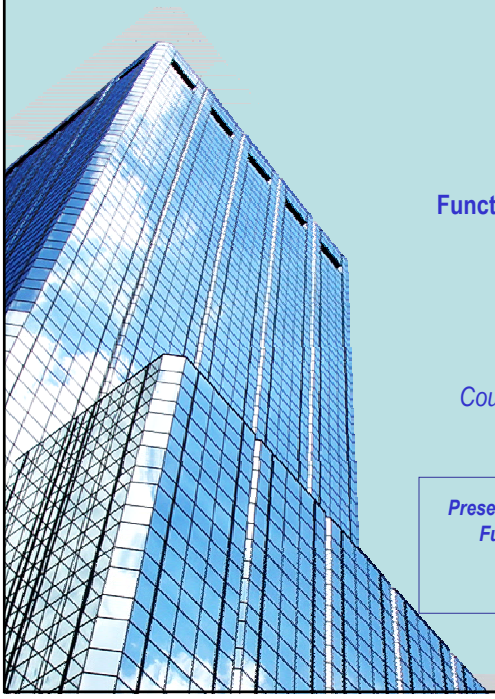


Database Systems

Session 7 – Main Theme

Functional Dependencies and Normalization

Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Agenda

Session Agenda

- Logical Database Design - Normalization
- Normalization Process Detailed
- Summary & Conclusion

What is the class about?

- Course description and syllabus:
  - » http://www.nyu.edu/classes/jcf/CSCI-GA.2433-001
  - » http://cs.nyu.edu/courses/fall11/CSCI-GA.2433-001/
- Textbooks:
  - » ***Fundamentals of Database Systems (6th Edition)***

Ramez Elmasri and Shamkant Navathe
Addition Wesley
ISBN-10: 0-1360-8620-9, ISBN-13: 978-0136086208 6th Edition (04/10)

Icons / Metaphors


Information


Common Realization


Knowledge/Competency Pattern


Governance


Alignment


Solution Approach

Agenda

1. Session Overview
2. Logical Database Design - Normalization
3. Normalization Process Detailed
4. Summary and Conclusion

Agenda

- Informal guidelines for good design
- Functional dependency
  - Basic tool for analyzing relational schemas
- Informal Design Guidelines for Relation Schemas
- Normalization:
  - 1NF, 2NF, 3NF, BCNF, 4NF, 5NF
    - Normal Forms Based on Primary Keys
    - General Definitions of Second and Third Normal Forms
    - Boyce-Codd Normal Form
    - Multivalued Dependency and Fourth Normal Form
    - Join Dependencies and Fifth Normal Form

Logical Database Design

- We are given a set of tables specifying the database
  - » The base tables, which probably are the community (conceptual) level
- They may have come from some ER diagram or from somewhere else
- We will need to examine whether the specific choice of tables is good for
  - » For storing the information needed
  - » Enforcing constraints
  - » Avoiding anomalies, such as redundancies
- If there are issues to address, we may want to restructure the database, of course not losing any information
- Let us quickly review an example from “long time ago”

A Fragment Of A Sample Relational Database

R	Name	SSN	DOB	Grade	Salary
	A	121	2367	2	80
	A	132	3678	3	70
	B	101	3498	4	70
	C	106	2987	2	80

- Business rule (one among several):
  - The value of Salary is determined only by the value of Grade
- Comment:
  - We keep track of the various Grades for more than just computing salaries, though we do not show it
  - For instance, DOB and Grade together determine the number of vacation days, which may therefore be different for SSN 121 and 106

Anomalies

Name	SSN	DOB	Grade	Salary
A	121	2367	2	80
A	132	3678	3	70
B	101	3498	4	70
C	106	2987	2	80

- "Grade = 2 implies Salary = 80" is written twice
- There are additional problems with this design.
  - We are unable to store the salary structure for a Grade that does not currently exist for any employee.
  - For example, we cannot store that Grade = 1 implies Salary = 90
  - For example, if employee with SSN = 132 leaves, we forget which Salary should be paid to employee with Grade = 3
  - We could perhaps invent a fake employee with such a Grade and such a Salary, but this brings up additional problems, e.g.,

    What is the SSN of such a fake employee? It cannot be NULL as SSN is the primary key

Better Representation Of Information

- The problem can be solved by replacing

R	Name	SSN	DOB	Grade	Salary
	A	121	2367	2	80
	A	132	3678	3	70
	B	101	3498	4	70
	C	106	2987	2	80

- by two tables

S	Name	SSN	DOB	Grade
	A	121	2367	2
	A	132	3678	3
	B	101	3498	4
	C	106	2987	2

T	Grade	Salary
	2	80
	3	70
	4	70

Decomposition

- SELECT INTO S
  Name, SSN, DOB, Grade
  FROM R;

- SELECT INTO T
  Grade, Salary
  FROM R;

Better Representation Of Information

- And now we can
  - » Store “Grade = 3 implies Salary = 70”, even after the last employee with this Grade leaves
  - » Store “Grade = 2 implies Salary = 90”, planning for hiring employees with Grade = 1, while we do not yet have any employees with this Grade

S	Name	SSN	DOB	Grade
	A	121	2367	2
	B	101	3498	4
	C	106	2987	2

T	Grade	Salary
	1	90
	2	80
	3	70
	4	70

No Information Was Lost

- Given S and T, we can reconstruct R using ***natural join***

S	Name	SSN	DOB	Grade
	A	121	2367	2
	A	132	3678	3
	B	101	3498	4
	C	106	2987	2

T	Grade	Salary
	2	80
	3	70
	4	70

```
SELECT INTO R
Name, SSN, DOB, S.Grade AS Grade, Salary
FROM T, S
WHERE T.Grade = S.Grade;
```

R	Name	SSN	DOB	Grade	Salary
	A	121	2367	2	80
	A	132	3678	3	70
	B	101	3498	4	70
	C	106	2987	2	80

Natural Join (Briefly, More Later)

- Given several tables, say R1, R2, …, Rn, their ***natural join*** is computed using the following "template":

  SELECT INTO R
  one copy of each column name
  FROM R1, R2, …, Rn
  WHERE equal named columns have to be equal

- The intuition is that R was "decomposed" into R1, R2, …,Rn by appropriate SELECT statements, and now we are putting it back together

Comment On Decomposition

- It does not matter whether we remove duplicate rows
- But some systems insist that that a row cannot appear more than once with a specific value of a primary key
- So this would be OK for such a system

T	Grade	Salary
	2	80
	3	70
	4	70

- This would not be OK for such a system

T	Grade	Salary
	2	80
	3	70
	4	70
	2	80

Comment On Decomposition

- We can always make sure, in a system in which DISTINCT is allowed, that there are no duplicate rows by writing

  SELECT INTO T
  DISTINCT Grade, Salary
  FROM R;

- And similarly elsewhere

Natural Join And Lossless Join Decomposition

- ***Natural Join*** is:
  - » Cartesian join with condition of equality on corresponding columns
  - » Only one copy of each column is kept
- ***"Lossless join decomposition"*** is another term for information not being lost, that is we can reconstruct the original table by "combining" information from the two new tables by means of natural join
- This does not necessarily always hold
- We will have more material about this later
- Here we just observe that our decomposition satisfied this condition at least in our example

Elaboration On "Corresponding Columns" (Using Semantically "Equal" Columns)

- It is suggested by some that no two columns in the database should have the same name, to avoid confusion, then we should have columns and join similar to these

S	S_Name	S_SSN	S_DOB	S_Grade
	A	121	2367	2
	A	132	3678	3
	B	101	3498	4
	C	106	2987	2

T	T_Grade	T_Salary
	2	80
	3	70
	4	70

```
SELECT INTO R S_Name AS R_Name, S_SSN AS R_SSN, S_DOB AS R_DOB,
S_Grade AS R_Grade, T_Salary AS R_Salary
FROM T, S
WHERE T_Grade = S_Grade;
```

R	R_Name	R_SSN	R_DOB	R_Grade	R_Salary
	A	121	2367	2	80
	A	132	3678	3	70
	B	101	3498	4	70
	C	106	2987	2	80

Mathematical Notation For Natural Join (We Will Use Sparingly)

- There is a special mathematical symbol for natural join
- It is not part of SQL, of course, which only allows standard ANSI font
- In mathematical, relational algebra notation, natural join of two tables is denoted by a bow-like symbol (this symbol appears only in special mathematical fonts, so we may use ∞ in these notes instead)
- So we have: R = S ∞ T
- It is used when "corresponding columns" means "equal columns"

Revisiting The Problem

- Let us look at

R	Name	SSN	DOB	Grade	Salary
	A	121	2367	2	80
	A	132	3678	3	70
	B	101	3498	4	70
	C	106	2987	2	80
	A	132	3678	3	70
	B	101	3498	4	70

- The problem is ***not*** that there are duplicate rows
- The problem is the same as before, business rule assigning Salary to Grade is written a number of time
- So how can we "generalize" the problem?

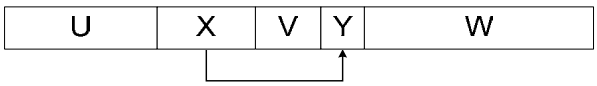
Stating The Problem In General

R	Name	SSN	DOB	Grade	Salary
	A	121	2367	2	80
	A	132	3678	3	70
	B	101	3498	4	70
	C	106	2987	2	80
	A	132	3678	3	70
	B	101	3498	4	70

- We have a problem whenever we have two sets of columns X and Y (here X is just Grade and Y is just Salary), such that
  1. **X does not contain a key either primary or unique** (thus there could be several/many **non-identical** rows with the same value of X)
  2. **Whenever two rows are equal on X, they must be equal on Y**
- Why a problem: the business rule specifying how X "forces" Y is "embedded" in different rows and therefore
  - » Inherently written redundantly
  - » Cannot be stored by itself

What Did We Do? Think X = Grade And Y = Salary

- We had a table



- We replaced this one table by two tables

Goodness of Relational Schemas

- Levels at which we can discuss *goodness* of relation schemas
  - Logical (or conceptual) level
  - Implementation (or physical storage) level
- Approaches to database design:
  - Bottom-up or top-down

Informal Design Guidelines for Relation Schemas

- Measures of quality
  - Making sure attribute semantics are clear
  - Reducing redundant information in tuples
  - Reducing NULL values in tuples
  - Disallowing possibility of generating spurious tuples

Imparting Clear Semantics to Attributes in Relations

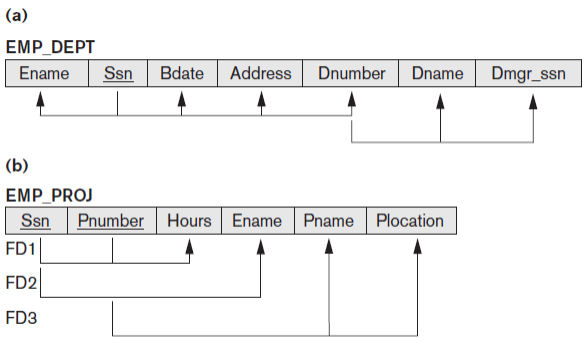
- Semantics of a relation
  - Meaning resulting from interpretation of attribute values in a tuple
- Easier to explain semantics of relation
  - Indicates better schema design

Guideline 1

- Design relation schema so that it is easy to explain its meaning
- Do not combine attributes from multiple entity types and relationship types into a single relation
- Example of violating Guideline 1: Figure 15.3

Guideline 1 (cont'd.)

Figure 15.3
Two relation schemas suffering from update anomalies. (a) EMP_DEPT and (b) EMP_PROJ.



Redundant Information in Tuples and Update Anomalies

- Grouping attributes into relation schemas
  - Significant effect on storage space
- Storing natural joins of base relations leads to **update anomalies**
- Types of update anomalies:
  - Insertion
  - Deletion
  - Modification

Guideline 2

- Design base relation schemas so that no update anomalies are present in the relations
- If any anomalies are present:
  - Note them clearly
  - Make sure that the programs that update the database will operate correctly

NULL Values in Tuples

- May group many attributes together into a "fat" relation
  - Can end up with many NULLs
- Problems with NULLs
  - Wasted storage space
  - Problems understanding meaning

Guideline 3

- Avoid placing attributes in a base relation whose values may frequently be NULL
- If NULLs are unavoidable:
  - Make sure that they apply in exceptional cases only, not to a majority of tuples

Generation of Spurious Tuples

- Figure 15.5(a)
  - Relation schemas EMP_LOCS and EMP_PROJ1
- NATURAL JOIN
  - Result produces many more tuples than the original set of tuples in EMP_PROJ
  - Called **spurious tuples**
  - Represent spurious information that is not valid

Guideline 4

- Design relation schemas to be joined with equality conditions on attributes that are appropriately related
  - Guarantees that no spurious tuples are generated
- Avoid relations that contain matching attributes that are not (foreign key, primary key) combinations

Summary and Discussion of Design Guidelines

- Anomalies cause redundant work to be done
- Waste of storage space due to NULLs
- Difficulty of performing operations and joins due to NULL values
- Generation of invalid and spurious data during joins

Logical Database Design

- We will discuss techniques for dealing with the above issues
- Formally, we will study ***normalization*** (decompositions as in the above example) and ***normal forms*** (forms for relation specifying some "niceness" conditions)
- There will be three very important issues of interest:
  - Removal of redundancies
  - Lossless-join decompositions
  - Preservation of dependencies
- We will learn the material mostly through comprehensive examples
- But everything will be precisely defined
- Algorithms will be fully and precisely given in the material
- Some of this will be part of the Advanced part of this Unit

Several Passes On The Material

- Practitioners do it (mostly) differently than the way researchers/academics like to do
- I will focus on the way IT practitioners do it
- In the advanced part, I will describe what researchers/academics and some computer scientists like to do

The Topic Is Normalization And Normal Forms

- Normalization deals with “reorganizing” a relational database by, generally, breaking up tables (relations) to remove various anomalies
- We start with the way practitioners think about it (as we have just said)
- We will proceed by means of a simple example, which is rich enough to understand what the problems are and how to fix them
- It is important (in this context) to understand what the various normal forms are (they may ask you this during a job interview!)

Normal Forms

- A normal form applies to a table/relation, not to the database
- So the question is individually asked about a table: is it of some specific desireable normal form?
- The ones you need to know about in increasing order of "quality" and complexity:
  - First Normal Form (**1NF**); it essentially states that we have a table/relation
  - Second Normal Form (**2NF**); intermediate form in some algorithms
  - Third Normal Form (**3NF**); very important; a final form
  - Boyce-Codd Normal Form (**BCNF**); very important; a final form
  - Fourth Normal Form (**4NF**); a final form but generally what is good about it beyond previous normal forms is easily obtained
- There are additional ones, which are more esoteric, and which we will not cover

Definitions of Keys and Attributes Participating in Keys

- Definition of **superkey** and **key**
- **Candidate key**
  - If more than one key in a relation schema
    - One is **primary key**
    - Others are **secondary keys**

Definition. An attribute of relation schema R is called a **prime attribute** of R if it is a member of *some candidate key* of R. An attribute is called **nonprime** if it is not a prime attribute—that is, if it is not a member of any candidate key.

First Normal Form

- Part of the formal definition of a relation in the basic (flat) relational model
- Only attribute values permitted are single **atomic (or indivisible) values**
- Techniques to achieve first normal form
  - Remove attribute and place in separate relation
  - Expand the key
  - Use several atomic attributes

First Normal Form (cont'd.)

- Does not allow **nested relations**
  - Each tuple can have a relation within it
- To change to 1NF:
  - Remove nested relation attributes into a new relation
  - Propagate the primary key into it
  - **Unnest** relation into a set of 1NF relations

Sample Normalization into First Normal Form

(a)

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations

(b)

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

(c)

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocation
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

Figure 15.9
Normalization into 1NF. (a) A relation schema that is not in 1NF. (b) Sample state of relation DEPARTMENT. (c) 1NF version of the same relation with redundancy.

Our Example

- We will deal with a very small fragment of a database dealing with a university
- We will make some assumptions in order to focus on the points that we need to learn
- We will identify people completely by their first names, which will be like Social Security Numbers
  - » That is, whenever we see a particular first name more than once, such as Fang or Allan, this will always refer to the same person: there is only one Fang in the university, etc.

Our Example

- We are looking at a single table in our database
- It has the following columns
  - S, which is a Student
  - B, which is the Birth Year of the Student
  - C, which is a Course that the student took
  - T, which is the Teacher who taught the Course the Student took
  - F, which is the Fee that the Student paid the Teacher for taking the course
- We will start with something that is not even a relation (Note this is similar to Employees having Children in Unit 2; a Student may have any number of (Course,Teacher,Fee) values

	S	B	C	T	F	C	T	F
	Fang	1990	DB	Zvi	1	OS	Allan	2
	John	1980	OS	Allan	2	PL	Marsha	4
	Mary	1990	PL	Vijay	1			

Alternative Depiction

- Instead of

	S	B	C	T	F	C	T	F
	Fang	1990	DB	Zvi	1	OS	Allan	2
	John	1980	OS	Allan	2	PL	Marsha	4
	Mary	1990	PL	Vijay	1			

you may see the above written as

	S	B	C	T	F
	Fang	1990	DB	Zvi	1
			OS	Allan	2
	John	1980	OS	Allan	2
			PL	Marsha	4
	Mary	1990	PL	Vijay	1

First Normal Form: A Table With Fixed Number Of Column

- This ***was not*** a relation, because we are told that each Student may have taken any number of Courses
- Therefore, the number of columns is not fixed/bounded
- It is easy to make this a relation, getting

R	S	B	C	T	F
	Fang	1990	DB	Zvi	1
	John	1980	OS	Allan	2
	Mary	1990	PL	Vijay	1
	Fang	1990	OS	Allan	2
	John	1980	PL	Marsha	4

- Formally, we have a relation in ***First Normal Form (1NF)***, this means that there are no repeating groups and the number of columns is fixed
  - » There are some variations to this definition, but we will use this one

Our Business Rules (Constraints)

- Our enterprise has certain ***business rules***
- We are told the following business rules
  1. A student can have only one birth year
  2. A teacher has to charge the same fee from every student he/she teaches.
  3. A teacher can teach only one course (perhaps at different times, different offerings, etc, but never another course)
  4. A student can take any specific course from one teacher only (or not at all)
- This means, that we are ***guaranteed*** that the information will always obey these business rules, as in the example

R	S	B	C	T	F
	Fang	1990	DB	Zvi	1
	John	1980	OS	Allan	2
	Mary	1990	PL	Vijay	1
	Fang	1990	OS	Allan	2
	John	1980	PL	Marsha	4

Functional Dependencies

- Formal tool for analysis of relational schemas
- Enables us to detect and describe some of the above-mentioned problems in precise terms
- Theory of functional dependency

Definition of Functional Dependency

- Constraint between two sets of attributes from the database

Definition. A **functional dependency**, denoted by $X \rightarrow Y$, between two sets of attributes X and Y that are subsets of R specifies a *constraint* on the possible tuples that can form a relation state r of R. The constraint is that, for any two tuples t_1 and t_2 in r that have $t_1[X] = t_2[X]$, they must also have $t_1[Y] = t_2[Y]$.

- Property of semantics or meaning of the attributes
- **Legal relation states**
  - Satisfy the functional dependency constraints

Definition of Functional Dependency (cont'd.)

- Given a populated relation
  - Cannot determine which FDs hold and which do not
  - Unless meaning of and relationships among attributes known
  - Can state that FD does not hold if there are tuples that show violation of such an FD

Normal Forms Based on Primary Keys

- Normalization process
- Approaches for relational schema design
  - Perform a conceptual schema design using a conceptual model then map conceptual design into a set of relations
  - Design relations based on external knowledge derived from existing implementation of files or forms or reports

Normalization of Relations

- Takes a relation schema through a series of tests
  - Certify whether it satisfies a certain normal form
  - Proceeds in a top-down fashion
- **Normal form tests**

Definition. The **normal form** of a relation refers to the highest normal form condition that it meets, and hence indicates the degree to which it has been normalized.

Normalization of Relations (cont'd.)

- Properties that the relational schemas should have:
  - **Nonadditive join property**
    - Extremely critical
  - **Dependency preservation property**
    - Desirable but sometimes sacrificed for other factors

Practical Use of Normal Forms

- Normalization carried out in practice
  - Resulting designs are of high quality and meet the desirable properties stated previously
  - Pays particular attention to normalization only up to 3NF, BCNF, or at most 4NF
- Do not need to normalize to the highest possible normal form

Definition. Denormalization is the process of storing the join of higher normal form relations as a base relation, which is in a lower normal form.

Functional Dependencies (Abbreviation: FDs)

- These rules can be formally described using ***functional dependencies***
- We will ignore NULLS
- Let P and Q be sets of columns, then:

  P ***functionally determines*** Q, written P → Q

  if and only if

  any two rows that are equal on (all the attributes in) P must be equal on (all the attributes in) Q
- In simpler terms, less formally, but really the same, it means that:

  If a value of P is specified, it "forces" some (specific) value of Q; in other words: Q is a function of P
- In our old example we looked at Grade → Salary

Our Given Functional Dependencies

R	S	B	C	T	F
	Fang	1990	DB	Zvi	1
	John	1980	OS	Allan	2
	Mary	1990	PL	Vijay	1
	Fang	1990	OS	Allan	2
	John	1980	PL	Marsha	4

- Our rules
  1. A student can have only one birth year: S → B
  2. A teacher has to charge the same fee from every student he/she teaches : T → F
  3. A teacher can teach only one course (perhaps at different times, different offerings, etc, but never another course) : T → C
  4. A student can take a course from one teacher only: SC → T

Possible Primary Key

- Our rules: S → B, T → F, T → C, SC → T
- ST possible primary key, because given ST
  1. S determines B
  2. T determines F
  3. T determines C
- A part of ST is not sufficient
  1. From S, we cannot get T, C, or F
  2. From T, we cannot get S or B

R	S	B	C	T	F
	Fang	1990	DB	Zvi	1
	John	1980	OS	Allan	2
	Mary	1990	PL	Vijay	1
	Fang	1990	OS	Allan	2
	John	1980	PL	Marsha	4

Possible Primary Key

- Our rules: S → B, T → F, T → C, SC → T
- SC possible primary key, because given SC
  1. S determines B
  2. SC determines T
  3. T determines F (we can now use T to determine F because of 2)
- A part of SC is not sufficient
  1. From S, we cannot get T, C, or F
  2. From C, we cannot get S, B, T, or F

R	S	B	C	T	F
	Fang	1990	DB	Zvi	1
	John	1980	OS	Allan	2
	Mary	1990	PL	Vijay	1
	Fang	1990	OS	Allan	2
	John	1980	PL	Marsha	4

Possible Primary Keys

- Our rules: S → B, T → F, T → C, SC → T
- Because ST can serve as primary key, in effect:
  - » ST → SBCTF
  - » This sometimes just written as ST → BCF, since always ST → ST (columns determine themselves)
- Because SC can serve as primary key, in effect:
  - » SC → SBCTF
  - » This sometimes just written as SC → BTF, since always SC → SC (columns determine themselves)

We Choose The Primary Key

- We choose SC as ***the primary key***
- This choice is arbitrary, but perhaps it is more intuitively justifiable than ST
- For the time being, we ignore the other key (ST)

R	S	B	C	T	F
	Fang	1990	DB	Zvi	1
	John	1980	OS	Allan	2
	Mary	1990	PL	Vijay	1
	Fang	1990	OS	Allan	2
	John	1980	PL	Marsha	4

Repeating Rows Are Not A Problem

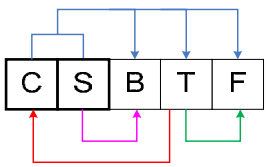
R	S	B	C	T	F
	Fang	1990	DB	Zvi	1
	John	1980	OS	Allan	2
	Mary	1990	PL	Vijay	1
	Fang	1990	OS	Allan	2
	John	1980	PL	Marsha	4

R	S	B	C	T	F
	Fang	1990	DB	Zvi	1
	John	1980	OS	Allan	2
	Mary	1990	PL	Vijay	1
	Fang	1990	OS	Allan	2
	John	1980	PL	Marsha	4
	Mary	1990	PL	Vijay	1

Review

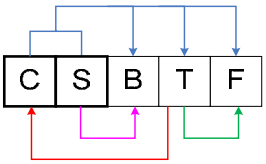
- To just review this
- Because S → B, given a specific S, either it does not appear in the table, or wherever it appears it has the same value of B
  - » John has 1980, everywhere it appears
  - » Lilian does not appear
- Because SC → BTF (and therefore SC → SCBTF, as of course SC → SC), given a specific SC, either it does not appear in the table, or wherever it appears it has the same value of BTF
  - » Mary,PL has 1990,Vijay,1, everywhere it appears
  - » Mary,OS does not appear

Drawing Functional Dependencies



- Each column in a box
- Our key (there could be more than one) is chosen to be the primary key and its boxes have thick borders and it is stored in the left part of the rectangle
- Above the boxes, we have functional dependencies ***"from the full key"*** (this is actually not necessary to draw)
- Below the boxes, we have functional dependencies ***"not from the full key"***
- Colors of lines are not important, but good for explaining

Classification Of Dependencies



- The three “not from the full key” dependencies are classified as:
- ***Partial dependency:*** From a part of the primary key to outside the key
- ***Transitive dependency:*** From outside the key to outside the key
- ***Into key dependency:*** From outside the key into (all or part of) the key

Anomalies

- These “not from the full key” dependencies cause the design to be bad
  - Inability to store important information
  - Redundancies
- Imagine a new Student appears who has not yet registered for a course
  - This S has a specific B, but this cannot be stored in the table as we do not have a value of C yet, and the attributes of the primary key cannot be NULL
- Imagine that Mary withdrew from the only Course she has
  - We have no way of storing her B
- Imagine that we “erase” the value of C in the row stating that Fang was taught by Allan
  - We will know that this was OS, as John was taught OS by Allan, and every teacher teaches only one subject, so we had a redundancy; and whenever there is a redundancy, there is potential for inconsistency

Anomalies

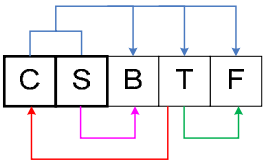
- The way to handle the problems is to replace a table with other equivalent tables that do not have these problems
- Implicitly we think as if the table had only one key (we are not paying attention to keys that are not primary)
- In fact, as we have seen, there is one more key, we just do not think about it (at least for now)

Review Of Our Example

R	S	B	C	T	F
	Fang	1990	DB	Zvi	1
	John	1980	OS	Allan	2
	Mary	1990	PL	Vijay	1
	Fang	1990	OS	Allan	2
	John	1980	PL	Marsha	4

- Our rules
  - A student can have only one birth year: S → B
  - A teacher has to charge the same fee from every student he/she teaches : T → F
  - A teacher can teach only one course (perhaps at different times, different offerings, etc, but never another course) : T → C
  - A student can take a course only from one teacher only : SC → T

Review Of Our "Not From The Full Key" Functional Dependencies



- S → B: partial; called partial because the left hand side is only a proper part of the key
- T → F: transitive; called transitive because as T is outside the key, it of course depends on the key, so we have CS → T and T → F; and therefore CS → F

  Actually, it is more correct (and sometimes done) to say that CS → F is a transitive dependency because it can be decomposed into SC → T and T → F, and then derived by transitivity
- T → C: into the key (from outside the key)

Classification Of The Dependencies: Warning

- Practitioners ***do not*** use consistent definitions for these
- I picked one set of definitions to use here

- We will later have formal machinery to discuss this

- Wikipedia seems to be OK, but other sources of material on the web are frequently wrong (including very respectable ones!)

Redundancies In Our Example

	S	B	C	T	F
	Fang	1990	DB	Zvi	1
	John	1980	OS	Allan	2
	Mary	1990	PL	Vijay	1
	Fang	?	?	Allan	?
	John	?	PL	Marsha	4

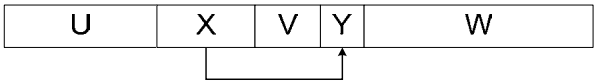
- What could be “recovered” if somebody covered up values (the values are not NULL)?
- All of the empty slots

Our Business Rules Have A Clean Format

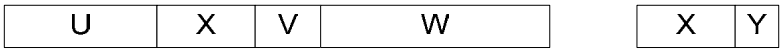
- Our business rules have a clean format
  - » Whoever gave them to us, understood the application very well
- The procedure we describe next assumes rules in such a clean format
- In the Advanced part, we can learn how to “clean” business rules without understanding the application
  - » Computer Scientists do not assume that they understand the application or that the business rules are clean, so they use algorithmic techniques to clean up business rules

A Procedure For Removing Anomalies

- Recall what we did with the example of Grade determining Salary
- In general, we will have sets of attributes: U, X, V, Y, W
- We replaced R(Name,SSN,DOB,Grade,Salary), where Grade → Salary; in the drawing "X" stands for "Grade" and "Y" stands for "Salary"



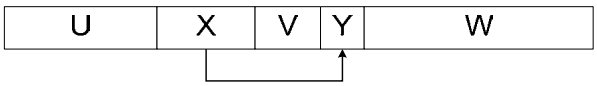
by two tables S(Name,SSN,DOB,Grade) and T(Grade,Salary)



- We will do the same thing, dealing with one anomaly at a time

A Procedure For Removing Anomalies

- While replacing



by two tables

- We do this if Y does not overlap (or is a part of) primary key
- We do not want to "lose" the primary key of the table UXVW, and if Y is not part of primary key of UXVYW, the primary key of UXVYW is part of UXVW and therefore it is a primary key there (a small proof is omitted)

Incorrect Decomposition (Not A Lossless Join Decomposition)

- Assume we replaced

R	Name	SSN	DOB	Grade	Salary
	A	121	2367	2	80
	A	132	3678	3	70
	B	101	3498	4	70
	C	106	2987	2	80

with two tables (note "Y" in the previous slide), which is SSN was actually the key, therefore we should not do it), without indicating the key for S to simplify the example

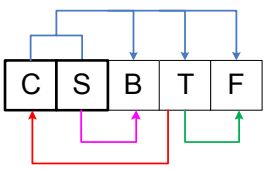
S	Name	DOB	Grade	Salary
	A	2367	2	80
	A	3678	3	70
	B	3498	4	70
	C	2987	2	80

T	SSN	Salary
	121	80
	132	70
	101	70
	106	80

- We cannot answer the question what is the Name for SSN = 121 (we lost information), so cannot decompose like this

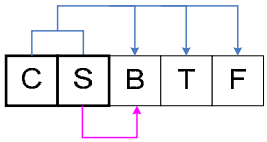
Our Example Again

	S	B	C	T	F
	Fang	1990	DB	Zvi	1
	John	1980	OS	Allan	2
	Mary	1990	PL	Vijay	1
	Fang	1990	OS	Allan	2
	John	1980	PL	Marsha	4



Partial Dependency: S → B

	S	B	C	T	F
	Fang	1990	DB	Zvi	1
	John	1980	OS	Allan	2
	Mary	1990	PL	Vijay	1
	Fang	1990	OS	Allan	2
	John	1980	PL	Marsha	4



Decomposition

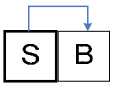
	S	B	C	T	F
	Fang	1990	DB	Zvi	1
	John	1980	OS	Allan	2
	Mary	1990	PL	Vijay	1
	Fang	1990	OS	Allan	2
	John	1980	PL	Marsha	4

	S	B
	Fang	1990
	John	1980
	Mary	1990
	Fang	1990
	John	1980

	S	C	T	F
	Fang	DB	Zvi	1
	John	OS	Allan	2
	Mary	PL	Vijay	1
	Fang	OS	Allan	2
	John	PL	Marsha	4

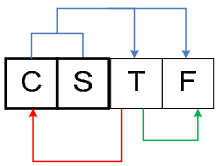
No Anomalies

	S	B
	Fang	1990
	John	1980
	Mary	1990
	Fang	1990
	John	1980



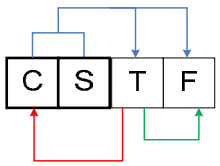
Some Anomalies

	S	C	T	F
	Fang	DB	Zvi	1
	John	OS	Allan	2
	Mary	PL	Vijay	1
	Fang	OS	Allan	2
	John	PL	Marsha	4

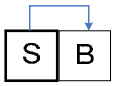


Decomposition So Far

	S	C	T	F
	Fang	DB	Zvi	1
	John	OS	Allan	2
	Mary	PL	Vijay	1
	Fang	OS	Allan	2
	John	PL	Marsha	4



	S	B
	Fang	1990
	John	1980
	Mary	1990



Second Normal Form

- Based on concept of **full functional dependency**
  - Versus **partial dependency**

Definition. A relation schema R is in 2NF if every nonprime attribute A in R is *fully functionally dependent* on the primary key of R.

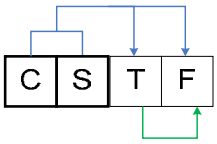
- Second normalize into a number of 2NF relations
  - Nonprime attributes are associated only with part of primary key on which they are fully functionally dependent

Second Normal Form: 1NF And No Partial Dependencies

- Each of the tables in our database is in Second Normal Form
- Second Normal Form means:
  - First Normal Form
  - No Partial dependencies
- The above is checked individually for each table
- Furthermore, our decomposition was a lossless join decomposition
- This means that by "combining" all the tables we get exactly the original table back
- This is checked "globally"; we do not discuss how this is done generally, but intuitively clearly true in our simple example

T → F

	S	C	T	F
	Fang	DB	Zvi	1
	John	OS	Allan	2
	Mary	PL	Vijay	1
	Fang	OS	Allan	2
	John	PL	Marsha	4



Decomposition

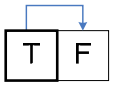
	S	C	T	F
	Fang	DB	Zvi	1
	John	OS	Allan	2
	Mary	PL	Vijay	1
	Fang	OS	Allan	2
	John	PL	Marsha	4

	S	C	T
	Fang	DB	Zvi
	John	OS	Allan
	Mary	PL	Vijay
	Fang	OS	Allan
	John	PL	Marsha

	T	F
	Zvi	1
	Allan	2
	Vijay	1
	Allan	2
	Marsha	4

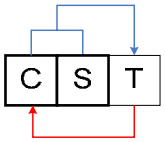
No Anomalies

	T	F
	Zvi	1
	Allan	2
	Vijay	1
	Allan	2
	Marsha	4



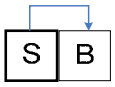
Anomalies

	S	C	T
	Fang	DB	Zvi
	John	OS	Allan
	Mary	PL	Vijay
	Fang	OS	Allan
	John	PL	Marsha

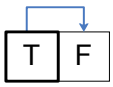


Decomposition So Far

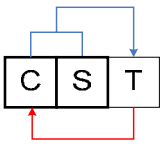
	S	B
	Fang	1990
	John	1980
	Mary	1990



	T	F
	Zvi	1
	Allan	2
	Vijay	1
	Marsha	4



	S	C	T
	Fang	DB	Zvi
	John	OS	Allan
	Mary	PL	Vijay
	Fang	OS	Allan
	John	PL	Marsha



Third Normal Form

- Based on concept of transitive dependency

  Definition. According to Codd's original definition, a relation schema R is in **3NF** if it satisfies 2NF *and* no nonprime attribute of R is transitively dependent on the primary key.

- Problematic FD
  - Left-hand side is part of primary key
  - Left-hand side is a nonkey attribute

General Definitions of Second and Third Normal Forms

Table 15.1 Summary of Normal Forms Based on Primary Keys and Corresponding Normalization

Normal Form	Test	Remedy (Normalization)
First (1NF)	Relation should have no multivalued attributes or nested relations.	Form new relations for each multivalued attribute or nested relation.
Second (2NF)	For relations where primary key contains multiple attributes, no nonkey attribute should be functionally dependent on a part of the primary key.	Decompose and set up a new relation for each partial key with its dependent attribute(s). Make sure to keep a relation with the original primary key and any attributes that are fully functionally dependent on it.
Third (3NF)	Relation should not have a nonkey attribute functionally determined by another nonkey attribute (or by a set of nonkey attributes). That is, there should be no transitive dependency of a nonkey attribute on the primary key.	Decompose and set up a relation that includes the nonkey attribute(s) that functionally determine(s) other nonkey attribute(s).

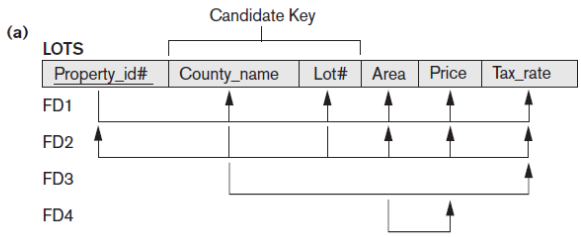
General Definitions of Second and Third Normal Forms (cont'd.)

- **Prime attribute**
  - Part of any candidate key will be considered as prime
- Consider partial, full functional, and transitive dependencies with respect to all candidate keys of a relation

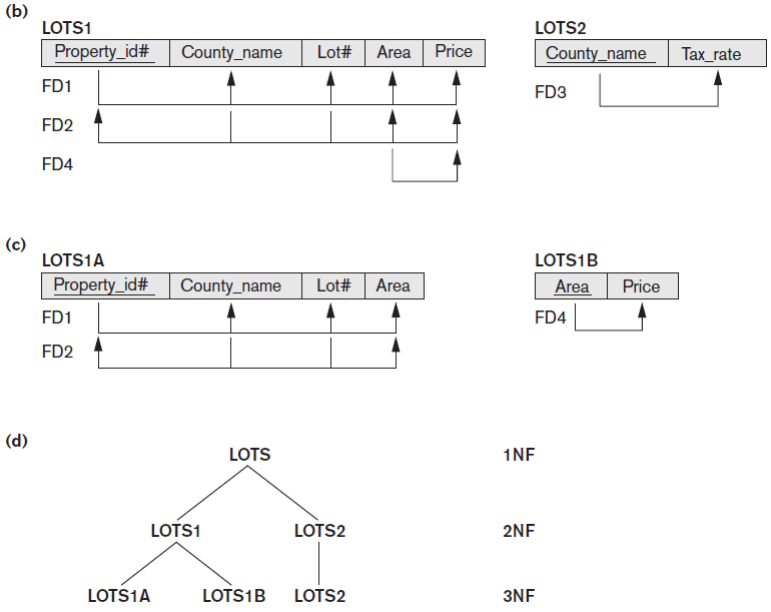
General Definition of Second Normal Form

Definition. A relation schema R is in **second normal form** (2NF) if every non-prime attribute A in R is not partially dependent on *any* key of R.[11]

Figure 15.12
Normalization into 2NF and 3NF. (a) The LOTS relation with its functional dependencies FD1 through FD4. (b) Decomposing into the 2NF relations LOTS1 and LOTS2. (c) Decomposing LOTS1 into the 3NF relations LOTS1A and LOTS1B. (d) Summary of the progressive normalization of LOTS.



Sample Normalization into 2NF and 3NF



General Definition of Third Normal Form

Definition. A relation schema R is in **third normal form** (3NF) if, whenever a *nontrivial* functional dependency $X \rightarrow A$ holds in R, either (a) X is a superkey of R, or (b) A is a prime attribute of R.

Alternative Definition. A relation schema R is in 3NF if every nonprime attribute of R meets both of the following conditions:

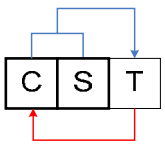
- It is fully functionally dependent on every key of R.
- It is nontransitively dependent on every key of R.

Third Normal Form: 2NF And No Transitive Dependencies

- Each of the tables in our database is in Third Normal Form
- Third Normal Form means:
  - » Second Normal Form (therefore in 1NF and no partial dependencies)
  - » No transitive dependencies
- The above is checked individually for each table
- Furthermore, our decomposition was a lossless join decomposition
- This means that by "combining" all the tables we get exactly the original table back
- This is checked "globally"; we do not discuss how this is done generally, but intuitively clearly true in our simple example

Anomaly

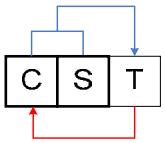
	S	C	T
	Fang	DB	Zvi
	John	OS	Allan
	Mary	PL	Vijay
	Fang	OS	Allan
	John	PL	Marsha



- We are worried about decomposing by "pulling out" C and getting CS and TC, as we are pulling out a part of the key
- But we can actually do it

An Alternative Primary Key

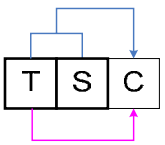
	S	C	T
	Fang	DB	Zvi
	John	OS	Allan
	Mary	PL	Vijay
	Fang	OS	Allan
	John	PL	Marsha



- Note that TS could also serve as primary key since by looking at the FD we have: T → C, we see that TS functionally determines everything, that is TSC
- Recall, that TS could have been chosen at the primary key of the original table

Anomaly

	S	C	T
	Fang	DB	Zvi
	John	OS	Allan
	Mary	PL	Vijay
	Fang	OS	Allan
	John	PL	Marsha



- Now our anomaly is a partial dependency, which we know how to handle

Decomposition

	S	C	T
	Fang	DB	Zvi
	John	OS	Allan
	Mary	PL	Vijay
	Fang	OS	Allan
	John	PL	Marsha

	S	T
	Fang	Zvi
	John	Allan
	Mary	Vijay
	Fang	Allan
	John	Marsha

	C	T
	DB	Zvi
	OS	Allan
	PL	Vijay
	OS	Allan
	PL	Marsha

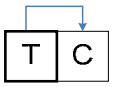
No Anomalies

	S	T
	Fang	Zvi
	John	Allan
	Mary	Vijay
	Fang	Allan
	John	Marsha

T	S

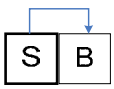
No Anomalies

	C	T
	DB	Zvi
	OS	Allan
	PL	Vijay
	OS	Allan
	PL	Marsha

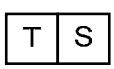


Our Decomposition

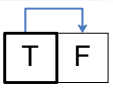
	S	B
	Fang	1990
	John	1980
	Mary	1990



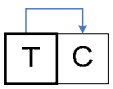
	S	T
	Fang	Zvi
	John	Allan
	Mary	Vijay
	Fang	Allan
	John	Marsha



	T	F
	Zvi	1
	Allan	2
	Vijay	1
	Marsha	4



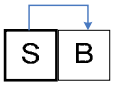
	C	T
	DB	Zvi
	OS	Allan
	PL	Vijay
	PL	Marsha



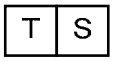
Our Decomposition

- We can also combine tables if they have the same key and we can still maintain good properties

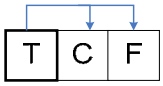
	S	B
	Fang	1990
	John	1980
	Mary	1990



	S	T
	Fang	Zvi
	John	Allan
	Mary	Vijay
	Fang	Allan
	John	Marsha



	T	F	C
	Zvi	1	DB
	Allan	2	OS
	Vijay	1	PL
	Marsha	4	PL



Boyce-Codd Normal Form

- Every relation in BCNF is also in 3NF
  - Relation in 3NF is not necessarily in BCNF

Definition. A relation schema R is in **BCNF** if whenever a *nontrivial* functional dependency $X \rightarrow A$ holds in R, then X is a superkey of R.

- Difference:
  - Condition which allows A to be prime is absent from BCNF
- Most relation schemas that are in 3NF are also in BCNF

Sample Normalization into Boyce-Codd Normal Form

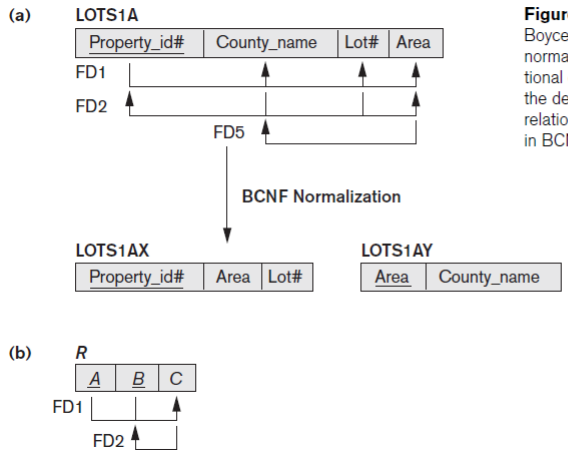


Figure 15.13
Boyce-Codd normal form. (a) BCNF normalization of LOTS1A with the functional dependency FD2 being lost in the decomposition. (b) A schematic relation with FDs; it is in 3NF, but not in BCNF.

Boyce-Codd Normal: 1NF And All Dependencies From Full Key

- Each of the tables in our database is in Boyce-Codd Normal Form
- Boyce-Codd Normal Form (BCNF) means:
  - First Normal Form
  - Every functional dependency is from a full key

  This definition is “loose.” Later, a complete, formal definition
- A table is BCNF is automatically in 3NF (elaboration later in the course)
- The above is checked individually for each table

- Furthermore, our decomposition was a lossless join decomposition
- This means that by “combining” all the tables we get exactly the original table back
- This is checked “globally”; we do not discuss how this is done generally, but intuitively clearly true in our simple example

A New Issue: Maintaining Database Correctness And Preservation Of Dependencies

- We can understand this just by looking at the table which we decomposed last
- We will not use drawings but write the constraints that needed to be satisfied in narrative
- We will examine an update to the database and look at two scenarios
- When we have one “imperfect” 3NF table SCT
- When we have two “perfect” BCNF tables ST and CT
- We will attempt an incorrect update and see how to detect it under both scenarios

Our Tables (For The Two Cases)

- SCT satisifies: SC → T and ST →C: keys SC and ST

	S	C	T
	Fang	DB	Zvi
	John	OS	Allan
	Mary	PL	Vijay
	Fang	OS	Allan
	John	PL	Marsha

- ST does not satisfy anything: key ST
- CT satisfies T → C: key T

	S	T
	Fang	Zvi
	John	Allan
	Mary	Vijay
	Fang	Allan
	John	Marsha

	C	T
	DB	Zvi
	OS	Allan
	PL	Vijay
	OS	Allan
	PL	Marsha

An Insert Attempt

- A user wants to specify that now John is going to take PL from Vijay
- If we look at the database, we realize this update should not be permitted because
  - » John can take PL from at most one teacher
  - » John already took PL (from Marsha)
- But can the system figure this out just by checking whether FDs continue being satisified?
- Let us find out what will happen in each of the two scenarios

Scenario 1: SCT

- We maintain SCT, knowing that its keys are SC and ST
- Before the INSERT, constraints are satisfied; keys are OK
- After the INSERT, constraints are not satisfied; SC is no longer a key
- INSERT rejected after the constraint is checked

	S	C	T
	Fang	DB	Zvi
	John	OS	Allan
	Mary	PL	Vijay
	Fang	OS	Allan
	John	PL	Marsha

	S	C	T
	Fang	DB	Zvi
	John	OS	Allan
	Mary	PL	Vijay
	Fang	OS	Allan
	John	PL	Marsha
	John	PL	Vijay

Scenario 2: ST And CT

- We maintain ST, knowing that its key ST
- We maintain CT, knowing that its key is T

- Before the INSERT, constraints are satisfied; keys are OK

S	T
Fang	Zvi
John	Allan
Mary	Vijay
Fang	Allan
John	Marsha

C	T
DB	Zvi
OS	Allan
PL	Vijay
OS	Allan
PL	Marsha

- After the INSERT, constraints are still satisfied; keys remain keys

S	T
Fang	Zvi
John	Allan
Mary	Vijay
Fang	Allan
John	Marsha
John	Vijay

C	T
DB	Zvi
OS	Allan
PL	Vijay
OS	Allan
PL	Marsha
PL	Vijay

- But the INSERT ***must*** still be rejected

Scenario 2: What To Do?

- The INSERT must be rejected
- This bad insert cannot be discovered as bad by examining only what happens in each individual table
- The formal term for this is: ***dependencies are not preserved***
- So need to perform non-local tests to check updates for validity
- For example, take ST and CT and reconstruct SCT

A Very Important Conclusion

- Generally, normalize up to 3NF and not up to BCNF
  - » So the database is not fully normalized
- Luckily, when you do this, frequently you "automatically" get BCNF
  - » But not in our example, which is set up on purpose so this does not happen

Multivalued Dependencies

- To have a smaller example, we will look at this separately, not by extending our previous example
  - » Otherwise, it would become too big
- In the application, we store information about Courses (C), Teachers (T), and Books (B)
- Each course has a set of books that have to be assigned during the course
- Each course has a set of teachers that are qualified to teach the course
- Each teacher, when teaching a course, has to use the set of the books that has to be assigned in the course

An Example table

	C	T	B
	DB	Zvi	Oracle
	DB	Zvi	Linux
	DB	Dennis	Oracle
	DB	Dennis	Linux
	OS	Dennis	Windows
	OS	Dennis	Linux
	OS	Jinyang	Windows
	OS	Jinyang	Linux

- This instance (and therefore the table in general) does not satisfy any functional dependencies
  - » CT does not functionally determine B
  - » CB does not functionally determine T
  - » TB does not functionally determent C

Redundancies

	C	T	B
	DB	Zvi	Oracle
	DB	Zvi	Linux
	DB	Dennis	?
	DB	Dennis	?
	OS	Dennis	Windows
	OS	Dennis	Linux
	OS	Jinyang	?
	OS	Jinyang	?

	C	T	B
	DB	Zvi	Oracle
	DB	?	Linux
	DB	Dennis	Oracle
	DB	?	Linux
	OS	Dennis	Windows
	OS	?	Linux
	OS	Jinyang	Windows
	OS	?	Linux

- There are obvious redundancies
- In both cases, we know exactly how to fill the missing data if it was erased
- We decompose to get rid of anomalies

Decomposition

	C	T	B
	DB	Zvi	Oracle
	DB	Zvi	Linux
	DB	Dennis	Oracle
	DB	Dennis	Linux
	OS	Dennis	Windows
	OS	Dennis	Linux
	OS	Jinyang	Windows
	OS	Jinyang	Linux

	C	T
	DB	Zvi
	DB	Dennis
	OS	Dennis
	OS	Jinyang

	C	B
	DB	Oracle
	DB	Linux
	OS	Windows
	OS	Linux

Multivalued Dependency and Fourth Normal Form Definitions

- Multivalued dependency (MVD)
  - Consequence of first normal form (1NF)

Definition. A multivalued dependency $X \twoheadrightarrow Y$ specified on relation schema R, where X and Y are both subsets of R, specifies the following constraint on any relation state r of R: If two tuples t_1 and t_2 exist in r such that $t_1[X] = t_2[X]$, then two tuples t_3 and t_4 should also exist in r with the following properties,[15] where we use Z to denote $(R - (X \cup Y))$:[16]

- $t_3[X] = t_4[X] = t_1[X] = t_2[X]$.
- $t_3[Y] = t_1[Y]$ and $t_4[Y] = t_2[Y]$.
- $t_3[Z] = t_2[Z]$ and $t_4[Z] = t_1[Z]$.

Multivalued Dependency and Fourth Normal Form (cont'd.)

- Relations containing nontrivial MVDs
  - **All-key relations**
- **Fourth normal form (4NF)**
  - Violated when a relation has undesirable multivalued dependencies

Definition. A relation schema R is in 4NF with respect to a set of dependencies F (that includes functional dependencies and multivalued dependencies) if, for every *nontrivial* multivalued dependency $X \twoheadrightarrow Y$ in F^{+17} X is a superkey for R.

Multivalued Dependencies And 4NF

- We had the following situation
- For each value of C there was
  - A set of values of T
  - A set of values of B
- Such that, every T of C had to appear with every B of C

  This is stated here rather loosely, but it is clear what it means
- The notation for this is: C $\rightarrow\!\!\rightarrow$ T | B
- The tables CT and CB where in ***Fourth Normal Form (4NF)***
- We will define this formally later in the next section of this deck

Introduction To Algorithmic Techniques

- We will go over an introduction to algorithmic techniques, which are more fully described in the advanced part

Closures Of Sets Of Attributes (Column Names)

- A database contains some tables, which of course, are defined by a set of their column names
- The database satisfies some business rules, which are specified by means of functional dependencies
- For example, we may be given that some table with attributes (column names):
  - Employee (E, for short, meaning really the SSN of the employee)
  - Grade (G, for short)
  - Salary (S, for short)
- Satisfies:
  1. $E \rightarrow G$
  2. $G \rightarrow S$
- We would like to find all the keys of this table
- A key is a minimal set of attributes, such that the values of these attributes, “force” some values for all the other attributes

Closures Of Sets Of Attributes

- In general, we have a concept of a the closure of the set of attributes
- Let X be a set of attributes, then X^+ is the set of all attributes, whose values are forced by the values of X
- In our example
  - » $E^+ = EGS$ (because given E we have the value of G and then because we have the value for G we have the value for E)
  - » $G^+ = GS$
  - » $S^+ = S$
- This is interesting because we have just showed that E is a key
- And here we could also figure out that this is the only key, as $GS^+ = GS$, so we will never get E unless we already have it
- Note that GS^+ really means $(GS)^+$

Computing Closures Of Sets Of Attributes

- There is a very simple algorithm to compute X^+

1. Let Y = X
2. Whenever there is an FD, say $V \rightarrow W$, such that
   1. $V \subseteq Y$, and
   2. $W - Y$ is not empty

   add $W - Y$ to Y
3. At termination $Y = X^+$

- The algorithm is very efficient
- Each time we look at all the functional dependencies
  - » Either we can apply at least one functional dependency and make Y bigger (the biggest it can be are all attributes), or
  - » We are finished

Example

- Let R = ABCDEGHIJK
- Given FDs:
  1. K → BG
  2. A → DE
  3. H → AI
  4. B → D
  5. J → IH
  6. C → K
  7. I → J
- We will compute: ABC^+
  1. We start with $ABC^+ = ABC$
  2. Using FD number 2, we now have: $ABC^+ = ABCDE$
  3. Using FD number 6, we now have $ABC^+ = ABCDEK$
  4. Using FD number 1, we now have $ABC^+ = ABCDEKG$

  No FD can be applied productively anymore and we are done

Keys Of Tables

- The notion of an FD allows us to formally define keys
- Given R, satisfying a set of FDs, a set of attributes X of R is a key, if and only if:
  - $X^+ = R$.
  - For any $Y \subseteq X$ such that $Y \neq X$, we have $Y^+ \neq R$.
- Note that if R does not satisfy any (nontrivial) FDs, then R is the only key of R
- Example, if a table is R(FirstName,LastName) without any functional dependencies, then its key is just the pair (FirstName,LastName)
- If we apply our algorithm to the EGS example given earlier, we can now just compute that E was (the only) key by checking all the subsets of E,G,S

Example

- Let R = ABCDEKGHIJ
- Given FDs:
  1. K → BG
  2. A → DE
  3. H → AI
  4. B → D
  5. J → IH
  6. C → K
  7. I → J
- Then
  - $ABCH^+$ = ABCDEGHIJK
  - And ABCH is a key or maybe contains a key as a proper subset
  - We could check whether ABCH is minimal by computing ABC^+, ABH^+, ACH^+, BCH^+

Example: Airline Scheduling

- We have a table PFDT, where
  - **P**ILOT
  - **F**LIGHT NUMBER
  - **D**ATE
  - SCHEDULED_**T**IME_of_DEPARTURE
- The table satisfies the FDs:
  - F → T
  - PDT → F
  - FD → P

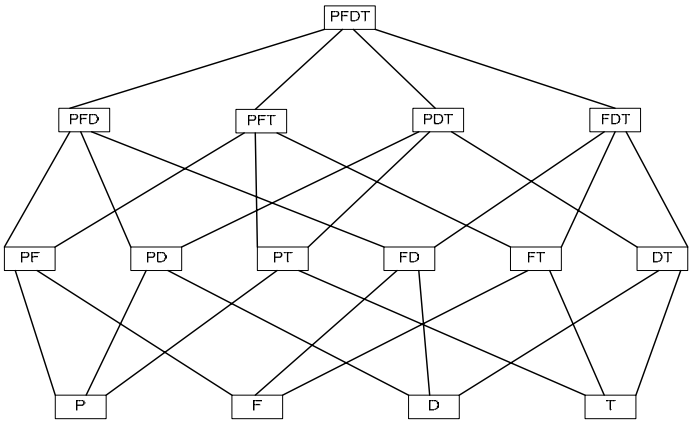
Computing Keys

- We will compute all the keys of the table
- In general, this will be an exponential-time algorithm in the size of the problem
- But there will be useful heuristic making this problem tractable in practice
- We will introduce some heuristics here and additional ones later
- We note that if some subset of attributes is a key, then no proper superset of it can be a key as it would not be minimal and would have superfluous attributes

Lattice Of Sets Of Attributes

- There is a natural structure (technically a lattice) to all the nonempty subsets of attributes
- I will draw the lattice here, in practice this is not done
  - » Not necessary and too big
- We will look at all the non-empty subsets of attributes
- There are 15 of them: $2^4 - 1$
- The structure is clear from the drawing

Lattice Of Nonempty Subsets



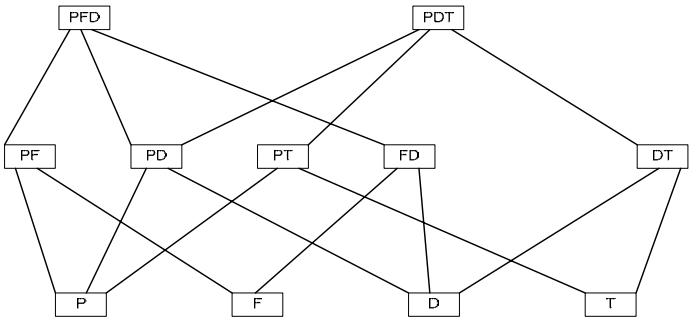
Keys Of PFDT

- The algorithm proceeds from bottom up
- We first try all potential 1-attribute keys, by examining all 1-attribute sets of attributes
  - $P^+ = P$
  - $F^+ = FT$
  - $D^+ = D$
  - $T^+ = T$

  There are no 1-attribute keys

- Note, that the it is impossible for a key to have ***both*** F and T
  - Because if F is in a key, T will be automatically determined as it is included in the closure of F
- Therefore, we can prune our lattice

Pruned Lattice



Keys Of PFDT

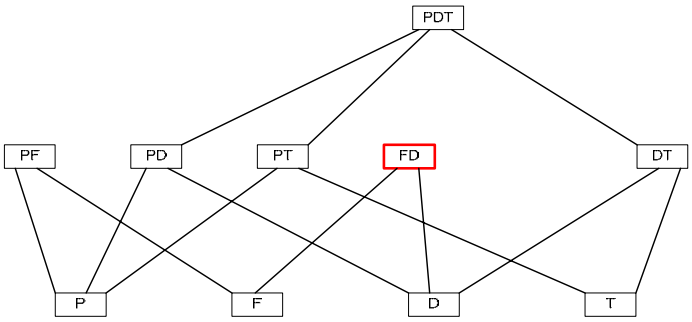
- We try all potential 2-attribute keys
  - $PF^+ = PFT$
  - $PD^+ = PD$
  - $PT^+ = PT$
  - FD+ = FDPT
  - DT+ = DT

  There is one 2-attribute key: FD

We can mark the tree

We can prune the lattice

Pruned Lattice

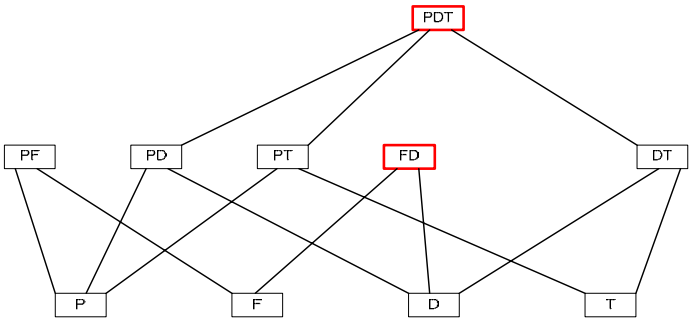


Keys Of PFDT

- We try all potential 3-attribute keys
  - » $PDT^+ = PDTF$

  There is one 3-attribute key: PDT

Final Lattice - We Only Care About The Keys



Finding A Decomposition

- Next, we will discuss by means of an example how to decompose a table into tables, such that

1. The decomposition is lossless join
2. Dependencies are preserved
3. Each resulting table is in 3NF

- This will just be an overview as the complete details are in the advanced section

The EmToPrHoSkLoRo Table

- The table deals with employees who use tools on projects and work a certain number of hours per week
- An employee may work in various locations and has a variety of skills
- All employees having a certain skill and working in a certain location meet in a specified room once a week

- The attributes of the table are:
  - Em: Employee
  - To: Tool
  - Pr: Project
  - Ho: Hours per week
  - Sk: Skill
  - Lo: Location
  - Ro: Room for meeting

The FDs Of The Table

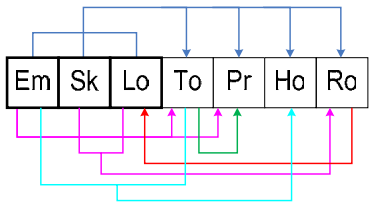
- The table deals with employees who use tools on projects and work a certain number of hours per week
- An employee may work in various locations and has a variety of skills
- All employees having a certain skill and working in a certain location meet in a specified room once a week
- The table satisfies the following FDs:
  - Each employee uses a single tool: Em $\rightarrow$ To
  - Each employee works on a single project: Em $\rightarrow$ Pr
  - Each tool can be used on a single project only: To $\rightarrow$ Pr
  - An employee uses each tool for the same number of hours each week: EmTo $\rightarrow$ Ho
  - All the employees working in a location having a certain skill always work in the same room (in that location): SkLo $\rightarrow$ Ro
  - Each room is in one location only: Ro $\rightarrow$ Lo

Sample Instance: Many Redundancies

	Em	To	Pr	Ho	Sk	Lo	Ro
	Mary	Pen	Research	20	Clerk	Boston	101
	Mary	Pen	Research	20	Writer	Boston	102
	Mary	Pen	Research	20	Writer	Buffalo	103
	Fang	Pen	Research	30	Clerk	New York	104
	Fang	Pen	Research	30	Editor	New York	105
	Fang	Pen	Research	30	Economist	New York	106
	Fang	Pen	Research	30	Economist	Buffalo	107
	Lakshmi	Oracle	Database	40	Analyst	Boston	101
	Lakshmi	Oracle	Database	40	Analyst	Buffalo	108
	Lakshmi	Oracle	Database	40	Clerk	Buffalo	107
	Lakshmi	Oracle	Database	40	Clerk	Boston	101
	Lakshmi	Oracle	Database	40	Clerk	Albany	109
	Lakshmi	Oracle	Database	40	Clerk	Trenton	110
	Lakshmi	Oracle	Database	40	Economist	Buffalo	107

Our FDs

1. Em → To
2. Em → Pr
3. To → Pr
4. EmTo → Ho
5. SkLo → Ro
6. Ro → Lo



- What should we do with this drawing? I do not know.
- We know how to find keys (we will actually do it later) and we can figure that EmSkLo could serve as the primary key, so we could draw using the appropriate colors
- But note that there for FD number 4, the left hand side contains an attribute from the key and an attribute from outside the key, so I used a new color
- Let's forget for now that I have told you what the primary key was, we will find it later

1: Getting A Canonical Cover

- We need to “simplify” our set of FDs to bring it to a “nicer” form, so called canonical of minimal cover
- But, of course, the power has to be the same as we need to enforce the same business rules
- The algorithm for this is in the advanced part of this unit
- The end result is:
  1. Em → ToHo
  2. To → Pr
  3. SkLo → Ro
  4. Ro → Lo
- From these we will built our tables directly

2: Creating Tables

- Create a table for each functional dependency
- We obtain the tables:
  1. EmToHo
  2. ToPr
  3. SkLoRo
  4. LoRo

3: Removing Redundant Tables

- LoRo is a subset of SkLoRo, so we remove it
- We obtain the tables:
  1. EmToHo
  2. ToPr
  3. SkLoRo

4: Ensuring The Storage Of The Global Key (Of The Original Table)

- We need to have a table containing the global key
- Perhaps one of our tables contain such a key
- So we check if any of them already contains a key of EmToPrHoSkLoRo:

1. EmToHo	$EmToHo^+$ = EmToHoPr, does not contain a key
2. ToPr	$ToPr^+$ = ToPr, does not contain a key
3. SkLoRo	$SkLoRo^+$ = SkLoRo, does not contain a key

- We need to add a table whose attributes form a global key

Finding Keys

- Let us list the FDs again (or could have worked with the minimal cover, does not matter):
  - Em → To
  - Em → Pr
  - To → Pr
  - EmTo → Ho
  - SkLo → Ro
  - Ro → Lo
- We can classify the attributes into 4 classes:
  1. Appearing on both sides of FDs; here To, Lo, Ro.
  2. Appearing on left sides only; here Em, Sk.
  3. Appearing on right sides only; here Pr, Ho.
  4. Not appearing in FDs; here none.

Finding Keys

- Facts:
  - Attributes of class 2 and 4 must appear in every key
  - Attributes of class 3 do not appear in any key
  - Attributes of class 1 may or may not appear in keys
- An algorithm for finding keys relies on these facts
  - Unfortunately, in the worst case, exponential in the number of attributes
- Start with the attributes in classes 2 and 4, add as needed (going bottom up) attributes in class 1, and ignore attributes in class 3

Finding Keys

- In our example, therefore, every key must contain EmSk
- To see, which attributes, if any have to be added, we compute which attributes are determined by EmSk
- We obtain
  - $EmSk^+ = EmToPrHoSk$
- Therefore Lo and Ro are missing
- It is easy to see that the table has two keys
  - EmSkLo
  - EmSkRo

Finding Keys

- Although not required strictly by the algorithm (which does not mind decomposing a table in 3NF into tables in 3NF) we can check if the original table was in 3NF
- We conclude that the original table is not in 3NF, as for instance, To $\rightarrow$ Pr is a transitive dependency and therefore not permitted for 3NF

4: Ensuring The Storage Of The Global Key

- None of the tables contains either EmSkLo or EmSkRo.
- Therefore, one more table needs to be added. We have 2 choices for the final decomposition
  1. EmToHo; satisfying Em → ToHo; primary key: Em
  2. ToPr; satisfying To → Pr; primary key To
  3. SkLoRo; satisfying SkLo → Ro and Ro → Lo; primary key SkLo or SkRo
  4. EmSkLo; not satisfying anything; primary key EmSkLo

     or
  1. EmToHo; satisfying Em → ToHo; primary key: Em
  2. ToPr; satisfying To → Pr; primary key To
  3. SkLoRo; satisfying SkLo → Ro and Ro → Lo; primary key SkLo or SkRo
  4. EmSkRo ; not satisfying anything; primary key SkRO
- We have completed our process and got a decomposition with the properties we needed; actually more than one

A Decomposition

Em	Sk	Lo
Mary	Clerk	Boston
Mary	Writer	Boston
Mary	Writer	Buffalo
Fang	Clerk	New York
Fang	Editor	New York
Fang	Economist	New York
Fang	Economist	Buffalo
Lakshmi	Analyst	Boston
Lakshmi	Analyst	Buffalo
Lakshmi	Clerk	Buffalo
Lakshmi	Clerk	Boston
Lakshmi	Clerk	Albany
Lakshmi	Clerk	Trenton
Lakshmi	Economist	Buffalo

Em	To	Ho
Mary	Pen	20
Fang	Pen	30
Lakshmi	Oracle	40

To	Pr
Pen	Research
Oracle	Database

Sk	Lo	Ro
Clerk	Boston	101
Writer	Boston	102
Writer	Buffalo	103
Clerk	New York	104
Editor	New York	105
Economist	New York	106
Economist	Buffalo	107
Analyst	Boston	101
Analyst	Buffalo	108
Clerk	Buffalo	107
Clerk	Albany	109
Clerk	Trenton	110

A Decomposition

Em	Sk	Lo
Mary	Clerk	Boston
Mary	Writer	Boston
Mary	Writer	Buffalo
Fang	Clerk	New York
Fang	Editor	New York
Fang	Economist	New York
Fang	Economist	Buffalo
Lakshmi	Analyst	Boston
Lakshmi	Analyst	Buffalo
Lakshmi	Clerk	Buffalo
Lakshmi	Clerk	Boston
Lakshmi	Clerk	Albany
Lakshmi	Clerk	Trenton
Lakshmi	Economist	Buffalo

Em	To	Ho
Mary	Pen	20
Fang	Pen	30
Lakshmi	Oracle	40

To	Pr
Pen	Research
Oracle	Database

Sk	Lo	Ro
Clerk	Boston	101
Writer	Boston	102
Writer	Buffalo	103
Clerk	New York	104
Editor	New York	105
Economist	New York	106
Economist	Buffalo	107
Analyst	Boston	101
Analyst	Buffalo	108
Clerk	Buffalo	107
Clerk	Albany	109
Clerk	Trenton	110

A Decomposition

Em	Sk	Ro
Mary	Clerk	101
Mary	Writer	102
Mary	Writer	103
Fang	Clerk	104
Fang	Editor	105
Fang	Economist	106
Fang	Economist	107
Lakshmi	Analyst	101
Lakshmi	Analyst	108
Lakshmi	Clerk	107
Lakshmi	Clerk	101
Lakshmi	Clerk	109
Lakshmi	Clerk	110
Lakshmi	Economist	107

Em	To	Ho
Mary	Pen	20
Fang	Pen	30
Lakshmi	Oracle	40

To	Pr
Pen	Research
Oracle	Database

Sk	Lo	Ro
Clerk	Boston	101
Writer	Boston	102
Writer	Buffalo	103
Clerk	New York	104
Editor	New York	105
Economist	New York	106
Economist	Buffalo	107
Analyst	Boston	101
Analyst	Buffalo	108
Clerk	Buffalo	107
Clerk	Albany	109
Clerk	Trenton	110

A Decomposition

Em	Sk	Ro
Mary	Clerk	101
Mary	Writer	102
Mary	Writer	103
Fang	Clerk	104
Fang	Editor	105
Fang	Economist	106
Fang	Economist	107
Lakshmi	Analyst	101
Lakshmi	Analyst	108
Lakshmi	Clerk	107
Lakshmi	Clerk	101
Lakshmi	Clerk	109
Lakshmi	Clerk	110
Lakshmi	Economist	107

Em	To	Ho
Mary	Pen	20
Fang	Pen	30
Lakshmi	Oracle	40

To	Pr
Pen	Research
Oracle	Database

Sk	Lo	Ro
Clerk	Boston	101
Writer	Boston	102
Writer	Buffalo	103
Clerk	New York	104
Editor	New York	105
Economist	New York	106
Economist	Buffalo	107
Analyst	Boston	101
Analyst	Buffalo	108
Clerk	Buffalo	107
Clerk	Albany	109
Clerk	Trenton	110

Properties Of The Decomposition

- The table on the left listed the values of the key of the original table
- Each row corresponded to a row of the original table
- The other tables had rows that could be “glued” to the “key” table and reconstruct the original table
- All the tables are in 3NF

DB Design Process (Roadmap)

- Produce a good ER diagram, thinking of all the issues
- Specify all dependencies that you know about
- Produce relational implementation
- Normalize to whatever extent feasible
- Specify all assertions and checks
- Possibly denormalize for performance
  - » May want to keep both EGS and GS
  - » This can be done also by storing EG and GS and defining EGS as a view

DB Design Process (Roadmap)

- If there is no UNIQUE constraint, that is there is only one key, the PRIMARY KEY, then 3NF and BCNF are the same

  But this is only a special case
- Sometimes, the exposition is oversimplified, look at http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp?topic=/com.ibm.db2.udb.admin.doc/doc/c0004100.htm

Join Dependencies and Fifth Normal Form

- **Join dependency**
- Multiway decomposition into fifth normal form (5NF)
- Very peculiar semantic constraint
  - Normalization into 5NF is very rarely done in practice

Join Dependencies and Fifth Normal Form (cont'd.)

Definition. A **join dependency** (**JD**), denoted by JD(R_1, R_2, ..., R_n), specified on relation schema R, specifies a constraint on the states r of R. The constraint states that every legal state r of R should have a nonadditive join decomposition into R_1, R_2, ..., R_n. Hence, for every such r we have

$* (\pi_{R_1}(r), \pi_{R_2}(r), ..., \pi_{R_n}(r)) = r$

Definition. A relation schema R is in **fifth normal form** (**5NF**) (or **project-join normal form** (**PJNF**)) with respect to a set F of functional, multivalued, and join dependencies if, for every nontrivial join dependency JD(R_1, R_2, ..., R_n) in F^+ (that is, implied by F),[18] every R_i is a superkey of R.

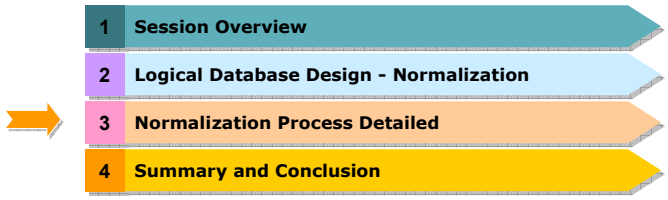
Key Ideas (1/2)

- Need for decomposition of tables
- Functional dependencies
- Some types of functional dependencies:
  - » Partial dependencies
  - » Transitive dependencies
  - » Into full key dependencies
- First Normal Form: 1NF
- Second Normal Form: 2NF
- Third Normal Form: BCNF
- Removing redundancies
- Lossless join decomposition
- Preservation of dependencies
- 3NF vs. BCNF

Key Ideas (2/2)

- Multivalued dependencies
- Fourth Normal Form: 4NF
- Canonical (minimal) cover for a set of functional dependencies
- Algorithmic techniques for finding keys
- Algorithmic techniques for computing an a canonical cover
- Algorithmic technique for obtaining a decomposition of relation into a set of relations, such that
  - » The decomposition is lossless join
  - » Dependencies are preserved
  - » Each resulting relation is in 3NF

Agenda



Agenda

- This section contains a more detailed and precise description of the normalization process
- Most importantly, how to compute a canonical cover
- The three concepts we understand precisely after this unit will be
  - Lossless-join decomposition
  - Normal forms, focusing on 3NF and BCNF
  - Preservation of dependencies
- We will learn an algorithm converting a relation into a set of relations in 3NF with lossless-join decomposition and preservation of dependencies
- We will touch on some additional topics

Algorithmic Normalization

- We abandon our ad-hoc approach and now move to precise specification and algorithms
- We have to work with a clean model
- It will not matter whether we have sets or multisets, as was the case before
  - » I may remove duplicates simply to save on space, whether they are removed or not makes no difference
- We will assume that there are no NULLs
  - » Could extend this to the case when there are NULLS
- We will assume that all the relations are in 1NF, which we have defined already

DESIGNING A SET OF RELATIONS (1)

- **The Approach of Relational Synthesis (Bottom-up Design):**
  - » Assumes that all possible functional dependencies are known.
  - » First constructs a minimal set of FDs
  - » Then applies algorithms that construct a target set of 3NF or BCNF relations.
  - » Additional criteria may be needed to ensure the the *set of relations* in a relational database are satisfactory.

DESIGNING A SET OF RELATIONS (2)

- **Goals:**
  - » Lossless join property (a must)
    - Algorithm 16.3 tests for general losslessness.
  - » Dependency preservation property
    - Algorithm 16.5 decomposes a relation into BCNF components by sacrificing the dependency preservation.
  - » Additional normal forms
    - 4NF (based on multi-valued dependencies)
    - 5NF (based on join dependencies)

1. Properties of Relational Decompositions (1)

- **Relation Decomposition and Insufficiency of Normal Forms:**
  - »Universal Relation Schema:
    - A relation schema R = {A1, A2, …, An} that includes all the attributes of the database.
  - »Universal relation assumption:
    - Every attribute name is unique.

Properties of Relational Decompositions (2)

- **Relation Decomposition and Insufficiency of Normal Forms (cont.):**
  - » Decomposition:
    - The process of decomposing the universal relation schema R into a set of relation schemas D = {R1,R2, …, Rm} that will become the relational database schema by using the functional dependencies.
  - » Attribute preservation condition:
    - Each attribute in R will appear in at least one relation schema Ri in the decomposition so that no attributes are “lost”.

Properties of Relational Decompositions (2)

- Another goal of decomposition is to have each individual relation Ri in the decomposition D be in BCNF or 3NF.
- Additional properties of decomposition are needed to prevent from generating spurious tuples

Properties of Relational Decompositions (3)

- **Dependency Preservation Property of a Decomposition:**
  - » Definition: Given a set of dependencies F on R, the **projection** of F on R_i, denoted by $p_{Ri}(F)$ where R_i is a subset of R, is the set of dependencies $X \rightarrow Y$ in F^+ such that the attributes in $X \cup Y$ are all contained in R_i.
  - » Hence, the projection of F on each relation schema R_i in the decomposition D is the set of functional dependencies in F^+, the closure of F, such that all their left- and right-hand-side attributes are in R_i.

Properties of Relational Decompositions (4)

- **Dependency Preservation Property of a Decomposition (cont.):**
  - » Dependency Preservation Property:
    - A decomposition D = {R1, R2, ..., Rm} of R is **dependency-preserving** with respect to F if the union of the projections of F on each Ri in D is equivalent to F; that is
      $((\pi_{R1}(F)) \cup \ldots \cup (\pi_{Rm}(F)))^+ = F^+$
    - (See examples in Fig 15.13a and Fig 15.12)
- Claim 1:
  - » It is always possible to find a dependency-preserving decomposition D with respect to F such that each relation Ri in D is in 3nf.

Properties of Relational Decompositions (5)

- **Lossless (Non-additive) Join Property of a Decomposition:**
  - » Definition: Lossless join property: a decomposition D = {R1, R2, ..., Rm} of R has the **lossless (nonadditive) join property** with respect to the set of dependencies F on R if, for *every* relation state r of R that satisfies F, the following holds, where * is the natural join of all the relations in D:

$$* (\pi_{R1}(r), \ldots, \pi_{Rm}(r)) = r$$

  - » Note: The word loss in lossless refers to loss of information, not to loss of tuples. In fact, for "loss of information" a better term is "**addition of spurious information**"

Properties of Relational Decompositions (6)

- **Lossless (Non-additive) Join Property of a Decomposition (cont.):**
- **Algorithm 16.3: Testing for Lossless Join Property**
  - » **Input**: A universal relation R, a decomposition D = {R1, R2, ..., Rm} of R, and a set F of functional dependencies.

1. Create an initial matrix S with one row i for each relation Ri in D, and one column j for each attribute Aj in R.

2. Set S(i,j):=bij for all matrix entries. (* each bij is a distinct symbol associated with indices (i,j) *).

3. For each row i representing relation schema Ri

{for each column j representing attribute Aj

{if (relation Ri includes attribute Aj) then set S(i,j):= aj;};};

  - » (* each aj is a distinct symbol associated with index (j) *)

» CONTINUED on NEXT SLIDE

Properties of Relational Decompositions (7)

- **Lossless (Non-additive) Join Property of a Decomposition (cont.):**
- **Algorithm 16.3: Testing for Lossless Join Property**

4. Repeat the following loop until a complete loop execution results in no changes to S

{for each functional dependency X →Y in F

{for all rows in S *which have the same symbols* in the columns corresponding to attributes in X

{make the symbols in each column that correspond to an attribute in Y be the same in all these rows as follows:

If any of the rows has an "a" symbol for the column, set the other rows to that *same* "a" symbol in the column.

If no "a" symbol exists for the attribute in any of the rows, choose one of the "b" symbols that appear in one of the rows for the attribute and set the other rows to that same "b" symbol in the column ;};

};

};

5. If a row is made up entirely of "a" symbols, then the decomposition has the lossless join property; otherwise it does not.

Properties of Relational Decompositions (8)

Lossless (nonadditive) join test for *n*-ary decompositions.
(a) Case 1: Decomposition of EMP_PROJ into EMP_PROJ1 and EMP_LOCS fails test.
(b) A decomposition of EMP_PROJ that has the lossless join property.

(a) R = {Ssn, Ename, Pnumber, Pname, Plocation, Hours} $D = \{R_1, R_2\}$

R_1 = EMP_LOCS = {Ename, Plocation}

R_2 = EMP_PROJ1 = {Ssn, Pnumber, Hours, Pname, Plocation}

F = {Ssn → Ename; Pnumber → {Pname, Plocation}; {Ssn, Pnumber} → Hours}

	Ssn	Ename	Pnumber	Pname	Plocation	Hours
R_1	b_{11}	a_2	b_{13}	b_{14}	a_5	b_{16}
R_2	a_1	b_{22}	a_3	a_4	a_5	a_6

(No changes to matrix after applying functional dependencies)

(b)

EMP

Ssn	Ename

PROJECT

Pnumber	Pname	Plocation

WORKS_ON

Ssn	Pnumber	Hours

Properties of Relational Decompositions (9)

- **Testing Binary Decompositions for Lossless Join Property**
  - **Binary Decomposition:** Decomposition of a relation R into two relations.
  - **PROPERTY LJ1 (lossless join test for binary decompositions):** A decomposition D = {R1, R2} of R has the lossless join property with respect to a set of functional dependencies F on R *if and only if* either
    - The f.d. ((R1 ∩ R2) → (R1- R2)) is in F^+, or
    - The f.d. ((R1 ∩ R2) → (R2 - R1)) is in F^+.

Properties of Relational Decompositions (10)

- **Successive Lossless Join Decomposition:**
  - **Claim 2 (Preservation of non-additivity in successive decompositions):**
    - If a decomposition D = {R1, R2, ..., Rm} of R has the lossless (non-additive) join property with respect to a set of functional dependencies F on R,
    - and if a decomposition Di = {Q1, Q2, ..., Qk} of Ri has the lossless (non-additive) join property with respect to the projection of F on Ri,
      - then the decomposition D2 = {R1, R2, ..., Ri-1, Q1, Q2, ..., Qk, Ri+1, ..., Rm} of R has the non-additive join property with respect to F.

A Canonical Example

- We focus on the relation schema R=R(E,G,S), simplified version of what we have seen in the informal synopsis of the course, where
  - E: Employee number
  - G: Grade
  - S: Salary
- The customers specified for us a set of semantic constraints (called "business rules" in businesses):
  - Each value of E has a single value of G associated with it
  - Each value of E has a single value of S associated with it
  - Each value of G has a single value of S associated with it
- For simplicity, we will sometimes refer to relations schemes by listing their attributes; thus we may write EGS instead of R above
- We will spend a lot of time discussing this example, if we understand it well, we understand more than half of normalization theory needed in practice

A Sample Instance

- Consider a sample instance of EGS:

EGS	E	G	S
	Alpha	A	1
	Beta	B	2
	Gamma	A	1
	Delta	C	1

- We have anomalies because G is "outside the key" and determines S, analogous to what he had before
- We will be more precise later
- We will only rarely use terms that we used before, such as
  - Partial dependency
  - Transitive dependency
  - Dependency into a/the key
- They are (especially the first two) essentially irrelevant/obsolete

General Approach: Decomposition

- Anomalies are removed from the design by decomposing a relation into a set of several relations
- So, here we will want to decompose EGS, and then reconstruct it, by natural-joining the new relations
- EGS has only three attributes, so the only decompositions that could be considered are decompositions into relations of two attributes
- There are three such relations
  - EG
  - GS
  - ES
- For our examples, we will consider decompositions into two relations, so possible decompositions are
  - EG and GS
  - EG and EG
  - ES and GS

Our Candidate Relations

- The three new relations are all good in the sense that there are no anomalies

EG	E	G
	Alpha	A
	Beta	B
	Gamma	A
	Delta	C

GS	G	S
	A	1
	B	2
	C	1

ES	E	S
	Alpha	1
	Beta	2
	Gamma	1
	Delta	1

Decomposing And Joining - An Acceptable Decomposition

- The chosen relations: EG and GS
- We got the original relation back

EG	E	G
	Alpha	A
	Beta	B
	Gamma	A
	Delta	C

GS	G	S
	A	1
	B	2
	C	1

EGS	E	G	S
	Alpha	A	1
	Beta	B	2
	Gamma	A	1
	Delta	C	1

Decomposing And Joining - An Acceptable Decomposition

- The chosen relations: EG and ES
- We got the original relation back

EG	E	G
	Alpha	A
	Beta	B
	Gamma	A
	Delta	C

ES	E	S
	Alpha	1
	Beta	2
	Gamma	1
	Delta	1

EGS	E	G	S
	Alpha	A	1
	Beta	B	2
	Gamma	A	1
	Delta	C	1

Decomposing And Joining - An Unacceptable Decomposition

- The chosen relations: ES and GS
- We ***did not*** get the original relation back (note: E is not even a key of the "reconstructed" EGS

ES	E	S
	Alpha	1
	Beta	2
	Gamma	1
	Delta	1

GS	G	S
	A	1
	B	2
	C	1

EGS	E	G	S
	Alpha	A	1
	Beta	B	2
	Gamma	A	1
	Delta	A	1
	Alpha	C	1
	Gamma	C	1
	Delta	C	1

Discussion

- In fact, both of the hypothetical instances below of the original relation EGS

EGS	E	G	S
	Alpha	A	1
	Beta	B	2
	Gamma	A	1
	Delta	C	1

EGS	E	G	S
	Alpha	A	1
	Beta	B	2
	Gamma	C	1
	Delta	A	1

produce exactly the same projected relations ES and GS, even with the same "duplications," with (A,1) appearing twice

- So given correct instances of ES and GS we cannot uniquely determine what EGS was, as each one of the above would be acceptable and we cannot decide between them

Decompositions vs. Reconstructions

- By examining the 3 decompositions we observe that:
  - » Some decompositions allow us to reconstruct the original relation.
  - » Some decompositions do not allow us to reconstruct the original relation
- In general, if we decompose a relation and try to reconstruct the original relation, it is not possible to do so, as many relations can give us the same "decomposed" relations.

Decompositions

- Formally we say that for a relation (schema) R, that is R with some constraints on it, some R_1, ..., R_m, form a ***decomposition*** iff (that is if and only if)
  - » Each R_i is the projection of R on some attributes

    This means that each R_i is obtained by means of a SELECT statement choosing some columns (attributes) with the empty WHERE condition (all rows are "good")
  - » Each attribute of R appears in at least one R_i

    This means that no column (attribute) is "forgotten"

Our Example

- In our case, the relation schema was R(EGS) with the constraints
  - » Each value of E has a single value of G associated with it
  - » Each value of E has a single value of S associated with it
  - » Each value of G has a single value of S associated with it
- So we did not only specify what the columns were, but also what the constraints were: together these form a schema
- And we considered three decompositions
  - » EG and GS
  - » EG and ES
  - » ES and GS

Lossless Join Decompositions

- We say that some decomposition of a relation schema R into relations R_1, ..., R_m is a ***lossless join decomposition*** iff for every instance of R (that is a specific value of R, which we continue denoting R):

  R is the natural join of R_1, ..., R_m

- We will also use the term "***valid decomposition***" for "lossless join decomposition"

Lossless Join Decompositions

- A very important property:
  - » Always: R $\subseteq$ the natural join of R_1, ..., R_m (Intuition: if you take things apart, and then try to put them together you always can rebuild what you had, but the "pieces" can perhaps be made to fit to create additional, "fake" originals)
  - » Therefore ***lossless*** means: you ***do not gain*** spurious tuples by joining, which seems the only reasonable way to reconstruct the original relation (this can be formalized more, but we do not do it)
- Note the decomposition into ES and GS caused the join to "gain" spurious tuples and therefore was not lossless

Precise Algorithmic Techniques Exist (Avoid Ad-Hoc Approaches)

- Determine whether a particular choice of relations in our database is "good"
- If the choice is not good, replace them by other relations, in general by decomposing some of the relations by means of projections
- The goal (simplified)
  - » The relations are in a "***good***" or perhaps only "***better***" form
  - » No information was lost (the decompositions were valid): ***we must not compromise this***
  - » Constraints (business rules) are well expressed and "***easy***" (or "***easier***") to maintain
- Our techniques will be based on two formal (but very real and practical) concepts:
  - » Functional dependencies
  - » Multivalued dependencies

Functional Dependencies

- Consider again the relation EGS with the semantic constraints:
  - » Each value of E has a single value of G associated with it
    We will write this as: E $\rightarrow$ G
  - » Each value of E has a single value of S associated with it
    We will write this as: E $\rightarrow$ S
  - » Each value of G has a single value of S associated with it
    We will write this as: G $\rightarrow$ S
- $\rightarrow$ formalizes the notion that the right hand side is a function of the left hand side
- The function is not computable by a formula, but is still a function

Functional Dependencies

- Generally, if X and Y are sets of attributes, then X $\rightarrow$ Y means:

  Any two tuples (rows) that are equal on (the vector of attributes) X

  are also

  equal on (the vector of attributes) Y

- Note that this ***generalizes*** the concept of a key (UNIQUE, PRIMARY KEY)
  - » We do not insist that X determines everything
  - » For instance we say that any two tuples that are equal on G are equal on S, but we ***do not*** say that any two tuples that are equal on G are "completely" equal

An Example

- Functional dependencies are properties of a schema, that is, ***all permitted*** instances
- For practice, we will examine an instance

	A	B	C	D	E	F	G	H
	a1	b1	c1	d1	e1	f1	g1	h1
	a2	b1	c1	d2	e2	f2	g1	h1
	a2	b2	c3	d3	e3	f3	g1	h2
	a1	b1	c1	d1	e1	f4	g2	h3
	a1	b2	c2	d2	e4	f5	g2	h4
	a2	b3	c3	d2	e5	f6	g2	h3

1. $A \rightarrow C$
2. $AB \rightarrow C$ Yes
3. $E \rightarrow CD$ Yes
4. $D \rightarrow B$ No
5. $F \rightarrow ABC$ Yes
6. $H \rightarrow G$ Yes
7. $H \rightarrow GE$ No

Relative Power Of Some FDs - $H \rightarrow G$ vs. $H \rightarrow GE$

- Let us look at another example first
- Consider some table talking about employees in which there are three columns:
  1. Grade
  2. Bonus
  3. Salary
- Consider now two possible FDs (functional dependencies)
  1. Grade $\rightarrow$ Bonus
  2. Grade $\rightarrow$ Bonus Salary
- FD (2) is more restrictive, fewer relations will satisfy FD (2) than satisfy FD (1)
  - » So FD (2) is stronger
  - » Every relation that satisfies FD (2), must satisfy FD (1)
  - » And we know this just because {Bonus} is a proper subset of {Bonus, Salary}

Relative Power Of Some FDs - H → G vs. H → GE

- An important note: H → GE is always at least as powerful as H → G

  that is

- If a relation satisfies H → GE it must satisfy H → G

- What we are really saying is that if GE = f(H), then of course G = f(H)
- An informal way of saying this: if being equal on H forces to be equal on GE, then of course there is equality just on G

- More generally, if X, Y, Z, are sets of attributes and Z ⊆ Y; then if X → Y is true than X → Z is true

Relative Power Of Some FDs - A → C vs. AB → C

- Let us look at another example first
- Consider some table talking about employees in which there are three columns:
  1. Grade
  2. Location
  3. Salary
- Consider now two possible FDs
  1. Grade → Salary
  2. Grade Location → Salary
- FD (2) is less restrictive, more relations will satisfy FD (2) than satisfy FD (1)
  - » So FD (1) is stronger
  - » Every relation that satisfies FD (1), must satisfy FD (2)
  - » And we know this just because {Grade} is a proper subset of {Grade, Salary}

Relative Power Of Some FDs - A → C vs. AB → C

- An important note: A → C is always at least as powerful as AB → C

  that is

- If a relation satisfies A → C it must satisfy AB → C

- What we are really saying is that if *C = f(A),* then of course *C = f(A,B)*
- An informal way of saying this: if just being equal on A forces to be equal on C, then if we ***in addition*** know that there is equality on B also, of course it is still true that there is equality on C

- More generally, if X, Y, Z, are sets of attributes and X ⊆ Y; then if X → Z is true than Y → Z is true

Trivial FDs

- An FD X → Y, where X and Y are sets of attributes is trivial

  if and only if

  Y ⊆ X

  (Such an FD gives no constraints, as it is always satisfied, which is easy to prove)

- Example
  - » Grade, Salary → Grade
    is trivial

- A trivial FD does not provide any constraints
- Every relations that contains columns Grade and Salary will satisfy this FD: Grade, Salary → Grade

Decomposition and Union of some FDs

- An FD $X \rightarrow A_1 A_2 \ldots A_m$, where A_i's are individual attributes

  is equivalent to

  the set of FDs:
  $X \rightarrow A_1$
  $X \rightarrow A_2$
  ...,
  $X \rightarrow A_m$

- Example
  FirstName LastName → Address Salary
  is equivalent to the set of the two FDs:
  Firstname LastName → Address
  Firstname LastName → Salary

Logical implications of FDs

- It will be important to us to determine if a given set of FDs ***forces*** some other FDs to be true
- Consider again the EGS relation
- Which FDs are satisfied?
  - » E → G, G → S, E → S are all true in the real world
- If the real world tells you only:
  - » E → G and G → S
- Can you deduce on your own (and is it even always true?), ***without understanding the semantics of the application***, that
  - » E → S?

Logical implications of FDs

- Yes, by simple logical argument: transitivity
  1. Take any (set of) tuples that are equal on E
  2. Then given E → G we know that they are equal on G
  3. Then given G → S we know that they are equal on S
  4. So we have shown that E → S must hold

- We say that E → G, G → S ***logically imply*** E → S and we write
- E → G, G → S |= E → S

- This means:

  If a relation satisfies E → G and G → S,

  then

  It must satisfy E → S

£

Logical implications of FDs

- If the real world tells you only:
  - » E → G and E → S,
- Can you deduce on your own, without understanding the application that
  - » G → S
- No, because of a counterexample:

EGS	E	G	S
	Alpha	A	1
	Beta	A	2

- This relation satisfies E → G and E → S, but violates G → S
- For intuitive explanation, think: G means Height and S means Weight

Conclusion/Question

- Consider a relation EGS for which the three constraints $E \rightarrow G$, $G \rightarrow S$, and $E \rightarrow S$ ***must all be obeyed***
- ***It is enough*** to make sure that the two constraints $E \rightarrow G$ and $G \rightarrow S$ are not violated
- ***It is not enough*** to make sure that the two constraints $E \rightarrow G$ and $E \rightarrow S$ are not violated
- But what to do in general, large, complex cases?

Convention

- We will use the letters P, …, Z, unless stated otherwise, to indicate sets of attributes and therefore also relations schema
- We will use the letter F, unless stated otherwise, to indicate a set of functional dependencies
- Other letters, unless stated otherwise, will indicate individual attributes
- We will use “FD” for “functional dependency” if it does not confuse with the above usage of “F”

Closures Of Sets Of Attributes

- We consider some relation schema, which is a set of attributes, R (say EGS, which could also write as R(EGS))
- A set F of FDS for this schema (say $E \rightarrow G$ and $G \rightarrow S$)
- We take some $X \subseteq R$ (Say just the attribute E)
- We ask if two tuples are equal on X, what is the largest set of attributes on which they must be equal
- We call this set ***the closure of X with respect to F*** and denote it by $\mathbf{X_F^+}$ (in our case $E_F^+ = EGS$ and $S_F^+ = S$, as is easily seen)
- If it is understood what F is, we can write just $\mathbf{X^+}$

Closures Of Sets Of Attributes

- There is a very simple algorithm to compute X^+

1. Let Y = X
2. Whenever there is an FD in F, say $V \rightarrow W$, such that
   1. $V \subseteq Y$, and
   2. $W - Y$ is not empty

   add $W - Y$ to Y
3. At termination $Y = X^+$

- The algorithm is very efficient
- Each time we look at all the FDs
  - Either we can apply at least one and make Y bigger (the biggest it can be are all attributes), or
  - We are finished

Closures Of Sets Of Attributes

- The algorithm is correct
- We do not prove it, an easy proof by induction exists
- Intuitively, we "prove" using the algorithm that whenever equality on some attribute must exist, we add that attribute to the answer

Example

- R = ABCDEGHIJK with FDs
  1. K → BG
  2. A → DE
  3. H → AI
  4. B → D
  5. J → IH
  6. C → K
  7. I → J
- Then applying FD when possible
  - » Because of 2, any two tuples that are equal on ABC must be equal on ABCDE
  - » Because of 6, any two tuples that are equal on ABCDE must be equal on ABCDEK
  - » Because of 1, any two tuples that are equal on ABCDEK, must be equal on ABCDEKG; note: we could not apply 1 earlier!
  - » We cannot apply any more FDs productively
- Therefore ABC → Z is true iff Z contains only attributes from ABCDEKG

Example

- Let R = ABCDEGHIJK
- Given FDs:
  1. K → BG
  2. A → DE
  3. H → AI
  4. B → D
  5. J → IH
  6. C → K
  7. I → J
- Then
  - ABC^+ = ABCDEGK
  - and ABC → Z if and only if Z ⊆ ABCDEGK
- Therefore, for example:
  - ABC → CK is true
  - ABC → IC is false

EGS again

- Using the algorithm, we immediately see that
  - E → G, G → S |= E → S
  - E → G, E → S does not |= G → S
- So what we did in an ad-hoc manner previously, is now a trivial algorithmic procedure!

Superkeys And Keys Of Relations

- The notion of an FD allows us to formally define superkeys and keys
- Given R, satisfying a set of FDs, a set of attributes X of R is a superkey, if and only if:
  - » $X^+ = R$.
- Given R, satisfying a set of FDs, a set of attributes X of R is a key, if and only if:
  - » $X^+ = R$.
  - » For any $Y \subseteq X$ such that $Y \neq X$, we have $Y^+ \neq R$.
- Note that if R does not satisfy any (nontrivial) FDs, then R is the only key of R.
- In our example
  - » $E \rightarrow G$, $G \rightarrow S$, $E \rightarrow S$
  - » E was the only key of EGS.

Example

- Let R = ABCDEKGHIJ
- Given FDs:
  1. $K \rightarrow BG$
  2. $A \rightarrow DE$
  3. $H \rightarrow AI$
  4. $B \rightarrow D$
  5. $J \rightarrow IH$
  6. $C \rightarrow K$
  7. $I \rightarrow J$
- Then
  - » $ABCH^+ = ABCDEGHIJK$
  - » And ABCH is a superkey for R and maybe also a key
  - » We could check whether ABCH is minimal by computing ABC^+, ABH^+, ACH^+, BCH^+

Example: Airline Scheduling

- We have a relation PFDT, where
  - **P**ILOT
  - **F**LIGHT NUMBER
  - **D**ATE
  - SCHEDULED_**T**IME_of_DEPARTURE

  and the relation satisfies the FDs (F is an attribute not set of FDs):

  - $F \rightarrow T$
  - $PDT \rightarrow F$
  - $FD \rightarrow P$
- Note that we have a problem with PFDT, similar to the one we had with EGS
- Any two tuples that are equal on F must be equal on T, and there could be many such tuples

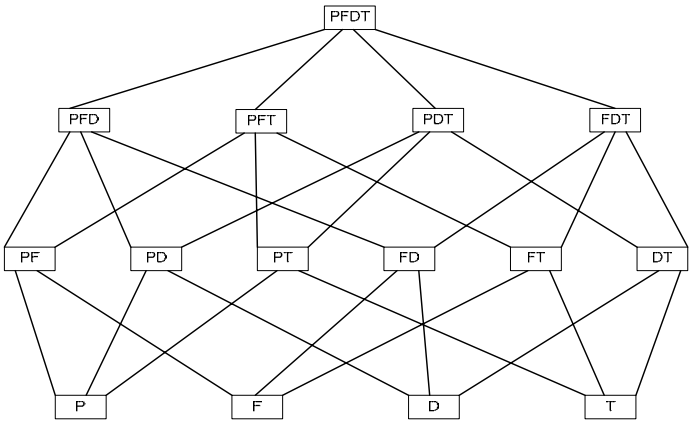
Computing Keys

- We will compute all the keys of the relation
- In general, this will be an exponential-time algorithm in the size of the problem
- But there will be useful heuristic making this problem tractable in practice
- We will introduce some heuristics here and additional ones later
- We note that if some subset of attributes is a key, then no proper superset of it can be a key as it would not be minimal and would have superfluous attributes

Lattice Of Sets Of Attributes

- There is a natural structure (technically a lattice) to all the nonempty subsets of attributes
- I will draw the lattice here, in practice this is not done
  - » Not necessary and too big
- We will look at all the non-empty subsets of attributes
- There are 15 of them: $2^4 - 1$
- The structure is clear from the drawing

Lattice Of Nonempty Subsets



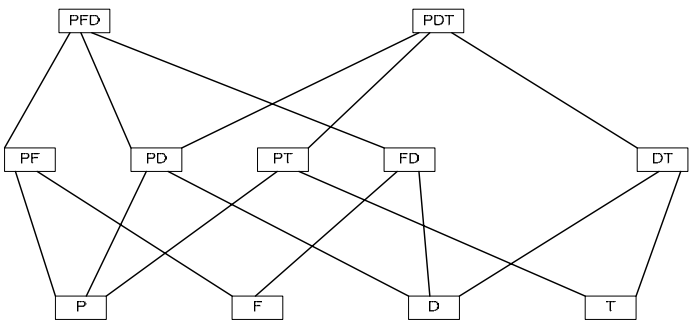
Keys Of PFDT

- The algorithm proceeds from bottom to top
- We first try all potential 1-attribute keys, by examining all 1-attribute sets of attributes
  - $P^+ = P$
  - $F^+ = FT$
  - $D^+ = D$
  - $T^+ = T$

  There are no 1-attribute keys

- Note, that the it is impossible for a key to have ***both*** F and T
  - Because if F is in a key, T will be automatically determined as it is included in the closure of F
- Therefore, we can prune our lattice

Pruned Lattice



Keys Of PFDT

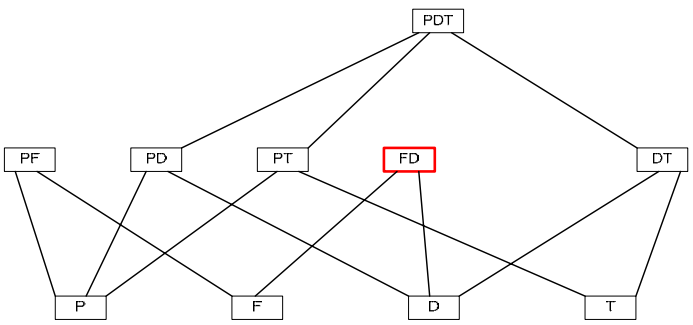
- We try all potential 2-attribute keys
  - » $PF^+ = PFT$
  - » $PD^+ = PD$
  - » $PT^+ = PT$
  - » FD+ = FDPT
  - » DT+ = DT

  There is one 2-attribute key: FD

We can mark the tree and we can prune the lattice

We can prune the lattice

Pruned Lattice

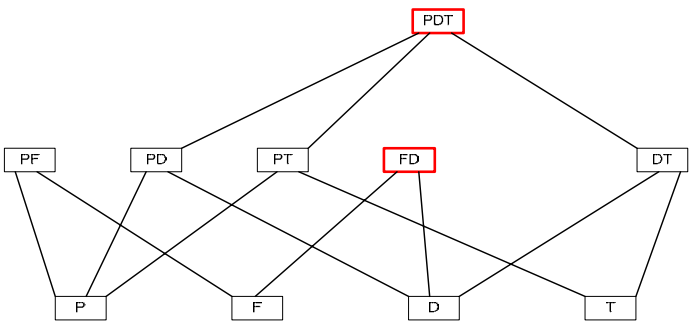


Keys Of PFDT

- We try all potential 3-attribute keys
  - » PDT^{+} = PDTF

  There is one 3-attribute key: PDT

Final Lattice - We Only Care About The Keys



Example: Airline Scheduling - The Anomaly

- In our design, we have “combined” several types of information in one relation:
  - Information about the flights in the schedule handed out to passengers, that is, which flights operate at what times of day
  - Information about assignments of pilots to flights and dates combinations.
- The functional dependency F → T “causes” redundancies
  - There are many tuples with the same value of F, and they have to have the same value of T.
- We can generalize this observation: As F did not contain a key of PFDT, there were many tuples with the same value of F, and all such tuples had to have the same value of T

The Problem In A General Setting

- In a fully general setting, we can say that we have a problem whenever a relation R satisfies an FD X → Y, and
  - X → Y is non-trivial
  - X does not contain a key
- Why? Because potentially there are many tuples with the same value in X, and they all must have the value in Y
- It is our goal to have relations for which all non-trivial FDs have the property that the left side contains a key

- Note for the future:

  X → Y is non-trivial
  if and only if
  X → A is non-trivial for some attribute A in Y

Review Of EGS

- Let us review the relation EGS
- The “new relations” (we will also refer to them as “small relations” were:
  - EG, with the key E and a non-trivial FD $E \rightarrow G$.
  - GS, with the key G and a non-trivial FD $G \rightarrow S$.
  - ES, with the key E and a non-trivial FD $E \rightarrow S$.
- Each of those relations was “good,”
  - I.e., there were no redundancies in any of them, each nontrivial FD contained a key on the left side
- However, we need to know how to test whether a decomposition was valid
- Algorithm will conclude that
  - EG and GS form a valid decomposition
  - EG and ES form a valid decomposition
  - ES and GS do not form a valid decomposition

Testing Whether A Decomposition Is Valid

- In the general case there is an algorithm that given
  - A relational schema
  - A set of FDs it satisfies
  - A proposed decomposition

  will determine whethe the proposed decomposition is valid
- We do not present it here
- We will look at a special case (which is sufficient to understand our example in full) of decomposition into two relations
- Warning: the general case is not just the extension of the algorithm we present next

Testing Whether A Decomposition Is Valid

- We use V and W to denote the set of attributes of the relations V and W respectively; thus $V \cup W = R$ means that no attribute is missing, and $V \cap W$ is the set of attributes common to V and W.
- Fact: if we decompose R into V and W where $V \cup W = R$, then the decomposition is valid if and only if
  - » $V \cap W \rightarrow V$ is true

    OR
  - » $V \cap W \rightarrow W$ is true
- Note: this **does not** generalize trivially to a decomposition into three or more relations
  - » There is a simple algorithm for this, which we will not need and therefore not cover

Testing Whether A Decomposition Is Valid

- Intuitive reason: say we have a relation R=ABCDE and we decompose it into two relations
  - » V = ABC.
  - » W = BCDE.
- If $BC \rightarrow BCDE$, then for any tuple (a,b,c) of ABC there is a unique tuple (b,c,d,e) of BCDE that can be "glued" to it
- Let us again review the three decompositions of EGS

The First Decomposition Of EGS

- EGS was decomposed into EG and GS.

  $EG \cap GS = G$.

  We need to check whether $G \rightarrow EG$ or $G \rightarrow GS$
  - » $G \rightarrow EG$ is false
  - » $G \rightarrow GS$ is true

- Therefore, the decomposition was valid

The Second Decomposition Of EGS

- EGS was decomposed into EG and ES

  $EG \cap ES = E$.

  We need to check whether $E \rightarrow EG$ or $E \rightarrow GS$
  - » $E \rightarrow EG$ is true
  - » $E \rightarrow ES$ is true

- Therefore, the decomposition was valid

The Third Decomposition Of EGS

- EGS was decomposed into ES and GS

  $ES \cap GS = S$.

  We need to check whether $S \rightarrow ES$ or $S \rightarrow GS$
  - » $S \rightarrow ES$ is false
  - » $S \rightarrow GS$ is false.

- Therefore, the decomposition was not valid

The Boyce-Codd Normal Form

- We summarize the desirable properties of a relation by defining the Boyce-Codd normal form (BCNF).
- A relation R is in BCNF if an only if whenever $X \rightarrow Y$ is true and nontrivial then X contains a key of R
  - » This is easy to test, just compute X^+ and check whether you get all of R
- To formulate the next claim concisely, assume there are no duplicates (it does not matter, just easier to phrase)
- ***A relation in BCNF does not have any redundancies (of the type we have been discussing)***
- Let $X \rightarrow Y$ be true, then either
  - » $Y \subseteq X$, and we are not saying anything meaningful, or
  - » There is (at most) only one tuple (perhaps with duplicates) with this value of X, so the constraint is stored in only this tuple

Decomposition Of EGS Into Relations In BCNF

- For reasons discussed earlier, we like relations to be in BCNF
- We return to EGS and its decompositions

- EGS was not in BCNF because
  - » G → S was not trivial and true
- EG was in BCNF because
  - » The only key was E and the only nontrivial FD was E → G
- ES was in BCNF because
  - » The only key was E and the only nontrivial FD was E → S
- GS was in BCNF because
  - » The only key was G and the only nontrivial FD was G → S

Decomposition Of EGS Into Relations In BCNF

- We considered three decompositions
  - » EG and GS was valid
  - » EG and ES was valid
  - » ES and GS was not valid
- How to choose between the valid decompositions?
- There are additional issues that need to be considered to select good designs
  - » They will let us decide which of the two valid, and therefore possible, decompositions is better

Preservation Of Dependencies

- Assume that we have a relation R satisfying the set F of FDs (there is a subtlety we gloss over, this is the set of satisfied FDs not only the ones given to us), and we decompose it into relations
  - » R_1 satisfying set F_1 of FDs
  - » R_2 satisfying set F_2 of FDs
- In general
  - » F_1 is the subset of F that is expressible using the attributes of R_1 only
  - » F_2 is the subset of F that is expressible using the attributes of R2 only
- Of course, $F_1 \cup F_2 \subseteq F$

Preservation Of Dependencies

- $F_1 \cup F2 \subseteq F$
- We can ask: **$F_1 \cup F_2$ |= F**

- That is, is $F_1 \cup F_2$ as powerful as the bigger set F?
- If yes, we say that ***dependencies are preserved***
- For, this of course enough to check: $F_1 \cup F_2$ |= F − ($F_1 \cup F_2$)
- In other words, does set $F_1 \cup F_2$ logically implies everything that is in F outside of $F_1 \cup F_2$

Preservation Of Dependencies

- We consider EGS again together with the two valid decompositions we found earlier:
- EGS was decomposed into EG and GS.
  - » EG satisfied E → G, and additional "boring" FDs, such as EG → G, G → G, …
  - » GS satisfied G → S, and additional "boring" FDs
- So we need to check whether
  - » E → G, G → S |= E → G, G → S, E → S
    that is whether
  - » E → G, G → S |= E → S
- This is true, as we have seen earlier, and therefore dependencies are preserved.

Preservation Of Dependencies

- EGS was decomposed into EG and ES
  - » EG satisfied E → G
  - » ES satisfied E → S
- We need to check whether
  - » E → G, E → S |= E → G, G → S, E → S
    that is whether
  - » E → G, E → S |= G → S
- This is false, as we have seen earlier, and therefore dependencies are not preserved

Preservation Of Dependencies

- If FDs are not preserved, some inconsistent updates cannot be determined as such by means of local tests only
- What are local tests?
- User likes R, F
- To avoid redundancies, etc., we decide
  - » To decompose R into R_1 (satisfying F_1) and R_2 (satisfying F_2)
  - » Store R_1 and R_2 as two separate relations
- User wants to insert r into R and expects us to check for consistency (F must be maintained)
  - » We will insert r_1 into R_1 and r_2 into R_2
- Is it enough to only check that F_1 and F_2 are satisfied by the two relations individually to assure that F is "globally" satisfied?

Updating EGS

- We return to our example of EGS, and consider a sample instance and a sample update, insertion in this case

EGS	E	G	S
	Alpha	A	1

- Insert (Epsilon,A, 2)
  - » Is this a permitted update, as far as the real world is concerned? What would happen if we did it?

EGS	E	G	S
	Alpha	A	1
	Epsilon	A	2

  - » This relation violates the FD G→ S it is supposed to satisfy. Thus we recognize this as an invalid update and reject it.
- However, instead of EGS we could have two valid decompositions of EGS
- What would happen if we used them to store the data?

Updating EG and GS

EG	E	G
	Alpha	A

GS	G	S
	A	1

- After the update we get:

EG	E	G
	Alpha	A
	Epsilon	A

GS	G	S
	A	1
	A	2

- We must either allow all the inserts or none
- We test the relations
  - » EG satisfies its only nontrivial FD: E → G
  - » GS does not satisfy its only nontrivial FD: G → S
- We ***are able*** to recognize the update as incorrect, because dependencies were preserved
- It is ***enough*** to rely on "local tests" only

Updating EG and ES

EG	E	G
	Alpha	A

ES	E	S
	Alpha	1

- After the update we get:

EG	E	G
	Alpha	A
	Epsilon	A

ES	E	S
	Alpha	1
	Epsilon	2

- We must either allow all the inserts or none
- We test the relations
  - » EG satisfies its only nontrivial FD: E → G
  - » ES satisfies its only nontrivial FD: E → S
- We *are* ***not able*** to recognize the update as incorrect, because dependencies were not preserved
- It is ***not enough*** to rely on "local tests" only

The Boyce-Codd Normal Form

- A relation R is in BCNF

  if and only if

  whenever $X \rightarrow Y$ is true and nontrivial then X contains a key of R
- But, of course, always, for any relation R

  if X contains a key than

  $X \rightarrow Y$ (of course, X, Y are subsets of R)
- For a relation in BCNF all the functional dependencies satisfied by the attributes of the relation are fully specified by listing all the keys of the relation

$X \rightarrow Y$ is true ***if and only if*** X contains a key

Benefits of BCNF

- So using SQL DDL by specifying the keys (primary and unique) we automatically specify all FDs satisfied by each of the relations individually, if our database consists of relations in BCNF

- Reminder: Easy to test if X contains a key (as we have seen before) just check whether $X^+ = R$
- It is easy to check whether a relation is in BCNF (even without knowing keys, just check the condition for each given FD), that is for each given $X \rightarrow Y$ check whether $X^+ = R$

Benefits of BCNF

- But what about FDs which constrain attributes not within a single relation of the database, that is involve attributes of more than one relation?
  - » If we decompose EGS into ES and GS, we need to maintain the "non-local" FD: G → S
- If FDs are not preserved, larger relations may need to be reconstructed in order to check for consistency of the database (such as after updates)
- The decomposition of EGS into EG and GS was wonderful:
  - » It was a valid decomposition (lossless join decomposition)
  - » EG and GS were in BCNF
  - » Functional dependencies were preserved
- Can we always satisfy all three conditions by appropriate decompositions?

Finding Keys

- We will discuss additional heuristics for finding keys, in addition to those we have already discussed in the context of the PDFT example
- Consider an example of a relation with attributes ABCDE and functional dependencies
  - » A → D
  - » B → C
  - » C → B
- We can classify the attributes into 4 classes:
  1. Appearing on both sides of FDs; here B, C
  2. Appearing on left sides only; here A
  3. Appearing on right sides only; here D
  4. Not appearing in FDs; here E

Finding Keys

- Facts:
  - » Attributes of class 2 and 4 must appear in every key
  - » Attributes of class 3 do not appear in any key
  - » Attributes of class 1 may or may not appear in keys
- An algorithm for finding keys relies on these facts
  - » Unfortunately, in the worst case, exponential in the number of attributes
- Start with the attributes in classes 2 and 4, add as needed (going bottom up) attributes in class 1, and ignore attributes in class 3
- But pay attention to previous heuristics in the PDFT example
- One could formulate a precise algorithm, which we will not do here as we understand all its pieces and not following automatically actually builds up intuition

Finding Keys

- Start with AE
- Compute $AE^+ = AED$
- B and C are missing, we will try adding each of them
- $AEB^+ = AEBDC$; AEB is a key
- $AEC^+ = AECDB$; AEC is a key
- These are the only keys of the relation

Some Goals May Not Be Achievable

- Given a relation R and a set of FDs, ***it is not always possible*** to decompose R into relations so that:
  - The decomposition is valid
  - The new relations are in ***BCNF***
  - Functional dependencies are preserved
- So what can we do in the general case?
- We have to compromise
- We will define a normal form, 3NF, which is not as good as BCNF, as it allows certain redundancies
- Given a relation R and a set of FDs, ***it is always possible*** to decompose R into relations so that:
  - The decomposition is valid
  - The new relations are in ***3NF***
  - Functional dependencies are preserved

3NF

- A relation R is in 3NF if and only if whenever $X \rightarrow Y$ is true
  - It is trivial, or
  - X contains a key, or
  - Every attribute of Y is in some key (different attributes could be in different keys)
- Could also phrase it as follows
- A relation R is in 3NF if and only if whenever $X \rightarrow A$ is true
  - It is trivial, or
  - X contains a key, or
  - A is in some key
- Compare with BCNF
- A relation R is in BCNF if and only if whenever $X \rightarrow Y$ is true
  - It is trivial, or
  - X contains a key
- 3NF is more permissive than BCNF

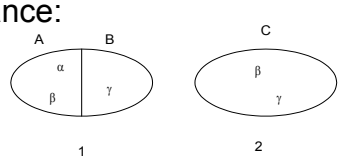
Testing For 3NF Condition

- Given a set of FDs F to we can check if the relation is in 3NF for each FD we check whether one of the 3 conditions is satisfied
- But we need to know what the keys are for full testing (to check the 3rd condition.

- For BCNF we do not need to do that (testing whether left hand side contains a key does not require knowing keys, as we have seen before)

The SCT Example - Sometimes We May Prefer 3NF to BCNF

- The attributes
  - STUDENT
  - COURSE
  - TUTOR (Teaching assistant with whom students "sign up")
- The functional dependencies
  - SC $\rightarrow$ T
  - T $\rightarrow$ C
- The semantics of the example is (written to fit on slide):
  - Students take courses; Tutors are assigned to courses; A tutor can be assigned to only one course; A student can only have one tutor in any particular course
- Instance:



	S	C	T
	Alpha	1	A
	Beta	1	A
	Beta	2	C
	Gamma	1	B
	Gamma	2	C

The SCT Example - Sometimes We May Prefer 3NF to BCNF

- Note that we have redundancies, for example the fact that tutor A is assigned to course 1 is written twice
- It is easy to see that the relation has two keys:
  - SC
  - ST
- As the T $\rightarrow$ C is nontrivial, and T does not contain a key, the relation is *not* in BCNF
- We could produce a valid decomposition of SCT into relations in BCNF
- However, it can be shown, that such a decomposition would not preserve FDs
  - Intuitively the reason is that the decomposed relations would only contain 2 attributes, and therefore only T $\rightarrow$ C could be satisfied, from which SC $\rightarrow$ T is not logically implied.
  - The above is easy to do, just tedious, so we do not do it here
- Therefore, local tests would not be sufficient

Towards A Minimal Cover

- This form will be based on trying to store a “concise” representation of FDs
- We will try to find a “small” number of “small” relation schemas that are sufficient to maintain the FDs
- The core of this will be to find “concise” description of FDs
  - Example: in ESG, E $\rightarrow$ S was not needed
- We will compute a ***minimal cover*** for a set of FDs
- Sometimes the term “***canonical***” is used instead of “***minimal***”
- The basic idea, simplification of a set of FDs by
  - Combining FDs when possible
  - Getting rid of unnecessary attributes
- We will start with examples to introduce the concepts and the tools
- Deviating from our convention, we will use H to denote a set of attributes

Union Rule: Combining Right Hand Sides (RHSs)

- F = { AB $\rightarrow$ C, AB $\rightarrow$ D }

  is equivalent to

  H = { AB $\rightarrow$ CD }

- We have discussed this rule before
- Intuitively clear
- Formally we need to prove 2 things
  - » F |= H is true; we do this (as we know) by showing that AB_F^+ contains CD; easy exercise
  - » H |= F is true; we do this (as we know) by showing that AB_H^+ contains C and AB_H^+ contains D; easy exercise
- Note: you ***cannot*** combine LHSs based on equality of RHS and get an equivalent set of FDS
  - » F = {A $\rightarrow$ C, B $\rightarrow$ C} ***is stronger than*** H = {AB $\rightarrow$ C}

Union Rule: Combining Right Hand Sides (RHSs)

- Stated formally:

  F = { X $\rightarrow$ Y, X $\rightarrow$ Z } ***is as powerful as*** H = { X $\rightarrow$ YZ }

- Easy proof, we omit

Relative Power Of FDs: Left Hand Side (LHS)

- F = { AB $\rightarrow$ C }

  is weaker than

  H = { A $\rightarrow$ C }

- We have discussed this rule before when we started talking about FDs
- Intuitively clear: in F, if we assume more (equality on both A and B) to conclude something (equality on C) than our FD is applicable in fewer case (does not work if we have equality is true on B's but not on C'S) and therefore F is weaker than H
- Formally we need to prove two things
  - » F |= H is false; we do this (as we know) by showing that A_F^+ does not contain C; easy exercise
  - » H |= F is true; we do this (as we know) by showing that AB_H^+ contains C; easy exercise

Relative Power Of FDs: Left Hand Side (LHS)

- Stated formally:

  F = { XB $\rightarrow$ Y } ***is weaker than*** H = { X $\rightarrow$ Y }, (if B $\notin$ X)

- Easy proof, we omit

- Can state more generally, replacing B by a set of attributes, but we do not need this

Relative Power Of FDs: Right Hand Side (RHS)

- F = { A → BC }

  is stronger than

  H = { A → B }

- Intuitively clear: in H, we deduce less from the same assumption, equality on A's
- Formally we need to prove two things
  - » F |= H is true; we do this (as we know) by showing that A_F^+ contains B; easy exercise
  - » H |= F is false; we do this (as we know) by showing that A_H^+ does not contain C; easy exercise

Relative Power Of FDs: Right Hand Side (RHS)

- Stated formally:

  F = { X → YC } ***is stronger than*** H = { X → Y },
  (if C ∉ Y
  and C ∉ X)

- Easy proof, we omit

- Can state more generally, replacing C by a set of attributes, but we do not need this

Simplifying Sets Of FDs

- At various stages of the algorithm we will have
  - » An “old” set of FDs
  - » A “new” set of FDs
- The two sets will not vary by “very much”
- We will indicate the parts that do not change by . . .
- Of course, as we are dealing with sets, the order of the FDs in the set does not matter

Simplifying Set Of FDs By Using The Union Rule

- X, Y, Z are sets of attributes
- Let F be:

  …
  $X \rightarrow Y$
  $X \rightarrow Z$

- Then, F is equivalent to the following H:

  …
  $X \rightarrow YZ$

Simplify Set Of FDS By Simplifying LHS

- Le X, Y are sets of attributes and B a single attribute not in X
- Let F be:

  …
  $XB \rightarrow Y$

- Let H be:

  …
  $X \rightarrow Y$

- Then if F |= $X \rightarrow Y$ holds, then we can replace F by H without changing the "power" of F
- We do this by showing that X_F^+ contains Y
  - » H could only be stronger, but we are proving it is not actually stronger, but equivalent

Simplify Set Of FDS - By Simplifying LHS

- H can only be stronger than F, as we have replaced a weaker FD by a stronger FD
- But if we F |= H holds, this "local" change does not change the overall power
- Example below
- Replace
  - » $AB \rightarrow C$
  - » $A \rightarrow B$

  by
  - » $A \rightarrow C$
  - » $A \rightarrow B$

Simplify Set Of FDS - By Simplifying RHS

- Le X, Y are sets of attributes and C a single attribute not in Y
- Let F be:

  …
  $X \rightarrow YC$
  …

- Let H be:

  …
  $X \rightarrow Y$
  …

- Then if H |= $X \rightarrow YC$ holds, then we can replace F by H without changing the "power" of F
- We do this by showing that X_H^+ contains YC
  - » H could only be weaker, but we are proving it is not actually weaker, but equivalent

Simplify Set Of FDS By Simplifying RHS

- H can only be weaker than F, as we have replaced a stronger FD by a weaker FD
- But if we H |= F holds, this "local" change does not change the overall power
- Example below
- Replace
  - » $A \rightarrow BC$
  - » $B \rightarrow C$

  by
  - » $A \rightarrow B$
  - » $B \rightarrow C$

Minimal Cover

- Given a set of FDs F, find a set of FDs F_m, that is (in a sense we formally define later) minimal
- Algorithm:

1. Start with F
2. Remove all trivial functional dependencies
3. Repeatedly apply (in whatever order you like), until no changes are possible
   - » Union Simplification (it is better to do it as soon as possible, whenever possible)
   - » RHS Simplification
   - » LHS Simplification
4. What you get is a a minimal cover

- We proceed through a largish example to exercise all possibilities

The EmToPrHoSkLoRo Relation

- The relation deals with employees who use tools on projects and work a certain number of hours per week
- An employee may work in various locations and has a variety of skills
- All employees having a certain skill and working in a certain location meet in a specified room once a week
- The attributes of the relation are:
  - » Em: Employee
  - » To: Tool
  - » Pr: Project
  - » Ho: Hours per week
  - » Sk: Skill
  - » Lo: Location
  - » Ro: Room for meeting

The FDs Of The Relation

- The relation deals with employees who use tools on projects and work a certain number of hours per week
- An employee may work in various locations and has a variety of skills
- All employees having a certain skill and working in a certain location meet in a specified room once a week
- The relation satisfies the following FDs:
  - Each employee uses a single tool: Em → To
  - Each employee works on a single project: Em → Pr
  - Each tool can be used on a single project only: To → Pr
  - An employee uses each tool for the same number of hours each week: EmTo → Ho
  - All the employees working in a location having a certain skill always work in the same room (in that location): SkLo → Ro
  - Each room is in one location only: Ro → Lo

Sample Instance

	Em	To	Pr	Ho	Sk	Lo	Ro
	Mary	Pen	Research	20	Clerk	Boston	101
	Mary	Pen	Research	20	Writer	Boston	102
	Mary	Pen	Research	20	Writer	Buffalo	103
	Fang	Pen	Research	30	Clerk	New York	104
	Fang	Pen	Research	30	Editor	New York	105
	Fang	Pen	Research	30	Economist	New York	106
	Fang	Pen	Research	30	Economist	Buffalo	107
	Lakshmi	Oracle	Database	40	Analyst	Boston	101
	Lakshmi	Oracle	Database	40	Analyst	Buffalo	108
	Lakshmi	Oracle	Database	40	Clerk	Buffalo	107
	Lakshmi	Oracle	Database	40	Clerk	Boston	101
	Lakshmi	Oracle	Database	40	Clerk	Albany	109
	Lakshmi	Oracle	Database	40	Clerk	Trenton	110
	Lakshmi	Oracle	Database	40	Economist	Buffalo	107

Our FDs

1. Em → To
2. Em → Pr
3. To → Pr
4. EmTo → Ho
5. SkLo → Ro
6. Ro → Lo

Run The Algorithm

- Using the union rule, we combine RHS of 1 and 2, getting:
  1. Em → ToPr
  2. To → Pr
  3. EmTo → Ho
  4. SkLo → Ro
  5. Ro → Lo

Run The Algorithm

- No RHS can be combined, so we check whether there are any redundant attributes.
- We start with FD 1, where we attempt to remove an attribute from RHS
  - » We check whether we can remove To. This is possible if we can derive Em → To using

    Em → Pr

    To → Pr

    EmTo → Ho

    SkLo → Ro

    Ro → Lo

    Computing the closure of Em using the above FDs gives us only EmPr, so the attribute To must be kept.

Run The Algorithm

- » We check whether we can remove Pr. This is possible if we can derive Em → Pr using

  Em → To

  To → Pr

  EmTo → Ho

  SkLo → Ro

  Ro → Lo

  Computing the closure of Em using the above FDs gives us EmToPrHo, so the attribute Pr is redundant

Run The Algorithm

- We now have
  1. Em → To
  2. To → Pr
  3. EmTo → Ho
  4. SkLo → Ro
  5. Ro → Lo
- No RHS can be combined, so we continue attempting to remove redundant attributes. The next one is FD 3, where we attempt to remove an attribute from LHS
  - » We check if Em can be removed. This is possible if we can derive To → Ho using *all* the FDs. Computing the closure of To using the FDs gives ToPr, and therefore To cannot be removed
  - » We check if To can be removed. This is possible if we can derive Em → Ho using *all* the FDs. Computing the closure of Em using the FDs gives EmToPrHo, and therefore To can be removed

Run The Algorithm

- We now have
  1. Em → To
  2. To → Pr
  3. Em → Ho
  4. SkLo → Ro
  5. Ro → Lo
- We can now combine RHS of 1 and 3 and get
  1. Em → ToHo
  2. To → Pr
  3. SkLo → Ro
  4. Ro → Lo

Run The Algorithm

- We now attempt to remove an attribute from the LHS of 3, and an attribute from RHS of 1, but neither is possible
- Therefore we are done
- We have computed a minimal cover for the original set of FDs

Minimal (Or Canonical) Cover

- A set of FDs, F_m, is a minimal cover for a set of FD F, if and only if

1. F_m is minimal, that is
   1. No two FDs in it can be combined using the union rule
   2. No attribute can be removed from a RHS of any FD in F_m without changing the power of F_m
   3. No attribute can be removed from a LHS of any FD in F_m without changing the power of F_m
2. F_m is equivalent in power to F

- Note that there could be more than one minimal cover for F, as we have not specified the order of applying the simplification operations

How About EGS

- Applying to algorithm to EGS with
  1. $E \rightarrow G$
  2. $G \rightarrow S$
  3. $E \rightarrow S$
- Using the union rule, we combine 1 and 3 and get
  1. $E \rightarrow GS$
  2. $G \rightarrow S$
- Simplifying RHS of 1 (this is the only attribute we can remove), we get
  1. $E \rightarrow G$
  2. $G \rightarrow S$
- We automatically got the two "important" FDs!

An Algorithm For "An Almost" - 3NF Lossless-Join Decomposition

- Input: relation schema R and a set of FDs F
- Output: almost-decomposition of R into R1, R2, …, Rn, each in 3NF
- Algorithm

1. Produce F_m, a minimal cover for F
2. For each $X \rightarrow Y$ in F_m create a new relation schema XY
3. For every new relation schema that is a subset (including being equal) of another new relation schema (that is the set of attributes is a subset of attributes of another schema or the two sets of attributes are equal) remove this relation schema (the "smaller" one or one of the equal ones); but if the two are equal, need to keep one of them
4. The set of the remaining relation schemas is an almost-decomposition

Back To Our Example

- For our EmToPrHoSkLoRo example, we previously computed the following minimal cover:
  1. Em → ToHo
  2. To → Pr
  3. SkLo → Ro
  4. Ro → Lo

Creating Relations

- Create a relation for each functional dependency
- We obtain the relations:
  1. EmToHo
  2. ToPr
  3. SkLoRo
  4. LoRo

Removing Redundant Relations

- LoRo is a subset of SkLoRo, so we remove it
- We obtain the relations:
  1. EmToHo
  2. ToPr
  3. SkLoRo

How About EGS

- The minimal cover was
  1. $E \rightarrow G$
  2. $G \rightarrow S$
- Therefore the relations obtained were:
  1. EG
  2. GS
- And this is exactly the decomposition we thought was best!

Assuring Storage Of A Global Key

- If no relation contains a key of the original relation, add a relation whose attributes form such a key
- Why do we need to do this?
  - » Because otherwise we may not have a decomposition
  - » Because otherwise the decomposition may not be lossless

Why It Is Necessary To Store A Global Key - Example

- Consider the relation LnFn:
  - » Ln: Last Name
  - » Fn: First Name
- There are no FDs
- The relation has only one key:
  - » LnFn
- Our algorithm (without the key included) produces no relations
- A condition for a decomposition: Each attribute of R has to appear in at least one Ri
- So we did not have a decomposition
- But if we add the relation consisting of the attributes of the key
  - » We get LnFn (this is fine, because the original relations had no problems and was in a good form, actually in BCNF, which is always true when there are no (nontrivial) FDs)

Why It Is Necessary To Store A Global Key - Example

- Consider the relation: LnFnVaSa:
  - Ln: Last Name
  - Fn: First Name
  - Va: Vacation days per year
  - Sa: Salary
- The functional dependencies are:
  - Ln → Va
  - Fn → Sa
- The relation has only one key
  - LnFn
- The relation is not in 3NF
  - Ln → Va: Ln does not contain a key and Va is not in any key
  - Fn → Sa: Fn does not contain a key and Sa is not in any key

Why It Is Necessary To Store A Global Key - Example

- Our algorithm (without the key being included) will produce the decomposition
  1. LnVa
  2. FnSa
- This is not a lossless-join decomposition
  - In fact we do not know who the employees are (what are the valid pairs of LnFn)
- So we decompose
  1. LnVa
  2. FnSa
  3. LnFn

Assuring Storage Of A Global Key

- If no relation contains a key of the original relation, add a relation whose attributes form such a key
- It is easy to test if a "new" relation contains a key of the original relation
- Compute the closure of the relation with respect to all FDs (either original or minimal cover, it's the same) and see if you get all the attributes of the original relation
- If not, you need to find some key of the original relation
- How do we find a key? We go "bottom up," but there are helpful heuristics we have learned

Back To EmToPrHoSkLoRo

- The FDs were (or could have worked with the minimal cover, does not matter):
  - Em $\rightarrow$ To
  - Em $\rightarrow$ Pr
  - To $\rightarrow$ Pr
  - EmTo $\rightarrow$ Ho
  - SkLo $\rightarrow$ Ro
  - Ro $\rightarrow$ Lo
- Our new relations and we check if any of them contains a key of EmToPrHoSkLoRo:
  1. EmToHo — $EmToHo^+$ = EmToHoPr, does not contain a key
  2. ToPr — $ToPr^+$ = ToPr, does not contain a key
  3. SkLoRo — $SkLoRo^+$ = SkLoRo, does not contain a key

Finding Keys

- So we need to find a key
- Let us list the FDs again (or could have worked with the minimal cover, does not matter):
  - Em → To
  - Em → Pr
  - To → Pr
  - EmTo → Ho
  - SkLo → Ro
  - Ro → Lo
- As discussed before, we can classify the attributes into 4 classes:
  1. Appearing on both sides of FDs; here To, Lo, Ro.
  2. Appearing on left sides only; here Em, Sk.
  3. Appearing on right sides only; here Pr, Ho.
  4. Not appearing in FDs; here none.

Finding Keys

- Facts:
  - Attributes of class 2 and 4 must appear in every key
  - Attributes of class 3 do not appear in any key
  - Attributes of class 1 may or may not appear in keys
- An algorithm for finding keys relies on these facts
  - Unfortunately, in the worst case, exponential in the number of attributes
- Start with the attributes in classes 2 and 4, add as needed (going bottom up) attributes in class 1, and ignore attributes in class 3

Finding Keys

- In our example, therefore, every key must contain EmSk
- To see, which attributes, if any have to be added, we compute which attributes are determined by EmSk
- We obtain
  - » $EmSk^+$ = EmToPrHoSk
- Therefore Lo and Ro are missing
- It is easy to see that the relation has two keys
  - » EmSkLo
  - » EmSkRo

Finding Keys

- Although not required strictly by the algorithm (which does not mind decomposing a relation in 3NF into relations in 3NF) we can check if the original relation was in 3NF
- We conclude that the original relation is not in 3NF, as for instance, To $\rightarrow$ Pr violates the 3NF conditions:
  - » This FD is nontrivial
  - » To does not contain a key
  - » Pr is not in any key

Example Continued

- None of the relations contains either EmSkLo or EmSkRo.
- Therefore, one more relation needs to be added. We have 2 choices for the final decomposition
  1. EmToHo
  2. ToPr
  3. SkLoRo
  4. EmSkLo

  or
  1. EmToHo
  2. ToPr
  3. SkLoRo
  4. EmSkRo
- We have completed our process and got a decomposition with the properties we needed

Applying the algorithm to EGS

- Applying the algorithm to EGS, we get our desired decomposition:
  - » EG
  - » GS
- And the “new” relations are in BCNF too, though we guaranteed only 3NF!

Returning to Our Example

- We pick the decomposition
  1. EmToHo
  2. ToPr
  3. SkLoRo
  4. EmSkLo
- We have the minimal set of FDs of the simplest form (before any combinations)
  1. Em $\rightarrow$ ToHo
  2. To $\rightarrow$ Pr
  3. SkLo $\rightarrow$ Ro
  4. Ro $\rightarrow$ Lo

Returning to Our Example

- Everything can be described as follows:
- The relations, their keys, and FDs that need to be explicitly mentioned are:
  1. EmToHo — key: Em
  2. ToPr — key: To
  3. SkLoRo — key: SkLo, key SkRo, and functional dependency Ro $\rightarrow$ Lo
  4. EmSkLo — key: EmSkLo
- In general, when you decompose as we did, a relation may have several keys and satisfy several FDs that do not follow from simply knowing keys
- In the example above there was one relation that had such an FD, which made is automatically not a BCNF relation (but by our construction a 3NF relation)

Back to SQL DDL

- How are we going to express in SQL what we have learned?
- We need to express:
  - keys
  - functional dependencies
- Expressing keys is very easy, we use the PRIMARY KEY and UNIQUE keywords
- Expressing functional dependencies is possible also by means of a CHECK condition
  - What we need to say for the relation SkLoRo is that each tuple satisfies the following condition

    There are no tuples in the relation with the same value of Ro and different values of Lo

Back to SQL DDL

- CREATE TABLE SkLoRo
  (Sk …,
  Lo …,
  Ro…,
  UNIQUE (Sk,Ro),
  PRIMARY KEY (Sk,Lo),
  CHECK (NOT EXISTS SELECT *
  FROM SkLoRo AS copy
  WHERE (SkLoRo.Ro = copy.Ro
  AND NOT SkLoRo.Lo = copy.Lo));
- But this is generally not supported by actual relational database systems
- Even assertions are frequently not supported
- Can use triggers to support this
- Whenever there is an insert or update, check that FDs holds, or reject these actions

Algorithms for Relational Database Schema Design (1)

- **Algorithm 16.4: Relational Synthesis into 3NF with Dependency Preservation (Relational Synthesis Algorithm)**
  - » **Input: A universal relation R and a set of functional dependencies F on the attributes of R.**

1. Find a minimal cover G for F (use Algorithm 16.2);
2. For each left-hand-side X of a functional dependency that appears in G,
   create a relation schema in D with attributes {X ∪ {A1} ∪ {A2} ... ∪ {Ak}},
   where X → A1, X → A2, ..., X → Ak are the only dependencies in G with X as left-hand-side (X is the key of this relation) ;
3. Place any remaining attributes (that have not been placed in any relation) in a single relation schema to ensure the attribute preservation property.
   - » **Claim 3: Every relation schema created by Algorithm 16.4 is in 3NF.**

Algorithms for Relational Database Schema Design (2)

- **Algorithm 16.5: Relational Decomposition into BCNF with Lossless (non-additive) join property**
  - » **Input: A universal relation R and a set of functional dependencies F on the attributes of R.**

1. Set D := {R};
2. While there is a relation schema Q in D that is not in BCNF
   do {
   choose a relation schema Q in D that is not in BCNF;
   find a functional dependency X → Y in Q that violates BCNF;
   replace Q in D by two relation schemas (Q - Y) and (X ∪ Y);
   };

Assumption: No null values are allowed for the join attributes.

Algorithms for Relational Database Schema Design (3)

- **Algorithm 16.6 Relational Synthesis into 3NF with Dependency Preservation and Lossless (Non-Additive) Join Property**
  - **Input: A universal relation R and a set of functional dependencies F on the attributes of R.**

1. Find a minimal cover G for F (Use Algorithm 16.2).
2. For each left-hand-side X of a functional dependency that appears in G,

   create a relation schema in D with attributes {X ∪ {A1} ∪ {A2} ... ∪ {Ak}},

   where X → A1, X → A2, ..., X –>Ak are the only dependencies in G with X as left-hand-side (X is the key of this relation).
3. If none of the relation schemas in D contains a key of R, then create one more relation schema in D that contains attributes that form a key of R. *(Use Algorithm 16.4a to find the key of R)*

Algorithms for Relational Database Schema Design (4)

- **Algorithm 16.2a Finding a Key K for R Given a set F of Functional Dependencies**
  - **Input: A universal relation R and a set of functional dependencies F on the attributes of R.**

1. Set K := R;
2. For each attribute A in K {
   Compute (K - A)+ with respect to F;
   If (K - A)+ contains all the attributes in R,
   then set K := K - {A};
   }

Algorithms for Relational Database Schema Design (6)

Figure 16.3
The dangling tuple problem.
(a) The relation EMPLOYEE_1 (includes all attributes of EMPLOYEE from Figure 16.2(a) except Dnum).

(a) EMPLOYEE_1

Ename	Ssn	Bdate	Address
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX
Narayan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX
Berger, Anders C.	999775555	1965-04-26	6530 Braes, Bellaire, TX
Benitez, Carlos M.	888665555	1963-01-09	7654 Beech, Houston, TX

Algorithms for Relational Database Schema Design (7)

- **Discussion of Normalization Algorithms:**
- Problems:
  - The database designer must first specify *all* the relevant functional dependencies among the database attributes.
  - These algorithms are *not deterministic* in general.
  - It is not always possible to find a decomposition into relation schemas that preserves dependencies and allows each relation schema in the decomposition to be in BCNF (instead of 3NF as in Algorithm 16.6).

Multivalued Dependencies - Putting Previous Material In Context

- To have a smaller example, we will look at this separately not by extending our previous example
- In the application, we store information about Courses (C), Teachers (T), and Books (B)
- Each course has a set of books that have to be assigned during the course
- Each course has a set of teachers that are qualified to teach the course
- Each teacher, when teaching a course, has to use the set of the books that has to be assigned in the course

An Example Relation - Putting Previous Material In Context

	C	T	B
	DB	Zvi	Oracle
	DB	Zvi	Linux
	DB	Dennis	Oracle
	DB	Dennis	Linux
	OS	Dennis	Windows
	OS	Dennis	Linux
	OS	Jinyang	Windows
	OS	Jinyang	Linux

- This instance (and therefore the relation in general) does not satisfy any functional dependencies
  - » CT does not functionally determine B
  - » CB does not functionally determine T
  - » TB does not functionally determent C

Redundancies - Putting Previous Material In Context

	C	T	B
	DB	Zvi	Oracle
	DB	Zvi	Linux
	DB	Dennis	
	DB	Dennis	
	OS	Dennis	Windows
	OS	Dennis	Linux
	OS	Jinyang	
	OS	Jinyang	

	C	T	B
	DB	Zvi	Oracle
	DB		Linux
	DB	Dennis	Oracle
	DB		Linux
	OS	Dennis	Windows
	OS		Linux
	OS	Jinyang	Windows
	OS		Linux

- There are obvious redundancies
- In both cases, we know exactly how to fill the missing data if it was erased
- We decompose to get rid of anomalies

Decomposition - Putting Previous Material In Context

	C	T	B
	DB	Zvi	Oracle
	DB	Zvi	Linux
	DB	Dennis	Oracle
	DB	Dennis	Linux
	OS	Dennis	Windows
	OS	Dennis	Linux
	OS	Jinyang	Windows
	OS	Jinyang	Linux

	C	T
	DB	Zvi
	DB	Dennis
	OS	Dennis
	OS	Jinyang

	C	B
	DB	Oracle
	DB	Linux
	OS	Windows
	OS	Linux

Multivalued Dependencies & 4NF - Putting Previous Material In Context

- We had the following situation
- For each value of C there was
  - » A set of values of T
  - » A set of values of B
- Such that, every T of C had to appear with every B of C

  This is stated here rather loosely, but it is clear what it means
- The notation for this is: $C \rightarrow\!\!\rightarrow T \mid B$
- The relations CT and CB were in ***Fourth Normal Form (4NF)***

Fourth Normal Form (4NF) – (Optional Slide)

- A relation R is in 4NF if and only if whenever $X \rightarrow\!\!\rightarrow Y \mid Z$ is true
  - » It is trivial, or
  - » X contains a key
- Trivial means that either Y or Z are empty
- We will not discuss it any further, but just mention for reference that multivalued dependencies generalize "regular" dependencies
  - ◆ In fact, if $X \rightarrow Y$ then $X \rightarrow\!\!\rightarrow Y \mid Z$, where Z is just "everything not in X and not in Y, that is $Z = R - (X \cup Y) = R - X - Y$

Formal Definition Of MVDs

- In general, let X, Y be subsets of R (all attributes), and then let Z = R − (X ∪ Y) = R − X − Y
- Then X →→ Y (or could write X →→ Y | Z) if and only if
  Whenever for some values x, y1, y2, z1, z2 there exist two tuples t1 and t2 of R such that
  - $\pi_X[t1] = x$, $\pi_X[t2] = x$, $\pi_Y[t1] = y1$, $\pi_Y[t2] = y2$, $\pi_Z[t1] = z1$, $\pi_Z[t2] = z2$
    then there exists a tuple t3 in R such that
  - $\pi_X[t3] = x$, $\pi_Y[t3] = y1$, $\pi_Z[t3] = z2$
- For a "general" example, let R = ABCD, X = AB, Y = BC. Then X →→ Y means that whenever we have some tuples abc_1d_1 and abc_2d_2, then we also have tuples abc_1d_2 and abc_2d_1
- Note that using our previous notation:
  - X →→ Y is the same as X →→ Y | (R − X − Y)
- To make intuitive sense, it is best to write MVDs so that the three sets X, Y, and R − X − Y are all disjoint

More About MVDS

- An MVD X →→ Y is ***trivial*** if and only if:
  - Y is a subset of X
    or
  - XY = R
- A trivial MVD always holds

  Proof:
  - Y is a subset of X . To avoid cumbersome notation assume that X = AB, Y = A, and R = ABC. Then the statement X →→ Y simply means that if we have tuples abc_1 and abc_2 then we have tuples abc_1 and abc_2.
  - XY = R. To avoid cumbersome notation assume that X = A and Y = B. Then the statement X →→ Y simply means that if we have tuples ab_1 and ab_2 then we have tuples ab_1 and ab_2.

More About MVDS

- If $X \rightarrow Y$, then $X \rightarrow\rightarrow Y$
  Proof:
  - Assume for simplicity that X = A, Y = B, and R = ABC. Let tuples ab_1c_1 and ab_2c_2 be in R. To show that $X \rightarrow\rightarrow Y$, we need to show that tuples ab_2c_1 and ab_1c_2 are in R. But because of $X \rightarrow Y$ we have that $b_1 = b_2$, say b. So our job reduces to showing that if tuples abc_1 and abc_2 are in R then tuples abc_2 and abc_1 are in R.
- If $X \rightarrow Y$ is a trivial FD, then $X \rightarrow\rightarrow Y$ is trivial MVD
  Proof
  - It follows from the definitions as the proof reduces to the statement that if Y is a subset of X then Y is a subset of X

4th Normal Form

- A relation is in ***4NF*** if and only if
  - Whenever $X \rightarrow\rightarrow Y$ is not trivial, then X contains a key
- Key is defined as before, by means of FDs only
- Note that this means also that relation is in ***4NF*** if and only if
  - Whenever $X \rightarrow\rightarrow Y$ is not trivial, then X contains a key
  - Every $X \rightarrow\rightarrow Y$ is also $X \rightarrow Y$ (because X is a key)
- If a relation is in 4NF, it is also in BCNF
- It is always possible to decompose a relation into relations in 4NF such that the decomposition is a lossless join decomposition
- This is a stronger statement then the possibility of a lossless join decompositions into relations in BCNF
- But what we probably want is some combination of removal of MVDs and 3NF

Projection Of FDs Revisited

- You are given R, which satisfies some set of FDs F
- You decompose R into relations R1, R2, …, Rm, so that
  - The decomposition is lossless join
  - Relations R1, R2, …, Rm are in some nice form
  - Dependencies are preserved
- "Dependencies are preserved" simply means that the union of FDs satisfied by R1, R2, …, Rm is "as powerful" as F, that is, it is as powerful as F^+
- So, in order to perform the "checks") you need to find F1, F2, …, Fm which are "small" subsets of F^+ such that
  - Ri satisfies Fi, for all i
  - Union of all Fi's is as powerful as F^+
- In general it is not the case that Fi's will be just subsets of F
- The set of such Fi's is called a projection of F on R1, R2, …, Rm (even though it is really a projection of F^+

Projection Of FDs Revisited

- Consider the example of R = ABC with the set of FD F = { AB $\rightarrow$ C, A $\rightarrow$ B}
- We know what to do
  - Execute our algorithm
  - We will get AB satisfying A $\rightarrow$ B and AC satisfying A $\rightarrow$ C
- Somebody else, who just does what seems OK, decomposes also
  - AB and AC
- But what FDs are satisfied there?
- If one only looks at the original F and asks which of these FDs are satisfied where, one thinks
  - We will get AB satisfying A $\rightarrow$ B and AC satisfying nothing (because neither AB nor AC have enough attributes to store AB $\rightarrow$ C)

Projection Of FDs Revisited

- So in summary, if you do your own decomposition of R satisfying F into R1, R2, … , Rm,
- You must show that the decomposition is lossless (there is a general algorithm, we did not cover)
- You should find all FDs that are satisfied by each Ri (or at least a subset that is equivalent to all of them: best minimal cover)
  - » There is an algorithm, which we did not cover, which tests whether for such decomposition dependencies are preserved
- Luckily everything is done for us if we use our algorithm and we do not have to check/test anything

Multivalued Dependencies and Fourth Normal Form (1)

Definition:

- A **multivalued dependency** (**MVD**) X —>> Y specified on relation schema R, where X and Y are both subsets of R, specifies the following constraint on any relation state r of R: If two tuples t_1 and t_2 exist in r such that $t_1[X] = t_2[X]$, then two tuples t_3 and t_4 should also exist in r with the following properties, where we use Z to denote (R 2 (X ∪ Y)):
  - » $t_3[X] = t_4[X] = t_1[X] = t_2[X]$.
  - » $t_3[Y] = t_1[Y]$ and $t_4[Y] = t_2[Y]$.
  - » $t_3[Z] = t_2[Z]$ and $t_4[Z] = t_1[Z]$.
- An MVD X —>> Y in R is called a **trivial MVD** if (a) Y is a subset of X, or (b) X ∪ Y = R.

Multivalued Dependencies and Fourth Normal Form (2)

(a) The EMP relation with two MVDs: ENAME —>> PNAME and ENAME —>> DNAME.
(b) Decomposing the EMP relation into two 4NF relations EMP_PROJECTS and EMP_DEPENDENTS.

(a) **EMP**

ENAME	PNAME	DNAME
Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John

(b) **EMP_PROJECTS**

ENAME	PNAME
Smith	X
Smith	Y

EMP_DEPENDENTS

ENAME	DNAME
Smith	John
Smith	Anna

Multivalued Dependencies and Fourth Normal Form (3)

(c) The relation SUPPLY with no MVDs is in 4NF but not in 5NF if it has the JD(R1, R2, R3). (d) Decomposing the relation SUPPLY into the 5NF relations R1, R2, and R3.

(c) **SUPPLY**

SNAME	PARTNAME	PROJNAME
Smith	Bolt	ProjX
Smith	Nut	ProjY
Adamsky	Bolt	ProjY
Walton	Nut	ProjZ
Adamsky	Nail	ProjX
Adamsky	Bolt	ProjX
Smith	Bolt	ProjY

(d) **R1**

SNAME	PARTNAME
Smith	Bolt
Smith	Nut
Adamsky	Bolt
Walton	Nut
Adamsky	Nail

R2

SNAME	PROJNAME
Smith	ProjX
Smith	ProjY
Adamsky	ProjY
Walton	ProjZ
Adamsky	ProjX

R3

PARTNAME	PROJNAME
Bolt	ProjX
Nut	ProjY
Bolt	ProjY
Nut	ProjZ
Nail	ProjX

Multivalued Dependencies and Fourth Normal Form (4)

- **Inference Rules for Functional and Multivalued Dependencies**:
  - IR1 (**reflexive rule for FDs**): If $X \supseteq Y$, then $X \rightarrow Y$.
  - IR2 (**augmentation rule for FDs**): $\{X \rightarrow Y\} \models XZ \rightarrow YZ$.
  - IR3 (**transitive rule for FDs**): $\{X \rightarrow Y, Y \rightarrow Z\} \models X \rightarrow Z$.
  - IR4 (**complementation rule for MVDs**): $\{X \twoheadrightarrow Y\} \models X \twoheadrightarrow (R - (X \cup Y))\}$.
  - IR5 (**augmentation rule for MVDs**): If $X \twoheadrightarrow Y$ and $W \supseteq Z$ then $WX \twoheadrightarrow YZ$.
  - IR6 (**transitive rule for MVDs**): $\{X \twoheadrightarrow Y, Y \twoheadrightarrow Z\} \models X \twoheadrightarrow (Z 2 Y)$.
  - IR7 (**replication rule for FD to MVD**): $\{X \rightarrow Y\} \models X \twoheadrightarrow Y$.
  - IR8 (**coalescence rule for FDs and MVDs**): If $X \twoheadrightarrow Y$ and there exists W with the properties that
    - (a) $W \cap Y$ is empty, (b) $W \rightarrow Z$, and (c) $Y \supseteq Z$, then $X \rightarrow Z$.

Multivalued Dependencies and Fourth Normal Form (5)

Definition:

- A relation schema R is in **4NF** with respect to a set of dependencies F (that includes functional dependencies and multivalued dependencies) if, for every *nontrivial* multivalued dependency $X \twoheadrightarrow Y$ in F^+, X is a superkey for R.
  - Note: F^+ is the (complete) set of all dependencies (functional or multivalued) that will hold in every relation state r of R that satisfies F. It is also called the **closure** of F.

Multivalued Dependencies and Fourth Normal Form (6)

Lossless (Non-additive) Join Decompositioninto 4NF Relations:

- **PROPERTY LJ1'**
  - The relation schemas R_1 and R_2 form a lossless (non-additive) join decomposition of R with respect to a set F of functional *and* multivalued dependencies if and only if
    - $(R_1 \cap R_2) \twoheadrightarrow (R_1 - R_2)$
  - or by symmetry, if and only if
    - $(R_1 \cap R_2) \twoheadrightarrow (R_2 - R_1))$.

Multivalued Dependencies and Fourth Normal Form (7)

Algorithm 16.7: Relational decomposition into 4NF relations with non-additive join property

- **Input:** A universal relation R and a set of functional and multivalued dependencies F.

1. Set D := { R };
2. While there is a relation schema Q in D that is not in 4NF do {
   choose a relation schema Q in D that is not in 4NF;
   find a nontrivial MVD $X \twoheadrightarrow Y$ in Q that violates 4NF;
   replace Q in D by two relation schemas $(Q - Y)$ and $(X \cup Y)$;
   };

Summary Of Some Normal Forms

- Let R be relation schema
- We are told that it satisfies $X \rightarrow Y$, where X and Y are sets of attributes
- Using the union rule "in reverse" we can decompose this FD into several FDs of the form $X \rightarrow A$, where A is a single attribute
- So will just talk about $X \rightarrow A$
- We will list what is permitted for three normal forms
- We will include an obsolete normal form, which is still sometimes considered by practitioners: second normal form (2NF)
- It is obsolete, because we can always find a desired decomposition in relations in 3NF, which is better than 2NF

Which FDs Are Allowed For Some Normal Forms

BCNF	3NF	2NF
The FD is trivial	The FD is trivial	The FD is trivial
X contains a key	X contains a key	X contains a key
	A is in some key	A is in some key
		X not a proper subset of some key

Which FDs Are Allowed For Some Normal Forms

- Example: EGS with
  - » $E \rightarrow G$
  - » $E \rightarrow S$
  - » $G \rightarrow S$
- The only key of EGS: E
- EGS is not in 3NF, because
  - » In $G \rightarrow S$, G does not contain a key and S is not in any key
- EGS is in 2NF, because
  - » In $E \rightarrow G$, E contains a key
  - » In $E \rightarrow S$, E contains a key
  - » In $G \rightarrow S$, G is not a proper subset of a key

Which FDs Are Allowed For Some Normal Forms

- Example: ABC with
  - » $A \rightarrow B$
- The only key of ABC: AC
- ABC is not in 2NF, because
  - » In $A \rightarrow B$, A does not contain a key, B is not in any key, and A is a proper subset of a key

What If You Are Given A Decomposition?

- You are given a relation R with a set of dependencies it satisfies
- You are given a possible decomposition of R into R_1, R_2, …, R_m
- You can check
  - » Is the decomposition lossless: must have
  - » Are the new relations in some normal forms: nice to have
  - » Are dependencies preserved: nice to have
- Algorithms exist for all of these, which you could learn, if needed and wanted

Join Dependencies and Fifth Normal Form (1)

Definition:

- A **join dependency** (**JD**), denoted by JD(R_1, R_2, ..., R_n), specified on relation schema R, specifies a constraint on the states r of R.
  - » The constraint states that every legal state r of R should have a non-additive join decomposition into R_1, R_2, ..., R_n; that is, for every such r we have
  - » $* (\pi_{R1}(r), \pi_{R2}(r), ..., \pi_{Rn}(r)) = r$

  ***Note**: an MVD is a special case of a JD where n = 2.*
- A join dependency JD(R_1, R_2, ..., R_n), specified on relation schema R, is a **trivial JD** if one of the relation schemas R_i in JD(R_1, R_2, ..., R_n) is equal to R.

Join Dependencies and Fifth Normal Form (2)

Definition:

- A relation schema R is in **fifth normal form** (**5NF**) (or **Project-Join Normal Form** (**PJNF**)) with respect to a set F of functional, multivalued, and join dependencies if,
  - » for every nontrivial join dependency JD(R_1, R_2, ..., R_n) in F^+ (that is, implied by F),
    - every R_i is a superkey of R.

Inclusion Dependencies (1)

Definition:

- An **inclusion dependency** $R.X < S.Y$ between two sets of attributes—X of relation schema R, and Y of relation schema S—specifies the constraint that, at any specific time when r is a relation state of R and s a relation state of S, we must have

$$\pi_X(r(R)) \supseteq \pi_Y(s(S))$$

- **Note**:
  - » The ? (subset) relationship does not necessarily have to be a proper subset.
  - » The sets of attributes on which the inclusion dependency is specified—X of R and Y of S—must have the same number of attributes.
  - » In addition, the domains for each pair of corresponding attributes should be compatible.

Inclusion Dependencies (2)

- **Objective of Inclusion Dependencies:**
  - » To formalize two types of interrelational constraints which cannot be expressed using F.D.s or MVDs:
    - Referential integrity constraints
    - Class/subclass relationships
- **Inclusion dependency inference rules**
  - » **IDIR1** (**reflexivity**): $R.X < R.X$.
  - » **IDIR2** (**attribute correspondence**): If $R.X < S.Y$
    - where $X = \{A_1, A_2, ..., A_n\}$ and $Y = \{B_1, B_2, ..., B_n\}$ and A_i *Corresponds-to* B_i, then $R.A_i < S.B_i$
    - for $1 \leq i \leq n$.
  - » **IDIR3** (**transitivity**): If $R.X < S.Y$ and $S.Y < T.Z$, then $R.X < T.Z$.

Other Dependencies and Normal Forms (1)

Template Dependencies:

- Template dependencies provide a technique for representing constraints in relations that typically have no easy and formal definitions.
- The idea is to specify a template—or example—that defines each constraint or dependency.
- There are two types of templates:
  - » tuple-generating templates
  - » constraint-generating templates.
- A template consists of a number of **hypothesis tuples** that are meant to show an example of the tuples that may appear in one or more relations. The other part of the template is the **template conclusion.**

Other Dependencies and Normal Forms (2)

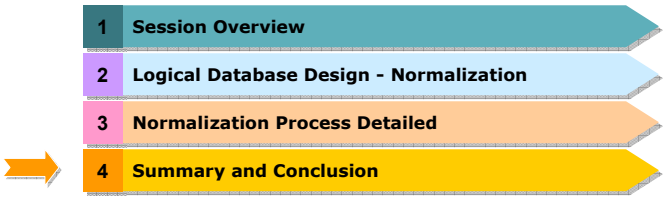
Domain-Key Normal Form (DKNF):

- **Definition:**
  - » A relation schema is said to be in **DKNF** if all constraints and dependencies that should hold on the valid relation states can be enforced simply by enforcing the domain constraints and key constraints on the relation.
- The **idea** is to specify (theoretically, at least) the "*ultimate normal form*" that takes into account all possible types of dependencies and constraints. .
- For a relation in DKNF, it becomes very straightforward to enforce all database constraints by simply checking that each attribute value in a tuple is of the appropriate domain and that every key constraint is enforced.
- The practical utility of DKNF is limited

Summary

- Designing a Set of Relations
- Properties of Relational Decompositions
- Algorithms for Relational Database Schema
- Multivalued Dependencies and Fourth Normal Form
- Join Dependencies and Fifth Normal Form
- Inclusion Dependencies
- Other Dependencies and Normal Forms

Agenda



Summary

- Logical Database Design - Normalization
- Normalization Process Detailed
- Summary & Conclusion

Assignments & Readings

- Readings
  - Slides and Handouts posted on the course web site
  - Textbook: Chapters 15 and 16
- Assignment #5
  - Textbook exercises: TBA
  - See Database Project (Part I) specifications and support material posted under handouts and demos on the course Web site.
- Project Framework Setup (ongoing)

Next Session: Physical Database Design

- Physical design of the database using various file organization and indexing techniques for efficient query processing

Any Questions?