrower_id, call_number→ last_name, first_name, call_number

(3) Example of transitivity: the above proof that

call_number, copy_number → last_name, first_name

b) Note that this is a minimal set (a desirable property of a set of mathematical axioms.) However, the task is made easier by using certain additional rules of inference that follow from Armstrong's axioms. These are also listed on pages 279-280.

PROJECT

(1) Example of Union rule:

Since call_number → title

and

call_number → author

it follows that

call_number → title, author

(2) Example of the Decomposition rule:

Since borrower_id → last_name, first_name,

it follows that

borrower_id → last_name

and

borrower_id → first_name

(3) Example of the Pseudo-transitivity rule: (Note: to illustrate this concept we will have to make some changes to our assumptions, just for the sake of this one illustration)

Suppose we require book titles to be unique - i.e. we require

title → call_number (but not, of course, title → copy_number!)

Then, given

call_number, copy_number → accession_number, borrower_id, date_due

by pseudo-transitivity, we would get

title, copy_number → accession_number, borrower_id, date_due

c) Each of these additional rules can be proved from the ones in the basic set of Armstrong's axioms - e.g.

(1) Proof of the union rule

Given: $\alpha \rightarrow \beta$, $\alpha \rightarrow \gamma$

Prove: $\alpha \rightarrow \beta\gamma$

Proof:

$\alpha\alpha \rightarrow \alpha\beta$	(augmentation of first given with α)
$\alpha \rightarrow \alpha\beta$	(since we are working with sets, $\alpha\alpha = \alpha$)
$\alpha\beta \rightarrow \beta\gamma$	(augmentation of second given with β)
$\alpha \rightarrow \beta\gamma$	(transitivity)

(2) Proof of the decomposition rule:

Given: $\alpha \rightarrow \beta\gamma$

Prove: $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$

Proof: $\beta\gamma \rightarrow \beta$ and $\beta\gamma \rightarrow \gamma$ (by reflexivity)
$\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$ (by transitivity using given)

(3) Proof of the psuedo-transitivity rule:

Given: $\alpha \rightarrow \beta$ and $\beta\gamma \rightarrow \delta$

Prove: $\alpha\gamma \rightarrow \delta$

Proof: $\alpha\gamma \rightarrow \beta\gamma$ (augmentation of first given with γ)
$\alpha\gamma \rightarrow \delta$ (transitive rule using second given)

d) Note that the union and decomposition rules, together, give us some choices as to how we choose to write a set of FD's.

For example, given the FD's

$\alpha \rightarrow \beta\gamma$ and $\alpha \rightarrow \delta\varepsilon$

We could choose to write them as

$\alpha \rightarrow \beta$
$\alpha \rightarrow \gamma$
$\alpha \rightarrow \delta$
$\alpha \rightarrow \varepsilon$

or as

$\alpha \rightarrow \beta\gamma\delta\varepsilon$

(or any one of a number of other ways)

Because the latter form requires a lot less writing, we used it when listing our initial set of library dependencies, and we will use it in many of the examples which follow.

e) In practice, F+ can be computed algorithmically. An algorithm is given in the text for determining F+ given F:

PROJECT - Figure 7.8 on page 280

f) Note that, using an algorithm like this, we end up with a rather large set of FD's. (Just the reflexivity rule alone generates lots of FD's.)

For this reason, it is often more useful to consider finding the closure of a given attribute, or set of attributes. (If we apply this process to all attributes appearing on the left-hand side of an FD, we end up with all the interesting FD's)

(1) The text gives an algorithm for this:

PROJECT - Figure 7.9 on page 281

(2) Example of applying the algorithm on page 281 to left hand sides of each of the FD's for our library:

Starting set (F):

borrower_id → last_name, first_name
call_number → title
call_number, copy_number → accession_number, borrower_id, date_due
accession_number → call_number, copy_number
call_number → author

(a) Compute borrower_id +:

Initial: { borrower_id }

On first iteration through loop, add

last_name
first_name

Additional iterations don't add anything

∴ borrower_id → borrower_id, last_name, first_name

(b) Compute call_number +

Initial: { call_number }

On first iteration through loop, add

title
author

Additional iterations don't add anything

∴ call_number → call_number, title, author

(c) Compute call_number, copy_number +

Initial: { call_number, copy_number }

On first iteration through loop, add

title
accession_number
borrower_id
date_due
author

On second iteration through loop, add

last_name
first_name

∴ call_number, copy_number → call_number, copy_number, title, accession_number, borrower_id, date_due, author, last_name, first_name

(d) Compute accession_number +:

Initial: { accession_number }

On first iteration thorough loop, add

call_number
copy_number

On second iteration through loop, add

title
borrower_id
date_due
author

On third iteration through loop, add

last_name
first_name

∴ accession_number → accession_number, call_number, copy_number, title, borrower_id, date_due, author, last_name, first_name

(3) Note that generating the closure of an attribute / set of attributes provides an easy way to test if a given set of attributes is a superkey: does/do the attribute(s) in the set determine <u>every</u> attribute in the scheme?

(a) Both { call_number, copy_number } and { accession_number } would qualify as superkeys for our entire scheme (if it were represented as a single table) - and therefore for any smaller table in which they occur.

(b) { borrower_id } is a superkey for any scheme consisting of just attributes from { borrower_id, last_name, first_name }

(c) If we had a scheme for which no set of attributes appearing on the left hand side of an initial dependency were a superkey, we could find a superkey by combining sets of attributes to get a set that determines everything.

3. Given that we can infer additional dependencies from a set of FD's, we might ask if there is some way to define a minimal set of FD's for a given reality.

a) We say that a set of FD's Fc is a CANONICAL COVER for some set of dependencies F if:

(1) Fc implies F and F implies Fc - i.e. they are equivalent.

(2) No dependency in Fc contains any extraneous attributes:

(3) No two dependencies have the same left side (i.e. the right sides of dependencies with the same left side are combined)

b) It turns out to be easier - I think - to find a canonical cover by first writing F as a set of dependencies where each has a single attribute on its <u>right</u> hand side - then eliminate redundant dependencies (dependencies implied by other dependencies) - then combine dependencies with the same left-hand side.

Example: Find a canonical cover for the dependencies in our library database:

(1) Start with the following closure of the various attributes we found earlier:

borrower_id → borrower_id, last_name, first_name
call_number → call_number, title, author
call_number, copy_number → call_number copy_number, title, accession_number, borrower_id, date_due, author, last_name, first_name
accession_number → accession_number, call_number, copy_number, title, borrower_id, date_due, author, last_name, first_name

(2) Rewrite with a single attribute on the right hand side of each

borrower_id → borrower_id
borrower_id → last_name
borrower_id → first_name
call_number → call_number
call_number → title
call_number → author
call_number, copy_number → call_number
call_number, copy_number → copy_number
call_number, copy_number → title
call_number, copy_number → accession_number
call_number, copy_number → borrower_id
call_number, copy_number → date_due
call_number, copy_number → author
call_number, copy_number → last_name
call_number, copy_number → first_name
accession_number → accession_number
accession_number →call_number
accession_number →copy_number
accession_number →title
accession_number →borrower_id
accession_number →date_due
accession_number →author
accession_number →last_name
accession_number →first_name

(3) Now eliminate the trivial dependencies

(Cross out on list)

(4) There are dependencies in this list which are implied by other dependencies in the list, and so should be eliminated. Which ones?

ASK

- call_number, copy_number → title
  call_number, copy_number → author

(Since the same RHS appears with only call_number on the LHS)

- accession_number → title
  accession_number → author

(These are implied by the transitive rule given that accession_number → call_number and call_number determines these).

• call_number, copy_number → last_name
  call_number, copy_number → first_name

(These are implied by the transitive rule given that call_number, copy_number → borrower_id and borrower_id determines these)

• accession_number → last_name
  accession_number → first_name

(These are implied by the transitive rule given that accession_number → borrower_id and borrower_id determines these)

• Either one of the following - but not both!

  call_number, copy_number → borrower_id
  call_number, copy_number → date_due

or

  accession_number → borrower_id
  accession_number → date_due

(Either set is implied by the transitive rule from the other set given accession_number → call_number, copy_number or call_number, copy_number → accession_number.)

(Assume we keep the ones with call_number, copy_number on the LHS)

(5) Result after eliminating redundant dependencies:

borrower_id → last_name
borrower_id → first_name

call_number → title
call_number → author

call_number, copy_number → accession_number
call_number, copy_number → borrower_id
call_number, copy_number → date_due

accession_number → call_number
accession_number → copy_number

(6) Rewrite in canonical form by combining dependencies with the same left-hand side:

borrower_id → last_name, first_name
call_number → title, author

call_number, copy_number → accession_number,borrower_id, date_due
accession_number → call_number, copy_number

c) Unfortunately, for any given set of FD's, the canonical cover is not necessarily unique - there may be more than one set of FD's that satisfies the requirement.

Example: For the above, we could have kept
accession_number → borrower_id, date_due
and dropped
call_number, copy_number → borrower_id, date_due.

D. Functional dependencies are used in two ways in database design

1. They are used as a guide to DECOMPOSING relations. For example, the problem with our original, single-relation scheme was that there were too many functional dependencies within one relation.

a) last_name and first_name depend only on borrower_id

b) title and author depend only on call_number

c) if a book is checked out, then borrower_id and date_due depend on call_number, copy_number

d) We run into a problem when all of these FD's appear in a single table - we will formalize this soon.)

2. They are used as a means of TESTING decompositions.

a) We can use the closure of a set of FD's to test a decomposition to be sure it are lossless join.

If we decompose a scheme R with set of dependencies F into two schemes R1 and R2, the resultant decomposition is lossless join iff

$(R1 \cap R2) \rightarrow R1$ is in F+

or

$R1 \cap R2 \rightarrow R2$ is in F+

(or both)

b) We also want to produce DEPENDENCY-PRESERVING decompositions wherever possible.

(1) A dependency-preserving decomposition allows us to test a new tuple being inserted into some table to see if it satisfies all relevant functional dependencies without doing a join.

Example: If our decomposition includes a scheme including the following attributes:

call_number, copy_number, accession_number ...

then when we are inserting a new tuple we can easily test to see whether or not it violates the following dependencies

accession_number → call_number, copy_number
call_number, copy_number → accession_number

Now suppose we decomposed this scheme in such a way that no table contains all three of these attributes - i.e. into something like:

call_number, accession_number ....
and
copy_number, accession_number ...

When inserting a new book entity (now as two tuples in two tables), we can still test

accession_number → call_number, copy_number

by testing each part of the right hand side separately for each table - but the only way we can test whether

call_number, copy_number → accession_number

is satisfied by a new entity is by joining the two tables to make sure that the same call_number and copy_number don't appear with a different accession_number

(2) To test whether a decomposition is dependency-preserving, we introduce the notion of the <u>restriction</u> of a set of dependencies to some scheme. Basically, the restriction of a set of dependencies to some scheme is the subset which have the property that all of the attributes of the dependency are contained in the scheme

Ex: The restriction of { A → B, A →C, A → D to (ABD) is { A →B, A →D }

(3) A decomposition is dependency preserving if the <u>transitive closure</u> of the original set is equal to the <u>transitive closure</u> of the set of restrictions to each scheme.

Example: if we have a scheme (ABCD) with dependencies

A → B
B → CD

(a) The decomposition into

(AB) (BCD)

is both lossless-join and dependency preserving.

(b) So is the following decomposition

(AB) (BC) (BD)

because

B → C and B → D together imply B → CD

(c) However, the following decomposition, while lossless join, is not dependency-preserving

(AB) (ACD)

i) The transitive closure of the original set of dependencies includes A → (all combinations of A,B,C,D) and
B → (all combinations of B, C, D)

ii) The restriction of this to the decomposed schemes is A → (all combinations of AB) and
A → (all combinations of A,C,D)

iii) Since B → CD is not a member of this restriction, it cannot be tested without doing a join

Example: suppose we have the tuple a1 b1 c1 d1 in the table, and try to insert a2 b1 c2 d2.

This violates B → CD. However, we cannot discover this fact unless we join the two tables, since B does not appear in the same table with C or D

(4) Again, suppose we have the scheme (ABCD) with dependencies

A → B
A → C
B → CD

If we decompose into

(AB) (BCD)

The decomposition is lossless join and dependency-preserving, even though we can't test A → C directly without doing a join, because A → C is implied by the dependencies A → B and B → C which we can test - it is therefore in the transitive closure of the restriction of the original set of dependencies to the decomposed scheme.

E. Note that the notions of superkey, candidate key, and primary key we developed earlier can now be stated in terms of functional dependencies.

1. Given a relation scheme R, a set of attributes K (K subset R) is a SUPERKEY iff K → R. (And therefore by the decomposition rule each individual attribute in R.)

2. A K is a CANDIDATE key iff there is no proper subset of K that is a superkey.

   a) A superkey consisting of a single attribute is always a candidate key.

   b) If K is composite, then for K to be a candidate key it must be the case that for each proper subset of K there is some attribute in R that is NOT functionally dependent on that subset, though it is on K.

3. The PRIMARY KEY of a relation scheme is the candidate key chosen for that purpose by the designer.

4. Since a relation is a set, it must have a superkey (possibly the entire set of attributes.) Therefore, it must have one or more candidate keys, and a primary key can be chosen. We assume, in all further discussions of design, that each relation scheme we work with has a primary key.

   Note: In our discussion of the ER model, we introduced the notion of a <u>weak entity</u> as an entity that has no superkey. However, the process by which we convert to tables guarantees that the corresponding table will have a superkey, since we include in the table the primary key(s) of the entity/entities on which the weak entity depends.

5. In the discussions that follow, we will say that an attribute is a KEY ATTRIBUTE if it is a candidate key or part of a candidate key. (Not necessarily the primary key.) Some writers call a key attribute a PRIME ATTRIBUTE.

III. Using Functional Dependencies to Design Database Schemes

A. Three major goals:

1. Avoid redundancies and the resulting update, insertion, and deletion anomalies, by decomposing schemes as necessary.

2. Ensure that all decompositions are lossless-join.

3. Ensure that all decompositions are dependency-preserving.

4. However, all three may not be achievable at the same time in all cases, in which case some compromise is needed. One thing we never compromise, however is lossless-join, since that involves the destruction of information. We may have to accept some redundancy to preserve dependencies, or we may have to give up dependency-preservation in order to eliminate all redundancies. (We'll see an example of this later.)

B. To ensure the first goal, database theorists have developed a hierarchy of NORMAL FORMS, plus a set of decomposition rules that can be used to convert a database not in a given normal form into one that is. (The decomposition rules ensure the lossless-join property, but not necessarily the dependency-preserving property.)

1. We will consider the normal forms as they were developed historically.
2. We will use the library database and set of FD's we just developed, and will progressively normalize it to 4NF.
3. As noted in the book, it is most common, in practice, to go straight to the highest normal form desired, rather than working through the hierarchy of forms. We present the forms in this order only for pedagogical reasons.

C. First Normal Form (1NF):

1. A relation scheme R is in 1NF iff, for each tuple t in R, each attribute of t is atomic - i.e. it has a SINGLE, NON-COMPOSITE VALUE

2. This rules out:
   a) Repeating groups.
   b) Composite fields in which we can access individual components e.g. dates that can be either treated as unit or can have month, day and year components accessed separately.

3. This is our motivation, at the present time, for requiring

   call_number → author

   - i.e. requiring that each book have a single author

   (If we didn't want to require that, we could still produce a 1NF scheme by "flattening" our scheme. This would result, for example, in having three book tuples for our course text - one each for Korth, Silberschatz, and Sudarshan.)

4. 1NF is desirable for most applications, because it guarantees that each attribute in R is functionally dependent on the primary key, and simplifies queries.

   However, there are some applications for which atomicity may be undesirable - e.g. keyword fields in bibliographic databases. There are some who have argued for <u>not</u> requiring normalization in such cases, though the pure relational model certainly does.

D. Second Normal Form (2NF):

1. A 1NF relation scheme R is in 2NF iff each non-key attribute of R is FULLY functionally dependent on each candidate key. By FULLY functionally dependent, we mean that it is functionally dependent on the whole candidate key, but not on any proper subset of it.

   NOTE: We only require attributes not part of a candidate key to be fully functionally dependent on each candidate key. An attribute that IS part of a candidate key CAN be dependent on just part of some other candidate key. We address this situation in conjunction with BCNF.)

2. Example: Suppose we had the following single scheme, which incorporates all of our attributes into a single table.

   Everything(borrower_id, last_name, first_name,
   call_number, copy_number, accession_number, title,
   author, date_due)

   a) What would our candidate keys be?

      ASK

      From the FD analysis we just did, we see that the candidate keys are (call_number, copy_number) and (accession_number).

b) One of our candidate keys is composite. Do we then have any attributes that depend only on call_number or only on copy_number?

ASK

Yes - title and author.

c) This means that we cannot record the fact that QA76.9.D3 S5637 is the call number for "Database System Concepts 4th ed" unless we actually own a copy of the book. (Maybe this is a problem, maybe not.) Moreover, if we do own a copy and it is lost, and we delete it from the database, then we have to re-enter this information when we get a new copy.

3. Any non-2NF scheme can be made 2NF by a decomposition in which we factor out the attributes that are dependent on only a portion of a candidate key, together with the portion they depend on.

   For example, in this case we would factor as follows

   Book_info(call_number, title, author)

   and

   Everything_else(borrower_id, last_name, first_name,
   call_number, copy_number, accession_number, date_due)

   This is now 2NF.

4. Observe that any 1NF relation scheme which does NOT have a COMPOSITE primary key is, of necessity, in 2NF.

5. 2NF is desirable because it avoids repetition of information that is dependent on part of the primary key, but not the whole key, and thus prevents various anomalies.

E. Third Normal Form (3NF):

1. A 2NF relation scheme R is in 3NF iff no non-key attribute of R is transitively-dependent (in a nontrivial way) on a candidate key through some other non-key attribute(s).

   NOTE: We only forbid attributes not part of a candidate key to be transitively dependent on the primary key. An attribute that IS part of a candidate key CAN be transitively dependent on the primary key. (We address this situation in conjunction with BCNF.

2. Example: Consider the Everything_else scheme we just derived, with candidate keys call_number, copy_number and accession_number. While this is 2NF, is is not 3NF, since certain attributes are dependent on borrower_id, which is in turn dependent on the candidate key call_number, copy_number. That is, we have:

   call_number, copy_number → borrower_id

   borrower_id → last_name
   borrower_id → first_name

   which are a transitive dependencies on the candidate key. This leads to anomalies like:

   a) We cannot record information about a borrower who does not have a book checked out.

   b) If a a borrower who has several books checked out changes his/her name, we must update several tuples.

   c) If a borrower has only one book checked out and returns it, all information about the borrower's name is also deleted.

3. Any non-3NF scheme can be decomposed into 3NF schemes by factoring out the attributes that are transitively-dependent on some non-key attribute, and putting them into a new scheme along with the attribute(s) they depend on.

   Example: We can decompose Everything_else into

   Borrower(borrower_id, last_name, first_name)

   Everything_left(borrower_id, call_number, copy_number,
   accession_number, date_due)

   which are now 3NF

4. Any non-3NF relation can be decomposed in a lossless-join, dependency preserving way. An informal approach like the one we just used will often work, but there is also a formal algorithm that can be used

   PROJECT: Figure 7.13 on page 291

   Note that this is actually a construction algorithm, not a decomposition algorithm - i.e. we start with nothing and construct a set of schemes, instead of starting with a scheme and decomposing it.

   Example: construct a 3NF scheme for our library database

a) Start with our canonical cover:

borrower_id → last_name, first_name
call_number → title, author
call_number, copy_number → accession_number,borrower_id, date_due
accession_number → call_number, copy_number

b) Each of the first three dependencies leads to adding a schema.

(borrower_id, last_name, first_name)
(call_number, title, author)
(call_number, copy_number, accession_number, borrower_id, date_due)

c) The fourth dependency does not lead to adding a schema, since all of its attributes occur together in the third scheme.

d) The set of schemas includes a candidate key for the whole relation - so we are done.

F. Boyce-Codd Normal Form (BCNF)

1. The first three normal forms were developed in a context in which it was tacitly assumed that each relation scheme would have a single candidate key. Later consideration of schemes in which there were multiple candidate keys led to the realization that 3NF was not a strong enough criterion, and led to the proposal of a new definition for 3NF. To avoid confusion with the old definition, this new definition has come to be known as Boyce-Codd Normal Form or BCNF.

2. BCNF is a strictly stronger requirement than 3NF. That is, every BCNF relation scheme is also 3NF (though the reverse may not be true.) It also has a cleaner, simpler definition than 3NF, since no reference is made to other normal forms (except for an implicit requirement of 1NF, since a BCNF relation is a normalized relation and a normalized relation is 1NF). Thus, for most applications, attention will be focused on finding a design that satisfies BCNF, and the previous definitions of 1NF, 2NF, and 3NF will not be needed. There will, however, be times when BCNF is not possible without sacrificing dependency-preservation; in these cases, we may use 3NF as a a compromise.

3. Definition of BCNF: A normalized relation R is in BCNF iff every nontrivial functional dependency that must be satisfied by R is of the form A → B, where A is a superkey for R.

4. We have noted that BCNF is basically a strengthening of 3NF. Often a relation that is in 3NF will also be in BCNF. But BCNF becomes of interest when a scheme contains two overlapping, composite candidate keys.

   a) Example: Consider the 3NF decompositions we generated earlier for our library example. It is also BCNF.

   b) Example: Suppose we remove the assumption that each book has a single author, and allow books to have multiple authors.

      (1) In this case, of course, we must drop the following FD:

      call_number → author

      (2) Suppose we now generate the following schema (along with others):

      (call_number, copy_number, accession_number, author)

      with FD's

      call_number, copy_number → accession_number
      accession_number → call_number, copy_number

      What are the candidate keys for this schema?

      ASK

      (call_number, copy_number, author)
      (accession_number, author)

      (3) Is it 3NF?

      ASK

      At first glance, it would appear not to be - because both of our FD's involve left-hand-sides that are not candidate keys. However, the 3NF definition includes a "loophole" - a <u>key attribute</u> can be transitively dependent on another attribute. Since call_number, copy_number, and accession_number are all part of one or the other of the candidate keys, the 3NF rules allow these dependencies.

(4) However, though the scheme is 3NF, it does involve an undesirable repetition of data - e.g. given that QA76.9.D3 S5637 is the call number for our text, if copy #1 of it has accession number 123456, then we must record this information three times - once for each of the three authors of the text.

(5) Of course, this scheme is not BCNF - the BCNF definition does not have the "loophole" and would force us to decompose further into something like:

(call_number, copy_number, accession_number)
(call_number, copy_number, author)

(6) The advantage of BCNF here is that it avoids a redundancy that 3NF would allow; the repetition of the accession_number for each occurrence of a given call_number, copy_number (which could occur many times paired with different authors).

5. Unfortunately, while it is always possible to decompose a non-3NF scheme into a set of 3NF schemes in a lossless, dependency-preserving way, it is not always possible to decompose a non-BCNF scheme into a set of BCNF schemes in a way that preserves dependencies. (A lossless decomposition is always possible, of course.)

a) Example: The previous example we used DOES allow a dependency preserving decomposition into BCNF - e.g. the BCNF decomposition above does preserve the FD's.

b) Example: The following non-BCNF scheme cannot be decomposed into BCNF in a way that preserves dependencies:

S(J,K,L)

with dependencies

JK → L
L → K

This is not BCNF, since the candidate keys are JK and JL, but K depends only on L. (L is therefore a determinant, but not a candidate key.) There are three possible decompositions into two schemes of two attributes each of which only one is lossless-join:

(J,L) and (K,L)

This does not allow the dependency to JK → L to be tested without a join.

(Fortunately, such messy situations are rare; usually a dependency-preserving BCNF decomposition is possible.)

6. The book gives an Algorithm for decomposing any non-BCNF scheme into a set of BCNF schemes.

   PROJECT - Figure 7.12 p. 289

   Let's apply it to our "multiple authors per book" example:

   result initially = { (call_number, copy_number,
   accession_number, author) }

   F+ = F =

   call_number, copy_number → accession_number
   accession_number → call_number, copy_number

   (i.e. in this case taking the transitive closure of F adds no new dependencies of interest.)

   candidate keys are (call_number, copy_number, author) and (accession_number, author)

   At the first iteration of the while, we find that the one scheme found in result is non-BCNF. We look at our dependencies and find that the first is of the form $\alpha \rightarrow \beta$, where α is call_number, copy_number and β is accession_number, but call_number, copy_number is not a key for this scheme- so we replace the scheme in result by:

   R - β = (call_number, copy_number, author)
   plus $\alpha \cup \beta$ = (call_number, copy_number, accession_number)

   At the second iteration of the while, we find that both schemes in result are BCNF, so we stop - which is the same as the BCNF scheme we introduced earlier.

G. We said earlier that we had three goals we wanted to achieve in design:

1. ASK

   a) Avoid redundancies and resulting update, insertion, and deletion anomalies, by decomposing schemes as necessary.

   b) Ensure that all decompositions are lossless-join.

   c) Ensure that all decompositions are dependency-preserving.

2. We have seen how to use FD's to help accomplish the first goal, and how to use FD's to test whether the second is satisfied. Obviously, the set of FD's is what we want to preserve, though this is not always attainable if we want to go to the highest normal form.

IV. Normalization Using Multivalued Dependencies

A. So far, we have based our discussion of good database design on functional dependencies. Functional dependencies are a particular kind of constraint imposed on our data by the reality we are modeling. However, there are certain important real-world constraints that cannot be expressed by functional dependencies.

1. Example: We have thus far avoided fully dealing with the problem of the relationship between a book and its author(s).
2. Initially, we developed our designs as if the following dependency held:

   call_number → author
3. Although we dropped that dependency, we don't want to say that there is no relationship between call_number and author - e.g. we would expect to see QA76.9.D3 S5637 (the call_number for our text book) in the database associated with Korth, Silberschatz, or Sudarshan, but we would not expect to see it associated with Peterson (who happens to be a joint author with Silberschatz on another text book we have used, but not this one!).

B. At this point, we introduce a new kind of dependency called a MULTIVALUED DEPENDENCY. We will define this two ways - first more intuitively, then more rigorously.

1. We say that a set of attributes A MULTI-DETERMINES a set of attributes B iff, in any relation including attributes A and B, for any given value of A there is a (non-empty) set of values for B such that we expect to see all of those B values (and no others) associated with the given A value and any given set of values for the remaining attributes. (The number of B values associated with a given A value may vary from A value to A value.)
2. We say that a set of attributes A MULTI-DETERMINES a set of attributes B iff, for any pair of tuples t1 and t2 on a scheme R including A and B such that t1[A] = t2[A], there must exist tuples t3 and t4 such that

   a) A] = t2[A] = t3[A] = t4[A] and
   b) t3[B] = t1[B] and t4[B] = t2[B] and
   c) t3[R-A-B] = t2[R-A-B] and t4[R-A-B] = t1[R-A-B]

   Note: if t1[B] = t2[B], then this requirement is satisfied by letting t3 = t2 and t4 = t1. Likewise, if t1[R-A-B] = t2[R-A-B], then the requirement is satisfied by setting t3 = t1 and t4 = t2. Thus, this definition is only interesting when t1[B] <> t2[B] and t1[R-A-B] <> t2[R-A-B].

3. We denote the fact that A multidetermines B by the following notation:

   A ->> B

   (Note the similarity to the notation for functional dependence.)

4. Example: Consider the following scheme:

   Author_info(call_number, copy_number, author)

   a) This scheme is BCNF

   b) It actually contains several MVD's:

      call_number ->> author
      call_number ->> copy_number
      author ->> call_number

      (The second occurs because, for any given book, we have some number of copies with specific copy_numbers; the third because any given author has written some specific books)

   c) Thus once we know that the author values associated with QA76.9.D3 S5637 (the call number for our textbook) are Korth, Silberschatz, and Sudarshan, the multivalued dependency from call_number to author tells us two things:

      (1) Whenever we see a tuple with call_number attribute QA76.9.D3 S5637, we expect that the value of the author attribute will be either Korth or Silberschatz or Sudarshan - but never some other name such as Peterson.

      (2) Further, if a tuple containing QA76.9.D3 S5637 and Korth (along with some copy number) appears in the database, then we also expect to see another tuple that is exactly the same except that it contains Silberschatz as its author value, and another tuple that is exactly the same except it contains Sudarshan as its author.

      (3) As an illustration of this latter point, consider the following instance:

QA76.9.D3 S5637	1	Silberschatz
QA76.9.D3 S5637	1	Korth
QA76.9.D3 S5637	1	Sudarshan
QA76.9.D3 S5637	2	Silberschatz
QA76.9.D3 S5637	2	Sudarshan

The multi-valued dependency call_number ->> author requires that we must add to the relation instance the tuple

QA76.9.D3 S5637 2 Korth

This can be shown from the rigorous definition as follows:

Let t1 be the tuple:

QA76.9.D3 S5637 2 Silberschatz

t2 be the tuple:

QA76.9.D3 S5637 1 Korth

since these tuples agree on the call_number value, our definition requires the existence of t3 and t4 tuples such that

• t1, t2, t3 and t4 all agree on call_number QA76.9.D3 S5637
• t3 agrees with t1 in having author Silberschatz, and t4 agrees with t2 in having author Korth
• t3 agrees with t2 on everything else - i.e. copy_number 1, and t4 agrees with t1 on everything else - i.e. copy_number 2

Thus t3 is

QA76.9.D3 S5637 1 Silberschatz

And t4 is

QA76.9.D3 S5637 2 Korth

While the former occurs in the database, the latter does not, and so must be added.

(4) On the other hand, suppose our database contains just one copy - i.e.

QA76.9.D3 S5637 1 Silberschatz
QA76.9.D3 S5637 1 Korth
QA76.9.D3 S5637 1 Sudarshan

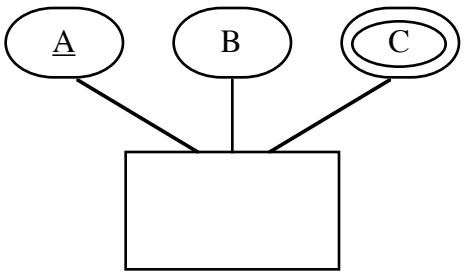
This satisfies the multivalued dependency call_number ->> author as it stands.

To see this, let t1 be the first tuple and t2 the second. Since they agree on call_number but differ on author, we require the presence of tuples t3 and t4 which have the same call_number, and with

t3 agreeing with t1 on author (Silberschatz)
t4 agreeing with t2 on author (Korth)
t3 agreeing with t2 on everything else (copy_number = 1)
t4 agreeing with t1 on everything else (copy_number = 1)

Of course, now t3 and t4 are already in the database (indeed, t3 is just t1 and t2 is just t4) so the definition is satisfied.

5. MVD's correspond to multi-valued attributes in an ER diagram - e.g. consider the the following diagram:



what dependencies does this translate into?

ASK

A → B
A ->> C

C. Note that, whereas we think of a functional dependency as prohibiting the addition of certain tuples to a relation, a multivalued dependency has the effect of REQUIRING that we add certain tuples when we add some other.

1. Example: If we add a new copy of QA76.9.D3 S5637, we need to add three tuples - one for each of the authors.
2. It is this kind of forced replication of data that 4NF will address.
3. Before we can introduce it, we must note a few additional points.

D. Multivalued dependencies are a lot like functional dependencies, however, their closure rules are a bit different

1. A functional dependency can be viewed as a special case of a multivalued dependency, in which the set of "B" values associated with a given "A" value contains a single value. In particular, the following holds:

   if A → B, then A ->> B

   a) To show this, note that if we have two tuples t1 and t2 such that t1[A] = t2[A], and A → B, then t1[B] must = t2[B]. But we have already seen that the t3 and t4 tuples required by the definition for A ->> B are simply t1 and t2 in the case that t1[B] = t2[B]; so any relation satisfying A → B must also satisfy A ->> B.

b) Of course, a functional dependency is a much stronger statement than a multi-valued dependency, so we don't want to simply replace FD's with MVD's in our set of dependencies.

2. We consider an FD to be trivial if its right-hand side is a subset of its left hand side. We consider an MVD to be trivial if either of the following is true:

   a) Its right-hand side is a subset of its left-hand side

      i.e. For any MVD on a relation R of the form α ->> β,

      if $\alpha \supseteq \beta$ the dependency is trivial

      or

   b) The union of its left hand and right hand sides is the whole scheme

      i.e. For any MVD on a relation R of the form α ->> β,

      if $\alpha \cup \beta = R$, the dependency is trivial. (

      This is because, for any scheme R, if α is a subset of R then α ->> R - α always holds. To see this, consider the definition of an MVD:

      Assume we have two tuples on R t1 and t2 s.t. $t1[\alpha] = t2[\alpha]$
      The MVD definition requires that R must necessarily contain tuples t3 and t4 s.t.

      $t1[\alpha] = t2[\alpha] = t3[\alpha] = t4[\alpha]$
      $t3[R - \alpha] = t1[R - \alpha]$
      $t4[R - \alpha] = t2[R - \alpha]$
      $t3[R - (R - \alpha)] = t2[R - (R - \alpha)]$
      $t4[R - (R - \alpha)] = t1[R - (R - \alpha)]$

      But since R - (R - α) is just α, t3 is simply t1 and t4 is t2.

3. Just as we developed the notion of the closure of a set of FD's, so we can consider the notion of the closure of a set of FD's and MVD's. Given a set of FD's and MVD's D, we can find their closure D+ by using appropriate rules of inference. These are discussed in Appendix C of the text.

   PROJECT: Rules of inference for FD's and MVD's

   a) Note that this set includes both the FD rules of inference we considered earlier, and new MVD rules of inference

   b) Note, in particular, that though there is a union rule for MVD's just like there is a union rule for FD's, there is no MVD rule analogous to the decomposition rule for FD's.

e.g. given A → BC, we can infer A → B and A → C.

However, given A ->> BC, we cannot necessarily infer A ->> B or A ->> C unless certain other conditions hold.

E. Just as the notion of functional dependencies led to the definition of various normal forms, so the notion of multivalued dependency leads to a normal form known as fourth normal form (4NF). 4NF addresses a redundancy problem that otherwise arises if we have two independent multivalued dependencies in the same relation - e.g. (in our example) the problem of having to add three tuples to add a new copy of a book with three authors.

A normalized relation R is in 4NF iff for any MVD A ->> B in R it is either the case that the MVD is trivial or else A functionally determines all the attributes of R (in which case the MVD is actually an FD)

1. Example: (call_number, copy_number, author) is not 4NF, since call_number ->> author is a nontrivial MVD that is not an FD

2. Note that every 4NF relation is also BCNF. BCNF requires that, for each nontrivial functional dependency A → B that must hold on R, A is a superkey for R.

   But if A → B, then A ->> B. Further, if R is in 4NF, then for every nontrivial multivalued dependency of the form A ->> B, A must be a superkey. This is precisely what BCNF requires.

3. An algorithm is given in the book for converting a non 4NF scheme to 4NF

   PROJECT - figure 7.17 p. 297

   It basically operates by isolating MVDs in their own relation, so that they become trivial.

   Example: application of this algorithm to our library database (with multiple authors).

   a) Our canonical cover for F+, with added MVDs for author

   borrower_id → last_name, first_name
   call_number → title
   call_number ->> author
   call_number, copy_number → accession_number,borrower_id, date_due
   accession_number → call_number, copy_number

Notes:

(1) We do not include call_number ->> copy_number

(a) It is not the case that if we have two different copies of some call_number, each copy_number value appears with each accession_number - just with the one for that book.

(b) Likewise, it is not the case that if we have two different copies of some call_number, each copy_number value appears with each borrower/date_due - just to the one (if any) that pertains to that particular copy.

(2) The definition of 4NF - and the 4NF decomposition algorithm - are both couched solely in terms of MVD's. However, since every FD is also an MVD, we will use the above set, remembering that when we have say

borrower_id → last_name, first_name

we necessarily also have

borrower_id ->> last_name, first_name

(3) The algorithm calls for using D+ - the transitive closure of D, the set of FD's and MVD's. As it turns out, all we really need to know for this problem is Fc (the canonical cover for the FD's) plus the MVD's. (The transitive closure of the set of MVD's is huge!)

b) Initial scheme:

{ (borrower_id, last_name, first_name, call_number,
copy_number, accession_number, title, author date_due)
}

c) Not in 4NF - LHS of first dependency is not a superkey - change to

{ (borrower_id, last_name, first_name),
(borrower_id, call_number, copy_number, accession_number,
title, author, date_due)
}

d) Second schema not in 4NF - LHS of second dependency is not a superkey - change to

{ (borrower_id, last_name, first_name),
(call_number, title),
(borrower_id, call_number, copy_number, accession_number,
author, date_due)
}

e) Third schema not in 4NF - LHS of third dependency is not a superkey - change to

{ (borrower_id, last_name, first_name),
(call_number, title),
(call_number, author)
(borrower_id, call_number, copy_number, accession_number, date_due)
}

f) Result is now in 4NF - we're done

V. Higher Normal Forms

A. For most applications, the normalizations we have considered are thoroughly adequate. In particular:

1. Wherever possible, we normalize to 4NF. The exception to this is if doing so would fail to preserve the ability to test certain dependencies without doing a join.

2. If we can't have 4NF for this reason, we accept BCNF.

3. However, since a BCNF decomposition may also fail to be dependency-preserving, in some cases we may even have to accept just 3NF.

4. We need never compromise below 3NF, since a lossless-join dependency-preserving decomposition into 3NF is always possible.

5. Sometimes, we may also accept a lower normal form for efficiency reasons - because joins are computationally expensive (the Achilles heel of the relational model.)

B. Two normal forms have been proposed as generalizations of the ones we have studied thus far. However, we will not discuss them further here. (If you are interested, see Appendix C of the text - available online).

VI. Some Final Thoughts About Database Design

A. At the risk of over-simplifying, he normalization rules we have considered can be reduced down to the following simple ditty:

In a good design, every attribute depends on the key, the whole key, and nothing but the key.

B. There are two general approaches to overall database design:

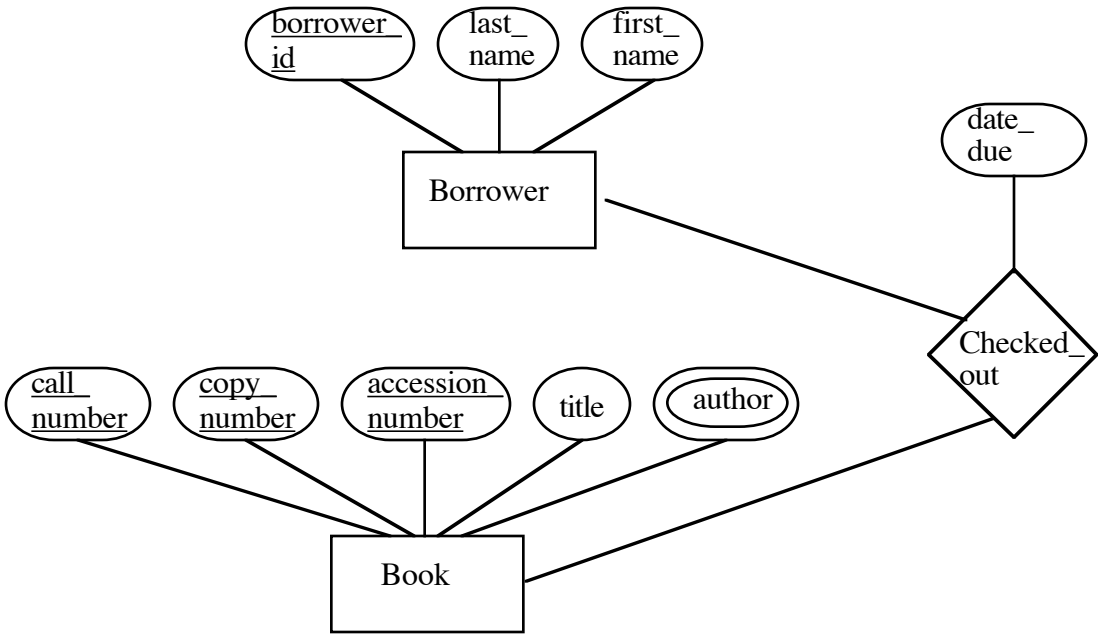
1. Start with a universal relation - a relation containing all the attributes we will ever need - and then normalize it.

   a) This has been the approach we followed in the running example in this series of lectures, where we started with a single universal relation and finished up with a 4NF decomposition.

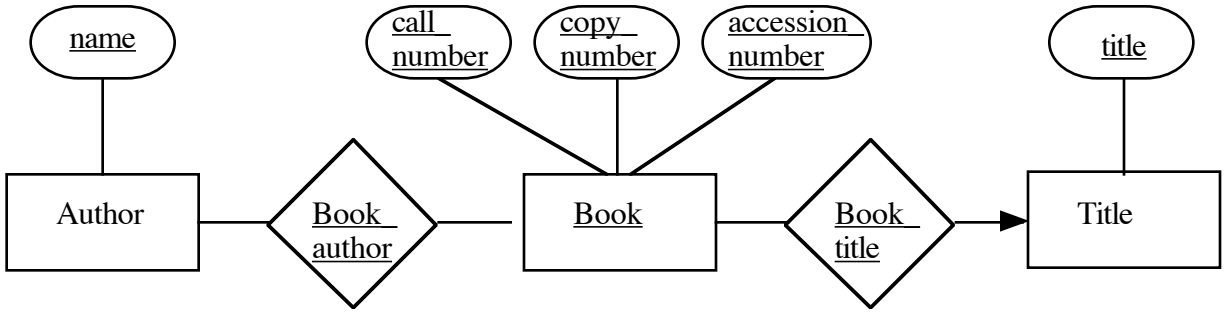
   b) This is often the way a naive user designs a database.- though the naive user may not get around to normalization!

2. Start with an ER diagram. If we do this, we may still need to do some normalization. This can lead to modifying our ER diagram, or we can simply do the normalization as part of creating the relational scheme.

   Example: Our running library example could be represented by the following initial ER diagram:



We could then think of our normalization process as requiring us to decompose book into three entities and two relationships:



However, a simplistic conversion into tables would, in this case, lead to more tables than we need, since the Author and Title tables contain no information other than their keys.

(If they did contain additional information, then making them separate entities would make sense. We can imagine having more information about authors, thus warranting a separate Author entity; it would be hard to imagine what would warrant a separate Title entity)

Thus, we may be better to take the set of tables arising from normalizing the original ER diagram, leading to the following set of tables:

Borrower(borrower_id, last_name, first_name)
Book(call_number, copy_number, accession_number)
Book_title(call_number, title)
Book_author(call_number, author)
Checked_out(borrower_id, call_number, copy_number, date_due)

(Note that, in general, a one-to-one or one-to-many relationship in an ER diagram can often be converted to a relational design in which the key of the "one" is folded into the table representing the "many" entity, thus avoiding the need for a separate table.)

3. Note that these two approaches lead, after normalization, to similar but not identical designs.

   a) How does the design that comes from normalizing our original ER diagram differ from the design we came to by normalization of a universal relation?

ASK

The former has a separate Checked_out table, rather than keeping borower_id and date_due in the Book table.

b) Which is better?

ASK

The latter design avoids the necessity of storing null for the borrower id of a book that is not checked out, at the expense of having an additional table. Thus,

(1) To record the return of a book under the first model, we set the borrower_id attribute of the Book tuple to null.

(2) To record the return of a book under the second model, we delete the Checked_out tuple

C. Although we have stressed the importance of normalization to avoid redundancies and anomalies, sometimes in practice partially-denormalized databases are used for performance reasons, since joins are costly.

1. The problem here is to ensure that all redundant data is updated consistently, and to deal with potential insertion and deletion anomalies, perhaps by use of nulls.

2. Note that views can be used to give the illusion of a denormalized design to users, but do not address the performance issue, since the DBMS must still do the join when a user accesses the view

3. A sophisticated DBMS may support materialized views in which the view is actually stored in the database, and updated in synch with the tables on which it is based. (In db2, these are called summary tables.)