

Review

A brief review of what we have seen so far:

- The entity-relationship model: conceptual data model
 - entities, relations, attributes
 - ER-diagram
 - cardinality of relationships
- The relational model: logical/record-based data model
 - relations, attributes, tuples
 - relation schema (which attributes)
 - relation instance
 - relational database schema, relational database instance
 - **superkey** – any group of attributes that can uniquely identify a tuple *in any instance*.
e.g., for a particular instance (FNAME, LNAME) may appear to be a key – but it can't be guaranteed.
 - **key** – a minimal superkey, e.g.,
(LNAME, SSN) is a superkey
(SSN) is a key
 - **primary key** – one key designated for general use.
 - **foreign key** – a set of attributes in relation R that is the primary key for relation S .
- Integrity constraints (defined over all instances):
 - **key constraint** – no two tuples can have identical keys
If a tuple is found that violates the condition, then either
 - * the tuple should be rejected or,
 - * the key is not truly a key (i.e., a bad choice)
 - **entity integrity constraints** – no primary key value can be null.
 - **referential integrity constraint** – can't have a tuple whose foreign key values don't exist in the foreign relation *instance*.
e.g. $\langle \text{John, B, Smith, ... , 9} \rangle$ and $\text{DNO}=9$ does not exist at present.
- Relational algebra:
 - Operators such as σ , Π and \bowtie .
- Other formal languages:

- Tuple relational calculus
- Domain relational calculus
- Application languages:
 - SQL
 - QUEL, QBE

Next: relational schema design.

Two levels of considering the effectiveness of a schema:

1. **Logical level** – whether a schema has intuitive appeal for users.
2. **Manipulation level** – whether it makes sense from an *efficiency* or *correctness* point of view.

Functional Dependencies and Normalization

- Guidelines for database schema design: how to decide on a particular schema?
- For example, consider these two schemas for the same database:
 - Schema *S1*:

EMPLOYEE (LNAME, FNAME, SSN, DNO)
DEPT (DNUM, DNAME, MGRSSN)

- Schema *S2*:

EMPLOYEE (LNAME, FNAME, SSN, DNO, DNAME, MGRSSN)?

Which one is better?

- Informal and formal methods.
- Formal methods: involve the notion of a *functional dependency*.

Informal Guidelines

1. *Try to make user interpretation easy.*

Compare

- Relational schema *S1*:

EMP (FNAME, LNAME, SSN)
WORKS_ON (SSN, PNO)
PROJECT_LOC (PNO, PLOC)

- Relational schema $S2$:
EMP (FNAME, LNAME, SSN, PNO, PLOC)

Perhaps $S2$ has too much information (to absorb) per tuple

2. *Try to reduce redundancy.*

In schema $S2$ above, suppose there are only a few projects
 \Rightarrow PLOC is unnecessarily repeated too often

On the other hand, $S1$ repeats SSN in WORKS_ON.
 \Rightarrow But SSN is a smaller attribute than PLOC (which may be a large string)

3. *Try to avoid update anomalies*

Consider the schemas:

- Schema $S1$:

EMPLOYEE (ENAME, SSN, BDATE, ADDR, DNO)
 DEPT (DNUM, MGRSSN, DNAME)
 WORKS_ON (SSN, PNO, HOURS)
 PROJECT (PNUM, PLOCATION)

- Schema $S2$:

EMP_DEPT (ENAME, SSN, BDATE, ADDR, DNO, DNAME, MGRSSN)
 EMP_PROJ (SSN, PNUM, HOURS, ENAME, PNAME, PLOCATION)

Both have same attributes. Unfortunately, $S2$ can create problems called *update anomalies*.

- **Insertion anomalies**

- (a) Consider inserting “John Smith works in Dept. 5”, i.e.,
 <John, Smith, 123456789, ... , 5, <dname>, <mgrssn> >.

Every time a tuple of this sort is entered, we have to check that DNAME is correct
 \Rightarrow we have to scan the whole relation (worst-case)

- (b) Consider creating a new department: DNO=9, DNAME=‘Sales’

Only one way to insert this info
 \Rightarrow create NULL values for employee info
 \Rightarrow but that means a NULL primary key value (SSN)!

- **Deletion anomalies**

If we delete the last employee in the ‘Research’ department, e.g.

<John, Smith, 123456789, ... , 5, ‘Research, ... >

then we will lose the information

“Department 5 is the Research department”.

- **Modification anomalies**

Suppose we change the manager of department 5

⇒ we have to change MGRSSN for all department 5 employees

⇒ full scan of database

Thus, schema *S2* has many problems. On the other hand:

- *S1* – has 4 relations
- *S2* – has 2 relations
- for many queries, we will need more joins using *S1*
- SQL code with *S1* will be more complicated because of the extra joins
(One solution: use *S1* but create views based on needed joins)

4. *Try to avoid too many NULL values.*

- This may occur in ‘fat’ relations (with too many attributes).
- Space is wasted.
- Problems occur when using aggregate functions like **count** or **sum**.
- NULLs can have different intentions:
 - (a) the attribute does not apply
 - (b) value is unknown, and will remain unknown
 - (c) value is unknown at present

5. *Beware of the Spurious Tuple Problem.*

Consider the following two schemas:

- Schema *S1*:

EMP_PROJ (SSN, PNUMBER, ENAME, PNAME, PLOCATION)

- Schema *S2*:

EMP_LOCS (ENAME, PLOCATION)

EMP_PROJ1 (SSN, PNUMBER, PNAME, PLOCATION)

First, let us see how a relation in *S1* can be converted to relations in *S2*, e.g.,

EMP_PROJ	(<u>SSN</u> ,	<u>PNUMBER</u> ,	ENAME,	PNAME,	PLOCATION)
	123456789	1	Smith	Pepsi	New York
	987654321	2	Jones	Coke	Atlanta
	111111111	3	Brown	Sprite	Atlanta

To create EMP_LOCS:

$$\text{EMP_LOCS} \leftarrow \Pi_{\text{ENAME, PLOCATION}}(\text{EMP_PROJ})$$

Thus,

EMP_LOCS	(ENAME,	PLOCATION)
	Smith	New York
	Jones	Atlanta
	Brown	Atlants

Similarly, to create EMP_PROJ1:

$$\text{EMP_PROJ1} \leftarrow \Pi_{\text{SSN,PNUMBER,PNAME,PLOCATION}}(\text{EMP_PROJ})$$

In this case,

EMP_PROJ1	(SSN,	PNUMBER,	PNAME,	PLOCATION)
	123456789	1	Pepsi	New York
	987654321	2	Coke	Atlanta
	111111111	3	Sprite	Atlanta

Now, suppose we are using S_2 and we want to recreate S_1 (say, as a *view*):

$$\text{EMP_PROJ} \leftarrow \text{EMP_PROJ1} * \text{EMP_LOCS}$$

We get the following join:

	(<u>SSN,</u>	<u>PNUMBER,</u>	ENAME,	PNAME,	PLOCATION)
	123456789	1	Smith	Pepsi	New York
	987654321	2	Jones	Coke	Atlanta
*	987654321	2	Brown	Coke	Atlanta
*	111111111	3	Jones	Sprite	Atlanta
	111111111	3	Brown	Sprite	Atlanta

Here, the *-tuples are *spurious*!

Summary of problems:

- Insertion, deletion and modification anomalies
- Too many NULLs
- Spurious tuples

⇒ we need a theory of schema design
⇒ *functional dependencies* and *normalization*

Functional Dependencies

- First, some convenient notions (and notation):
 - Suppose our relational database has attributes A_1, \dots, A_n .
 - Let R denote the schema $R = (A_1, \dots, A_n)$.
 - Typically, of course, we will have several relations, e.g., EMP(A_1, A_3, A_9), DEPT(A_9, A_7, A_8) ... etc.
 - For convenience, let us consider a relation with *all* the attributes, i.e., with schema $R = (A_1, A_2, \dots, A_n)$.

- **Definition.** A **functional dependency** (FD) between two sets of attributes X and Y , denoted by $X \rightarrow Y$, specifies a certain relationship or connection between X and Y . Specifically, it says:
if t_1 and t_2 are any two tuples in any instance of R such that

$$t_1[X] = t_2[X]$$

then

$$t_1[Y] = t_2[Y]$$

Intuitively, $X \rightarrow Y$ means: if you know the X -values of a tuple, then that uniquely determines the Y -values.

Note: X and Y can be single attributes or *groups* of attributes.

- Example:
Consider the relational database schema:

(SSN, PNUMBER, ENAME, PNAME, PLOCATION)

Suppose

$$\begin{aligned} X &= \{\text{SSN}\} \\ Y &= \{\text{ENAME}\} \end{aligned}$$

Then $X \rightarrow Y$, i.e., SSN uniquely determines ENAME

From the definition, if we are given two tuples t_1 and t_2 , e.g.,

$$t_1 = \langle 123456789, \dots, \text{Smith}, \dots \rangle$$

$$t_2 = \langle 123456789, \dots, \text{Smith}, \dots \rangle$$

where

$$t_1[X] = t_2[X] \quad (\text{i.e., } t_1[\text{SSN}] = t_2[\text{SSN}])$$

then it must be true that

$$t_1[Y] = t_2[Y] \quad (\text{i.e., } t_1[\text{ENAME}] = t_2[\text{ENAME}]).$$

Thus, we can't have two tuples with the same SSN and different ENAME's.

Similarly, the functional dependency

$$\{\text{SSN}, \text{PNUMBER}\} \rightarrow \{\text{PNAME}, \text{PLOCATION}\}$$

is a reasonable assumption or a reasonable constraint to declare.

- Functional dependencies are specified to capture dependencies for *all* instances.

It may just happen that project locations (PLOCATION) are all different for a particular instance.

⇒ we may be led to believe that $\{\text{ENAME}\} \rightarrow \{\text{PLOCATION}\}$.

But, later on, 'Smith' might work on projects in different locations

⇒ this would be a poor choice of a FD.

A FD is a property of the meaning of attributes

⇒ it should hold for all possible instances.

- FD's for specific relations.

Although we have defined FD's on the universe of attributes, we will often discuss FD's within particular relations.

For example, the database schema might be:

$$(\text{SSN}, \text{PNUMBER}, \text{ENAME}, \text{PNAME}, \text{PLOCATION}).$$

We might have the relation

$$\text{WORKS_ON} (\text{SSN}, \text{PNUMBER}, \text{ENAME}, \text{PNAME}).$$

Here, we can identify the FD

$$\{\text{SSN}\} \rightarrow \{\text{ENAME}\}$$

in WORKS_ON.

- Sometimes a diagram is used to show FD's, (e.g., Fig. 12.3)

Sets of Functional Dependencies

- Consider the following example (Fig.12.3):

EMP_DEPT(ENAME,SSN,BDATE,ADDRESS,DNUMBER,DNAME,DMGRSSN)

The obvious FD's are:

$$\begin{array}{ll} \{SSN\} & \rightarrow \{ENAME,BDATE,ADDRESS,DNUMBER\} \\ \{DNUMBER\} & \rightarrow \{DNAME,DMGRSSN\} \end{array}$$

Let F denote the above *collection* of FD's.

From F , we can infer

$$\{SSN\} \rightarrow \{DNAME,DMGRSSN\}.$$

Why? Because, SSN uniquely determines DNUMBER and DNUMBER uniquely determines {DNAME,DMGRSSN}.

Also, the following FD's are examples of trivial FD's inferred from F .

$$\begin{array}{ll} \{SSN\} & \rightarrow \{SSN\} \\ \{SSN,DNUMBER\} & \rightarrow \{DNUMBER\} \end{array}$$

- **Definition.** Suppose F is a set of FD's. Then, F^+ , the **closure** of F is the set of FD's that includes F and all the FD's that can be *inferred* from F .

Inference Rules for FD Sets

- NOTE the following:

1. When we say $X \rightarrow Y$, X and Y are *subsets* of the universe of attributes.
2. For convenience, we will sometimes drop the set notation and commas within sets.

Suppose F is the set of FD's:

$$\begin{array}{l} X \rightarrow Y \\ X \rightarrow Z. \end{array}$$

Then, from F we can infer:

$$X \rightarrow YZ.$$

Here, YZ denotes the *union* of Y and Z .

For example, F is

$$\begin{aligned} X = \{\text{SSN}\} &\rightarrow \{\text{ENAME}\} = Y \\ X = \{\text{SSN}\} &\rightarrow \{\text{BDATE}\} = Z. \end{aligned}$$

From which we conclude

$$X = \{\text{SSN}\} \rightarrow \{\text{ENAME}, \text{BDATE}\} = Y \cup Z = YZ$$

- From above, we can devise a rule: if $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$.

Such a rule is called an **inference rule**.

- Standard inference rules for FD's:

1. **Reflexive rule.** *If $Y \subseteq X$ then $X \rightarrow Y$.*

e.g. $\{\text{SSN}, \text{BDATE}\} \rightarrow \{\text{BDATE}\}$

2. **Augmentation rule.** *If $X \rightarrow Y$ then $XZ \rightarrow YZ$ for any group of attributes Z .*

e.g.

$$\begin{aligned} X &= \{\text{SSN}\} \\ Y &= \{\text{BDATE}\} \\ Z &= \{\text{PLOCATION}\} \end{aligned}$$

Then, $\{\text{SSN}\} \rightarrow \{\text{BDATE}\}$ implies

$$\{\text{SSN}, \text{PLOCATION}\} \rightarrow \{\text{BDATE}, \text{PLOCATION}\}$$

3. **Transitive rule.** *If $X \rightarrow Y$ and $Y \rightarrow Z$ then $X \rightarrow Z$.*

e.g., the FD's

$$\begin{aligned} \{\text{SSN}\} &\rightarrow \{\text{DNUMBER}\} \\ \{\text{DNUMBER}\} &\rightarrow \{\text{DNAME}\} \end{aligned}$$

together imply

$$\{\text{SSN}\} \rightarrow \{\text{DNAME}\}.$$

4. **Decomposition rule.** *If $X \rightarrow YZ$ then $X \rightarrow Y$.*

e.g. the FD

$$\{\text{SSN}\} \rightarrow \{\text{ENAME}, \text{DNUMBER}\}$$

implies

$$\{\text{SSN}\} \rightarrow \{\text{DNUMBER}\}.$$

5. **Union rule.** *If $X \rightarrow Y$ and $X \rightarrow Z$ then $X \rightarrow YZ$.*

e.g. the FD's

$$\begin{aligned} \{\text{SSN}\} &\rightarrow \{\text{ENAME}\} \\ \{\text{SSN}\} &\rightarrow \{\text{BDATE}\} \end{aligned}$$

imply

$$\{\text{SSN}\} \rightarrow \{\text{ENAME,BDATE}\}.$$

6. **Pseudotransitive rule.** *If $X \rightarrow Y$ and $WY \rightarrow Z$ then $WX \rightarrow Z$.*

e.g. the FD's

$$\begin{array}{ll} \{\text{PNO}\} & \rightarrow \{\text{PNAME}\} \\ \{\text{SSN,PNAME}\} & \rightarrow \{\text{HOURS}\} \end{array}$$

imply

$$\{\text{SSN,PNO}\} \rightarrow \{\text{HOURS}\}.$$

- One can use Rules 1-6 to determine F^+ .
- It turns out that Rules 1-3 are sufficient to completely determine F^+ .
Rules 1-3 are called **Armstrong's Rules** in honor of the person who proved this result.
- *Equivalent FD sets.*
 - For most FD-sets F , the closure F^+ is probably quite large.
 - While we are interested in the *theoretical* implications of F^+ for any F , we are rarely going to *compute* F^+ .
⇒ Working with F turns out to be good enough.
 - **Definition.** FD-sets E and F are **equivalent** if $E^+ = F^+$, i.e., if their closures are equal.
 - If E is a set of FD's smaller than F and yet $E^+ = F^+$, then it is easier to work with E .
 - If $E^+ = F^+$ we also say that E **covers** F or is a **cover for** F .

Attribute Set Closures

- If X is an attribute set, we are often interested in *all* attributes (functionally) determined by X .
i.e., what is the largest Y for which $X \rightarrow Y$?
- **Definition.** If F is a set of FD's and X is a set of attributes, then X^+ is the **closure of X under F** if X^+ is the largest set of attributes functionally determined by X using inference rules on FD's in F .
- Algorithm for determining X^+ .

Algorithm: COMPUTE X^+

Input: X, F .

Output: X^+ .

```
1.   $X^+ \leftarrow X$ ;
2.  repeat
3.     $\text{old\_}X^+ \leftarrow X^+$ ;
4.    for each FD  $Y \rightarrow Z$  in  $F$  do
5.      if  $Y \subseteq X^+$  then
6.         $X^+ \leftarrow X^+ \cup Z$ ;
7.    until  $\text{old\_}X^+ = X^+$ ;
8.  return  $X^+$ ;
```

For example, suppose F is:

$\{\text{SSN}\}$	\rightarrow	$\{\text{ENAME}\}$
$\{\text{PNUMBER}\}$	\rightarrow	$\{\text{PNAME}, \text{PLOCATION}\}$
$\{\text{SSN}, \text{PNUMBER}\}$	\rightarrow	$\{\text{HOURS}\}$

and suppose we want the closure of $X = \{\text{SSN}, \text{PNUMBER}\}$ under F .

Initially, $X^+ \leftarrow \{\text{SSN}, \text{PNUMBER}\}$.

In the first iteration of the outerloop, $\text{old_}X^+ = \{\text{SSN}, \text{PNUMBER}\}$.

Then, there are 3 iterations of the innerloop (one for each FD). This gives us

$$X^+ = \{\text{SSN}, \text{PNUMBER}, \text{ENAME}, \text{PNAME}, \text{PLOCATION}\}.$$

The second iteration of the outerloop creates no changes.

\Rightarrow algorithm terminates with

$$X^+ = \{\text{SSN}, \text{PNUMBER}, \text{ENAME}, \text{PNAME}, \text{PLOCATION}\}.$$

Normal Forms

- Normal forms are properties of *relations*.
- There are many normal forms:

1NF - First Normal Form
2NF - Second Normal Form
3NF - Third Normal Form
 \vdots

- We say a relation is in x NF if its attributes satisfy certain properties (via their FD's).
- Generally, these properties are desirable.
- For example, if we desire the 3NF for a relational database:
 - We will test the relations in the database to see which are in 3NF.
 - Those that are not in 3NF will be decomposed into smaller relations (smaller in numbers of attributes) until we have each relation satisfying the 3NF properties.
- In the real world, most people achieve 3NF. It is slightly better to achieve BCNF (Boyce-Codd), but 3NF is considered 'not bad'.
- The end result is: if a database is in BCNF (or 3NF), many anomalies are avoided.
- FD's were designed to test for Normal Forms.

Normal Forms: Primary Key Version

- It is easier to understand normal forms by first considering a simpler version – *normal forms for primary keys*.
- Recall definitions and notation:
 - Let R denote a relation schema.
 - A **superkey** $S \subseteq R$ can be used to uniquely determine tuples.
 - A **key** is a minimal superkey.
 - A **candidate key** – same as a key. Used when several keys are under consideration.
 - A **primary key** – a **key** designated for common use.
 - A **prime attribute** – an attribute belonging to *some candidate key* (not necessarily the primary key).
 - A **nonprime attribute** – not belonging to any candidate key.

1NF: First Normal Form

- **Definition.** A relation is in 1NF if:
 1. the value of any attribute in any tuple is a single value, and
 2. domains of attributes contain only atomic values.

- Example of relations not satisfying 1NF (Fig 12.8(b)):
 - ⇒ Does not satisfy first part of definition. (Fig 12.9 b)
 - ⇒ Does not satisfy second part of definition (contains nested relations).
- It is easy to transform the above relations to satisfy 1NF by:
 1. adding tuples or
 2. creating new relations

For the first example: Fig 12.8(b),(c)

For the second example: Fig 12.9 (b),(c)

Note: we could have created additional tuples but this would have required more space.

- 1NF is nowadays taken for granted (i.e., later formulations of relational theory assume relations to be in 1NF).
 - ⇒ we will assume all relations are in 1NF.

2NF: Second Normal Form

- **Definition.** A FD $X \rightarrow Y$ is a **partial dependency** if there exists an attribute $A \in X$ such that

$$X - A \rightarrow Y.$$

We say that Y is **partially dependent** on X .

Example: consider the relation

EMP_PROJ(SSN,PNUMBER,HOURS,ENAME,PNAME,PLOCATION).

Here we can identify some FD's such as

$$\begin{aligned} \{SSN,PNUMBER\} &\rightarrow \{HOURS\} \\ \{SSN,PNUMBER\} &\rightarrow \{ENAME\} \end{aligned}$$

Now, neither one of

$$\begin{aligned} \{SSN\} &\rightarrow \{HOURS\} \\ \{PNUMBER\} &\rightarrow \{HOURS\} \end{aligned}$$

is true.

Thus, $\{SSN,PNUMBER\} \rightarrow \{HOURS\}$ is not a partial dependency.

But, $\{SSN\} \rightarrow \{ENAME\}$.

Hence, ENAME is partially dependent on the primary key $\{SSN,PNUMBER\}$.

- **Definition.** A relation schema is in 2NF if *no* nonprime attribute is partially dependent on the primary key.
- Thus, in the above example, EMP_PROJ is not in 2NF. (It is, however, in 1NF).
- Why is this a problem?
Suppose an instance looks like:

SSN	PNUMBER	HOURS	ENAME	PNAME	PLOCATION
123456789	1	10	Smith	Pepsi	New York
987654321	2	15	Jones	Coke	Atlanta

Suppose we want to insert the tuple

<987654321, 3, 12, Jones, Sprite, Atlanta>.

Because we have the FD: {SSN} → {ENAME} we will have to check that

<987654321, ..., Jones, ...>

is valid, i.e., that it matches other SSN, ENAME values for Jones.

⇒ we have to scan the whole relation (worst-case) to check

⇒ *insertion anomaly*.

Similarly, if project # 1 is deleted:

⇒ we will lose the information “123456789 is the SSN of Smith”

⇒ *deletion anomaly*.

This relation also has a *modification anomaly*:

⇒ if Smith changes name to Brown

⇒ have to propagate change to all relevant tuples.

To solve the problem, consider the decomposition of

EMP_PROJ(SSN, PNUMBER, HOURS, ENAME, PNAME, PLOCATION).

into

EP1 (SSN, PNUMBER, HOURS)
 EP2 (SSN, ENAME)
 EP3 (PNUMBER, PNAME, PLOCATION)

These relations are in 2NF, with the important FD's:

$$\begin{array}{lcl}
\{\text{SSN,PNUMBER}\} & \rightarrow & \{\text{HOURS}\} \\
\{\text{SSN}\} & \rightarrow & \{\text{ENAME}\} \\
\{\text{PNUMBER}\} & \rightarrow & \{\text{PNAME,PLOCATION}\}
\end{array}$$

There are no partial dependencies of nonprime attributes on primary keys.

3NF: Third Normal Form

- **Definition.** The FD $X \rightarrow Y$ is a **transitive dependency** in relation R if there exists a set of attributes Z in R such that

1. $X \rightarrow Z$ and $Z \rightarrow Y$
2. Z is not a subset of any key of R

Example: consider the relation

EMP_DEPT(ENAME,SSN,BDATE,ADDRESS,DNO,DNAME,MGRSSN).

- Observe that EMP_DEPT is in 2NF since no partial dependencies exist at all (and hence partial dependencies on the primary key don't exist).
- Next, we have the FD's:

$$\begin{array}{lcl}
\{\text{SSN}\} & \rightarrow & \{\text{DNO}\} \\
\{\text{DNO}\} & \rightarrow & \{\text{MGRSSN}\}
\end{array}$$

Note that DNO is not part of any key. Thus, there is a transitive dependency of MGRSSN on SSN.

- **Definition.** A relation R is in 3NF if

1. it is in 2NF and,
2. no nonprime attribute is transitively dependent on the primary key.

In the above example:

- EMP_DEPT is in 2NF
- MGRSSN is a nonprime attribute transitively dependent on the primary key SSN.

\Rightarrow EMP_DEPT is *not* in 3NF.

- Why should we care about 3NF?

Consider the following instance of EMP_DEPT:

ENAME	SSN	BDATE	ADDRESS	DNO	DNAME	MGRSSN
Smith	5	Research	111111111
Jones	6	Sales	444444444

Suppose we insert the tuple $\langle \text{Brown}, \dots, 5, \text{Research}, 333333333 \rangle$

\Rightarrow we would have to check that the DNO matches the MGRSSN (wrong in this case) \Rightarrow scanning the database \Rightarrow *insertion anomaly*.

Similarly, if Smith's DNO changes to 6

\Rightarrow we will have to also insert the correct MGRSSN \Rightarrow *modification anomaly*.

A deletion anomaly also occurs, if we delete Smith's tuple and Smith is the last Research employee.

\Rightarrow we will lose the information "Dept# 6 is Research".

- To solve a 3NF problem, we can decompose relations.

In the above example:

ED1 (ENAME, SSN, BDATE, ADDRESS, DNO)
ED2 (DNO, DNAME, MGRSSN)

Note:

1. ED1 and ED2 are in 3NF.
2. EMP_DEPT can be recovered by *joining* ED1 and ED2.
3. The join will not create spurious tuples.

General Definitions of 2NF and 3NF

- We have defined 2NF and 3NF using primary keys.
- General definitions based on any candidate key are desirable.
- 2NF:
 - *Primary version*: A relation schema R is in 2NF if every nonprime attribute A in R is not partially dependent on the primary key.
 - *General version*: A relation schema R is in 2NF if every nonprime attribute A in R is not partially dependent on any key of R .

Consider the following example:

LOTS (PROP_ID, COUNTY, LOT#, AREA, PRICE, TAX_RATE)

Here,

- The relation represents statewide information about properties, their areas (sizes), price and tax rates assessed.

- PROP_ID is the primary key.
- LOT# is the code used within a county (by the county government).
 \Rightarrow (COUNTRY, LOT#) is a candidate key.

Suppose tax rates are set by county, i.e., we have the partial FD

$$\{\text{COUNTY}\} \twoheadrightarrow \{\text{TAX_RATE}\}.$$

Then, LOTS is in 2NF according to the primary version of the definition (no partial dependency on the primary key).

But the partial dependency on COUNTY causes the general definition to fail.
 \Rightarrow LOTS is not in 2NF.

- We *want* the general definition to hold, because otherwise we will have to check the FD $\{\text{COUNTY}\} \rightarrow \{\text{TAX_RATE}\}$ for every insertion.

We can use the decomposition:

LOTS1 (PROP_ID, COUNTY, LOT#, AREA, PRICE)
 LOTS2 (COUNTY, TAX_RATE)

- Thus we see how our elaborate definitions of normal forms helps us catch problems in seemingly innocuous schemas (like LOTS).

- 3NF:

- *Primary version:* A relation R is in 3NF if
 1. it is in 2NF, and
 2. no nonprime attribute is transitively dependent on the primary key of R .
- *General version:* A relation R is in 3NF if
 1. it is in 2NF, and
 2. no nonprime attribute is transitively dependent on any key of R .

Now observe this:

- If $X \subseteq \{A_1, \dots, A_n\}$ is any key then $X \rightarrow Y$ for any $Y \subseteq \{A_1, \dots, A_n\}$
- Suppose for some relation R
 1. X is the primary key.
 2. Y is some other key.
 3. Z is *transitively dependent* on Y , i.e., there are FD's

$$Y \rightarrow W \text{ and } W \rightarrow Z.$$

But $X \rightarrow W$ since X is a key.
 $\Rightarrow Z$ is transitively dependent on X (the primary key)
 \Rightarrow the two 3NF definitions are identical.

BCNF: Boyce-Codd Normal Form

- Consider the relation ADDR_INFO (CITY, ADDRESS, ZIP).
 [Here, ADDRESS denotes street address only].

We can identify the natural FD's:

$$\begin{array}{lcl} \{CITY, ADDRESS\} & \rightarrow & \{ZIP\} \\ \{ZIP\} & \rightarrow & \{CITY\} \end{array}$$

Possible keys are: {CITY,ADDRESS} or {ADDRESS,ZIP}.

Is ADDR_INFO in 2NF?

- Recall: no nonprime attribute should have a partial dependence on a key.
- Here, there are no nonprime attributes

\Rightarrow it passes the 2NF test

Is ADDR_INFO in 3NF?

- It is in 2NF.
- Recall: we should not have any transitive dependence on a key.
- When considering {CITY,ADDRESS} as a key, there is only one attribute left (ZIP)
 \Rightarrow can't have a transitive dependency.
- Same story when {ADDRESS,ZIP} is considered as a key.

\Rightarrow ADDR_INFO is in 3NF.

On the other hand, ADDR_INFO (CITY, ADDRESS, ZIP) has all the anomalies (insertion, deletion and modification).

- The FD $\{ZIP\} \rightarrow \{CITY\}$ is the real problem.
- Suppose, we delete the tuple
 $\langle \text{Williamsburg, 55 Richmond Road, VA 23189} \rangle$.

If this is the only 23189 tuple in the relation, we will lose the information
 “VA 23189 is in Williamsburg”.

\Rightarrow *deletion anomaly*.

It is easy to check that insertion and modification anomalies are also present.

- The problem appears to be:
 - In 2NF: we did not allow FD's *from* parts of keys *to* nonprime attributes.
 - Here we have an FD *from* an attribute *to* part of a key.
- One option is to introduce the following rule for every relation:
 1. it should be in 3NF
 2. there should be no FD $X \rightarrow Y$ such that Y is part of a key.

Unfortunately, this rule is too *restrictive*.

e.g., consider

LOTS_PRICE (PROP_ID, COUNTY, LOT#, PRICE).

Here,

- PROP_ID is the primary key.
- {COUNTY, LOT#} is another key.
- The FD {PROP_ID} \rightarrow {COUNTY} follows.

Thus, if we disallow relations of this sort, we will be essentially barring all non-primary keys from having multiple attributes.

\Rightarrow may be too restrictive in practice.

- Let us try to soften the rule:
 1. it should be in 3NF
 2. for every FD $X \rightarrow Y$, such that Y is part of a key, X should itself be a key.

That is, we do allow *part of keys* to be dependent on things – provided those things are keys.

This is a reasonable assumption because:

- If X is a key, we would likely have to check uniqueness anyway (and that's all we have to do).
- Deletion causes less of an anomaly, e.g., in

LOTS_PRICE (PROP_ID, COUNTY, LOT#, PRICE)

the FD {PROP_ID} \rightarrow {COUNTY} is not important.

Deleting the only tuple with 'James City County', e.g.,

$\langle 169, \text{James City}, 23, \$50,000 \rangle,$

we lose the information “Prop_id 169 is in James City”.

But, we wouldn't be deleting this if we weren't prepared to lose “169” forever.

\Rightarrow it's OK to lose “Prop_id 169 is in James City”.

- The softened rule above needs a small modification:

Observe that in the above example, we have the FD

$$\{\text{PROP_ID}\} \rightarrow \{\text{COUNTY}\}.$$

This passes our new test.

However, the following is also an FD:

$$\{\text{PROP_ID}, \text{PRICE}\} \rightarrow \{\text{COUNTY}\}.$$

This fails the test because $\{\text{PROP_ID}, \text{PRICE}\}$ is not a key.

However, it is a *superkey* (contains a key).

- Final form:

Definition. A relation R is in Boyce-Codd Normal Form (BCNF) if:

1. it is in 3NF
2. for every FD $X \rightarrow Y$ such that Y is part of a key, X is a superkey.

3NF and BCNF: An Alternate Definition

- First recall the 3NF definition:

1. 2NF
2. no transitive dependency from a key to a nonprime attribute should exist.

Here, a transitive dependency means:

- $X \rightarrow Y$ and $Y \rightarrow Z$
- Y is not part of any key
- X is a key
- Z is a nonprime attribute

Now, since X is a key, the FD $X \rightarrow Y$ must be true for any Y .

Thus, the condition is really saying (given X is a key) that for $Y \rightarrow Z$:

- (a) Y is not part of any key

- (b) Z is a nonprime attribute

Next, recall that 2NF is essentially:

- if $Y \rightarrow Z$ then Y cannot be a proper subset of a key.

Combine this with the first item (a) in 3NF and write (b) separately:

A relation R is in 3NF if for every FD $Y \rightarrow Z$ either

1. Y is a superkey, or
2. Z is a prime attribute.

- This is an alternate definition of 3NF which does not mention 2NF.
- Note that if Z is a prime attribute, we allow $Y \rightarrow Z$ even if Y is not a superkey, e.g. in

ADDR_INFO (CITY, ADDRESS, ZIP)

we allowed $\{ZIP\} \rightarrow \{CITY\}$ because CITY is a prime attribute (it is part of the key $\{CITY, ADDRESS\}$).

But BCNF does not allow this.

Hence, an alternate definition of BCNF is:

A relation R is in BCNF if for every FD $Y \rightarrow Z$, Y is a superkey of R .

- This definition does not use 3NF.

- Finally, observe that

$$\begin{aligned} R \text{ is in BCNF} &\Rightarrow R \text{ is in 3NF} \\ &\Rightarrow R \text{ is in 2NF} \\ &\Rightarrow R \text{ is in 1NF} \end{aligned}$$

Another View of Normal Forms

- If the discussion so far has been confusing, let us try to explain normal forms a little differently.
- First, some simplifications:
 - Let us only consider relations with a single key – a primary key.
 - Assume this key has several attributes.

Note: this simplification is only for conveying the key idea behind normal forms. In practice you would have to use the full definition.

- Let $R(A_1, \dots, A_6)$ be a relation with primary key $\{A_1, A_2, A_3\}$.

- 2NF says: FD's like $A_2 \rightarrow A_5$ are not allowed.
 \Rightarrow a proper subset of a key should not be on the left side of an FD:

$$R(\underline{A_1, A_2, A_3}, A_4, A_5, A_6)$$

- 3NF says:

1. at least 2NF, and
2. FD's like $A_4 \rightarrow A_6$ are not allowed.
 \Rightarrow an FD between non-key attributes is not allowed.

$$R(\underline{A_1, A_2, A_3}, A_4, A_5, A_6)$$

- Next, BCNF:

Unfortunately 3NF allows an FD like $A_5 \rightarrow A_3$, where A_5 is nonprime and A_3 is part of the key:

$$R(\underline{A_1, A_2, A_3}, A_4, A_5, A_6)$$

We saw why this was a problem in the BCNF example.

On the other hand we did not want to be too restrictive: if A_5 happened to be a key we would allow it.

- The key ideas above are generalized to allow for:
 - multiple keys in a relation
 - keys consisting of groups of attributes.

Decomposition and its Problems

- We have seen that it is desirable to have relations in BCNF (or at least 3NF).
- We have seen how to *test* for BCNF and 3NF.
- But how do we create a BCNF (or 3NF) database?
 - One approach: Ad-hoc
 - * Create relations intuitively
 - * Test each for BCNF
 - More formal approach:

- * Start with a single large relation with all attributes
- * Systematically decompose relations not in BCNF
- * Repeat until all relations are in BCNF
- Unfortunately, decomposition can create problems:
 - Dependencies may be *lost* after decomposition.
 - Joins of decomposed relations may create *spurious tuples*.
- Thus far, we have identified problems with *individual* relations
 \Rightarrow we have not placed constraints *among* multiple relations.

Dependency Preservation

- Suppose we decompose $R = (A_1, \dots, A_n)$ into relations R_1, \dots, R_m .
- Of course, we should have **attribute preservation**, i.e., attributes should not be lost in the shuffle:

$$R_1 \cup R_2 \cup \dots \cup R_m = R$$

- Unfortunately, FD's can be lost, e.g.,
 - Suppose F is a set of FD's containing the dependency
 $\{\text{COUNTY}\} \rightarrow \{\text{TAX_RATE}\}$.
 - Suppose also that $\{\text{COUNTY}\}$ is put in relation R_3 and $\{\text{TAX_RATE}\}$ is put in relation R_7
 \Rightarrow we can't check the dependency.
- In the above example, the lost FD would be OK, if the dependency were somehow not important
 \Rightarrow we need to consider the *closure* of FD's.
- **Definition.** Let F be a set of FD's and suppose R is decomposed into relations R_1, \dots, R_m . Let G be the set of FD's in R_1, \dots, R_m . Then the decomposition is **dependency preserving** if $G^+ = F^+$.
- **Fact:** It is always possible to decompose any relation R into 3NF relations R_1, \dots, R_m such that the decomposition is dependency preserving.
 (Proof is not easy).

Nonadditive Decompositions

- Suppose we decompose R into R_1, \dots, R_m . Later, we wish to recover R (perhaps as a view).

- The natural join on R_1, \dots, R_m should return R .
- If we're not careful, this join can create spurious tuples
e.g, consider the relation r with schema $R=(\text{CAR},\text{OWNER},\text{COLOR})$:

CAR	OWNER	COLOR
Toyota	Smith	blue
Ford	Jones	blue

Suppose we decompose this into r_1 and r_2 with schemas $R_1=(\text{CAR},\text{COLOR})$ and $R_2=(\text{OWNER},\text{COLOR})$.

How? Let $r_1 = \Pi_{\text{CAR},\text{COLOR}}(r)$ and $r_2 = \Pi_{\text{OWNER},\text{COLOR}}(r)$:

CAR	COLOR
Toyota	blue
Ford	blue

OWNER	COLOR
Smith	blue
Jones	blue

What happens when we join r_1 and r_2 ?

CAR	OWNER	COLOR
Toyota	Smith	blue
Toyota	Jones	blue *
Ford	Smith	blue *
Ford	Jones	blue

\Rightarrow Spurious tuples!

- A decomposition of R into R_1, \dots, R_m is **nonadditive** if for every instance r of R , the natural join of the corresponding instances r_1, \dots, r_m is equal to r , i.e.,

$$r_1 * r_2 * \dots * r_m = r$$

where $r_i = \Pi_{R_i}(r)$.

- Note: **nonadditive** is the same as 'creates no spurious tuples'.
Also, called a **lossless** join.

Testing for Nonadditivity

- It would be useful, if given a decomposition, to test whether the decomposition is nonadditive.

- **A useful fact.** A decomposition of R into R_1 and R_2 is nonadditive with respect to a set of FD's F , if and only if either one of the FD's

$$- R_1 \cap R_2 \rightarrow R_1 - R_2$$

$$- R_1 \cap R_2 \rightarrow R_2 - R_1$$

is in F^+ .

Intuition:

- Observe: the attributes $R_1 \cap R_2$ are in both R_1 and R_2
 \Rightarrow these are the join attributes.
- Suppose the FD $R_1 \cap R_2 \rightarrow R_1 - R_2$ holds.
 This is the same as $R_1 \cap R_2 \rightarrow R_1 - (R_1 \cap R_2)$
 $\Rightarrow R_1 \cap R_2$ is a *key* for R_1 .
 \Rightarrow weird, unwanted tuple combinations can't occur.

- In the $R=(\text{CAR},\text{OWNER},\text{COLOR})$ example:

We joined

	CAR COLOR		OWNER COLOR
	Toyota blue		Smith blue
	Ford blue		Jones blue

to get

CAR	OWNER	COLOR
Toyota	Smith	blue
Toyota	Jones	blue *
Ford	Smith	blue *
Ford	Jones	blue

Note that $\{\text{COLOR}\}$ is not a key for either relation.

Here, $R_1=(\text{CAR},\text{COLOR})$ and $R_2=(\text{OWNER},\text{COLOR})$ and,

$$\begin{aligned} R_1 \cap R_2 &= \text{COLOR} \\ R_1 - R_2 &= \text{CAR} \\ R_2 - R_1 &= \text{OWNER} \end{aligned}$$

Clearly, the neither of FD's

$$\begin{aligned} \{\text{COLOR}\} &\rightarrow \{\text{CAR}\} \\ \{\text{COLOR}\} &\rightarrow \{\text{OWNER}\} \end{aligned}$$

hold
⇒ the decomposition is *not* nonadditive.

Decomposition into Nonadditive BCNF Relations

- An algorithm for decomposing a relation into a collection of *nonadditive BCNF* relations.

Algorithm: Nonadditive BCNF decomposition

Input: Relation R , set of FD's F .

Output: A decomposition $D = \{R_1, R_2, \dots\}$

1. Set $D \leftarrow \{R\}$;
2. **while** \exists non-BCNF relations in D **do**
3. Pick a non-BCNF relation Q from D
4. Find an FD $X \rightarrow Y$ in Q that violates BCNF
5. $D \leftarrow D - \{Q\}$;
6. Create relation $R' \leftarrow Q - Y$;
7. Create relation $R'' \leftarrow X \cup Y$;
8. $D \leftarrow D \cup \{R', R''\}$;
9. **endwhile**
10. **return** D ;

Intuition:

If $X \rightarrow Y$ is in some Q and it violates BCNF

⇒ X is not a superkey of Q (definition of BCNF)

⇒ we create a relation $R'' = X \cup Y$

Here $X \rightarrow Y$ implies X is a superkey for R'' .

Also, we take out Y from Q (if not, it would violate BCNF).

- Unfortunately, we can't always decompose R into BCNF relations that are *both* dependency preserving and nonadditive.
- **Fact.** It is possible to decompose a relation into 3NF relations that are dependency preserving and nonadditive. (See textbook for method).

Joins That Leave Out Tuples

- Sometimes, one can inadvertently lose tuples in a join.
- Example:

Suppose EMP(SSN, ENAME, ADDR, DNO) has too many NULLs in the DNO attribute (because many employees have no assigned department).

⇒ One solution is to decompose EMP into two relations:

EMP1(SSN, ENAME, ADDR)
DEPT(SSN, DNO)

e.g.,

SSN	ENAME	ADDR	SSN	DNO
111111111	Smith	12 Main St.	111111111	5
222222222	Jones	55 Pleasant Lane	222222222	6
333333333	Brown	4C Maple Way		

Here,

- Only those employees assigned to a department will have department numbers
⇒ DEPT is small.
- Both EMP1 and DEPT are in BCNF.
- The decomposition is nonadditive and dependency preserving.

Consider the join EMP1 * DEPT:

SSN	ENAME	ADDR	DNO
111111111	Smith	12 Main St.	5
222222222	Jones	55 Pleasant Lane	6

⇒ the ‘Brown’ tuple is lost! ⇒ we have to be careful in letting joins replace relations.

Formal Schema Design: A Summary

- We saw that ad-hoc designs led to anomalies with insertions, deletions and modifications.
- To analyze relations, we developed the theory of normalization:
 - Definition of functional dependency (FD).
 - Properties of FD’s.
 - Computation of attribute closures.
 - Definition of Normal forms (primary and general versions).
- In practice: try to achieve BCNF. If not possible, live with 3NF.

If your design is not in 3NF
⇒ you have a weird schema.

- Also need to check for *nonadditivity* and *dependency preservation*.
- Before using a join to replace existing relations, check to see tuples don't get lost.
- Some issues we have not covered:
 - Finding minimal FD-sets (algorithm in book).
 - General mechanisms for testing nonadditivity. (Algorithm called Tableau Chase Method in book).
 - Algorithm for decomposition into 3NF relations.
 - Multivalued FD's and 4NF.
 - Join and other dependencies.