

Lecture 4c: Normalization ...

Schema Refinement and Normal Forms

- Normalization
 - Motivation
 - Anomalies, Redundancy
 - NULL values: not a good solution
 - Decomposition: a good solution
 - Keys and Functional Dependencies (FDs)
 - BCNF and Redundancy
 - Lossless Decompositions
 - Dependency Preserving Decompositions, Projections
 - Third Normal Form

What is Normalization?

- **Normalization** is the process of replacing a table with two or more tables.
- For example, consider this schema

EmpDept

<u>EID</u>	Name	DeptID	DeptName
A01	Ali	12	Wing
A12	Eric	10	Tail
A13	Eric	12	Wing
A03	Tyler	12	Wing

- Versus this one:

Emp

<u>EID</u>	Name	DeptID
A01	Ali	12
A12	Eric	10
A13	Eric	12
A03	Tyler	12

Dept

<u>DeptID</u>	DeptName
12	Wing
10	Tail

- Which schema do you think is best? Why?

What's wrong?

The EmpDept schema combines **two different ideas**:
employee information and department information.

But what is wrong with this?

1. If we separated the two concepts we could save **space** but some queries would run slower.
2. If we combine the two ideas we have **update anomalies** but some queries would run faster.

So we have a tradeoff!

The update anomalies are caused by the fact that for one **DeptID** there is only one **DeptName** that goes with it; but **DeptID** is not a key.

What anomalies do we have?

EmpDept

<u>EID</u>	Name	DeptID	DeptName
A01	Ali	12	Wing
A12	Eric	10	Tail
A13	Eric	12	Wing
A03	Tyler	12	Wing

- **Update Anomaly:** If the Wing department changes its name, we must change multiple rows in EmpDept
- **Insertion Anomaly:** If a department has no employees, where do we store its id and name?
- **Deletion Anomaly:** If A12 Eric quits, the information about the Tail department will be lost.

We also have redundancy: dept. name is stored multiple times. (Note: there are other types of redundancy.)

Example: Anomalies, Redundancies

- Identify anomalies associated with these schemas. Include update, insertion and deletion anomalies.

EnrollStud(StudID, ClassID, Grade, ProfID, StudName)

- Update Anomaly: If a student changes his name, we must change each row for which the student has taken a class. If a class changes the profID, we must change it for every row in which the class appears.
- Insertion Anomaly: If a student has not taken a class, where do we store her id and name? If a class has no student grades recorded yet, where do we store its ProfID?
- Deletion Anomaly: If a student drops her last course, the information about the student's name will be lost. If the last student drops the course, the info about the ProfID will be lost.

What is the redundancy?

Why do these anomalies occur?

Another example: redundancy/anomalies

Assigned(EmpID, JobID, EmpName, percent)

- Update Anomaly: If an employee changes his name, we must change each row for which the employee has a job.
- Insertion Anomaly: If an employee has no job, where do we store his name?
- Deletion Anomaly: If an employee quits his last job, the information about the employee's id and name will be lost.

What is the redundancy?

Why do these anomalies occur?

Null Values – Not a good solution

EmpDept

<u>EID</u>	Name	DeptID	DeptName
A01	Ali	12	Wing
NULL	NULL	10	Tail
A13	Eric	12	Wing
A03	Tyler	12	Wing

Insertion and deletion anomalies can be solved with **NULL values**.

But

- NULL values are **difficult to manage**
 - They make aggregate operators hard to compute
 - Not clear if NULL means does not apply, unknown, or known but absent
 - NULL value logic is complex
 - May need outer joins rather than ordinary joins
 - **EID is a primary key and you can't have null values in a key!**
- They do not eliminate all **update anomalies** or **wasted space**.

Decomposition: A good solution

- To normalize - EmpDept becomes

Emp

<u>EID</u>	Name	DeptID
A01	Ali	12
A12	Eric	10
A13	Eric	12
A03	Tyler	12

Dept

<u>DeptID</u>	DeptName
12	Wing
10	Tail

- Check to see if the update anomalies still occur.

The secret to understanding when and how to decompose schemas is **Functional Dependencies**, a generalization of **keys**.

Review Keys

EmpDept

<u>EID</u>	Name	DeptID	DeptName
A01	Ali	12	Wing
A12	Eric	10	Tail
A13	Eric	12	Wing
A03	Tyler	12	Wing

- Note that EID being a **key*** of EmpDept means that **if two rows have the same value of EID then they have the same value of every other attribute**, and EID is minimal.
- If the table has no duplicate rows, this is the same as: the key has **no duplicate values**.
- Remember: you cannot determine keys from an instance, only from “natural” information or from a domain expert.
- Let’s practice keys by identifying possible keys in an instance.
- *sometimes called a *candidate key*

Identify Possible Keys

- Identify some possible Keys based on this instance:

Time	Flight	Plane	Origin	Destination
9:57AM	157	abc	SEA	PDX
10:42AM	233	def	PDX	SEA
11:44AM	155	des	ORD	ATL
12:44PM	244	xdy	ATL	PDX
1:43PM	074	xyz	SEA	ATL
2:44PM	233	def	PDX	ATL
3:55PM	455	eff	MSP	SEA
5:44PM	120	ikk	MSP	PDX
7:55PM	233	abf	CHI	SEA

Possible keys are:

{Time}, {Plane, Dest}, {Origin, Dest}

Functional Dependencies

EmpDept

<u>EID</u>	Name	DeptID	DeptName
A01	Ali	12	Wing
A12	Eric	10	Tail
A13	Eric	12	Wing
A03	Tyler	12	Wing

- Remember what a key (EID) means: if two rows have the same EID then they have the same value of every other attribute.
- We say EID **functionally determines** all other attributes and write this **Functional Dependency (FD)**. All of these FDs are implied by the key (EID).

$EID \rightarrow Name, DeptID, DeptName$

or

$EID \rightarrow Name, EID \rightarrow DeptID, EID \rightarrow DeptName$

We look for FDs that are not implied by the key(s) for the table

EmpDept

<u>EID</u>	Name	DeptID	DeptName
A01	Ali	12	Wing
A12	Eric	10	Tail
A13	Eric	12	Wing
A03	Tyler	12	Wing

- Is **Name** \rightarrow **DeptID** true? Is this an FD?
 - No, because two rows can have the same Name but not the same DeptID. Look at rows 2 and 3.
- Is **DeptID** \rightarrow **DeptName** true? Is this an FD?

Functional Dependencies, ctd.

- **DEFINITION:** If A and B are sets of attributes in a relation, we say that A (functionally) determines B, or $A \rightarrow B$ is a Functional Dependency (FD), if whenever two rows agree on A, they agree on B. The value of a row on A functionally determines its value on B.
- An FD defines a function – in the mathematical sense.
- There are two special kinds of FDs:
 - Key FDs, $X \rightarrow A$ where X contains a key (also called a superkey)
 - Trivial FDs, such as $\text{Name} \rightarrow \text{Name}$, or $\{\text{Name}, \text{DeptID}\} \rightarrow \text{DeptID}$
(boring but important when we get to the formal part of this)

Update anomalies are caused by other FDs (not key or trivial).

Functional Dependencies

(based on the semantics of the application)

Likely **functional dependencies**:

ssn → *employee-name*

course-number → *course-title*

Unlikely **functional dependencies**

course-number ~~X~~→ *book*

course-number ~~X~~→ *car-color*

Will FDs be enforced?

Consider this table:

Emp(ssn, name, phone, dept, dept-name)



There is an FD from *dept* → *dept-name*

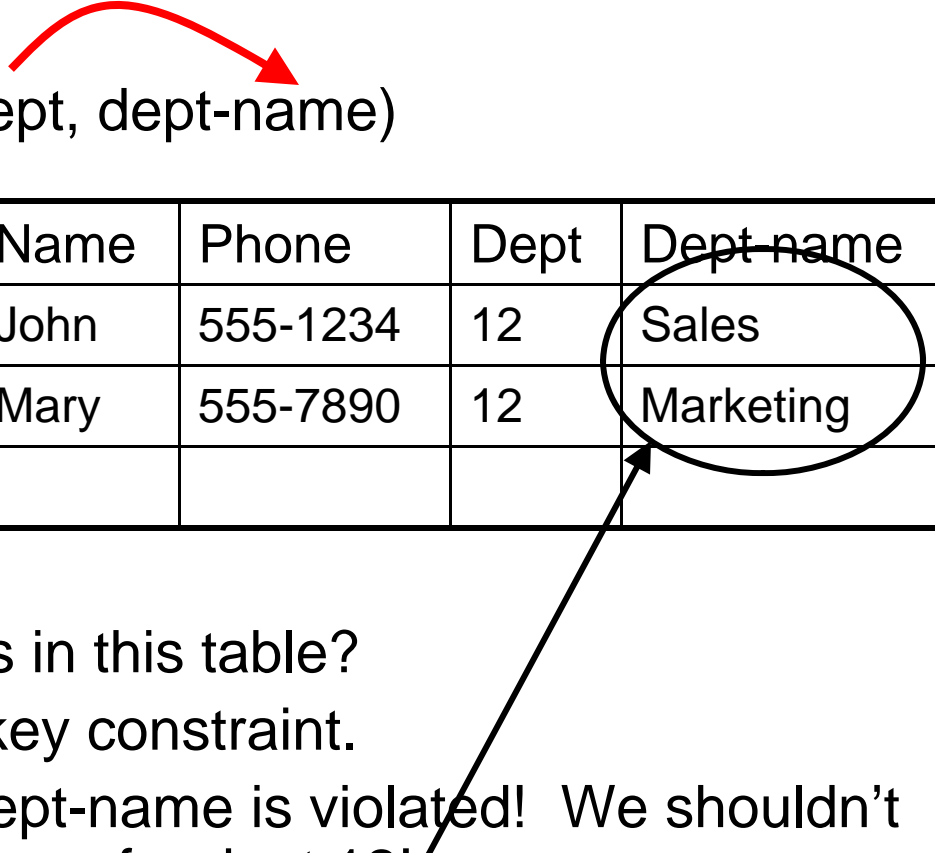
But *ssn* is the key for this table.

Is it possible for there to be two names for one dept?

Will this FD be enforced? Let's try it.

Consider this table:

Emp(ssn, name, phone, dept, dept-name)



Employee	<u>ssn</u>	Name	Phone	Dept	Dept-name
	111111111	John	555-1234	12	Sales
	222222222	Mary	555-7890	12	Marketing
	...				

Can we put these two rows in this table?

Yes, it doesn't violate the key constraint.

But, the FD from dept to dept-name is violated! We shouldn't
haven't two different names for dept 12!

We might say that *dept* "ought to be a key".

Every key implies a set of FDs

Each key implies a set of functional dependencies (FDs) from the key to the non-key attributes.

Employee (SSN, Name, Salary, Job-code)

A diagram illustrating the functional dependencies implied by the primary key SSN. Three red curved arrows originate from the underlined 'SSN' and point to 'Name', 'Salary', and 'Job-code' respectively.

FDs implied by the key:

SSN → Name

SSN → Salary

SSN → Job-code

But, there can be other FDs that are NOT implied by the key.

(table repeated from slide 20 and 21):

Emp(ssn, name, phone, dept, dept-name)



There is an FD from *dept* → *dept-name*

Keys and FDs (notation)

Given a table R, with a and b (together) as a key for the table, the following FDs are implied by the key.

R (a, b, c, d, e) then

$ab \rightarrow c$

$ab \rightarrow d$

$ab \rightarrow e$

Note we also write this as: $ab \rightarrow cde$

Functional Dependencies can Suggest Keys

If we know these FDs:

SSN → *name*

SSN → *hire-date*

SSN → *phone*

then **SSN** should be the key for a table with these attributes:

Employee (SSN, name, hire-date, phone)

Identify key FDs; identify other non-trivial FDs

Customer(CustID, Address, City, Zip, State)

EnrollStud(StudID, ClassID, Grade, ProfID, StudName, ProfName)

Non-trivial FDs (cont.)

Customer(CustID, Address, City, Zip, State)



Will this FD be enforced? (Is it possible for one zip to have more than one state?)

EnrollStud(StudID, ClassID, Grade, ProfID, StudName, ProfName)



Will StudID → StudName be enforced?

Will ProfID → ProfName be enforced?

How to determine FDs

- Sometimes it's not clear what FDs apply in a relation, e.g., zip codes vs cities, or ...

Supplier(Name, Address, Crating, Discount) – unclear what are the FDs.

- As with keys, you **cannot determine FDs from an instance!**
- You need to talk to a domain expert.
- But you can look at the instance and determine that something is NOT an FD or NOT a key.

Sometimes redundancy has NOTHING to do with FDs

Suppose we have two tables for employee information.

Employee(ssn, name, salary, birthdate)

Employee2(ssn, name, home-address)

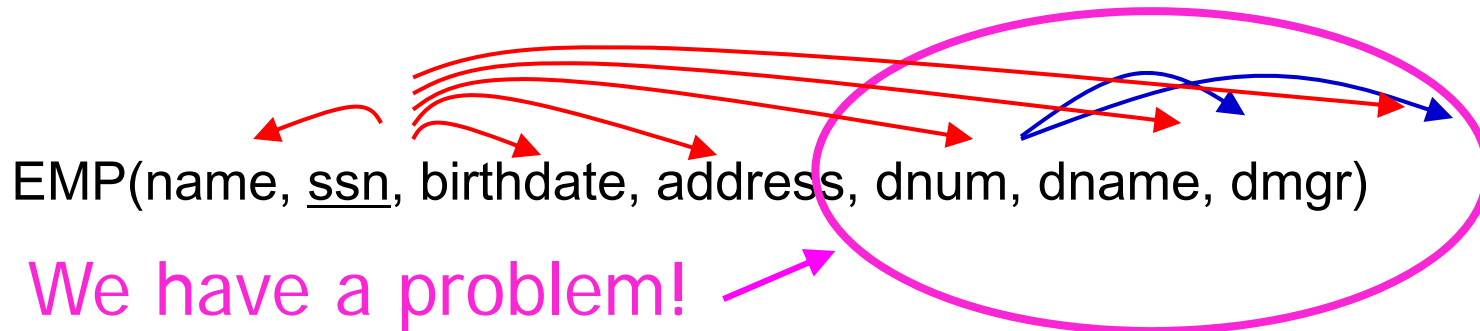
And, we put the name in both tables, for convenience.

Name is stored redundantly.

And, it is possible for the two names to be inconsistent, if you change it in one place but not in another.

This redundancy is NOT caused by an FD that is not key/trivial.
ssn → name is implied by the key in both tables.

Example: Troublesome FDs



We have a problem!
dnum is NOT the key for this table!

So these blue FDs will not be enforced automatically by the DBMS (using only keys).

We say, informally, that these blue FDs are "troublesome".

Example decomposition based on FDs

“Lifting” Troublesome FDs

EMP(name, ssn, birthdate, address, dnum, dname, dmgr)

1. Lift the “troublesome” FDs into their own table with dnum as the key. Now they will be enforced.

DEPARTMENT(dnum, dname, dmgr)

2. Leave the LHS of the “troublesome” FDs behind. Define a foreign key where Employee.dnum REFERENCES Department.dnum

EMPLOYEE(name, ssn, birthdate, address, dnum)

Basic Idea: Normalize based on FDs

- Identify all the (non-trivial) FDs in an application.
 - Identify FDs that are implied by the keys.
 - Identify FDs that are NOT implied by the keys – the “troublesome” ones.
- Decompose a table with a “troublesome” FD into two or more tables by “lifting” the troublesome FDs into a table of their own. Note: when there are two or more “troublesome” FDs with the same LHS, then they can be lifted, together, into a single table.

Decomposition Example

Assigned-to (A-project, A-emp, emp-name, percent)

Employee (A-emp, emp-name)

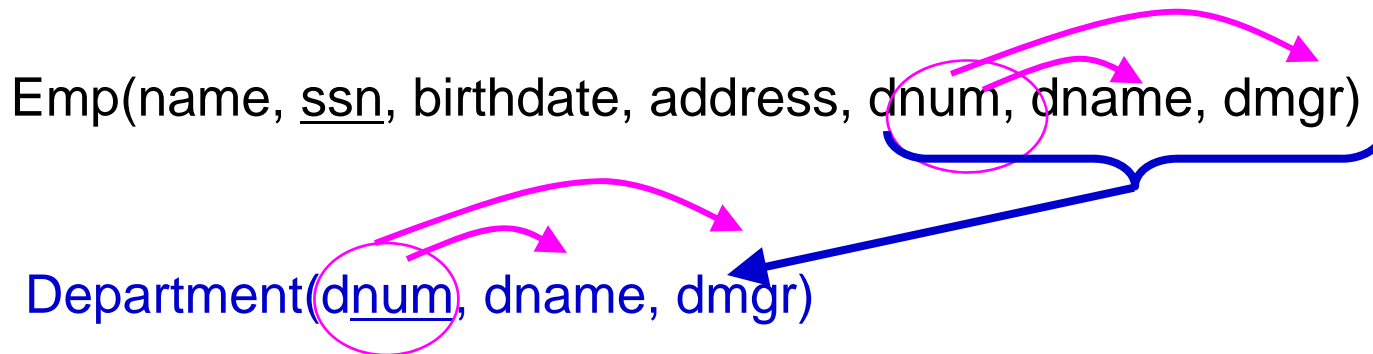
1. Lift the troublesome FDs into a table of their own.
Key for new table is left hand side of the troublesome FD.

2. Leave the left side of the FD behind in the original table.

Assigned-to (A-project, A-emp, percent)

3. Eliminate *emp-name* from the *Assigned-to* table.

Decomposition Example



1. Lift the troublesome FD(s) (with the same LHS) into a table of their own. Key for new table is left hand side of the troublesome FD.
2. Leave the left side of the FD behind in the original table.

NewEmp (name, ssn, birthdate, address, dnum)

3. Eliminate *dname* and *dmgr* from the *NewEmp* table.

Let's Check the Update Anomalies

Insertion anomalies:

if you insert an employee with a department then you need to know the descriptive information for that department. **NO – ONLY THE NUMBER**
if you want to insert a department, you can't ... until there is at least one employee. **NO PROBLEM**

Deletion anomalies: if you delete an employee, is that dept. gone? Was this the last employee in that dept.? **NO PROBLEM**

Modification anomalies: If you want to change *dname*, for example, you need to change it everywhere! And you have to find them all first. **dname is only stored once!**

Is there any redundancy? Yes – in the foreign key. But there is less redundancy.

Normalization

- Practical side:
 - Look for non-trivial FDs that are NOT implied by the key
 - Decompose – by “lifting” those FDs out into their own table; leave the LHS of those FDs in the original table
 - Eliminates update anomalies
 - Is correct (we don’t lose any information)
 - All “lifted” FDs are enforced by the DBMS
- Formal side:
 - FDs are functions
 - Armstrong’s axioms are sound and complete rules of inference
 - Normal forms and dependency preservation

Reasoning about FDs

EmpDept(EID, Name, DeptID, DeptName)

- Two natural FDs are

EID→**DeptID** and **DeptID**→**DeptName**

- These two FDs imply the FD **EID**→**DeptName**
 - Because if two tuples agree on EID, then by the first FD they agree on DeptID, then by the second FD they agree on DeptName.
- The set of FDs implied by a given set F of FDs is called **the closure of F** and is denoted **F⁺**

Armstrong's Axioms

- The closure of F can be computed using these axioms
 - Reflexivity: If $X \supseteq Y$, then $X \rightarrow Y$
 - Augmentation: If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z
 - Transitivity: If $X \rightarrow Y$ and $Y \rightarrow Z$ then $X \rightarrow Z$
- Armstrong's axioms are **sound** (they generate only FDs in F^+ when applied to FDs in F) and **complete** (repeated application of these axioms will generate all FDs in F^+).

Determining Keys

- In order to determine if X is a key of a relation R based on a set of FDs, use this algorithm, which computes the *attribute closure of X* :

AttClos = X ; // Note: X is a set of attributes

Repeat until there is no change {

 If there is an FD $U \rightarrow V$ with $U \subseteq \text{AttClos}$, then set

$\text{AttClos} = \text{AttClos} \cup V$

}

AttClos = R if and only if **X is a key**

Determining the keys of R (example)

- Given the schema: $R(\underline{A}, B, C, D, E)$ $BC \rightarrow A$, $DE \rightarrow C$.
- What are the keys of this schema, besides A?
- Let's start with $BC \rightarrow A$ as one example.
 - BC determines A is given
 - $A \rightarrow ABCDE$ because A is a key
 - $BC \rightarrow ABCDE$ by transitivity. Therefore, BC is a key.
- I won't be asking you to prove that you have found all keys using Armstrong's axioms but you should understand the algorithm on the preceding page. You can use it if you ever need it, in a real situation.

Redundancy and FDs

- Recall the FDs we have found in previous examples:

`EmpDept`(EID, Name, DeptID, DeptName)

`Assigned`(EmpID, JobID, EmpName, percent)

`EnrollStud`(StudID, ClassID, Grade, ProfID, StudName)

- Note that every non-key FD is associated with some redundancy and vice-versa.
- Our game plan is to use non-key, non-trivial FDs to decompose any relation into a form that has no redundancy, a so-called **normal form**.

Boyce-Codd Normal Form (BCNF)

- A relation is said to be in **Boyce-Codd Normal Form** if all its FDs are either trivial FDs or key FDs.

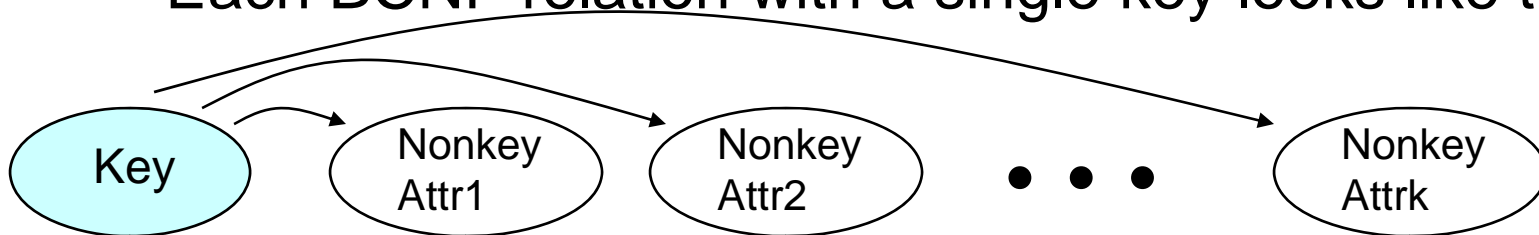
- Which of these relations is BCNF?

`EmpDept`(EID, Name, DeptName)

`Assigned`(EmpID, JobID, EmpName, percent)

`EnrollStud`(StudID, ClassID)

- Each BCNF relation with a single key looks like this



BCNF and Redundancy

- Theorem: **BCNF relations have no redundancy caused by FDs.**

Proof: A relation has redundancy if there is an FD between two sets of attributes, say **DeptID**→**DeptName**, and there can be repeated entries of data for those attributes.

For example, consider (12,Wing) in this example:

DeptID	DeptName
12	Wing
10	Tail
12	Wing

But if the relation is BCNF, then the FD must be a key FD, and DeptID must be a key. Thus any pair such as (12,Wing) can appear only once.

Decomposition into BCNF (“lifting”)

- Here is an algorithm for decomposing an arbitrary relation into a collection of BCNF relations:
 1. If R is not in BCNF and $X \rightarrow A$ is a non-key FD, then decompose R into $R - A$ and XA .
 2. If $R - A$ and/or XA is not in BCNF, recursively apply step 1.

Decomposing to BCNF

Begin with: `EnrollStud(StudID, ClassID, Grade, ProfID, StudName)`

Use the non-key FD **StudID→StudName**.

This results in the decomposition

`EnrollProf(StudID, ClassID, Grade, ProfID)`

`Stud(StudID, StudName)`

Stud is BCNF, but **EnrollProf is not**.

The non-key FD **ClassID→ProfID** gives the further decomposition

`Enroll(StudID, ClassID, Grade)`

`Prof(ClassID, ProfID)`

`Stud(StudID, StudName)`

in which all schemas are BCNF

LO8.5: Decompose to BCNF

- Given the schema
LoansBC(Branch#, Loan#, Amt, Assets, Cust#, CustName)
including the FDs **Branch**→**Assets**, **Cust#**→**CustName**,
decompose it into BCNF relations.

Where are we?

- We've accomplished a lot!
 - We began with a relational schema
 - We identified (redundancy) problems with it
 - We learned how to use FDs to eliminate those problems with decompositions into BCNF.
 - Along the way, we learned a powerful tool: how to determine keys from FDs.
- There are two steps left
 - Showing that the BCNF decompositions do not lose information.
 - Discovering that they may lose FDs, and how to deal with that.

Lossless Decompositions

- Some decompositions **lose information**. Suppose we got carried away and further decomposed

Enroll(StudID, ClassID, Grade) into

StudGrade(StudID, Grade) and **ClassGrade**(ClassID, Grade)

- Here a row (123,B) in **StudGrade** means that student 123 got a B in some course, and (386,A) in **ClassGrade** means that some student got an A in course 386.
- But now we have no way of knowing which student got which grade in which class.
- This decomposition is **lossy**. It contains less information than the original schema. We want to generate only **lossless** decompositions when we design our databases.

Lossless Decompositions

Definition: A decomposition of a schema R with FDs F , into attribute sets X and Y , is **lossless** with respect to F if for every instance r of R that satisfies F

$$r = \pi_X(r) \bowtie \pi_Y(r)$$

In other words, we can recover r from the natural join of the decomposed versions of r .

Example of a Lossless Decomposition

R=EmpDept

<u>EID</u>	Name	DeptID	DeptName
A01	Ali	12	Wing
A12	Eric	10	Tail
A13	Eric	12	Wing
A03	Tyler	12	Wing

= r

X=EID,Name,DeptID

<u>EID</u>	Name	DeptID
A01	Ali	12
A12	Eric	10
A13	Eric	12
A03	Tyler	12

= $\pi_X(r)$

Y=DeptID,DeptName

<u>DeptID</u>	DeptName
12	Wing
10	Tail

= $\pi_Y(r)$

r =

<u>EID</u>	Name	DeptID	DeptName
A01	Ali	12	Wing
A12	Eric	10	Tail
A13	Eric	12	Wing
A03	Tyler	12	Wing

= $\pi_X(r) \bowtie \pi_Y(r)$

Example of a Lossy Decomposition

R = Enroll

StudID	ClassID	Grade
123	CS386	A
456	CS410	A

 = r

X = StudID, Grade

StudID	Grade
123	A
456	A

 = $\pi_X(r)$

Y = ClassID, Grade

ClassID	Grade
CS386	A
CS410	A

 = $\pi_Y(r)$

$r \neq$

StudID	ClassID	Grade
123	CS386	A
123	CS410	A
456	CS410	A
456	CS386	A

 = $\pi_X(r) \bowtie \pi_Y(r)$

Note that the join has extra rows. This always happens in lossy decompositions

Producing only Lossless Decompositions

- In our design of database schemas we certainly want to produce only lossless decompositions. Fortunately this is easy to guarantee.

Theorem: The decomposition of R with respect to FDs F into attribute sets R_1 and R_2 is lossless if and only if $R_1 \cap R_2$ contains a key for either R_1 or R_2 .

Proof: Page 620 in the text.

Corollary: **The BCNF decomposition algorithm produces only lossless decompositions.**

Proof: In this case F includes the FD $\mathbf{X} \rightarrow \mathbf{A}$ and the decomposition is into $R_1 = R - A$ and $R_2 = XA$. Then $R_1 \cap R_2 = X$ is a key for XA .

L08.6: Proving lossless/lossy

- Given the schema $R(S C G)$ with FDs

$$SC \rightarrow G$$

Is the decomposition into SG and CG lossless or lossy and why?

Take the intersection of the two sets of attributes for the two tables. $SG \cap CG = G$

Then figure out whether G is a key for either table.

Does $G \rightarrow C$? No, we have no way to infer this FD.

Thus, G is not a key for the second table.

Does $G \rightarrow S$? No, we have no way to infer this FD.

Therefore, this decomposition is lossy!

Where are we?

- We have learned how to transform an **ER Diagram** into a **Relational Schema** and to transform that **losslessly** into a **BCNF schema**.
- We recall from a previous picture that BCNF tables are particularly simple, so this looks like a perfect solution to a very general problem.
- But real schemas are not always BCNF. There are two more complexities to deal with.

Dependency Preserving Decompositions

- Decompositions should **preserve FDs**.
- FDs are business requirements that **must be enforced**.
- Consider an example:
- Emp(Addr, City, State, Zip) **ACS** → **Z**, **Z** → **S**
- Keys are ACS and ACZ. Consider the decomposition:
(Address, City, Zip) (Zip, State)
- This is BCNF but it does not preserve **ACS** → **Z**
- Consider the values
(7315 SW84, Portland, 97223), (97223, OR),
(7315 SW84, Portland, 00000), (00000, OR)

Definition of Projection, Dependency Preserving

- Let R be a schema with FDs F and X, Y sets of attributes in R .
- Definition: **The projection of F on X , F_X** , is the closure of the FDs in $F \cap X$, i.e., $(F \cap X)^+$. Note: one FD $A \rightarrow B$ is in X , for example, if all attributes of A and all attributes of B are in X .

Definition: The decomposition of R into schemas with attributes X and Y is **dependency preserving** if

$$(F_X \cup F_Y)^+ = F^+ .$$

* Note that **Armstrong's Axioms** are needed to compute F_X and F^+ .

Example: Projection, Dependency Preserving*

- Consider Emp(Addr, City, State, Zip) with FDs:

$$F: \mathbf{ACS} \rightarrow \mathbf{Z}, \mathbf{Z} \rightarrow \mathbf{S}$$

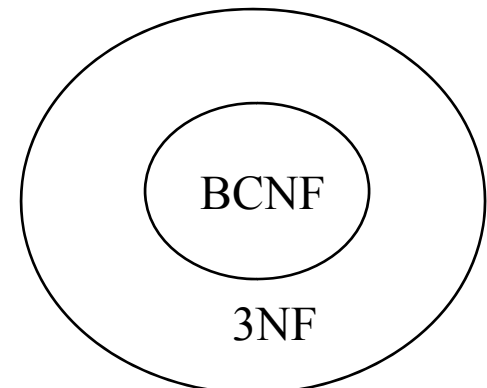
Consider its decomposition into $X=ACZ$, $Y=ZS$. What are the **projections** F_X , F_Y ?

- F_X : **empty**
- F_Y : $\mathbf{Z} \rightarrow \mathbf{S}$
- Is X, Y a **dependency preserving** decomposition?
 - No, because $(\mathbf{Z} \rightarrow \mathbf{S})^+$ does not contain $\mathbf{ACS} \rightarrow \mathbf{Z}$, so it can never equal F^+

Third Normal Form

- Some schemas do not have a lossless, dependency preserving, decomposition into BCNF schemas.
- Because of this dilemma, researchers created another normal form called Third Normal Form (3NF), with the property that every schema has a lossless dependency preserving decomposition into 3NF schemas.
- A schema R with FDs F is in **Third Normal Form** if for every $X \rightarrow A$ in F, one of these is true:
 - $X \rightarrow A$ is a trivial FD (i.e., X contains A)
 - $X \rightarrow A$ is a key FD (i.e. X contains a key*)
 - A is a part of some key for R

Definition
of BCNF!



*The text uses the terminology “X is a superkey”

Conclusion

- Almost all schemas in real life can be decomposed into BCNF schemas that preserve all FDs. In this case, life is wonderful.
- But every once in a while we get a schema like Emp(Addr, City, State, Zip) **ACS** → **Z**, **Z** → **S**
- Recall that its keys are ACS and ACZ. There is no decomposition into BCNF that preserves FDs!
- On the other hand, this schema is 3NF. Check it!

Conclusion, ctd.

- So in the rare case that we don't have an ideal decomposition (lossless, dependency preserving, into BCNF), rest assured that we can decompose into 3NF instead of BCNF and have lossless and dependency preservation.
- The proof of this assertion is in section 19.6.2 .
- However, there is one more hitch. You'll learn it in a few minutes.