Part 7: Relational Normal Forms

References:

- Elmasri/Navathe: Fundamentals of Database Systems, 3rd Ed., Ch. 14, "Functional Dependencies and Normalization for Relational Databases" Ch. 15, "Relational Database Design Algorithms and Further Dependencies"
- Silberschatz/Korth/Sudarshan: Database System Concepts, 3rd Ed., Ch. 7, "Relational Database Design"
- Ramakrishnan/Gehrke: Database Management Systems, 2nd Ed., Mc-Graw Hill, 2000. Ch. 15, "Schema Refinement and Normal Forms"
- Simsion/Witt: Data Modeling Essentials, 2nd Ed.. Coriolis, 2001. Ch. 2: "Basic Normalization", Ch. 8: "Advanced Normalization".
- Batini/Ceri/Navathe: Conceptual Database Design, An Entity-Relationship Approach. Benjamin/Cummings, 1992.
- Kemper/Eickler: Datenbanksysteme (in German), Oldenbourg, 1997. Ch. 6, "Relationale Entwurfstheorie"
- Rauh/Stickel: Konzeptuelle Datenmodellierung (in German). Teubner, 1997.
- Kent: A Simple Guide to Five Normal Forms in Relational Database Theory. Communications of the ACM 26(2), 120–125, 1983.
- Thalheim: Dependencies in Relational Databases. Teubner, 1991, ISBN 3-8154-2020-2.
- Lipeck: Skript zur Vorlesung Datenbanksysteme (in German), Univ. Hannover, 1996.
- Ehrich/Neumann: Skript zur Vorlesung Datenbanksysteme I (in German), TU Braunschweig, 2000.

Objectives

After completing this chapter, you should be able to:

- work with functional dependencies (FDs),

  Define them, detect them in applications, decide whether an FD is implied by other FDs, determine a key based on FDs.
- explain insert, update, and delete anomalies.
- explain BCNF, check a given relation for BCNF, and transform a relation into BCNF.
- detect and correct violations to 4NF.
- detect normal form violations on the ER-level.
- decide about denormalization.

Overview

Introduction (1)

- Relational database design theory is based mainly on a class of constraints called "Functional Dependencies" (FDs). FDs are a generalization of keys.
- This theory defines when a relation is in a certain normal form (e.g. Third Normal Form, 3NF) for a given set of FDs.
- It is usually bad if a schema contains relations that violate the conditions of a normal form.

  However, there are exceptions and tradeoffs.

Introduction (2)

- If a normal form is violated, data is stored redundantly, and information about different concepts is intermixed. E.g. consider the following table:

COURSES			
CRN	TITLE	INAME	PHONE
22268	DB Management	Brass	9404
42232	Data Structures	Brass	9404
31822	Client-Server	Spring	9429

- The phone number of "Brass" is stored two times. In general, the phone number of an instructor will be stored once for every course he/she teaches.

Introduction (3)

- Third Normal Form (3NF) is today considered part of the general database education.
- Boyce-Codd Normal Form (BCNF) is a little bit stronger, easier to define, and better matches our intuition.

  BCNF should really replace 3NF. The only problem is that in rare circumstances, a relation cannot be transformed into BCNF with the FDs preserved. However, every relation can be transformed into 3NF with the FDs preserved.

- In short, BCNF means that all functional dependencies are already enforced by keys.

Introduction (4)

- Normalization algorithms can construct tables from a set of attributes and a set of functional dependencies.

  So in theory, database design can be done by only collecting attributes and FDs. No ER-design is needed.

- In practice, normalization is only used as an additional check.

  E.g. one does the ER-design, translates the ER-schema into the relational model, and check the resulting tables for BCNF.

Introduction (5)

- When an Entity-Relationship design is done well, the resulting tables will automatically be in BCNF (even 4NF).

  If the resulting tables are not in BCNF, one must go back to the ER-design, and correct the normal form violation there.

- Awareness of normal forms can help to detect design errors already in the ER-design phase.

  There is a normal form theory for the ER-model, too, but it is quite complicated. It is easier to understand relational normal forms and combine them with the ER-to-relational translation.

Theory vs. Intuition

- Once one understood normal forms, the intuition should be sufficient in 97% of the cases.

  It will all seem very obvious. But in order to develop the intuition, one needs the theory. Good students showed me non-normalized designs.

- But in the remaining difficult 3% of the cases, it might be necessary to apply the formal definitions.

  In order to convince other people, it is also better if one can argue with generally accepted formal definitions.

- Even Codd needed three tries to get the normal form definition right (2NF, 3NF, BCNF).

  To be fair, 2NF and 3NF were defined in the same paper.

First Normal Form (1)

- First Normal Form only requires that all table entries are atomic (not lists, sets, records, relations).
- Today, the relational model is already defined in this way. All further normal forms assume that the tables are in 1NF (First Normal Form).
- Some modern database management systems allow structured attributes. Such systems are call NF^2 systems ("Non First Normal Form").

  Structured attributes are also usually considered a requirement for an object-relational DBMS.

First Normal Form (2)

- Example of an NF^2-relation (not 1NF):

COURSES

CRN	TITLE	TAUGHT_BY	STUDENTS	
			FNAME	LNAME
22332	DB Management	Brass	John Ann	Smith Miller
31864	Client-Server	Spring	Ann	Miller

- 1NF doesn't really belong in this chapter.

  It has nothing to do with functional dependencies.

First Normal Form (3)

- Some authors feel that 1NF is already violated if there are string-valued attributes that have an inner structure (i.e. which could be further decomposed).
- Simple example: Last name and first name are put together in one attribute, separated by a comma.

  This means that one will have to use string operations in some queries.

- Really bad example: The CRNs of all courses a student is registered for are put into an attribute of the students table (separated by spaces).

  Some interesting queries will need real programming now.

First Normal Form (4)

- Some practical DB designers argue that 1NF is already violated if there are repeated attributes like DEGREE1, DEGREE2, DEGREE3 in in the instructors table.
- Normally, such attributes make queries and updates more difficult, and should be avoided.

  And is there any guarantee that there cannot be an instructor with four degrees? It is better to have a separate degree table with one row for each degree (together with the key of the instructor).

- However, formally, this is no violation of First Normal Form.

Functional Dependencies (1)

- An example of a functional dependency (FD) is

  INAME $\rightarrow$ PHONE.

- It means that whenever two rows agree in the instructor name column INAME, they must also have the same value in the column PHONE:

COURSES			
<u>CRN</u>	TITLE	INAME	PHONE
22268	DB Management	Brass	9404
42232	Data Structures	Brass	9404
31822	Client-Server	Spring	9429

Functional Dependencies (2)

- The reason for the validity of INAME $\rightarrow$ PHONE is that the contact phone number for the course depends only on the instructor, not on the other course data.
- The FD is read as: "INAME (functionally, uniquely) determines PHONE".
- One says also that INAME is a determinant for PHONE.

  Saying that A is a determinant for B is slightly stronger than the FD $A \rightarrow B$, see below (A must be minimal and $\neq B$).

- A determinant is like a partial key: It uniquely determines some attributes, but in general not all.

Functional Dependencies (3)

- A key uniquely determines every attribute, i.e. the FDs CRN$\rightarrow$TITLE, CRN$\rightarrow$INAME, CRN$\rightarrow$PHONE hold, too.

  There will never be two distinct rows with the same CRN, so the condition is trivially satisfied.

- E.g. the FD "INAME$\rightarrow$TITLE" is not satisfied. There are rows with the same INAME, but different TITLE:

COURSES			
<u>CRN</u>	TITLE	INAME	PHONE
22268	DB Management	Brass	9404
42232	Data Structures	Brass	9404
31822	Client-Server	Spring	9429

Functional Dependencies (4)

- In general, an FD has the form

$$A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m.$$

- Sequence and multiplicity of attributes in an FD are unimportant, since both sides are formally sets of attributes: $\{A_1, \ldots, A_n\} \rightarrow \{B_1, \ldots, B_m\}$.
- In discussing FDs, the focus is on a single relation R. All attributes A_i, B_i are from this relation.

Functional Dependencies (5)

- The FD $A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$ holds for a relation R in a database state I if and only if for all tuples $t, u \in I(R)$:

  If $t.A_1 = u.A_1$ and ... and $t.A_n = u.A_n$,
  then $t.B_1 = u.B_1$ and ... and $t.B_m = u.B_m$.

- I.e. $A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$ holds if there no two rows contain the same values in all columns A_i, but different values in one of the columns B_j.

Functional Dependencies (6)

- An FD with m attributes on the right hand side

$$A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$$

  is equivalent to the m FDs:

$$\begin{array}{ccc} A_1, \ldots, A_n & \rightarrow & B_1 \\ \vdots & \vdots & \vdots \\ A_1, \ldots, A_n & \rightarrow & B_m. \end{array}$$

- Thus, it suffices to consider FDs with a single column on the right hand side.

  However, sometimes it is a useful abbreviation to put multiple columns on the right hand side.

FDs are Constraints (1)

- For database design, only FDs are interesting that must hold in all possible database states.
- I.e. FDs are constraints (like keys).
- E.g. in the example state for "`COURSES`", also the FD "`TITLE` → `CRN`" holds, because no two courses have the same title.
- But probably this is not true in general, only in this small example state.

  It would be important for the design to find out whether this holds in general, i.e. there can never be two sessions of the same course.

FDs are Constraints (2)

- There are tools for analyzing example data for possible FDs, and then asking the designer whether these FDs hold in general.
- If an FD (or any constraint) does not hold in the example state, it certainly cannot hold in general.
- If one wants to use normal forms, one needs to collect all FDs that hold in general. This is a design task, it cannot be done automatically.

  Actually, only a representative subset is needed (that implies the remaining ones, see below). Of course, such tools could help.

FDs vs. Keys (1)

- FDs are a generalization of keys: $A_1, \ldots, A_n$ is a key of $R(A_1, \ldots, A_n, B_1, \ldots, B_m)$ if and only if the FD "$A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$" holds.

  Under the assumption that there are no duplicate rows. Two distinct rows that are identical in every attribute would not violate the FD, but they would violate the key. In theory, this cannot happen, because relations are sets of tuples, and tuples are defined only by their attribute values. In practice, SQL permits two identical rows in a table as long as one did not define a key (therefore, always define a key).
- Given the FDs for a relation, one can compute a key i.e. a set of attributes $A_1, \ldots, A_n$ that functionally determines the other attributes (see below).

FDs vs. Keys (2)

- Conversely, FDs can be explained with keys. FDs are keys for some columns, e.g. "INAME → PHONE" means that INAME is a key of $\pi_{\mathtt{INAME,\ PHONE}}(\mathtt{COURSES})$:

INAME	PHONE
Brass	9404
Spring	9429

  I.e. one removes all columns from the table that do not appear in the FD and then eliminates duplicate rows. $A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$ is satisfied in the original table if the projection result has no two rows with the same A_i-values (these would have to differ in a B_j).
- So an FD describes a key for part of the attributes. The goal of normalization is to make it a real key.

FDs Describe Functions (1)

- A key means that a relation (a table) really describes a partial function (with values for the key attributes as inputs and values for the other attributes as outputs).
- In the example, another function is represented in the table: It maps instructors' names to phone numbers.

  It is a partial function, because it is not defined for arbitrary strings, but only for instructor names that appear in the table.

FDs Describe Functions (2)

- Sometimes there are functions which can be computed: E.g. if the date of birth and the age are stored in the same table. Then of course the FD "BIRTHDATE $\longrightarrow$ AGE" holds.

  If two persons have the same birthdate, they have the same age.

- But we actually know much more than what the FD expresses: we know the exact formula.
- Therefore, normalization theory does not help in this case. AGE is simply redundant.

Example (1)

- The following table is used to store information about books and their authors:

BOOKS				
AUTHOR	NO	TITLE	PUBLISHER	ISBN
Elmasri	1	Fund. of DBS	Addison-W.	0805317554
Navathe	2	Fund. of DBS	Addison-W.	0805317554
Silberschatz	1	DBS Concepts	Mc-Graw H.	0471365084
Korth	2	DBS Concepts	Mc-Graw H.	0471365084
Sudarshan	3	DBS Concepts	Mc-Graw H.	0471365084

A book can have multiple authors. There is one row for every author of a book. "NO" is used to keep track of their sequence (it is not necessarily alphabetic, e.g. Silberschatz/Korth/Sudarshan).

Example (2)

- The ISBN uniquely identifies a single book. Thus the following FD holds:

  ISBN $\longrightarrow$ TITLE, PUBLISHER

- Equivalently, one can use the two FDs:

  ISBN $\longrightarrow$ TITLE
  ISBN $\longrightarrow$ PUBLISHER

- Since a book can have several authors, the following FD does not hold:

  ISBN $\longrightarrow$ AUTHOR **[Not Satisfied]**

  In the same way, ISBN does not determine NO.

Example (3)

- One author can write many books, thus the following FD cannot be assumed, although it happens to hold in the given example state:

  AUTHOR $\longrightarrow$ TITLE **[Not in general true]**

- It is possible that there are books with the same title but different authors and different publishers.

  E.g. there are several unrelated books called "Database Management". So TITLE determines none of the other attributes.

- Books can have the same author and title, but different ISBNs (paperback and hardcover edition).

Example (4)

- At every position, there can be only one author:

  `ISBN, NO` $\longrightarrow$ `AUTHOR.`

- At first, it seems impossible that the same author appears in two different positions in the author list of the same book:

  `ISBN, AUTHOR` $\longrightarrow$ `NO` **[Questionable]**

  This would be violated if there is a book from Smith & Smith.

Example (5)

- Only unquestionable conditions should be used as constraints.

  The table structure depends on the FDs. Therefore, if it should turn out later that an FD is too restrictive, it will normally not suffice to simply remove a constraint (e.g. a key) with an `ALTER TABLE` statement. Instead, one has to create new tables, copy data, and change application programs. If conversely, new FDs are discovered later, the old tables can still be used, but they violate a normal form.

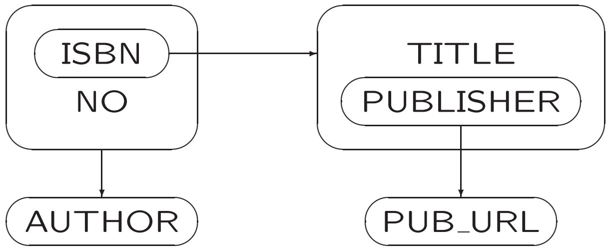
- One might also be tempted to assume this FD:

  `PUBLISHER, TITLE, NO` $\longrightarrow$ `AUTHOR` **[Questionable]**

  It is probably unlikely, but if e.g. in a new edition the author sequence changes, one is in trouble.

Example (6)

- A set of FDs can be displayed as a "hypergraph":



In a hypergraph the edges are between sets of nodes, not only between two nodes as in a standard graph.

A publisher URL was added to make the example more interesting.

Exercise (1)

- Consider a one-relation version of the homework grades DB:

HOMEWORK_RESULTS					
STUD_ID	FIRST	LAST	EX_NO	POINTS	MAX_POINTS
100	Andrew	Smith	1	9	10
101	Dave	Jones	1	8	10
102	Maria	Brown	1	10	10
101	Dave	Jones	2	11	12
102	Maria	Brown	2	10	12

- Which FDs should hold for this table (in general)?

Exercise (2)

- What does the FD "`LAST` → `FIRST`" mean?
  - ☐ If students have the same first name, they must have the same last name.
  - ☐ There cannot be siblings or other students with the same last name, different first name.
  - ☐ There cannot be different students with the same name (first and last).
- Name one FD which holds in this state, but not in general.
- Draw the hypergraph displaying the FDs.

Implication of FDs (1)

- `CRN`→`PHONE` is nothing new when one knows already `CRN`→`INAME` and `INAME`→`PHONE`.

  Whenever $A \rightarrow B$ and $B \rightarrow C$ are satisfied, $A \rightarrow C$ automatically holds.
- `PHONE`→`PHONE` holds, but is not interesting.

  FDs of the form $A \rightarrow A$ always hold (for every DB state).
- A set of FDs $\{\alpha_1 \rightarrow \beta_1, \ldots, \alpha_n \rightarrow \beta_n\}$ implies an FD $\alpha \rightarrow \beta$ if and only if every DB state which satisfies the $\alpha_i \rightarrow \beta_i$ for $i = 1, \ldots, n$ also satisfies $\alpha \rightarrow \beta$.

  α and β stand here for sets of attributes/columns. Note that this notion of implication is not specific to FDs, the same definition is used for general constraints.

Implication of FDs (2)

- One is normally not interested in all FDs which hold, but only in a representative set that implies all other FDs.
- Implied dependencies can be computed by applying the Armstrong Axioms:
  - ◇ If $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ trivially holds (Reflexivity).
  - ◇ If $\alpha \rightarrow \beta$, then $\alpha \cup \gamma \rightarrow \beta \cup \gamma$ (Augmentation).
  - ◇ If $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ (Transitivity).

Implication of FDs (3)

- However, a simpler way to check whether $\alpha \rightarrow \beta$ is implied by given FDs is to compute first the cover α^+ of α and then to check whether $\beta \subseteq \alpha^+$.
- The cover α^+ of a set of attributes α is the set of all attributes B that are uniquely determined by the attributes α (with respect to given FDs).

  $$\alpha^+ := \{B \mid \text{The given FDs imply } \alpha \rightarrow B\}.$$

  The cover α^+ depends on the given FDs, although the set of FDs is not explicitly shown in the usual notation α^+. If necessary, write $\alpha^+_{\mathcal{F}}$.
- A set of FDs $\mathcal{F}$ implies $\alpha \rightarrow \beta$ if and only if $\beta \subseteq \alpha^+_{\mathcal{F}}$.

Implication of FDs (4)

- The cover is computed as follows:

Input: α (Set of attributes)
$\alpha_1 \rightarrow \beta_1, \ldots, \alpha_n \rightarrow \beta_n$ (Set of FDs)

Output: α^+ (Set of attributes, Cover of α)

Method: $x := \alpha$;
while x did change **do**
  for each given FD $\alpha_i \rightarrow \beta_i$ **do**
    if $\alpha_i \subseteq x$ **then**
      $x := x \cup \beta_i$;
output x;

Implication of FDs (5)

- Consider the following FDs:

`ISBN` $\longrightarrow$ `TITLE, PUBLISHER`
`ISBN, NO` $\longrightarrow$ `AUTHOR`
`PUBLISHER` $\longrightarrow$ `PUB_URL`

- Suppose we want to compute $\{$`ISBN`$\}^+$.
- We start with $x = \{$`ISBN`$\}$.

x is the set of attributes for which we know that there can be only a single value. We start with the assumption that for the given attributes in α, i.e. `ISBN`, there is only one value. Then the cover α^+ is the set of attributes for which we can derive under this assumption that their value is uniquely determined (using the given FDs).

Implication of FDs (6)

- The first of the given FDs, namely

`ISBN` $\longrightarrow$ `TITLE, PUBLISHER`

has a left hand side that is entirely contained in the current set x (actually, it happens to be the same).

I.e. there is a unique value for these attributes. Then the FD means that also for the attributes on the right hand side have a unique value.

- Therefore, we can extend x by the attributes on the right hand side of this FD, i.e. `TITLE`, and `PUBLISHER`:

$x = \{$`ISBN`, `TITLE`, `PUBLISHER`$\}$.

Implication of FDs (7)

- Now the third of the FDs, namely

`PUBLISHER` $\longrightarrow$ `PUB_URL`

is applicable: Its left hand side is contained in x.

- Therefore, we can add the right hand side of this FD to x and get

$x = \{$`ISBN`, `TITLE`, `PUBLISHER`, `PUB_URL`$\}$.

- The last FD, namely

`ISBN, NO` $\longrightarrow$ `AUTHOR`

is still not applicable, because `NO` is missing in x.

Implication of FDs (8)

- After checking again that there is no way to extend the set x any further with the given FDs, the algorithm terminates and prints

  $\{\mathtt{ISBN}\}^+ = \{\mathtt{ISBN}, \mathtt{TITLE}, \mathtt{PUBLISHER}, \mathtt{PUB\_URL}\}$.
- From this, we can conclude that the given FDs imply e.g. ISBN $\rightarrow$ PUB_URL.
- In the same way, one can compute e.g. the cover of $\{\mathtt{ISBN}, \mathtt{NO}\}$. It is the entire set of attributes.

  This means that $\{\mathtt{ISBN}, \mathtt{NO}\}$ is a key of the relation, see next slide.

How to Determine Keys (1)

- Given a set of FDs (and the set of all attributes $\mathcal{A}$ of a relation), one can determine all possible keys for that relation.

  Again, one must assume that duplicate rows are excluded.
- $\alpha \subseteq \mathcal{A}$ is a key if and only if $\alpha^+ = \mathcal{A}$.
- Normally, one is only interested in minimal keys.

  The superset of a key is again a key, e.g. if $\{\mathtt{ISBN}, \mathtt{NO}\}$ uniquely identifies all other attributes, this automatically holds also for $\{\mathtt{ISBN}, \mathtt{NO}, \mathtt{TITLE}\}$. Therefore, one usually requires in addition that every $A \in \alpha$ must be necessary, i.e. $(\alpha - \{A\})^+ \neq \mathcal{A}$. Most authors make the minimality requirement part of the key definition. But then a key is not only a constraint, it also says that stronger constraints do not hold.

How to Determine Keys (2)

- One constructs a key x iteratively starting with the empty set ($x = \emptyset$).
- In order to avoid non-minimal keys, one makes sure that the set x never contains the right hand side B of an FD $\alpha \rightarrow B$, when it already contains the left hand side (i.e. $\alpha \subseteq x$).

  We assume here that all FDs have only a single attribute on the right hand side, this is no restriction. Only FDs with $B \notin \alpha$ can destroy the minimality.

How to Determine Keys (3)

- As long as x does not yet determine all attributes (i.e. $x^+ \neq \mathcal{A}$), one chooses any of the remaining attributes $X \in \mathcal{A} - x^+$.

  It is easier if one first chooses attributes that appear on as few as possible right hand sides of FDs.
- Now there are different possibilities, and one must try all:
  - ◇ One can simply add the attribute X to x.
  - ◇ For each FD $\alpha \rightarrow X$ with X on the right hand side, one can add the left hand side α to x.

How to Determine Keys (4)

- I.e. the search for the key branches. If
  - ◇ one runs into a dead end, i.e. x became non-minimal, or
  - ◇ one has found a key and wants to look for an alternative one,

  one must backtrack to the last branch and try another option.
- In the procedure shown on the next slide, the backtracking is implicitly done by the recursive calls (with "call by value" parameters).

How to Determine Keys (5)

procedure key(x, $\mathcal{A}$, $\mathcal{F}$) /* start with $x = \emptyset$ */
    for each $\alpha \rightarrow B \in \mathcal{F}$ **do**
        if $\alpha \subseteq x$ **and** $B \in x$ **and** $B \notin \alpha$ **then**
            return; /* not minimal */
    if $x^+ = \mathcal{A}$ **then**
        print x; /* minimal key found */
    else
        let X **be any element of** $\mathcal{A} - x^+$;
        key($x \cup \{X\}$, $\mathcal{A}$, $\mathcal{F}$);
        for each $\alpha \rightarrow X \in \mathcal{F}$ **do**
            key($x \cup \alpha$, $\mathcal{A}$, $\mathcal{F}$);

How to Determine Keys (6)

- One can construct a key also in a less formal way.
- So one starts with the set of required attributes (that do not appear on any right side).

  In the example one is already done: ISBN and NO appear at no right side, but their cover is the set of all attributes.
- If the required attributes do not already form a key, one adds attributes: The left hand side of an FD or directly one of the missing attributes.

  Only make sure at the end that the set is minimal. If it contains attributes that are functionally determined by other attributes in the set, remove them.

Exercise (1)

- The following relation is used for storing homework results:

RESULTS

STUD_ID	EX_NO	POINTS	MAX_POINTS
100	1	9	10
101	1	8	10
102	1	10	10
101	2	11	12
102	2	10	12

Exercise (2)

- It is known that these FDs hold:

  STUD_ID, EX_NO $\longrightarrow$ POINTS

  EX_NO $\longrightarrow$ MAX_POINTS

- Do these FDs imply the following FD?

  STUD_ID, EX_NO $\longrightarrow$ MAX_POINTS

- Does this FD imply "EX_NO $\longrightarrow$ MAX_POINTS"?

  So which of the two is the stronger restriction?

- Determine a key of the relation RESULTS.

Determinants

- $A_1, \ldots, A_n$ is called a determinant for a set of attributes $B_1, \ldots, B_m$ if and only if
  - The FD $A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$ holds.
  - The left hand side is minimal, i.e. whenever an attribute A_i is removed from the left hand side

    $$A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n \rightarrow B_1, \ldots, B_m$$

    does not hold.
  - Left and right hand side are distinct, i.e. $\{A_1, \ldots, A_n\} \neq \{B_1, \ldots, B_m\}$.

    Together with the minimality requirement this excludes trivial FDs.

Overview

1. Functional Dependencies (FDs)
2. Anomalies, FD-Based Normal Forms
3. Multivalued Dependencies and 4NF
4. Normal Forms and ER-Design
5. Denormalization

Problems (1)

- It is usually bad if a tables contains an FD that is not implied by a key (e.g. INAME$\rightarrow$PHONE).
- Among other problems, this leads to the redundant storage of certain facts. E.g. in the example table, the phone number of "Brass" is stored two times:

COURSES

CRN	TITLE	INAME	PHONE
22268	DB Management	Brass	9404
42232	Data Structures	Brass	9404
31822	Client-Server	Spring	9429

Problems (2)

- If a schema contains redundant data, a constraint must be specified to ensure that the copies of the information agree.
- In the example, this constraint is precisely the FD "INAME→PHONE".
- But FDs are not supported as one of the standard constraints of the relational model.
- Therefore, one should try to avoid FDs and transform them into key constraints. This is what normalization does.

Problems (3)

- Redundant data leads to the Update Anomalies: When a single fact needs to be changed, e.g. the phone number of "Brass", multiple tuples must be updated. This complicates application programs.
- If one forgets to change one of the tuples, the redundant copies get out of sync, and it is not clear which is the correct information.

  Application programs may break if the data does not satisfy the assumptions of the programmer. E.g. querying the phone number of an instructor should normally yield a unique value, so "SELECT INTO may be used. This gives an error if the query returns more than one value.

Problems (4)

- In the example, information about two different entities (Course and Instructor) is stored together in one table.
- This leads to insertion and deletion anomalies.
- Insertion Anomalies: The phone number of a new faculty member cannot be inserted until it is known what course he/she will teach.

  Since CRN is a key, it cannot be filled with "null".

Problems (5)

- Deletion Anomalies: When the last course of a faculty member is deleted, his/her phone number is lost.

  Even if null values were possible, it would be strange that all courses by an instructor can be deleted except the last. Then the course data must be replaced by null values instead. This complicates the logic of application programs.

Boyce-Codd Normal Form (1)

- A Relation is in Boyce-Codd Normal Form (BCNF) if and only if all its FDs are already implied by the key constraints.
- So for every FD "$A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$" for R one of the following conditions hold:
  - ◇ The FD is trivial, i.e. $\{B_1, \ldots, B_m\} \subseteq \{A_1, \ldots, A_n\}$.
  - ◇ The FD follows from a key, because $\{A_1, \ldots, A_n\}$ or some subset of it is already a key of R.

    It can be any key, not necessarily the primary key.

Boyce-Codd Normal Form (2)

- BCNF $\Longleftrightarrow$ Every determinant is a candidate key.
- So if a relation is in BCNF, FD constraints are not needed, only key constraints.
- The anomalies do not occur.

  At least not because of FDs: Since every (non-trivial) FD has a key on the left-hand side, and each key value can appear only once in the table, no redundancies occur because of the FDs. Again because every FD has a key on the left-hand side, there is no indication that information about different entities was merged.

Examples (1)

- COURSES(<u>CRN</u>, TITLE, INAME, PHONE) with the FDs
  - ◇ CRN → TITLE, INAME, PHONE
  - ◇ INAME → PHONE

  is not in BCNF because the FD "INAME → PHONE" is not implied by a key:
  - ◇ "INAME" is not a key of the entire relation.
  - ◇ The FD is not trivial.
- However, without the attribute PHONE (and its FD), the relation COURSE(<u>CRN</u>, TITLE, INAME) is in BCNF:
  - ◇ CRN → TITLE, INAME corresponds to the key.

Examples (2)

- Suppose that each course meets only once per week and that there are no cross-listed courses. Then

  CLASS(CRN, TITLE, DAY, TIME, ROOM)

  satisfies the following FDs (plus implied ones):
  - ◇ CRN → TITLE, DAY, TIME, ROOM
  - ◇ DAY, TIME, ROOM → CRN
- The keys are CRN and DAY, TIME, ROOM.
- Both FDs have a key on the left hand side, so the relation is in BCNF.

Examples (3)

- Consider the relation `PRODUCT(NO, NAME, PRICE)` with the following functional dependencies:

  (1) `NO` $\longrightarrow$ `NAME` (3) `PRICE, NAME` $\longrightarrow$ `NAME`
  (2) `NO` $\longrightarrow$ `PRICE` (4) `NO, PRICE` $\longrightarrow$ `NAME`
- This relation is in BCNF:
  - ◇ The first two FDs show that `NO` is a key. Thus, they have a key on the left hand side.
  - ◇ The third FD is trivial and can be ignored.
  - ◇ The fourth FD has a superset of the key on the left hand side, which is also no problem.

Exercises

- Is `RESULTS(STUD_ID, EX_NO, POINTS, MAX_POINTS)` with the following FDs in BCNF?

  (1) `STUD_ID, EX_NO` $\longrightarrow$ `POINTS`
  (2) `EX_NO` $\longrightarrow$ `MAX_POINTS`
- Is the relation

  `INVOICE(INV_NO, DATE, AMOUNT, CUST_NO, CUST_NAME)`

  with the following FDs in BCNF?

  (1) `INV_NO` $\longrightarrow$ `DATE, AMOUNT, CUST_NO`
  (2) `INV_NO, DATE` $\longrightarrow$ `CUST_NAME`
  (3) `CUST_NO` $\longrightarrow$ `CUST_NAME`
  (4) `DATE, AMOUNT` $\longrightarrow$ `AMOUNT`

Third Normal Form (1)

- Third Normal Form is a bit weaker than BCNF. If a relation is in BCNF, it is automatically in 3NF.

  However, the differences are small. For most practical applications, the two can be considered as equivalent.
- BCNF is easy and clear: We want no non-trivial FDs except those which are implied by a key constraint.
- In some rare circumstances, the "preservation of FDs" (see below) is lost when a relation is transformed into BCNF, whereas 3NF can always be reached without this problem.

Third Normal Form (2)

- A Relation R is in Third Normal Form (3NF) if and only if every FD "$A_1, \ldots, A_n \;\longrightarrow\; B$" satisfies at least one of the following conditions:
  - ◇ The FD is trivial, i.e. $B \in \{A_1, \ldots, A_n\}$.
  - ◇ The FD follows from a key, because $\{A_1, \ldots, A_n\}$ or some subset of it is already a key of R.
  - ◇ B is a key attribute, i.e. element of a key of R.
- FDs with multiple attributes on the right hand side $A_1, \ldots, A_n \longrightarrow B_1, \ldots, B_m$ are treated as abbreviations for the FDs $A_1, \ldots, A_n \longrightarrow B_i$ ($i = 1, \ldots, m$).

Third Normal Form (3)

- The only difference to BCNF is the additional third possibility for showing that an FD does not violate the normal form.
- An attribute is called a non-key attribute if it does not appear in any minimal key.

  Not necessarily the primary key, but any candidate key. The minimality requirement is important here because otherwise the entire set of attributes of a relation would always qualify as a key.
- 3NF means that every determinant of a non-key attribute is a key.

  Again, not necessarily the primary key, but any candidate key.

Transitive Dependencies (1)

- BCNF and 3NF can also be defined via transitive dependencies.
- A relation is in BCNF iff there are no attribute sets α, β, γ such that
  - ◇ $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$ are implied by the given FDs,
  - ◇ $\beta \not\rightarrow \alpha$ (i.e. $\beta \rightarrow \alpha$ is not implied),
  - ◇ $\gamma \not\subseteq \beta$.
- It suffices to consider sets γ that consist of a single attribute C.

Transitive Dependencies (2)

- The charactization of 3NF is again the same, except that now γ must consist of non-key attributes.

  Or alternatively, its single element C must be a non-key attribute.
- In the literature, transitive dependencies are often mentioned in connection with 3NF. But they apply equally well to BCNF.

  The reason is probably that transitive dependencies motivate why 2NF is not enough, and most textbooks use the sequence (1NF), 2NF, 3NF, BCNF. It is quite obvious that a definition with transitive dependencies is more complicated than the definition given above. Note especially that here implied FDs must be taken into account.

Second Normal Form (1)

- 2NF is only interesting for historical reasons. It is too weak, e.g. the "COURSES" example actually satisfies 2NF.
- If a relation is in 3NF or BCNF, it is automatically in 2NF.
- A relation is in Second Normal Form (2NF) if and only if every non-key attribute depends on the complete key.

Second Normal Form (2)

- I.e. a relation is in 2NF if and only if the given FDs do not imply an FD $A_1, \ldots, A_n \rightarrow B$ such that:
  - ◇ $A_1, \ldots, A_n$ is a strict subset of a minimal key, and
  - ◇ B is not contained in any minimal key.

    I.e. a non-key attribute.
- E.g., the homework results table is not in 2NF:

  `RESULTS(STUD_ID, EX_NO, POINTS, MAX_POINTS)`

  `MAX_POINTS` depends only on part of the key.

Splitting Relations (1)

- When a table is not in BCNF, one can split it into two tables ("decomposition").
- E.g. `COURSES(CRN, TITLE, INAME, PHONE)` is split into

  `COURSES_NEW(CRN, TITLE, INAME)`
  `INSTRUCTORS(INAME, PHONE)`
- If the FD $A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$ violates BCNF, one creates a new relation $S(\underline{A_1}, \ldots, \underline{A_n}, B_1, \ldots, B_m)$ and removes $B_1, \ldots, B_m$ from the original relation.

  In unusual cases (multiple violations), it might be necessary to repeat the splitting step with one or both of the resulting relations. Then you have to consider also implied FDs.

Splitting Relations (2)

- It is important that this transformation is "lossless", i.e. that the original relation can be reconstructed by means of a join:

  $$\mathtt{COURSES} = \mathtt{COURSES\_NEW} \bowtie \mathtt{INSTRUCTORS}.$$
- I.e. the original relation can be defined as a view:

```
CREATE VIEW COURSES(CRN, TITLE, INAME, PHONE)
AS
SELECT C.CRN, C.TITLE, C.INAME, I.PHONE
FROM   COURSES_NEW C, INSTRUCTORS I
WHERE  C.INAME = I.INAME
```


Splitting Relations (3)

- The split of the relations is guaranteed to be lossless if the intersection of the attributes of the new tables is a key of at least one of them ("decomposition theorem"):

  $$\{\mathtt{CRN}, \mathtt{TITLE}, \mathtt{INAME}\} \cap \{\mathtt{INAME}, \mathtt{PHONE}\} = \{\mathtt{INAME}\}.$$
- The above method for transforming relations into BCNF does only splits that satisfy this condition.
- It is always possible to transform a relation into BCNF by lossless splitting (if necessary repeated).

Splitting Relations (4)

- Not every lossless split is reasonable:

STUDENTS		
SSN	FIRST_NAME	LAST_NAME
111-22-3333	John	Smith
123-45-6789	Maria	Brown

- Splitting this into `STUD_FIRST(SSN,FIRST_NAME)` and `STUD_LAST(SSN,LAST_NAME)` is lossless, but
  - ◇ is not necessary to enforce a normal form and
  - ◇ only requires costly joins in later queries.

Splitting Relations (5)

- Losslessness means that the resulting schema can represent all states which were possible before.

  We can translate states from the old schema into the new schema and back (if the FD was satisfied). The new schema supports all queries which the old schema supported: We can define the old relations as views.

- However, the new schema allows states which do not correspond to a state in the old schema: Now instructors without courses can be stored.
- Thus, the two schemas are not equivalent: The new one is more general.

Splitting Relations (6)

- If instructors without courses are possible in the real world, the decomposition removes a fault in the old schema (insertion and deletion anomaly).
- If they are not,
  - ◇ a new constraint is needed that is not necessarily easier to enforce than the FD, but at least

    None of the two can be specified declaratively in the `CREATE TABLE` statement. Thus, nothing is gained or lost.

  - ◇ the redundancy is avoided (update anomaly).

Splitting Relations (7)

- It should also be remarked that although computed columns (such as `AGE` from `BIRTHDATE`) lead to violations of BCNF, splitting the relation is not the right solution.

  The split would give a table `R(BIRTHDATE, AGE)` which does not have to be stored because it can be computed.

- The right solution is to eliminate `AGE` from the table, but to define a view which contains all columns of the table plus the computed column `AGE`.

Preservation of FDs (1)

- Another property, which a good decomposition of a relation should satisfy, is the preservation of FDs.
- The problem is that an FD can refer only to attributes of a single relation.

  Of course, you could still have a general constraint which states that the join of the two tables must satisfy an FD.
- When you split a relation, there might be FDs that can no longer be expressed.

  Of course, you can try to find implied FDs such that each FD refers only to the attributes of one of the resulting relations, but together imply the global FD. But even this is not always possible.

Preservation of FDs (2)

- A classical example is

  `ADDRESSES(STREET_ADR, CITY, STATE, ZIP)`

  with the FDs:

  (1) `STREET_ADR, CITY, STATE` $\longrightarrow$ `ZIP`

  (2) `ZIP` $\longrightarrow$ `STATE`
- The second FD violates BCNF and would force us to split the relation into
  - ◇ `ADDRESSES1(STREET_ADR, CITY, ZIP)`
  - ◇ `ADDRESSES2(ZIP, STATE)`
- But now the first FD can no longer be expressed.

Preservation of FDs (3)

- Probably most database designers would not split the table (it is actually in 3NF, but violates BCNF).
- Textbooks say that it is more important to preserve FDs than to achieve BCNF.
- This is probably not the real reason: Few DB designers would actually write programs/triggers which check the FD.
- It does not often happen that there are two customers with exactly the same address.

  Only then the first FD could be potentially violated.

Preservation of FDs (4)

- If there are many addresses with the same ZIP code, there will be redundancies. If you split, you need expensive joins.
- Here the dependencies between ZIP code and other parts of the address are not considered as an independent fact, they are only interesting in the context of a given address (so insertion and deletion anomalies do not arise).

  Modifications are also relatively uncommon.

Preservation of FDs (5)

- If this were a DB for a mailing company, a table of ZIP codes might be of its own interest. Then the split should be done.
- The following table (with added customer number) is not even in 3NF, yet the same considerations apply:

  CUSTOMER(<u>NO</u>, NAME, STREET_ADR, CITY, STATE, ZIP)

Algorithm for 3NF (1)

- The following algorithm ("Synthesis Algorithm") produces a lossless decomposistion of a given relation into 3NF relations that preserve the FDs.
- First, one determines a minimal (canonical) set of FDs that is equivalent to the given FDs as follows:
  - ◇ Replace every FD $\alpha \to B_1, \ldots, B_m$ by the corresponding FDs $\alpha \to B_i$ ($i = 1, \ldots, m$). Let the result be F.
  - ◇ Continued on next slide ...

Algorithm for 3NF (2)

- Computation of canonical set of FDs, continued:
  - ◇ Minimize the left hand side of every FD:

    For each FD $A_1, \ldots, A_n \to B$ and each $i = 1, \ldots, n$ compute the cover $\{A_1, \ldots, A_n\}^+$ with respect to the set of functional dependencies $\mathcal{F}' := (\mathcal{F} - \{A_1, \ldots, A_n \to B\}) \cup \{A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n \to B\}$. If the result contains B, remove the attribute A_i from the left hand side of the FD, i.e. continue with $\mathcal{F} := \mathcal{F}'$ (A_i is not necessary to uniquely determine B).

  - ◇ Remove implied FDs.

    For each FD $\alpha \to B$ in $\mathcal{F}$, compute the cover α^+ with respect to $\mathcal{F}' := \mathcal{F} - \{\alpha \to B\}$. If the cover contains B, remove the FD $\alpha \to B$, i.e. continue with $\mathcal{F} := \mathcal{F}'$ ($\alpha \to B$ is implied by the other FDs).

Algorithm for 3NF (3)

- Synthesis Algorithm:
  - ◇ Compute the above minimal set of FDs $\mathcal{F}$.
  - ◇ For each left hand side α of an FD in $\mathcal{F}$, create a relation with attributes $\mathcal{A} := \alpha \cup \{B \mid \alpha \to B \in \mathcal{F}\}$.

    Assign to this relation all FDs $\alpha' \to B' \in \mathcal{F}$ with $\alpha' \cup \{B'\} \subseteq \mathcal{A}$.
  - ◇ If none of the constructed relations contains a key of the given relation, add one relation with all attributes of a key.
  - ◇ If one of the constructed relations has a set of attributes that is a subset of another relation, remove the relation with the subset of attributes.

Summary

- Tables should not contain FDs other than those implied by the keys (i.e. all tables should be in BCNF).

  Such FDs indicate a redundancy created by combining pieces of information which should be stored separately.

- You can transform tables into BCNF by splitting them.
- Sometimes there is no really good solution, and not doing the split (which would give BCNF) might be the better of two bad things. But you should know what you are doing.

Overview

Introduction (1)

- The development of BCNF has been guided by a particular type of constraints: FDs.
- The goal of 3NF/BCNF is to
  - ◇ eliminate the redundant storage of data that follows from these constraints, and to
  - ◇ transform the tables such that the constraints are automatically enforced by means of keys.
- However, there are other types of constraints which are also useful for determining the table structure.

Introduction (2)

- The condition in the decomposition theorem is only
  - ◇ sufficient (it guarantees losslessness),
  - ◇ but not necessary (a decomposition may be lossless even if the condition is not satisfied).
- Multivalued dependencies are constraints which give a necessary and sufficient condition for lossless decomposition.
- They lead to Fourth Normal Form (4NF).

Introduction (3)

- Intuitively, 4NF means: Whenever it is possible to split a table (i.e. the decomposition is lossless), and this is not superfluous (see, e.g., slide 7-73), do it.
- Still shorter: 4NF means "Do not store unrelated information in the same relation."
- Probably, in practice it is very seldom that a relation violates 4NF, but not BCNF.

  However, I have seen students merging two binary relationships to one ternary relationship, which gives such a case. See below.
- But theoretically, it is a nice roundoff.

Multivalued Dependencies (1)

- Suppose that every employee knows a set of programming languages and a set of DBMS and that these are independent facts about the employees:

EMP_KNOWLEDGE		
ENAME	PROG_LANG	DBMS
John Smith	C	Oracle
John Smith	C	DB2
John Smith	C++	Oracle
John Smith	C++	DB2
Maria Brown	Prolog	Access
Maria Brown	Java	Access

Multivalued Dependencies (2)

- If the sets of known DBMS and known programming languages are independent facts, the table contains redundant data, and must be split:

EMP_LANG	
ENAME	PROG_LANG
John Smith	C
John Smith	C++
Maria Brown	Prolog
Maria Brown	Java

EMP_DBMS	
ENAME	DBMS
John Smith	Oracle
John Smith	DB2
Maria Brown	Access

- The original table is in BCNF (no non-trivial FDs).

Multivalued Dependencies (3)

- The table can only be decomposed if the knowledge of DBMS and programming languages is indeed independent.
- If a row means that the employee knows the interface between the language and DBMS, the split would lead to a loss of information.

  Then it would be only by chance that e.g. "John Smith" knows all four possible interfaces. If e.g. he would know only the interface between C and Oracle, and the interface between C++ and DB2, the contents of the two tables would be the same. One would not know exactly which interface the employee knows.

Multivalued Dependencies (4)

- The multivalued dependency

  $$\texttt{ENAME} \twoheadrightarrow \texttt{PROG\_LANG}$$

  means that the set of values for PROG_LANG that is associated with every ENAME is independent of the other columns.

  Hidden in the table is a mapping from "ENAME" to sets of "PROG_LANG".

- Formally, "ENAME $\twoheadrightarrow$ PROG_LANG" holds if whenever two tuples agree in the value for ENAME, one can exchange their values for PROG_LANG and get two other tuples in the table.

Multivalued Dependencies (5)

- E.g. from the two table rows

ENAME	PROG_LANG	DBMS
John Smith	C	Oracle
John Smith	C++	DB2

  and the multivalued dependency, one can conclude that the table must contain also these two rows:

ENAME	PROG_LANG	DBMS
John Smith	C++	Oracle
John Smith	C	DB2

- This expresses the independence of PROG_LANG for a given ENAME from the rest of the table.

Multivalued Dependencies (6)

- A multivalued dependency (MVD)

  $$A_1, \ldots, A_n \twoheadrightarrow B_1, \ldots, B_m.$$

  is satisfied in a DB state I if and only if for all tuples t and u that have the same values for the columns $A_1, \ldots, A_n$ (i.e. $t.A_i = u.A_i$ for $i = 1, \ldots, n$), there are two further tuples t' and u' such that
  - ◇ t' has the same values as t for all columns except that $t'.B_j = u.B_j$ for $j = 1, \ldots, m$, and
  - ◇ u' agrees with u except that $u'.B_j = t.B_j$

  (i.e. the values for the B_j are exchanged).

Multivalued Dependencies (7)

- Multivalued dependencies come always in pairs:

  If "ENAME $\twoheadrightarrow$ PROG_LANG" holds,

  then "ENAME $\twoheadrightarrow$ DBMS" is automatically satisfied.

  For $R(A_1, \ldots, A_k, B_1, \ldots, B_n, C_1, \ldots, C_m)$ the multivalued dependency $A_1, \ldots, A_k \twoheadrightarrow B_1, \ldots, B_n$ is equivalent to $A_1, \ldots, A_k \twoheadrightarrow C_1, \ldots, C_m$. Exchanging the B_j values in the two tuples is the same as exchanging the values for all other columns (the A_i values are identical, so exchanging them has no effect).

Multivalued Dependencies (8)

- An MVD $A_1, \ldots, A_n \twoheadrightarrow B_1, \ldots, B_m$ for a relation R is trivial if and only if
  - $\{B_1, \ldots, B_m\} \subseteq \{A_1, \ldots, A_n\}$ or

    The precondition for exchanging the B_j-values in the two tuples is that they agree in the A_i-values. But if every B_j is also an A_i, only equal values are exchanged, which has no effect.
  - $\{A_1, \ldots, A_n\} \cup \{B_1, \ldots, B_m\}$ are all columns of R.

    In this case, exchanging the B_j values of tuples t and u gives the tuples u and t, and no new tuples.

Multivalued Dependencies (9)

- If the FD $A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$ holds, the corresponding MVD $A_1, \ldots, A_n \twoheadrightarrow B_1, \ldots, B_m$ is trivially satisfied.

  The functional dependency means that if t and u have the same values for the A_i, then they also have the same values for the B_j. But then exchanging their B_j values changes nothing.
- As an FD can be implied (i.e. automatically true) given a set of FDs, FDs and MVDs can also follow from a set of given FDs and MVDs.

  A constraint φ is implied by constraints $\{\varphi_1, \ldots, \varphi_n\}$ if and only if every database state which satisfies $\{\varphi_1, \ldots, \varphi_n\}$ also satisfies φ.

FD/MVD Deduction Rules

- The following deduction rules permit to derive all implied FDs/MVDs (they are sound and complete):
  - The three Armstrong axioms for FDs.
  - If $\alpha \twoheadrightarrow \beta$ then $\alpha \twoheadrightarrow \gamma$, where γ are all remaining columns of the relation.
  - If $\alpha_1 \twoheadrightarrow \beta_1$ and $\alpha_2 \supseteq \beta_2$, then $\alpha_1 \cup \alpha_2 \twoheadrightarrow \beta_1 \cup \beta_2$.
  - If $\alpha \twoheadrightarrow \beta$ and $\beta \twoheadrightarrow \gamma$, then $\alpha \twoheadrightarrow (\gamma - \beta)$.
  - If $\alpha \rightarrow \beta$, then $\alpha \twoheadrightarrow \beta$.
  - If $\alpha \twoheadrightarrow \beta$, $\beta' \subseteq \beta$, and there is γ with $\gamma \cap \beta = \emptyset$ and $\gamma \rightarrow \beta'$, then $\alpha \rightarrow \beta'$.

Fourth Normal Form (1)

- A relation is in Fourth Normal Form (4NF) with respect to given FDs and MVDs if and only if no MVD $A_1, \ldots, A_n \twoheadrightarrow B_1, \ldots, B_m$ is implied which is
  - not trivial and
  - not directly implied by a key, i.e. $A_1, \ldots, A_n$ does not functionally determine all other attributes.
- The definition of 4NF is very similar to the BCNF definition: It only looks at implied MVDs instead of given FDs.

Fourth Normal Form (2)

- Since every FD is also an MVD, 4NF is stronger than BCNF (i.e. if a relation is in 4NF, it is automatically in BCNF).

  If might first seem that an FD violating BCNF could lead to a trivial MVD: The second case for trivial MVDs has no counterpart for FDs. But if $\{A_1, \ldots, A_n\} \cup \{B_1, \ldots, B_m\}$ are all columns of R, the FD corresponds to a key and cannot violate BCNF.

- EMP_KNOWLEDGE(ENAME, PROG_LANG, DBMS) is an example of a relation that is in BCNF, but not in 4NF.

  It has no non-trivial FDs (the key consists of all attributes).

Fourth Normal Form (3)

- But if there are other columns besides the key of the entity (ENAME) and a multivalued attribute, even 2NF is violated:

EMPLOYEES		
ENAME	PROG_LANG	SALARY
John Smith	C	58000
John Smith	C++	58000
Maria Brown	Prolog	62000
Maria Brown	Java	62000

- It is not very common that 4NF is violated, but BCNF is not.

Fourth Normal Form (4)

- Splitting a relation

$$R(A_1, \ldots, A_n, B_1, \ldots, B_m, C_1, \ldots, C_k)$$

  into relations

$$R_1(A_1, \ldots, A_n, B_1, \ldots, B_m) \quad \text{and}$$
$$R_2(A_1, \ldots, A_n, C_1, \ldots, C_k)$$

  is lossless, i.e.

$$R = \pi_{A_1,\ldots,A_n,B_1,\ldots,B_m}(R) \bowtie \pi_{A_1,\ldots,A_n,C_1,\ldots,C_k}(R)$$

  if and only if $A_1, \ldots, A_n \twoheadrightarrow B_1, \ldots, B_m$.

  Or equivalently $A_1, \ldots, A_n \twoheadrightarrow C_1, \ldots, C_k$.

Fourth Normal Form (5)

- Any relation can be transformed into 4NF by splitting it as shown above.

  It might be necessary to split it multiple times.

- So the essence of 4NF is:
  - ◇ If a decomposition into two relations is lossless (i.e. the original relation can always be reconstructed by a join),
  - ◇ and the two resulting relations do not have identical keys (then the split would be superfluous),
  - ◇ then this decomposition must be done.

Fifth Normal Form (1)

- Fifth normal form is very seldom used in practice.

  Many textbooks actually do not discuss it at all (more than half of those I checked). 5NF is also called projection-join normal form (PJNF).

- 4NF handles all cases where a decomposition into two tables is needed.
- However, it is theoretically possible that only a decomposition into three or more tables is lossless, but no decomposition in two tables is lossless.

  This means that the required decomposition cannot be reached by repeated splitting into two tables. Instead, one needs additional tables with overlapping columns which only serve as a filter in the join.

Fifth Normal Form (2)

- E.g. consider

  `COURSE_OFFER(TITLE, TERM, INSTRUCTOR)`

- Normally, information is lost by the split into
  - ◇ `OFFERED(TITLE, TERM)`,
  - ◇ `EMPLOYED(INSTRUCTOR, TERM)`,
  - ◇ `TEACHES(TITLE, INSTRUCTOR)`.
- But the split would be lossless if following constraint were true: "If a course C is offered in a term T, and instructor I ever taught C and teaches some course in T, then I teaches C in T."

Fifth Normal Form (3)

- A join dependency (JD) states that a split into n relations is lossless:

  $$R = \pi_{A_{i_{1,1}},\ldots,A_{i_{1,k_1}}}(R) \bowtie \cdots \bowtie \pi_{A_{i_{n,1}},\ldots,A_{i_{n,k_n}}}(R).$$

  Of course, every attribute of R must appear in at least one of the projections. Then "$\subseteq$" is always satisfied, only "$\supseteq$" is interesting. It states that if there are n tuples $t_1, \ldots, t_n$ in R that agree in the values for the attributes listed in more than one projection, then one can construct a tuple from them that must also appear in R.

- An MVD $A_1, \ldots, A_n \twoheadrightarrow B_1, \ldots, B_m$ corresponds to

  $$R = \pi_{A_1,\ldots,A_n,B_1,\ldots,B_m}(R) \bowtie \pi_{A_1,\ldots,A_n,C_1,\ldots,C_k}(R).$$

  where $C_1, \ldots, C_k$ are the remaining columns of R.

Fifth Normal Form (4)

- A relation R is in Fifth Normal Form (5NF) if and only if every join dependency that holds for it is already implied by a key for R.

  Note that trivial constraints are always implied, so they do not have to be treated specially.

- Of course, 5NF implies 4NF, BCNF, 3NF, 2NF.
- If a relation is in 3NF, and all its keys consist of a single column each, it is automatically in 5NF.

  So in this case, it is not necessary to consider multivalued dependencies and join dependencies.

Domain-Key Normal Form (1)

- Consider a table for possible answers in multiple choice tests:

ANSWERS			
QUESTION	ANSWER	TEXT	CORRECT
1	A	...	Y
1	B	...	N
1	C	...	N
2	A	...	N
2	B	...	Y
2	C	...	N

Domain-Key Normal Form (2)

- The following is an example for a constraint that is not an FD, MVD, or JD:

  Each question can have only one correct answer.

- Domain-Key Normal Form (DKNF) requires that every constraint on the relation is implied by the domains and keys defined for that relation.
- It is of course nice if a relation is in DKNF: One only has to enforce domains and keys.
- Many relations cannot be transformed into DKNF.

Domain-Key Normal Form (3)

- Here, a horizontal decomposition might be useful:

CORRECT_ANSWERS		
QUESTION	ANSWER	TEXT
1	A	...
2	B	...

WRONG_ANSWERS		
QUESTION	ANSWER	TEXT
1	B	...
1	C	...
2	A	...
2	C	...

Domain-Key Normal Form (4)

- Now the key of CORRECT_ANSWERS enforces that every question has only one correct answer.

  CORRECT_ANSWERS may even be merged with a table QUESTIONS that contains the text of each question. In this way, it is certain that every question has exactly one correct answer.

- Each relation (considered in isolation) is in DKNF.
- However, a new inter-relational constraint must be enforced: The same question with the same answer may not appear in both relations.
- Exercise: Discuss which schema is better.

Domain-Key Normal Form (5)

- If every domain contains at least two values, DKNF implies 4NF.

  It also implies 5NF if every domain contains at least as many values as there are components in a join dependency.

- One important goal of normalization is indeed to replace general constraints as far as possible by standard constraints.

  Domain and key constraints are very simple constraints, supported in (nearly) every DBMS. Today, also `CHECK`-constraints defined on entire tuples could be used (domain constraints are basically `CHECK`-constraints for single attributes).

Domain-Key Normal Form (6)

Summary:

- If one has only FDs, BCNF is probably the best one can do. The next step was to look at more general constraints.
- 5NF is the end of vertical decomposition, i.e. using projections for normalization.
- But, as the example showed, also horizontal decomposition or other schema transformations should be considered.
- DKNF is the ultimate normal form.

Overview

Introduction (1)

- If a good ER-schema is transformed into the relational model, the resulting tables will satisfy all normal forms.

  Otherwise it was not a good ER-schema . . .

- If normal form violations are detected in the relational schema, one must go back to the ER-schema and correct them there.

  Unless one has done special tricks during the logical design, a normal form violation always means that the same problem occurs in the ER-schema. It is important that ER-schema and relational schema remain in sync, otherwise the ER-schema loses its value as a documentation for the actually implemented database.

Introduction (2)

- Of course, it is better to detect the errors directly in the ER-schema.

  The earlier one detects an error, the cheaper it is to correct it.

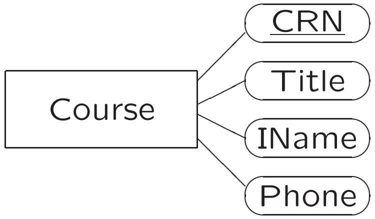
- There is no independent normalization theory for ER-schemas.

  It is possible to define normal forms like BCNF for ER-schemas, but this is significantly more complicated than for relational schemas, and does not really give something new (as far as I know).

- The definition is simply: An entity or a relationship is in BCNF if and only if the corresponding table is in BCNF.

Examples (1)

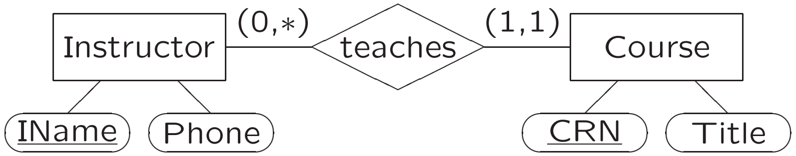
- Consider again the first example of this chapter:



- Here the functional dependency "IName → Phone" leads to the violation of BCNF discussed above.
- Also in the ER-model, FDs between attributes of an entity should be implied by a key constraint.

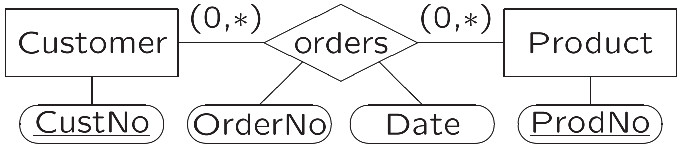
Examples (2)

- In the ER-model the solution is the same as in the relational model: We have to split the construct.
- In this case, we discover that "Instructor" is an independent entity:



Examples (3)

- Functional dependencies between attributes of a relationship always violate BCNF:



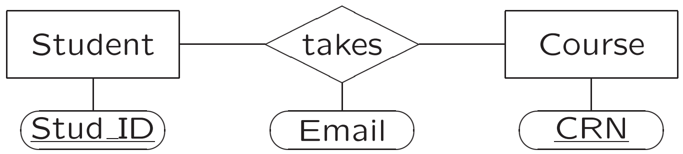
- The FD "OrderNo → Date" violates BCNF.

  The key of the table corresponding to the relationship "orders" consists of "CustNo" and "ProdNo".

- This shows that "Order" is an independent entity.

Examples (4)

- Violations of BCNF might also be due to the wrong placement of an attribute:



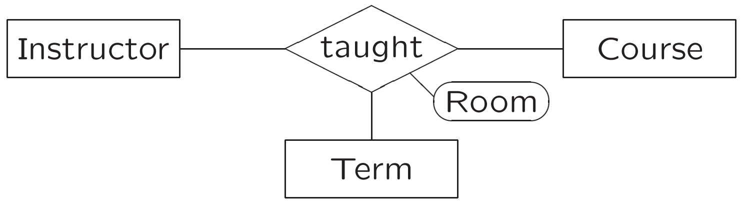
- The relationship is translated into

  Takes(<u>Stud_ID</u>, <u>CRN</u>, Email).

- Then the FD "Stud_ID → Email" violates BCNF.
- Obviously "Email" is an attribute of "Student".

Examples (5)

- If an attribute of a ternary relationship depends only on two of the entities, this violates BCNF (2NF):

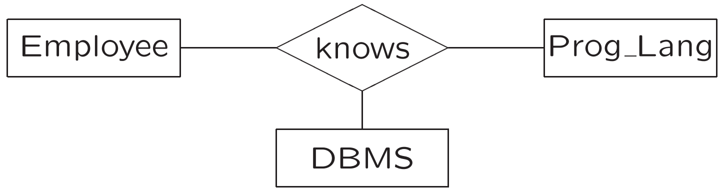


- If every course is taught only once per term, the "Room" depends only on "Course" and "Term".

  Solution hint: Create "Course_Offer" as association entity between "Course" and "Term".

Examples (6)

- If independent relationships are combined, 4NF is violated:



- If the knowledge of programming languages and DBMSs is independent, one must use two binary relationships instead.

  One between "Employee" and "Prog_Lang", and one between "Employee" and "DBMS".

Summary (1)

- Many errors in ER-design manifest themselves later as violations of relational normal forms.
- So it is a good test to think about FDs on the created tables and check for normal forms.

  One should check whether the tables make sense and not blindly trust the automatic generation from an ER-schema. Of course, the automatic translation preserves equivalence. However, the ER-schema might contain errors which have been overlooked earlier.

- Think about FDs already when developing the ER-schema, and not only after the translation!

Summary (2)

- Normalization is about:
  - ◇ Avoiding redundancy.
  - ◇ Storing separate facts separately.
  - ◇ Transforming general integrity constraints into constraints that are supported by the DBMS.
- Relational normalization theory is based mainly on FDs, but there are other types of constraints.

  The ER-model is also richer in constructs and built-in constraints.
- Instead of simply applying relational normal forms to ER-schemas, follow the above three goals!

Overview

Denormalization (1)

- Denormalization is the process of adding redundant columns and tables to the database in order to improve performance.
- E.g. if the phone number of the instructor of a course is often needed, the column `PHONE` can be added to the table `COURSES`.
- Then the join between `COURSES` and `INSTRUCTORS` can often be avoided, because the required instructor information (`PHONE`) is stored redundantly in the same row as the course information.

Denormalization (2)

- Since there is a separate table `INSTRUCTORS` (which contains the phone number, too), insertion and deletion anomalies are avoided.
- But there will be update anomalies (changing a single phone number requires updating many rows).
- Thus, one must pay for the performance gain with a more complicated application logic, the need for triggers, and some remaining insecurity (will all copies always agree?).

Denormalization (3)

- If the tables `COURSES` and `INSTRUCTORS` are small, it is certainly a bad decision to violate BCNF here, since performance is anyway no problem.
- Koletzke/Dorsey state that if your tables have less than 100 000 rows, you need no denormalization.

  Of course, it also depends on the number of queries per second.
- Too much denormalization can make a database nearly unmodifiable.

  The requirements often change, so one must anticipate that the DB application system will need changes.

Denormalization (4)

- In the above example, denormalization helped to avoid joins.
- Denormalization also includes creating tables or columns which hold aggregated values.

  In this case, formally no normal form is violated, but the information is of course redundant.
- E.g. suppose one stores invoices to a customer, and payments from a customer. One could e.g. store in the `CUSTOMERS` table the current balance.

  This is redundant, because it can be computed as the sum of all payments minus the sum of all invoices.