CPS352 Lecture - Database Normalization

last revised February 10, 2021

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 ER diagram for example unnormalized database
2. Projectable of Everything library scheme
3. Projectable of Lossy decomposition of Everything scheme
4. Projectable of Series of steps showing it is a problem
5. Projectable of notation for FD
6. Projectable of first cut at library FDs
7. Projectables of ER to FD Examples (4)
8. Projectable of review of Everything scheme and FDs
9. Projectable of normalization of Everything to 2NF
10. Projectable of normalization of Everything to 3NF, BCNF
11. Projectable of alternative that eliminates need for nulls for book not out
12. Projectable of Armstrong’s Axioms (from 5th ed slides augmented with examples)
13. Projectable of additional rules of FD inference (from 5th ed slide augmented with examples)
14. Projectable of proofs of additional inference rules from Armstrong's axioms
15. Projectable of equivalent ways of writing the same set of FDs
16. Projectable of algorithm for computing F+ (from 5th ed slides)
17. Projectable of algorithm for computing closure of an attribute (From 5th ed slides)
18. Projectable of finding closures for library FD attributes
19. Projectable of determining canonical cover for library FDs
20. Projectable of BCNF decomposition algorithm
21. Projectable of applying BCNF algorithm to example
22. Projectable of 3NF algorithm

23. Projectable of applying 3NF algorithm to example
24. Projectable of definition and notation for an MVD
25. Projectable of example showing problem that can result from an MVD
26. Projectable of example of an MVD that causes no problem
27. Projectable of ER to MVD example
28. Projectable of rules of inference for FD's and MVD's (5th ed slide)
29. Projectable of additional MVD Rules (5th ed slide)
30. Projectable of 4NF algorithm
31. Projectable of review of initial Everything scheme and dependencies for example database with multiple authors allowed
32. Projectable of applying 4NF algorithm to this
33. Projectable of ER diagram for original unnormalized database and relational created directly
34. Projectable of converting this to BCNF (without multiple authors) and 4NF (if multiple authors allowed)

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 some 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)

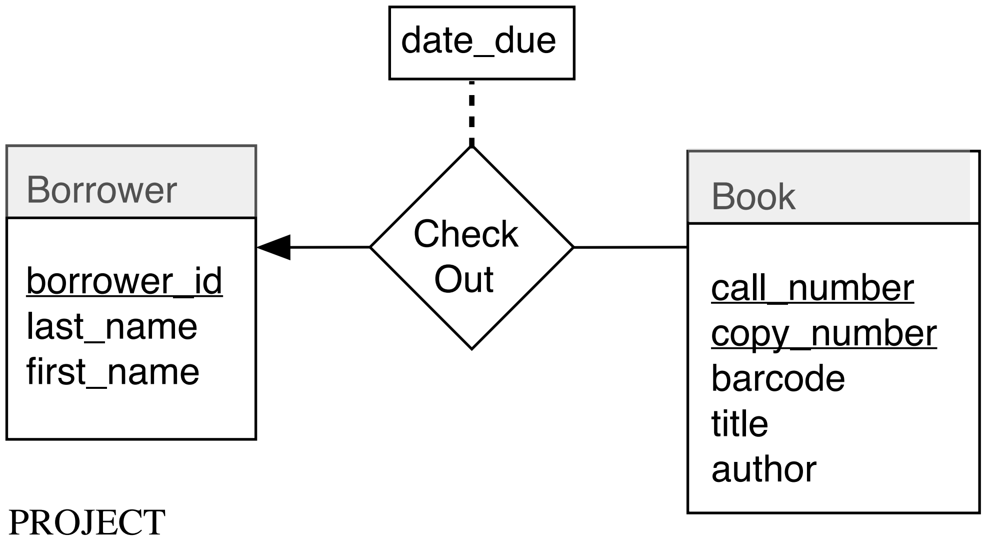
a) We will allow for the possibility of having multiple copies of a given book, so we include a copy_number attribute

b) We will also include a barcode attribute for books. (The barcode 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 user designs a database for the following ER diagram,



PROJECT

Suppose the individual purchases a commercial database and naievely decides to use the following scheme. Note that it incorporates all of the attributes of the separate tables relating to borrowers, books and checkouts from into a single table - including the two new ones just added)

```
Everything(borrower_id, last_name, first_name, // borrower info
           call_number, copy_number, barcode,  // copy info
           title, author,                      // other book
           date_due)                            // check out info
```

PROJECT

(Don't laugh - people actually 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) 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, barcode, title, author)
   Checked_out(date_due)

   PROJECT

   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 cat charlene AB123.40 1 17 Karate   elephant 2016-11-15
89754 dog donna    LM925.04 1 24 Cat Cook dog      2016-11-10
```

PROJECT Series of Steps

(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

2016-11-15
2016-11-10
```

(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 lost. 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 information 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 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 → B (Read: A determines B)

   PROJECT

   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 → 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
barcode
title
author
date_due

WRITE ON BOARD AND ASK FOR FD'S - PROJECT

borrower_id → last_name, first_name
call_number → title
call_number, copy_number → barcode
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
barcode → 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 $\rightarrow$ 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 $\rightarrow$ author

in our set of dependencies

b) Example: Should we include the dependency

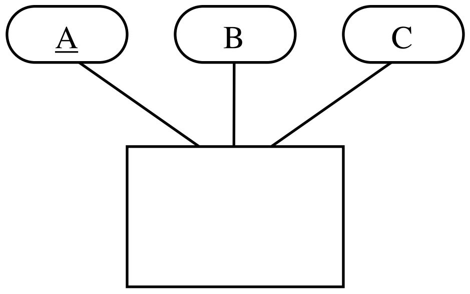
last_name, first_name $\rightarrow$ borrower_id

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.

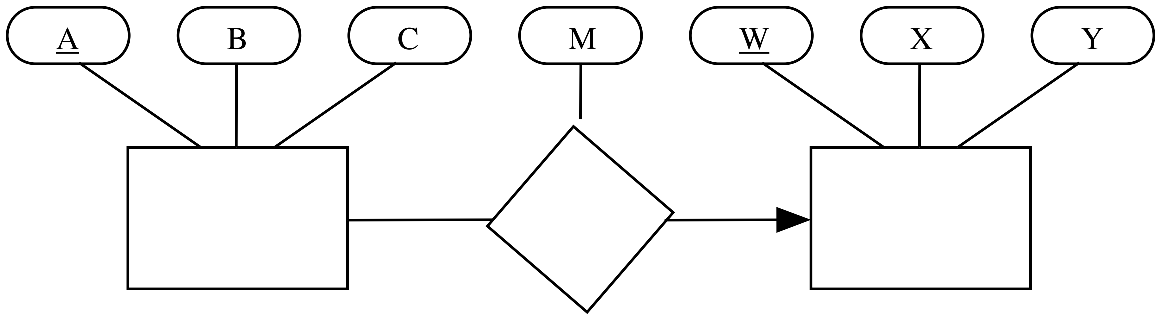
   a) What does the following pattern in an ER diagram translate into in terms of FD's? (Note that A is underlined)



   PROJECT, THEN ASK

   $A \rightarrow BC$

   b) How about this? (Note that A and W are underlined)



   PROJECT, THEN ASK

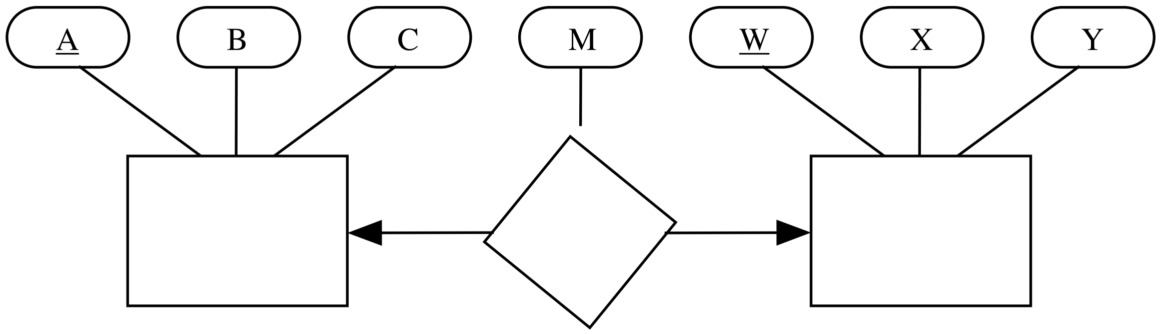
   $A \rightarrow BCMWXY$
   $W \rightarrow XY$

   But not

   $W \rightarrow XY$

   because relationship is one to many

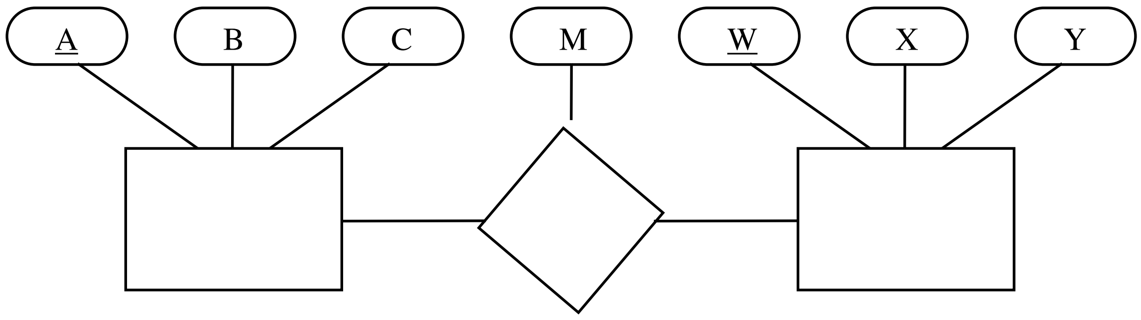
c) How about this?



PROJECT, ASK

A → BCMWXY
W → XYMABC

d) Or this?



PROJECT, ASK

A → BC
W → XY
AW → M

3. 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. We will say more about this later, but for now we can introduce the basic idea.

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:

      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].

      (1) 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]

      (2) 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.

(3)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. Given an initial set of dependencies F, we refer to the set of all dependencies <u>imply</u> as the transitive closure of F - written F+. We will discuss later how we can actually go about discovering this.

D. 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 key 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 weak entity 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.

E. 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)

Lossless-join is an absolutely essential property of a valid decomposition, of course.

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, barcode ...

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

barcode → call_number, copy_number
call_number, copy_number → barcode

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, barcode ....
and
copy_number, barcode ...

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

barcode → 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 → barcode

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 barcode

(2)To test whether a decomposition is dependency-preserving, we introduce the notion of the restriction 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 transitive closure of the original set is equal to the transitive closure 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 $A \rightarrow$ (all combinations of A,B,C,D) and $B \rightarrow$ (all combinations of B, C, D)

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

iii)Since $B \rightarrow CD$ is not a member of the transitive closure of these restrictions, 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 $B \rightarrow CD$. 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

$A \rightarrow B$
$A \rightarrow C$
$B \rightarrow CD$

If we decompose into

(AB) (BCD)

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

(5)The book gives an algorithm for testing dependency-preservation, but it is computationally expensive to compute the closures. However, we can often identify by inspection which dependencies do not appear in the restrictions of the the decomposition, and then check to see whether these are implied by dependencies that do appear.

(6)Dependency preservation is highly desirable, since it avoids needing to use natural joins to test a new row being inserted - but it is not absolutely essential and sometimes must be done without.

III.Decomposition Using Functional Dependencies

A. We will now develop an example of how the functional dependencies we identified earlier can be used to decompose our original "Everything" scheme.

In contrast to the approach used in the book, we will proceed along lines closer to the way the various normal forms were actually discovered, rather than starting with Boyce-Codd Normal Form (BCNF) which is actually the end result of a process of invention.

1. First, let's review our "Everything" scheme and its functional dependencies.

   PROJECT

2. What should be use as the primary key for this scheme?

   Recall that a primary key must functionally determine all the attributes of a scheme. There are actually two possibilities. What are they?

   ASK

   a) call_number and copy_number would work.

      • call_number directly determines title and author
      • call_number and copy_number together determine barcode, borrower_id, and date_due

• since borrower_id determines last_name and first_name, call_number and copy_number together determine these as well.

b) barcode would also work, since it directly determines call_number and copy_number, which together determine everything else

c) We will use call_number and copy_number as the primary key for this example

B. Actually, as this design stands we have already made a design decision related to normalization.

1. Recall that we decided at least for the time being - to include the FD

   call_number → author

2. Suppose we had not done this, allowing a single book to have multiple authors (as is normally this case).   One way to do this would be to allow the author attribute to be multivalued - to be a list of authors.

3. Recall that, in the relational model, the domains of attributes must be atomic (single, non-composite values).   In his original paper on the relational model, the inventor - Edgar Codd - referred to this requirement as "normalization". In the history of database normalization, this has become known as first normal form, and all the higher normal forms assume it.

4. An alternative would be to store multiple rows for a book with multiple authors - one for each author. But in that case, author would no longer depend on the primary key, since a given copy number could be associated with multiple values of author.   (And adding author to the PK would create other problems which we will talk about later.)

5. Our scheme, as it now stands - with only one author per book - needs no further work is needed to achieve 1NF.   But we will come back to the question of allowing multiple authors later.

C. Earlier, we identified some update, insertion, and deletion anomalies present in this scheme. There are actually also some more subtle ones as well.

1. Note that, in the Everything scheme, all of the information about a book (call number, copy number, barcode, title, and author) is in a single scheme. But looking at our functional dependencies, we find that title and author depend only on call_number. This gives rise to several anomalies:

   a) Suppose we discovered that, when creating our database, we had made an error spelling the author's name. If we had several copies of the book, we would need to correct each row in the database. What kind of anomaly is this an example of?

      ASK

      Update anomaly

   b) Suppose that, while we were waiting for the first copy of a new book to come in, we wanted to make an entry for the author and title of the book in our database. We couldn't do this unless we created a row with a copy number and barcode - something we generally want to avoid. What kind of anomaly is this an example of?

      ASK

      Insertion anomaly

   c) Suppose our only copy of a single book is lost. While we are waiting for a replacement to come in, we delete the row in the database describing the book since the barcode refers to a book that we no longer have. In the process, we lose the information on the title and author, which we then have to re-enter when the replacement arrives. What kind of anomaly is this an example of?

      ASK

      Deletion anomaly

2. It turns out that these anomalies are all a result of what is known as a <u>partial dependency</u> - since author and title depend only on a subset of the key. Codd went on to define 2nd normal form as excluding partial dependencies like this - i.e. no attribute can be allowed to depend on a proper subset of any candidate key.

3. So to put our Everything scheme into 2NF, we could decompose as follows:

```
EverythingElse(borrower_id, last_name, first_name,
               call_number, copy_number, barcode,
               date_due)

BookInfo(call_number, title, author)
```

PROJECT

(Notice how call_number appears in both the decomposed scheme and the added scheme to make the decomposition lossless - since the intersection of the attributes of the two schemes is `call_number`, which is a superkey for `BookInfo`.)

D. While this has eliminated the anomalies related to information about a book's title and author, it still leaves the anomalies related to borrower.

1. What were they?

ASK

a) Update - if a borrower starts using a different name (e.g. as a result of marriage or change of legal name), we need to update the entry in the table for every book that this borrower has out.

b) Insertion - we can only add a new borrower if the individual has a book checked out. (It might seem we could address this by creating a row with the borrower information and nulls for the book information, but then we would end up with a null primary key, which is not allowed!)

c) Deletion - if a borrower has only one book checked out and the book is returned (or if it is lost), deleting the row loses the information about the borrower. (Again, simply nulling the information about the book to preserve the borrower information results in a null primary key.)

2. It turns out that these anomalies are all a result of a <u>transitive dependency</u> - the borrower's name depends on the primary key only indirectly - through the non-key attribute `borrower_id`. Codd went on to define 3rd normal form as excluding transitive dependencies like this - i.e. no attribute can be allowed to depend on a non-key attribute. (This also implicitly excludes partial dependencies, so any 3NF relation is also in 2NF, though the reverse is not necessarily the case.)

   In the case of our everything scheme, this leads us to decompose to:

```
BorrowerInfo(borrower_id, last_name, first_name)
BookInfo(call_number, title, author)
CopyInfo(call_number, copy_number, barcode,
         borrower_id, date_due)
```

   PROJECT

   (Notice how borrower_id appears in both the decomposed scheme and the added scheme to make the decomposition lossless - since the intersection of the attributes of the two schemes is `borrower_id`, which is a superkey for BorrowerInfo.)

3. Sometimes the normalization criterion for 3NF is stated this way: each non-key attribute must depend on the key (1NF), the whole key (2NF), and nothing but the key (3NF).

E. Actually, there is still a subtle problem with 3NF. To deal with some unusual situations (which we can't illustrate with this particular database), 3NF allows a <u>key attribute</u> to be on the right hand side of a transitive dependency.

1. One proposal to address this was to redefine 3NF to forbid this.

2. However, treating key attributes specially sometimes turns out to be necessary for the sake producing dependency-preserving decomposition. Pages 315-318 give an example of such a case.

3. This led, a few years later, to the introduction of a new normal form called Boyce-Codd Normal Form (BCNF).

4. The BCNF definition is strictly stronger than 3NF - every BCNF scheme is also in 3NF, but the reverse is not true - there are some 3NF schemes that are not BCNF. It also turns out that it is simpler!

5. A scheme is in BCNF if every FD $\boldsymbol{\alpha} \rightarrow \boldsymbol{\beta}$ (where $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ are sets of attributes that appear in the scheme) that holds on the scheme is it either the case that

   a) It is trivial - i.e. $\boldsymbol{\alpha} \supseteq \boldsymbol{\beta}$

   b) $\boldsymbol{\alpha}$ is a superkey.

F. It turns out that our 3NF decomposition of Everything is also in BCNF, so we need to nothing more.

However, it might be expedient to consider one more change to simplify our use of the scheme.

How should we handle the case of a copy of a book that is not checked out to anyone, since the `CopyInfo` scheme contains attributes `borrower_id` and `date_due` which are not relevant to a copy that is not checked out?

1. This would be a case where storing null for these attributes for a copy that is not checked out makes good sense - in which case when a copy that is checked out is returned, we replace the values with null.

2. Alternately, we might decompose further

   ```
   BorrowerInfo(borrower_id, last_name, first_name)
   BookInfo(call_number, title, author)
   CopyInfo(call_number, copy_number, barcode)
   CheckedOut(call_number, copy_number, borrower_id, date_due)
   ```

   In this case, we store nothing in `CheckedOut` when a copy is acquired, add a row to `CheckedOut` when it is checked out, and delete the row in `CheckedOut` when it is returned.

   PROJECT

G. There are algorithms for going directly to BCNF or 3NF without passing through the other normal forms. But before we can look at these, we need to delve into the theory of functional dependencies a bit more.

IV. **Theory of Functional Dependencies**

A. 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

1. 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 339

   PROJECT 5th ed slide augmented with examples

   a) 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 $\boldsymbol{\alpha} \rightarrow \boldsymbol{\beta}$, where $\boldsymbol{\alpha} \supseteq \boldsymbol{\beta}$ is called trivial.

b) Example of augmentation:

Given:

borrower_id → last_name, first_name

It follows that

borrower_id, call_number → last_name, first_name, call_number

c) Example of transitivity:

call_number, copy_number → last_name, first_name

d) 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 page 321 in the book.

PROJECT 5th ed slide augmented with examples

(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 → barcode, borrower_id, date_due

by pseudo-transitivity, we would get

title, copy_number → barcode, borrower_id, date_due

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

PROJECT

(1)Proof of the union rule

Given: $\boldsymbol{\alpha} \rightarrow \boldsymbol{\beta}, \boldsymbol{\alpha} \rightarrow \boldsymbol{\gamma}$

Prove: $\boldsymbol{\alpha} \rightarrow \boldsymbol{\beta\gamma}$

Proof: $\boldsymbol{\alpha\alpha} \rightarrow \boldsymbol{\alpha\beta}$ (augmentation of first given with $\boldsymbol{\alpha}$)

$\boldsymbol{\alpha \rightarrow \alpha\beta}$ (since we are working with sets, $\boldsymbol{\alpha\alpha = \alpha}$)

$\boldsymbol{\alpha\beta \rightarrow \beta\gamma}$ (augmentation of second given with $\boldsymbol{\beta}$)

$\boldsymbol{\alpha \rightarrow \beta\gamma}$ (transitivity)

(2)Proof of the decomposition rule:

Given: $\boldsymbol{\alpha \rightarrow \beta\gamma}$

Prove: $\boldsymbol{\alpha \rightarrow \beta}$ and $\boldsymbol{\alpha \rightarrow \gamma}$

Proof: $\boldsymbol{\beta\gamma \rightarrow \beta}$ and $\boldsymbol{\beta\gamma \rightarrow \gamma}$ (by reflexivity)

$\boldsymbol{\alpha \rightarrow \beta}$ and $\boldsymbol{\alpha \rightarrow \gamma}$ (by transitivity using given)

(3)Proof of the psuedo-transitivity rule:

Given: $\boldsymbol{\alpha \rightarrow \beta}$ and $\boldsymbol{\beta\gamma \rightarrow \delta}$

Prove: $\boldsymbol{\alpha\gamma \rightarrow \delta}$

Proof: $\boldsymbol{\alpha\gamma \rightarrow \beta\gamma}$ (augmentation of first given with $\boldsymbol{\gamma}$)

$\boldsymbol{\alpha\gamma \rightarrow \delta}$ (transitive rule using second given)

2. 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

$\boldsymbol{\alpha \rightarrow \beta\gamma}$ and $\boldsymbol{\alpha \rightarrow \delta\varepsilon}$

We could choose to write them as

$\boldsymbol{\alpha \rightarrow \beta}$

$\boldsymbol{\alpha \rightarrow \gamma}$

$\boldsymbol{\alpha \rightarrow \delta}$

$\boldsymbol{\alpha \rightarrow \varepsilon}$

or as

$\boldsymbol{\alpha \rightarrow \beta\gamma\delta\varepsilon}$

(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.

PROJECT

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

   PROJECT - 5th ed slide

b) 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 - 5th ed slide

   (2)Example of applying the algorithm 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 → barcode, borrower_id, date_due
      barcode → 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
barcode
borrower_id
date_due
author

On second iteration through loop, add

last_name
first_name

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

(d)Compute barcode +:

Initial: { barcode }

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

∴ barcode → barcode, 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 { barcode } 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 on either side (see the book for definition of "extraneous", which is a non-trivial concept!)

      (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:

      PROJECT steps

      (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, barcode, author,
                           borrower_id, date_due,
                            last_name, first_name
barcode → barcode, call_number,
           copy_number, title, author,
          borrower_id, date_due,
           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 → barcode
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
barcode → barcode
barcode → call_number
barcode → copy_number
barcode → title
barcode → borrower_id
barcode → date_due
barcode → author
barcode → last_name
barcode → first_name
```

(3)Now eliminate the trivial dependencies

(Cross out on list)

~~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 → barcode

```
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
~~barcode → barcode~~
barcode → call_number
barcode → copy_number
barcode → title
barcode → borrower_id
barcode → date_due
barcode → author
barcode → last_name
barcode → first_name
```

(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)

- `barcode → title`
  `barcode → author`

(These are implied by the transitive rule given that `barcode → 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)

- `barcode  → last_name`
  `barcode → first_name`

(These are implied by the transitive rule given that barcode → 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

```
barcode → borrower_id
 barcode → date_due
```

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

(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 → barcode
call_number, copy_number → borrower_id
call_number, copy_number → date_due

barcode → call_number
barcode → 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 → barcode, borrower_id, date_due
barcode → 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

```
        barcode → borrower_id, date_due
```

and dropped

```
        call_number, copy_number → borrower_id, date_due.
```

V. **Algorithms for Decomposing to Normal Forms**

A. Now we can combine what we have said about normal forms and our more careful study of FDs to develop algorithms for a database design. Our goals will be these:

1. Achieve a design that is in BCNF

2. Ensure that all our decompositions are lossless join

3. Ensure that our 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.

   a) We may have to accept some redundancy by settling for 3NF instead of BCNF to preserve dependencies,

   (But we will never have to settle for anything less.)

   b) or we may have to give up dependency-preservation in order to eliminate all redundancies. (We'll see an example of this later.)

B. There is an algorithm we can use to go directly from an initial scheme (like the `Everything` scheme we started with in our example) to a lossless-join BCNF decomposition.

```
result = {R };
done = false;
compute F +;
while (not done) do
    if (there is a schema $R_i$ in result that is not in BCNF)
        let $\alpha \to \beta$ be a nontrivial functional dependency that
          holds on $R_i$ such that $\alpha \to R_i$ is not in F+,
          and $\alpha \cap \beta = \varnothing$;
        result = (result - $R_i$) $\cup$ ($R_i$ - $\beta$) $\cup$ ($\alpha$,$\beta$);
    else
        done = true;
```

1. PROJECT

   a)When the loop exits, each Ri is in BCNF

   b)Each decomposition is lossless-join.

2. **An important note**: the test for BCNF on each of the decomposed relations must be done using the closure of the original set of dependencies.

   The book discusses an alternative test based on the closure of the attributes in R_i.

3. The algorithm takes time exponential in the size of the original scheme, so in practice one may use a certain amount of intuition to minimize the effort.

4. Example: Applying this to our original `Everything` scheme

   PROJECT

   a)
```
result = { Everything(borrower_id, last_name, first_name,
                      call_number, copy_number, barcode,
                      title, author,
                      date_due)
         }
```

F+:

```
borrower_id → borrower_id, last_name, first_name
call_number → call_number, title, author
call_number, copy_number → call_number copy_number,
                           title, barcode, author,
                           borrower_id, date_due,
                           last_name, first_name
barcode → barcode, call_number,
          copy_number, title, author,
          borrower_id, date_due,
          last_name, first_name
```

b) `Everything` violates `borrower_id → borrower_id, last_name, first_name` because it is not trivial and `borrower_id` is not a superkey.

Decompose to get

```
result = { Everything(borrower_id,
                      call_number, copy_number, barcode,
                      title, author,
                      date_due),
           Borrower(borrower_id, last_name, first_name)
         }
```

c) Though `Borrower` is now OK because the only FD that applies to it is trivial. However, `Everything` violates `call_number → call_number, title, author`
because it is not trivial and `call_number` is not a superkey.

Decompose to get

```
result = { Everything(borrower_id,
                      call_number, copy_number, barcode,
                      date_due),
           Borrower(borrower_id, last_name, first_name),
           Book(call_number, title, author)
         }
```

d) Now `Book` is now OK because the only FD that applies to it is trivial. and `Everything` satisfies BCNF because the in FDs that apply the LHS is a superkey for it - so we're done.

C. The book also gives an algorithm for creating a 3NF scheme which can be used if the BCNF decomposition is not dependency-preserving (which the above is, so we wouldn't need to use it in this case.)

1. PROJECT

```
Let Fc be a canonical cover for F;
i = 0;
for each functional dependency α → β in Fc do
    if none of the schemas Rj, 1≤j ≤i contains αβ then
        i= i+1;
        Ri =αβ
if none of the schemas Rj, 1 ≤ j ≤ i contains a candidate
key for R then begin
     i=i +1;
     Ri = any candidate key for R;
/* Optionally, remove redundant relations */
repeat
    if any schema Rj is contained in another schema Rk
        /* delete Rj */
        Rj = Ri;
        i=i-1;
return (R1, R2, ..., Ri)
```

2. Clearly this produces a dependency-preserving decomposition,

a) We include a scheme based on each dependency in the canonical cover unless it is already included in another scheme.

b) Each scheme is based on a dependency, so its LHS is necessarily a superkey for that scheme.

3. Applying this to our example - PROJECT

   F_c =

```
borrower_id → last_name, first_name
call_number → title, author

call_number, copy_number → barcode, borrower_id, date_due
barcode → call_number, copy_number
```

   a) i = 1: to include the first FD in the canonical cover, add

      R_1 = (`borrower_id, last_name, first_name`)

   b) i = 2: to include the second FD in the canonical cover, add

      R_2 = (`call_number, title, author`)

   c) i = 3: include the third FD in the canonical cover, add

      R_3 = (`call_number, copy_number, barcode, borrower_id, date_due`)

   d) `i = 4:` last FD is contained in R_3, so no need to add anything. is a candidate key for the original set of attributes, so no need to add anything.

   e) No redundant schemas to remove

   f) Return same as what we have gotten previously:

      R_1 = (`borrower_id, last_name, first_name`)
      R_2 = (`call_number, title, author`)
      R_3 = (`call_number, copy_number, barcode, borrower_id, date_due`)

VI. **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 <u>no</u> 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 <u>not</u> expect to see it associated with Peterson (who happens to be a joint author with Silberschatz on another widely-used text book, 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 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

t1[A] = t2[A] = t3[A] = t4[A] and
t3[B] = t1[B] and t4[B] = t2[B] and
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].

PROJECT

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

   A ->> B

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

   REFER TO ON SLIDE GIVING DEFINITION

4. Example: Consider the following "all-key" scheme:

   Author_info(call_number, copy_number, author)

   a) This scheme is BCNF

   b) It actually contains the following MVD:

      call_number ->> author

      (That is, every copy of a book with a given call number has the exact same authors - something which is always true, even with revised editions with different authors since such a revised edition would have a different call number)

   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

PROJECT

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

PROJECT

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

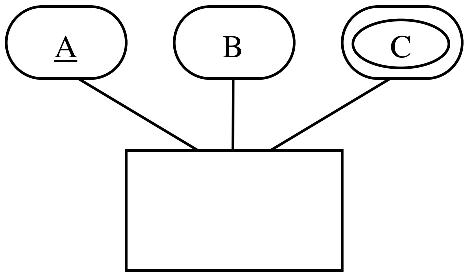
To see this, let t1 be the first tuple and t2 the second. Since they agree on call_number but differ on author, we require the presence of tuples t3 and t4 which have the same call_number, and with

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:

PROJECT



what dependencies does this translate into?

ASK

$A \rightarrow B$
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 $A \rightarrow B$, 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 <u>replace</u> 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 <u>either</u> 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 $\boldsymbol{\alpha}$ ->> $\boldsymbol{\beta}$,

if $\boldsymbol{\alpha} \supseteq \boldsymbol{\beta}$ the dependency is trivial

b) OR: 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 $\boldsymbol{\alpha}$ ->> $\boldsymbol{\beta}$,

if $\boldsymbol{\alpha} \cup \boldsymbol{\beta} = R$, the dependency is trivial.

This is because, for <u>any</u> scheme R, if $\boldsymbol{\alpha}$ is a subset of R then $\boldsymbol{\alpha}$ ->> R - $\boldsymbol{\alpha}$ always holds. To see this, consider the definition of an MVD:

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

t1[a] = t2[a] = t3[a] = t4[a]
t3[R - a] = t1[R - a]
t4[R - a] = t2[R - a]
t3[R - (R - a)] = t2[R - (R - a)]
t4[R - (R - a)] = t1[R - (R - a)]

But since R - (R - $\boldsymbol{\alpha}$) is just $\boldsymbol{\alpha}$, 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 chapter 28 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 <u>cannot necessarily</u> infer A ->> B or A ->> C unless certain other conditions hold.

   c) In general, the closure for a set of dependencies is <u>huge</u>!

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.

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

```
result: = {R};
done = false;
compute D+;
/* Let Di denote the restriction of D+ to Ri */
while (not done)
    if (there is a schema Ri in result that is not in 4NF)
        let α →→ β be a nontrivial multivalued dependency that
          holds on Ri such that α→Ri is not in Di and α∩β=φ;
        result = (result-Ri)∪(Ri-β) ∪(α,β);
    else
        done = true;
```

1. Note: upon completion, each Ri is in 4NF, and decomposition is lossless-join

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

3. 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 → barcode,borrower_id, date_due
barcode → call_number, copy_number,borrower_id, date_due
```

PROJECT

(1)The cover does <u>not</u> 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 <u>each</u> `barcode` - 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 <u>each</u> `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!)

PROJECT steps in decomposition

b) Initial scheme:

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

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

```
{ R(borrower_id, call_number,  copy_number,  barcode,
    title, author, date_due)
  R1(borrower_id, last_name, first_name)
}
```

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

```
{ R(borrower_id, call_number,  copy_number,  barcode,
      author, date_due)
  R1(borrower_id, last_name, first_name),
  R2(call_number, title),
}
```

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

```
{ R(borrower_id, call_number,  copy_number,  barcode,
     date_due)
  R1(borrower_id, last_name, first_name),
  R2(call_number, title),
  R3(call_number, author)
}
```

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

VII. **Higher Normal Forms**

A. For most applications, the kinds of dependencies we have considered are thoroughly adequate.

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. They come from situations having certain kinds of constraints that cannot be expressed as FDs or MVDs.

C. These are discussed in sections 28.2 and 28.3 of the online part of the text.

VIII. **Database Design**

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

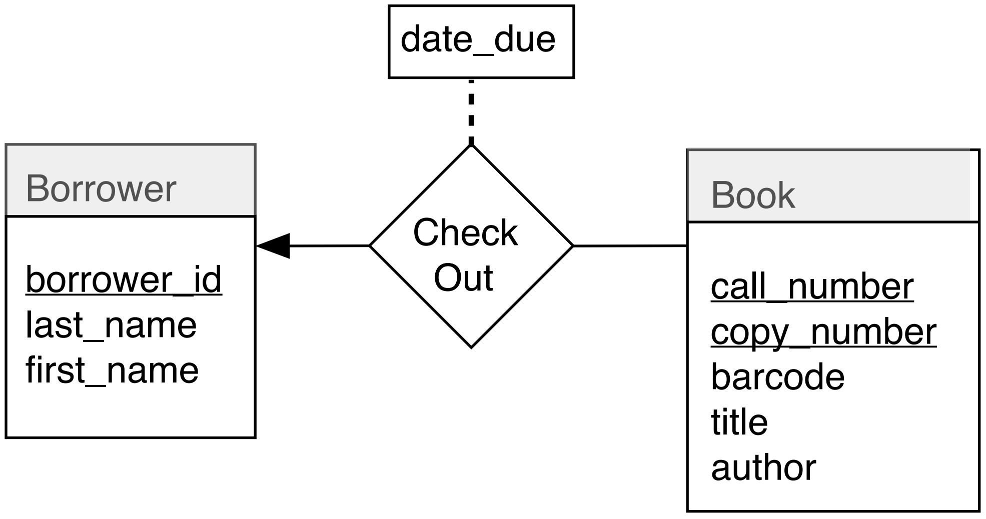
1. Start with a <u>universal relation</u> - a relation containing all the attributes we will ever need - and then normalize it.

   a) This has been the approach we followed in the examples 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 recall that our running library example could be represented by the following initial ER diagram:



PROJECT

From this, we might generate the following initial scheme:

```
Borrower(borrower_id, last_name, first_name)
Book(call_number, copy_number, barcode, title, author)
Checkout(borrower_id, call_number, copy_number, date_due)
```

PROJECT

(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.)

a) Even if we assume only one author per book, as we have done most of the time, this is not a BCNF design. Why?

   ASK

   Book contains a partial dependency - title and author depend just on `call_number`, not on the whole key. Normalizing this leads to

   ```
   Borrower(borrower_id, last_name, first_name)
   BookInfo(call_number, title, author)
   Copy(call_number, copy_number, barcode)
   Checkout(borrower_id, call_number, copy_number, date_due)
   ```

   PROJECT

b) If we allow a book to have multiple authors we also have to deal with the MVD call_number ->> author

   What would this lead to?

   ASK

   ```
   Borrower(borrower_id, last_name, first_name)
   BookTitle(call_number, title)
   BookAuthor(call_number, author)
   Copy(call_number, copy_number, barcode)
   Checkout(borrower_id, call_number, copy_number, date_due)
   ```

   PROJECT

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 Checkout table, rather than keeping `borrower_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

B. 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 <u>materialized views</u> 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.)