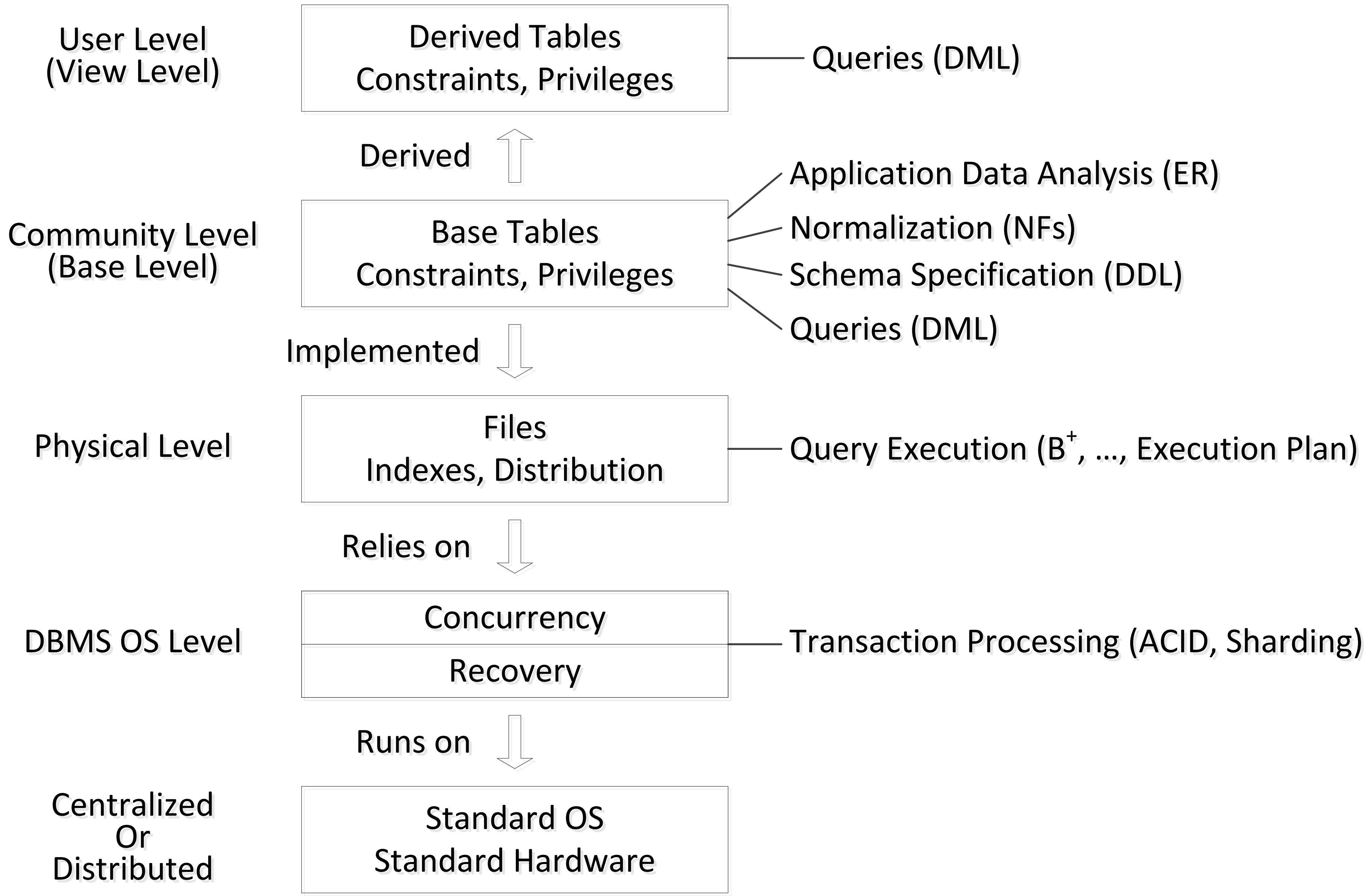
Unit 7
Logical Database Design
With Normalization

Normalization in Context



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
  - **Storing the information needed**
  - **Enforcing constraints**
  - **Avoiding anomalies, such as redundancies**
- If there are problems 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, that is a semantic constraint, (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

- "**If** Grade = 2 **then** 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 one table

R	Name	<u>SSN</u>	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	<u>SSN</u>	DOB	Grade
	A	121	2367	2
	A	132	3678	3
	B	101	3498	4
	C	106	2987	2

T	<u>Grade</u>	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 “If Grade = 3 then Salary = 70”, even after the last employee with this Grade leaves
  - Store “If Grade = 2 then 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	<u>SSN</u>	DOB	Grade
	A	121	2367	2
	A	132	3678	3
	B	101	3498	4
	C	106	2987	2

T	<u>Grade</u>	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	<u>SSN</u>	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

- 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 them back together to reconstruct the original R

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 å (this symbol appears only in special mathematical fonts, so we may use ∞ in these notes instead)

- So we have: R = S åT

- 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 times
- 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** (so possibly 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

U	X	V	Y	W

- We replaced this one table by two tables

U	X	V	W

X	Y

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

Several Passes On The Material

- Practitioners do it (mostly) differently than the way researchers/academics like to do

- **Pass 1**: I will focus on how IT practitioners do it or at least like to talk about it

  Ad-hoc treatment, but good for building intuition and having common language and concepts with IT people

- **Pass 2**: I focus on how computer scientists like to do or at least can do it this way if they want to

  Good for actually using algorithms that guarantee correct results

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 think about fixing them
- It is important (in this context) to understand what the various normal forms are even the ones that are obsolete/ unimportant (your maybe asked about this during a job interview!)

Normal Forms

- A normal form applies to a table/relation schema, not to the whole database schema
- So the question is individually asked about a table: is it of some specific desirable 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 obsolete algorithms
  - Third Normal Form (**3NF**); very important; a final form
  - Boyce-Codd Normal Form (**BCNF**); very important in theory (but less used in practice and we will understand why); a final form
  - Fourth Normal Form (**4NF**); a final form but generally what is good about it beyond previous normal forms is easily obtained without formal treatment
- There are additional ones, which are more esoteric, and which we will not cover

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 New 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 and getting a good grade
- 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: in other words this is a relation, nothing new, defined for historical reasons
  - There are some variations to this definition, but we use this one

Historical Reason For First Normal Form

- Originally, there were only file systems
- Such systems, frequently consisted of ***variable-length records***
- Transition to tables, which have fixed-length tuples, one needs to restrict files to have ***fixed-length records***
- This was phrased as normalization

- Note: we are not discussing how tables are actually stored, which is invisible to SQL
- It may actually be advantageous to store relations using files with variable-length records

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 (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: $S \rightarrow B$
  2. A teacher has to charge the same fee from every student he teaches : $T \rightarrow F$
  3. A teacher can teach only one course (perhaps at different times, different offerings, etc, but never another course) : $T \rightarrow C$
  4. A student can take a course from one teacher only: $SC \rightarrow T$

Possible Primary Key

- Our rules: $S \rightarrow B$, $T \rightarrow F$, $T \rightarrow C$, $SC \rightarrow T$
- ST is a 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	<u>S</u>	B	C	<u>T</u>	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: $S \rightarrow B$, $T \rightarrow F$, $T \rightarrow C$, $SC \rightarrow T$
- SC is a 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 rule 2)
- A part of SC is not sufficient
  1. From S, we cannot get T, C, or F
  2. From C, we cannot get B, S, 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
- ST can serve as primary key, in effect:
  - ST → SBCTF
  - This sometimes just written as ST → BCF, since always ST → ST (columns determine themselves)
- 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 possible primary key (ST)

R	<u>S</u>	B	<u>C</u>	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

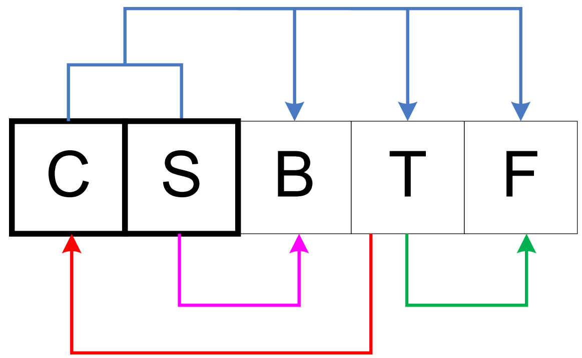
R	S	B	C	T	F
	Fang	1990	DB	Zvi	1
	John	1980	OS	Allan	2
	Mary	1990	PL	Vijay	1
	Fang	1990	OS	Allan	2
	John	1980	PL	Marsha	4
	Mary	1990	PL	Vijay	1

- The two tables store the same information and both obey all the business rules, note that (Mary,PL) fixes the rest

Review

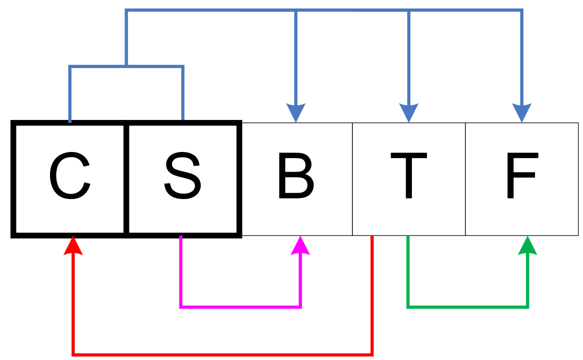
- To just review this
- Because S → B, given a specific S, either it does not appear in the table, or wherever it appears it has the same value of B
  - John has 1980, everywhere it appears
  - Lilian does not have B anywhere (in fact she does not appear in the relation)
- Because SC → BTF (and therefore 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

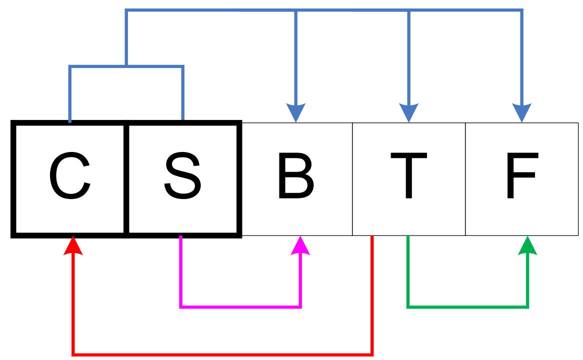
- Implicitly we think as if the table had only one key (we are not paying attention to keys that are not primary)
- In fact, as we have seen, there is one more key, we just do not think about it (at least for now)

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

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!)
- http://en.wikipedia.org/wiki/Database_normalization if you want to know more, but the coverage of the material we need to know is too skimpy and not sufficiently intuitive

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, marked here with “?”

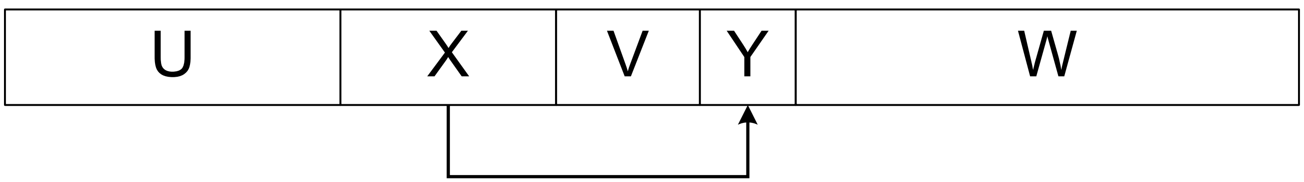
Our Business Rules Have A Clean Format

- 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
- Later we will learn how to “clean” business rules without understanding the application

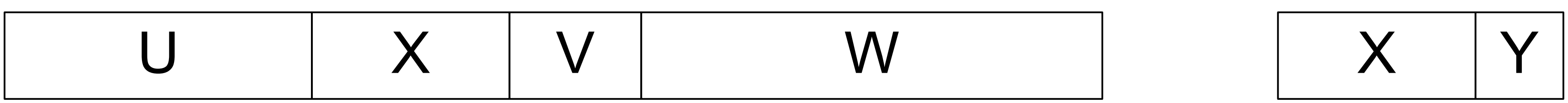
- Computer Scientists do not assume that they understand the application or that the business rules are clean, so they use algorithmic techniques to clean up business rules
- And Computer Scientists prefer to use algorithms and rely less on intuition

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,<u>SSN</u>,DOB,Grade,Salary), where Grade → Salary; in the drawing “X” stands for “Grade” and “Y” stands for “Salary”



by two tables S(Name,<u>SSN</u>,DOB,Grade) and T(<u>Grade</u>,Salary)



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

A Procedure For Removing Anomalies

- While replacing

U	X	V	Y	W

by two tables

U	X	V	W

X	Y

- 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

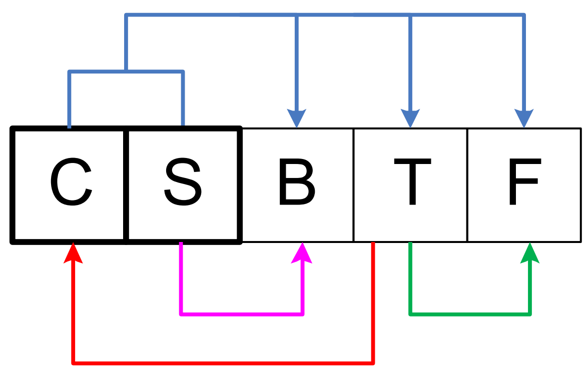
S	Name	DOB	Grade	Salary
	A	2367	2	80
	A	3678	3	70
	B	3498	4	70
	C	2987	2	80

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

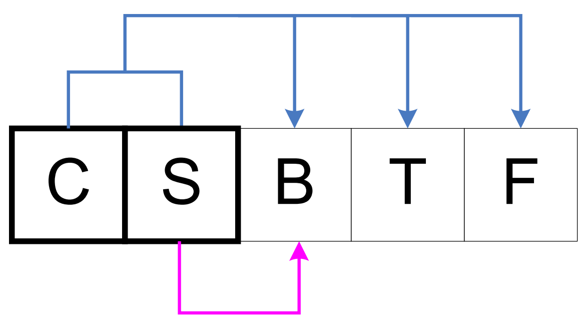
Our Example Again

	<u>S</u>	B	<u>C</u>	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

	<u>S</u>	B	<u>C</u>	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

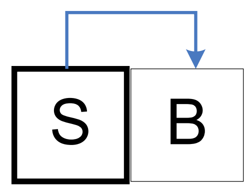
	<u>S</u>	B	<u>C</u>	T	F
	Fang	1990	DB	Zvi	1
	John	1980	OS	Allan	2
	Mary	1990	PL	Vijay	1
	Fang	1990	OS	Allan	2
	John	1980	PL	Marsha	4

	<u>S</u>	B
	Fang	1990
	John	1980
	Mary	1990
	Fang	1990
	John	1980

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

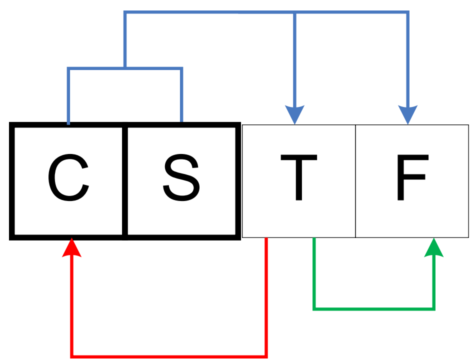
No Anomalies

	<u>**S**</u>	**B**
	Fang	1990
	John	1980
	Mary	1990
	Fang	1990
	John	1980



Some Anomalies

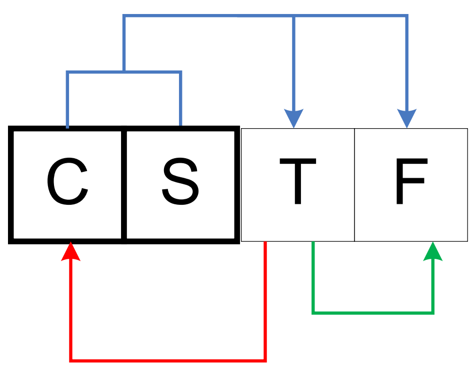
	<u>S</u>	<u>C</u>	T	F
	Fang	DB	Zvi	1
	John	OS	Allan	2
	Mary	PL	Vijay	1
	Fang	OS	Allan	2
	John	PL	Marsha	4

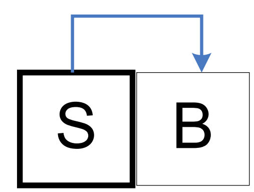


Decomposition So Far

	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





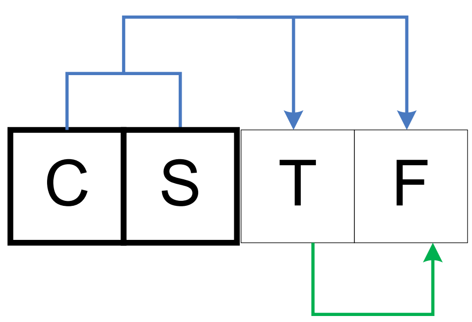
Second Normal Form: 1NF And No Partial Dependencies

- 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 using the natural join, 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

Transitive Dependency: T → F

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



Decomposition

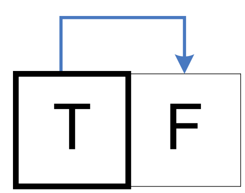
	S	C	T	F
	Fang	DB	Zvi	1
	John	OS	Allan	2
	Mary	PL	Vijay	1
	Fang	OS	Allan	2
	John	PL	Marsha	4

	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

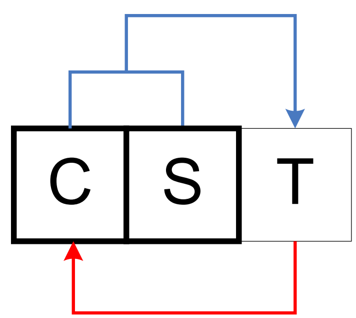
No Anomalies

	<u>T</u>	F
	Zvi	1
	Allan	2
	Vijay	1
	Allan	2
	Marsha	4



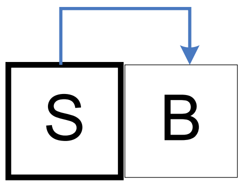
Anomalies

	<u>S</u>	<u>C</u>	T
	Fang	DB	Zvi
	John	OS	Allan
	Mary	PL	Vijay
	Fang	OS	Allan
	John	PL	Marsha

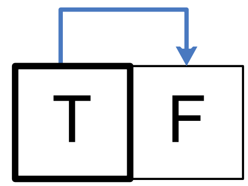


Decomposition So Far

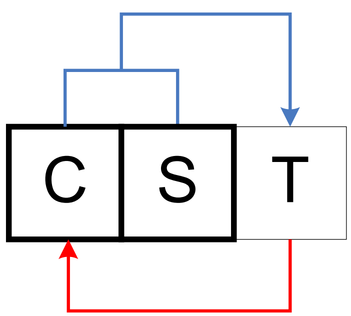
	<u>S</u>	B
	Fang	1990
	John	1980
	Mary	1990



	<u>T</u>	F
	Zvi	1
	Allan	2
	Vijay	1
	Marsha	4



	<u>S</u>	<u>C</u>	T
	Fang	DB	Zvi
	John	OS	Allan
	Mary	PL	Vijay
	Fang	OS	Allan
	John	PL	Marsha



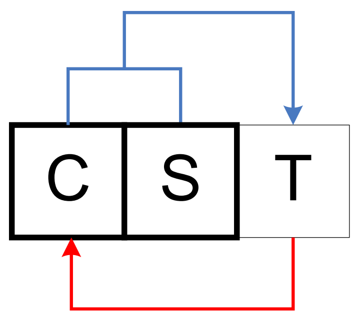
Third Normal Form: 2NF And No Transitive Dependencies

- 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

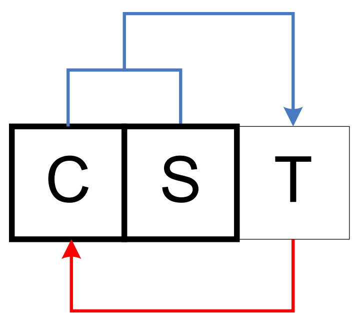
	<u>S</u>	<u>C</u>	T
	Fang	DB	Zvi
	John	OS	Allan
	Mary	PL	Vijay
	Fang	OS	Allan
	John	PL	Marsha



- 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: TS

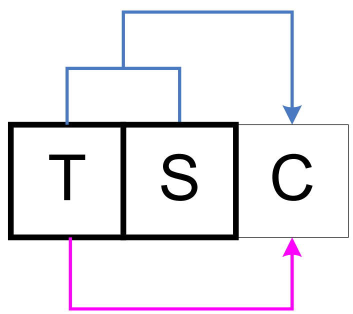
	S	C	T
	Fang	DB	Zvi
	John	OS	Allan
	Mary	PL	Vijay
	Fang	OS	Allan
	John	PL	Marsha



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

Anomaly

	<u>S</u>	C	<u>T</u>
	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

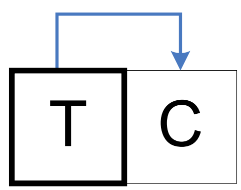
No Anomalies

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

T	S

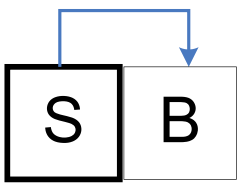
No Anomalies

	C	<u>T</u>
	DB	Zvi
	OS	Allan
	PL	Vijay
	OS	Allan
	PL	Marsha

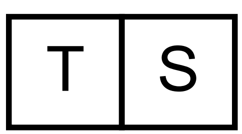


Our Decomposition

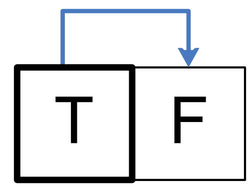
	S	B
	Fang	1990
	John	1980
	Mary	1990



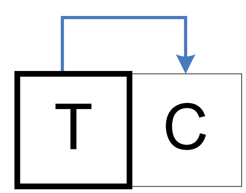
	S	T
	Fang	Zvi
	John	Allan
	Mary	Vijay
	Fang	Allan
	John	Marsha



	T	F
	Zvi	1
	Allan	2
	Vijay	1
	Marsha	4



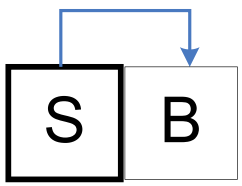
	C	T
	DB	Zvi
	OS	Allan
	PL	Vijay
	PL	Marsha



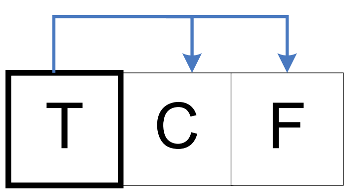
Our Decomposition

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

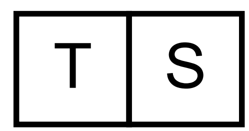
	S	B
	Fang	1990
	John	1980
	Mary	1990



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



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



Boyce-Codd Normal Form:
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 as no bad dependencies are possible
- 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

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

- **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
	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 I 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 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: $C \twoheadrightarrow T \mid B$

- The tables CT and CB where in ***Fourth Normal Form (4NF)***
- We do not define formally here

Now: To Algorithmic Techniques

- So far, our treatment was not algorithmic and we just looked at an interesting case exploring within the context of that case 3 issues

1. Avoiding (some) redundancies by converting tables to 3NF (and sometimes getting BCNF)
2. Preserving dependencies/constraints by making sure that dependencies (business rules) can be easily checked and enforced
3. Making sure that the decomposition of tables to obtain tables in better form does not cause us to lose information (lossless join) decomposition

- But we did not have an algorithmic procedure to do this
- We now continue with building up intuition and actually learning an algorithmic procedure

Closures Of Sets Of Attributes (Column Names)

- Closure of a set of attributes is an easy to use but extremely powerful tool for everything that follows
- "On the way" we may review some concepts
- We return to our old example, in which we are given a table with three columns (attributes)
  - 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 a 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 $(GS)^+$ **and not** $G(S)^+$

Computing a Closure: An Example

- Our table is ABCDE
- Our only functional dependency (FD) is BC → D
  - This means: any tuples that are equal on both B and on C must be equal on D also
- We look at all the tuples of the table in which ABC has a specific fixed value, that is all the values of A are the same, all the values of B are the same and all the values of C are the same
  - We discuss soon why this is interesting
- What other columns from D and E have specific fixed values for the set of tuples we are considering?

- D has to have a specific fixed value
- E does not have to have a specific fixed value

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. $K \rightarrow BG$
  2. $A \rightarrow DE$
  3. $H \rightarrow AI$
  4. $B \rightarrow D$
  5. $J \rightarrow IH$
  6. $C \rightarrow K$
  7. $I \rightarrow J$
- 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 (relation schema which is always denoted by its set of attributes), 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
- "Trivial" means $P \rightarrow Q$ and $Q \subseteq P$: we saying something that is always true and not interesting
- Example, $AB \rightarrow A$ is always true and does not say anything interesting
- Example, if a table is R(FirstName,LastName) without any functional dependencies, then its key is just the pair (FirstName,LastName)

Keys of Tables

- 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}

- Of course, in general, our algorithm is not efficient, but in practice what we do will be very efficient (most of the times)

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 key or maybe contains a key as a proper subset
  - We could check whether ABCH is a key by computing ABC^+, ABH^+, ACH^+, BCH^+ and showing that none of them is ABCDEGHIJK

Another 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 $\rightarrow$ T
  - PDT $\rightarrow$ F
  - FD $\rightarrow$ 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

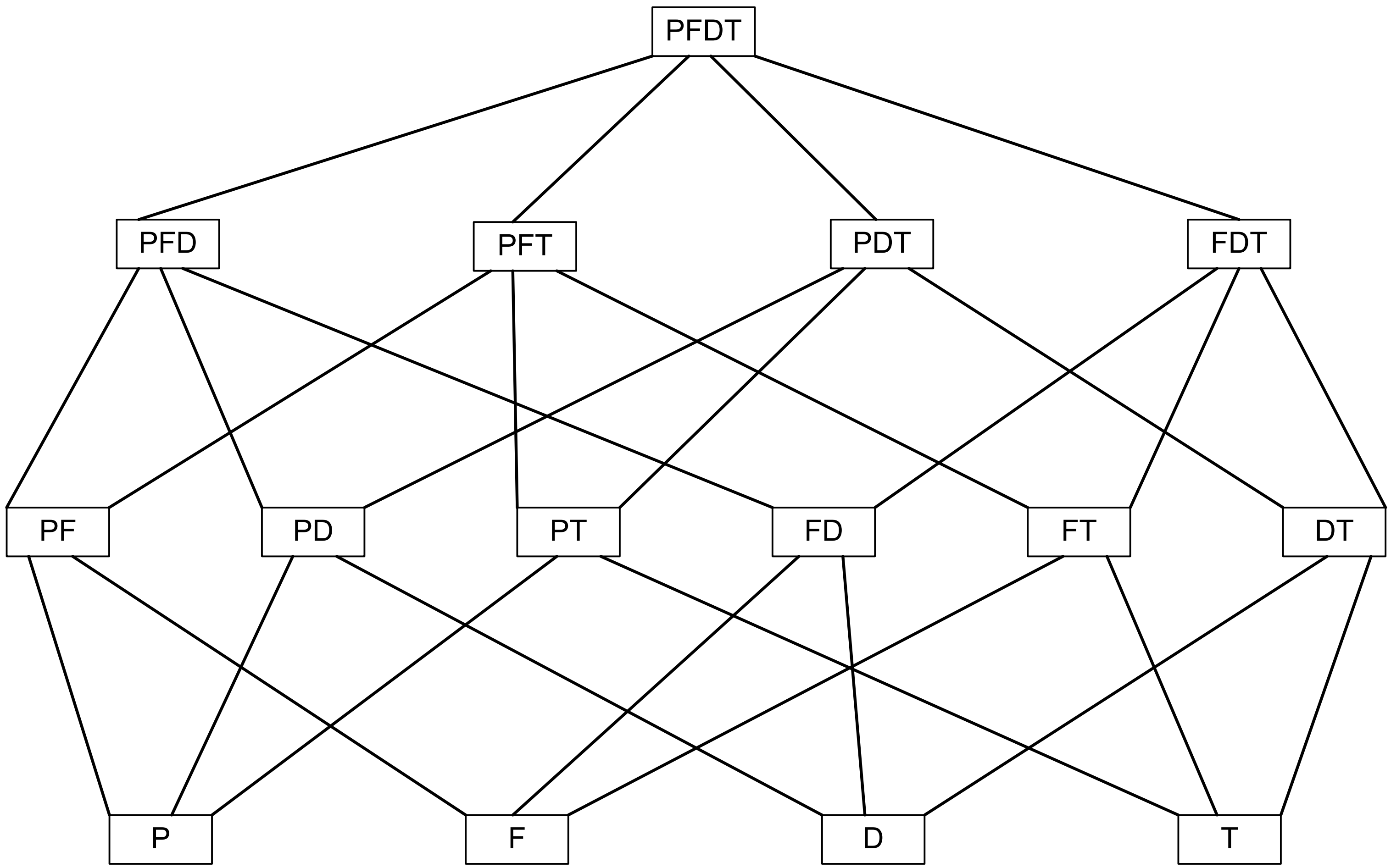
- We note that if some subset of attributes is a key, then no proper superset of it can be a key as it would not be minimal and would have superfluous attributes

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



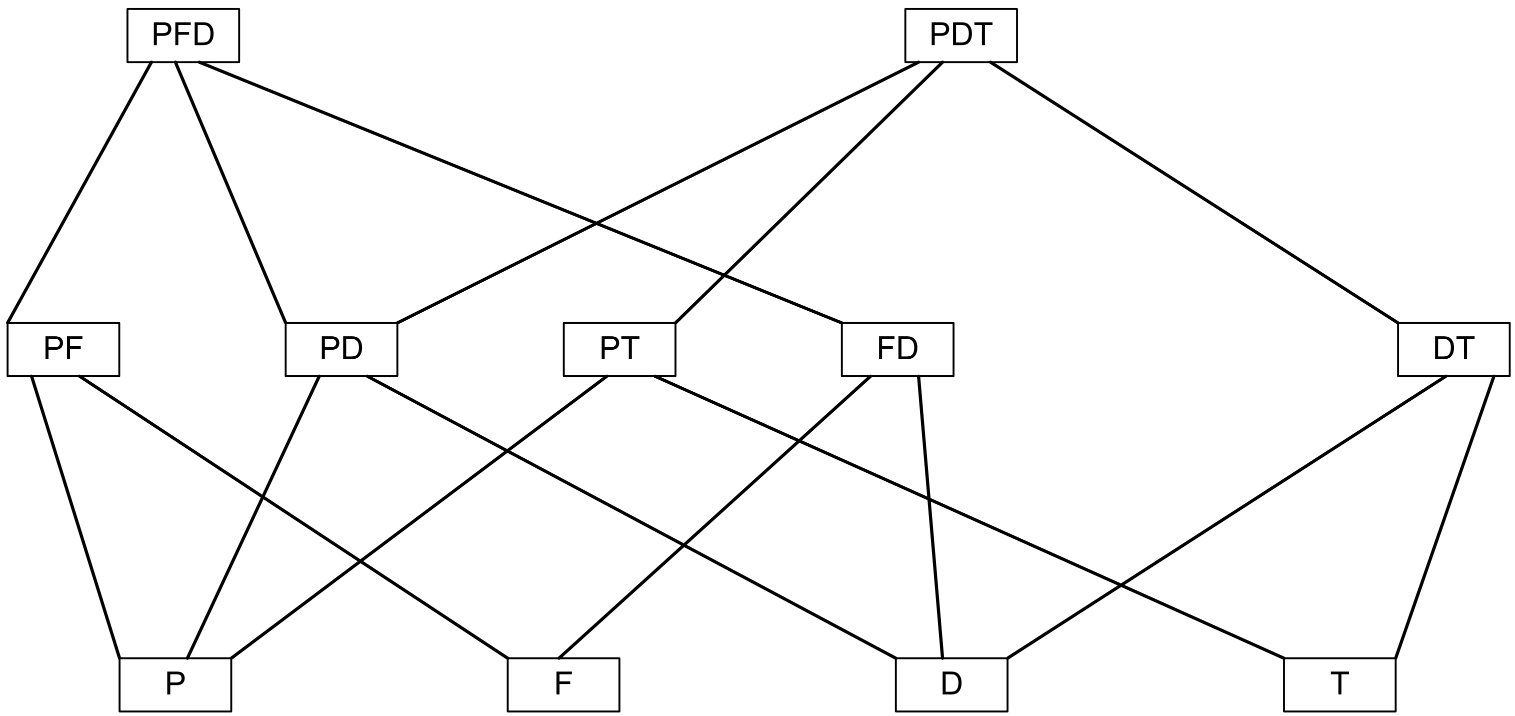
Keys Of PFDT

- 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

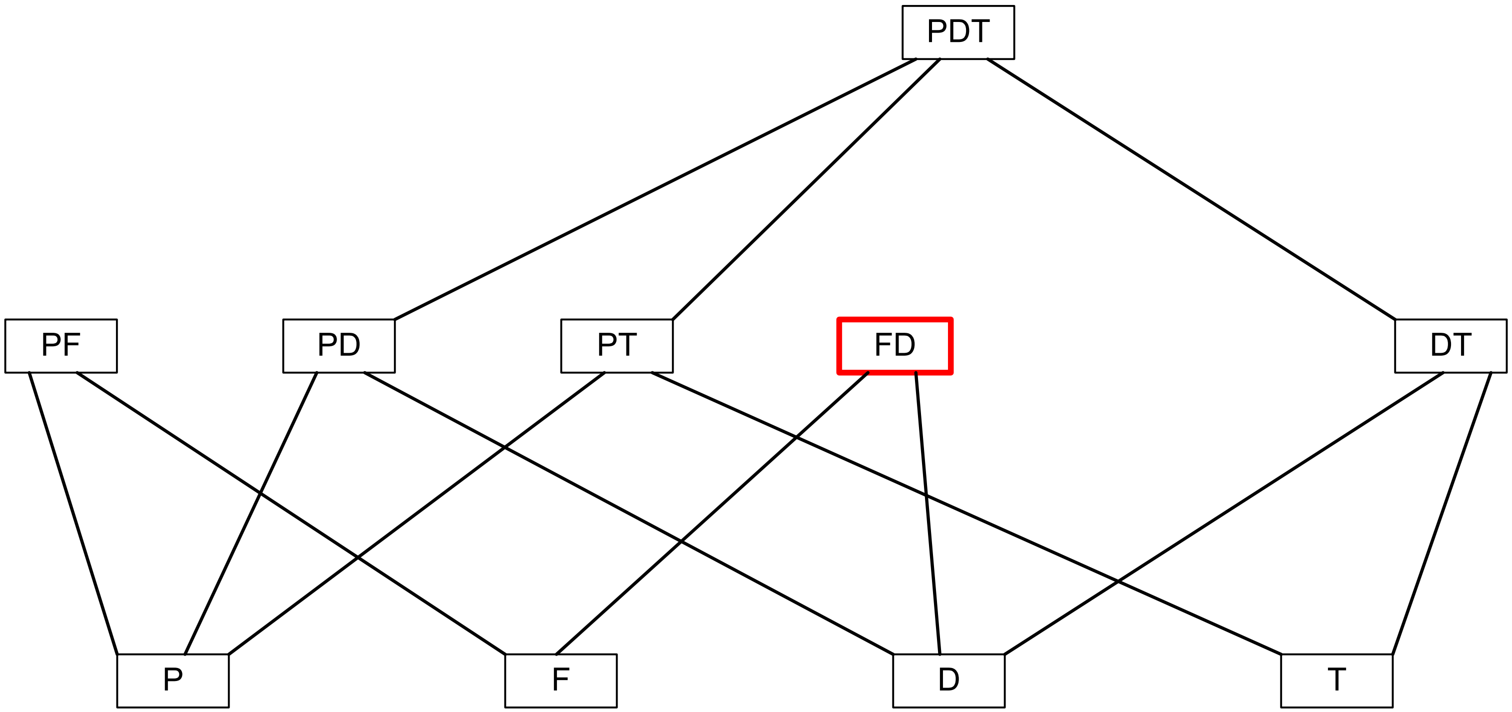
  There is one 2-attribute key: FD

- We can mark the lattice

- We can prune the lattice

Marked And Pruned Lattice

- The key we found is marked with red
- Some nodes can be removed



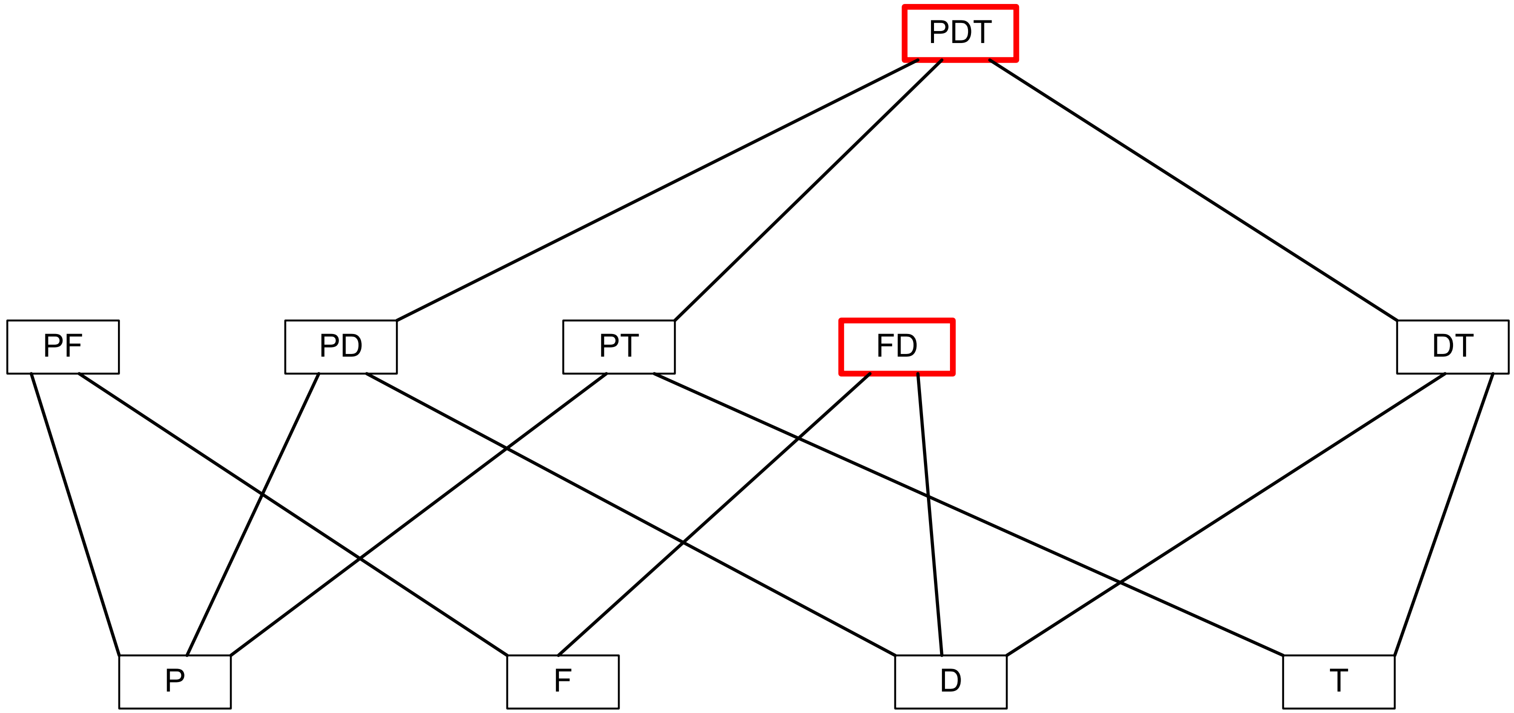
Keys Of PFDT

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

  There is one 3-attribute key: PDT

Final Lattice
We Only Care About The Keys

- We could have removed some nodes, but we did not need to do that as we found all the possible 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

- Although this will be an example, the example will be sufficiently general so that the general procedure will be covered

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

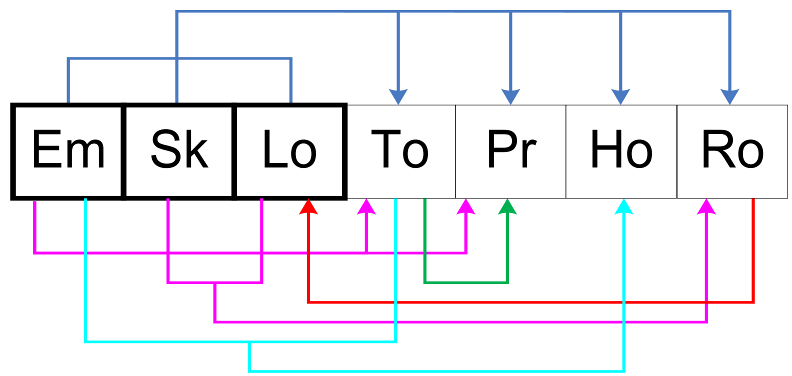
- The table deals with employees who use tools on projects and work a certain number of hours per week
- An employee may work in various locations and has a variety of skills
- All employees having a certain skill and working in a certain location meet in a specified room once a week
- The table satisfies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

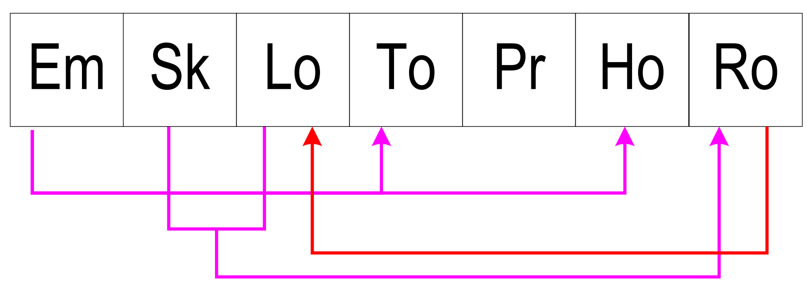
1. Em → To
2. Em → Pr
3. To → Pr
4. EmTo → Ho
5. SkLo → Ro
6. Ro → Lo



- What should we do with this drawing? I do not know. We need an algorithm
- 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 Minimal Cover

- We need to "simplify" our set of FDs to bring it into a "nicer" form, so called ***minimal cover*** or (sometimes called also ***canonical cover***)
- But, of course, the power has to be the same as we need to enforce the same business rules
- The algorithm for this will be covered later, it is very important
- The end result is:
  1. Em → ToHo
  2. To → Pr
  3. SkLo → Ro
  4. Ro → Lo
- From these we will build our tables directly, but just for fun, we can look at a drawing



2: Creating Tables From a Minimal Cover

- 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 Using a Good Heuristic

- 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 → 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 Decompostion

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 Decompostion

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 based on the given business rules and thus reconstruct the original table
- All the tables are in 3NF

Computing Minimal Cover

- What remains to be done is to learn how to start with a set of FDs and to “reduce” them to a “clean” set ***with equivalent constraints power***
- This “clean” set is a minimal cover
- So we need to learn how to do that next

- We need first to understand better some properties of FDs

To Remind: 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. $A \rightarrow C$ No
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
8. $HGE \rightarrow GE$ Yes

Relative Power Of Some FDs
H → G vs. H → 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 → Bonus
  2. Grade → 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 *$H \rightarrow G$ vs. $H \rightarrow GE$*

- An important note: $H \rightarrow GE$ is always at least as powerful as $H \rightarrow G$

that is

- If a relation satisfies $H \rightarrow GE$ it must satisfy $H \rightarrow G$

- 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 $Z \subseteq Y$; then if $X \rightarrow Y$ is true than $X \rightarrow Z$ is true

Relative Power Of Some FDs
$A \rightarrow C$ vs. $AB \rightarrow C$

- 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 $A \rightarrow C$ vs. $AB \rightarrow C$

- An important note: $A \rightarrow C$ is always at least as powerful as $AB \rightarrow C$

that is

- If a relation satisfies $A \rightarrow C$ it must satisfy $AB \rightarrow C$

- 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 $X \subseteq Y$; then if $X \rightarrow Z$ is true than $Y \rightarrow Z$ is true

Trivial FDs

- An FD $X \rightarrow Y$, where X and Y are sets of attributes is trivial

  if and only if

  $Y \subseteq X$

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

- Example
  - Grade, Salary $\rightarrow$ 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 $\rightarrow$ 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?
  - $E \rightarrow G$, $G \rightarrow S$, $E \rightarrow S$ are all true in the real world
- If the real world tells you only:
  - $E \rightarrow G$ and $G \rightarrow S$
- Can you deduce on your own (and is it even always true?), ***without understanding the semantics of the application***, that
  - $E \rightarrow S$?

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	<u>E</u>	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 → G, G → S, and E → S ***must all be obeyed***

- ***It is enough*** to make sure that the two constraints E → G and G → S are not violated

- ***It is not enough*** to make sure that the two constraints E → G and E → S are not violated

- But what to do in general, large, complex cases?

To Remind: 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 $\mathbf{X^+}$

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
- 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

Union Rule: Combining Right Hand Sides (RHSs)

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

  is equivalent to

  $H = \{ AB \to CD \}$

- We have discussed this rule before
- Intuitively clear
- Formally we need to prove 2 things
  - $F \models H$ is true; we do this (as we know) by showing that AB_F^+ contains CD; easy exercise
  - $H \models F$ 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
  - $F = \{A \to C, B \to C\}$ ***is stronger than*** $H = \{AB \to C\}$

Union Rule: Combining Right Hand Sides (RHSs)

- Stated formally:
  F = { X → Y, X → Z } ***is as powerful as*** H = { X → YZ }

- Easy proof, we omit

Relative Power Of FDs: Left Hand Side (LHS)

- F = { AB → C }

  is weaker than

  H = { A → 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 → Y } ***is weaker than*** H = { X → Y }, (if B ∉ 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)

- $F = \{ A \rightarrow BC \}$

  is stronger than

  $H = \{ A \rightarrow B \}$

- 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:

  $F = \{ X \rightarrow YC \}$ ***is stronger than*** $H = \{ X \rightarrow Y \}$, (if $C \notin Y$ and $C \notin X$)

- 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 $F \models X \rightarrow Y$ 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 $H \models X \rightarrow YC$ 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 Em 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
  - This, of course, needs to be checked
- Therefore we are done
- We have computed a minimal cover for the original set of FDs

Minimal 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 final 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
- We have studied this before

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 → 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 do it differently
- Whenever there is an insert or update, check that FDs hold, or reject these actions

Maintaining FDs During Insertion

- We have a table R satisfying some FDs
- We have a table T of “candidates” for inserting into R
- We want to construct a subset of U of T consisting only of those tuples whose insertion into R would not violate FDs

- We show how to do it for the simple example of R = EGS, where we need to maintain:
  - E is the primary key
  - G → S holds
- We replace

```
INSERT INTO R
(SELECT *
FROM T);
```

By the following

Maintaining FDs During Insertion

```
INSERT INTO R
 (SELECT *
 FROM T
 WHERE NOT EXISTS
  (SELECT *
   FROM R
   WHERE (R.G = T.G AND R.S <> T.S) OR (R.E = T.E)
  )
 );
```

- The WHERE condition will only insert only those tuples from T to R that satisfy the conditions
  - There is no tuple in R with the same value of the primary key E
  - There is no tuple in R with the same G but a different S

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
- We do not have time to do it in this class

Denormalization

- After Normalization, we may want to ***denormalize***
- The idea is to introduce redundant information in order to speed up some queries
- So the design not so clean, but more efficient
- We do not cover more, you can read in http://en.wikipedia.org/wiki/Denormalization

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 each table 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

A Review And Some Additional Material

What We Will Cover Here

- Review concepts dealing with Functional Dependencies
- Review algorithms
- Add some material extending previous material

Functional Dependencies (Abbreviation: FDs)

- We will ignore NULLS
- Let X and Y be sets of columns, then:

  X ***functionally determines*** Y, written X → Y

  if and only if

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

  If a value of X is specified, it "determines" some (specific) value of Y; in other words: Y is a function of X

- We will generally look at ***sets of FDs*** and will denote them as needed by ***M*** and ***N***

Trivial FDs

- If $\boldsymbol{Y} \subseteq \boldsymbol{X}$ then FD $\boldsymbol{X} \rightarrow \boldsymbol{Y}$
  - Holds always
  - Does not say anything
- Such FD is called ***trivial***
- Can always remove the “trivial part” from an FD without changing the constraint expressed by that FD
- Example: Replace

  ABCD $\rightarrow$ CDE

  by

  ABCD $\rightarrow$ E

Having CD on the right side does not add anything

Union Rule/Property

- An FD with n attributes on the right hand side

  $X \rightarrow A1\ A2\ \ldots\ An$

  is equivalent to the set of n FDs

  $X \rightarrow A1$
  $X \rightarrow A2$
  ….
  $X \rightarrow An$

- Example:

  $ABC \rightarrow DEFG$

  is equivalent to set of 4 FDs

  $ABC \rightarrow D$
  $ABC \rightarrow E$
  $ABC \rightarrow F$
  $ABC \rightarrow G$

Closures of a Sets of Attributes

- In general, we have a concept of a ***the closure*** of a set of attributes in a relational schema ***R***
- We are given a set of functional dependencies, say ***M***
- Let ***X*** be a set of attributes,
- X_M^+ is the set of all the attributes whose values are "determined" by the values of X because of M
  - If M is understood, we do not need to write it and can just write X^+

Computing Closures Of Sets Of Attributes

- There is a very simple algorithm to compute X^+ (given some set of FDs)

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

Keys Of Tables

- Given R (relation schema which is always denoted by its set of attributes), 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)

Anomalies And Boyce-Codd Normal Form (BCNF)

- We are given ***R*** (relation schema) and ***M*** (set of FDs)
- We have an anomaly whenever

  $X \rightarrow Y$ is non-trivial and holds

  but

  X does not contain a key of R

- Because there could be different tuples with the same value of X and they all have to have the same value of Y
- A relation is in ***BCNF*** if anomalies as described in this slide do not happen

How To Prove That A Relation Is Not In BCNF

- To prove that relation R is not in BCNF it is enough to show that there is a non-trivial FD $X \rightarrow Y$ and X does not contain a key of R

- And to show that X does not contain a key of R it is enough to show that $X^+ \neq R$

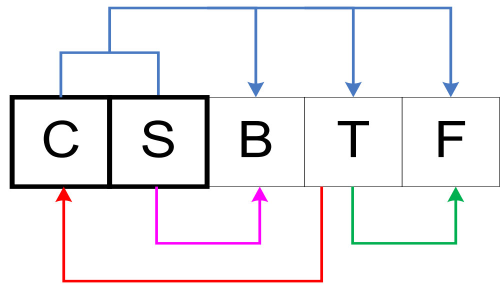
Some Normal Form

- We have discussed several additional normal forms pertaining to FDs
  - Second Normal Form (2NF)
  - Third Normal Form (3NF)
- We did not look at the most general definitions

- Let us review what we did using an old example

- We have, in general, FDs of the form $X \to Y$
- But by the union rule, we can decompose them and consider FDs of the form $X \to A$, where A is a single attribute

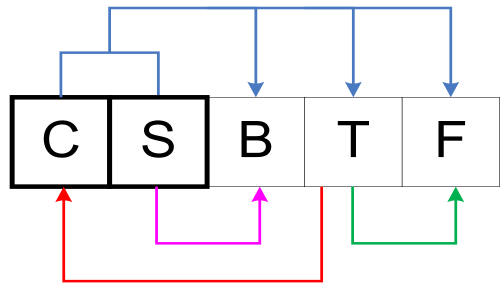
Classification Of FDs (Our Old Example Focusing Only on One Key)



- 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

- But what if we have X → Y where X is partially in the key and partially outside the key?

It is Incomplete to Focus on Only One Key (The Primary Key)



- By looking at the diagram we immediately can deduce that ST is also a key
  - Because T determines C and therefore as SC determined R, so did ST

- And we discussed it too.

General Definition of Some Normal Forms

- Let R be relation schema
- 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

- The interesting is a general definition of 3NF

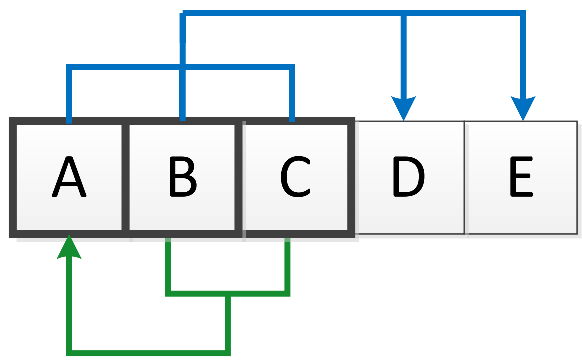
- Note: no discussion of which key is chosen to be primary as this is formally really “an arbitrary decision” though perhaps important for the application

Which FDs Are Allowed For Some Normal Forms
Consider $X \rightarrow A$ (X set, A single)

BCNF	3NF	2NF
$X \rightarrow A$ is trivial (A is inside X)	$X \rightarrow A$ is trivial (A is inside X)	$X \rightarrow A$ is trivial (A is inside X)
X contains a key	X contains a key	X contains a key
	A is in some key (informally: into a key, but X can overlap a key)	A is in some key (informally: into a key, but X can overlap a key
		X not a proper subset of some/any key (not worth making it more precise, as obsolete)

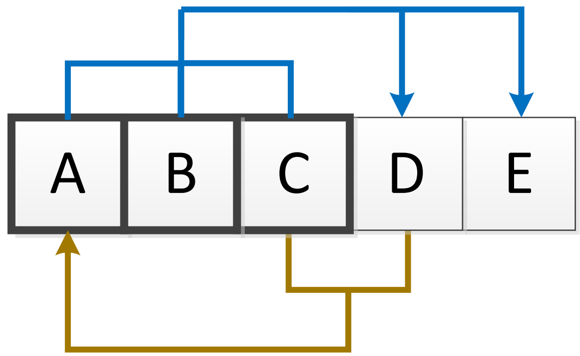
Cannot Have an FD From a Key Into Itself

- It is not possible to have a non-trivial functional dependency from a part of key into that same key
- Proof by example:



- In such a situation ABC is “too big” and actually BC is a key (and also the drawing does not follow standards)

Example: Relation in BCNF And Not in 3NF



- Given functional dependencies: ABC → DE and CD → A
- ABC is a key and designated as primary
- This relation is not in BCNF as we have CD → A and CD does not contain a key as is easily seen
- But CD → A is of the form: (something not containing a key) → (attribute in a key) and this is permitted by 3NF
- Note there is another key that could have been the primary key: BCD
- Originally people were confused as they considered only one key and did not realize that in general 3NF ≠ BCNF

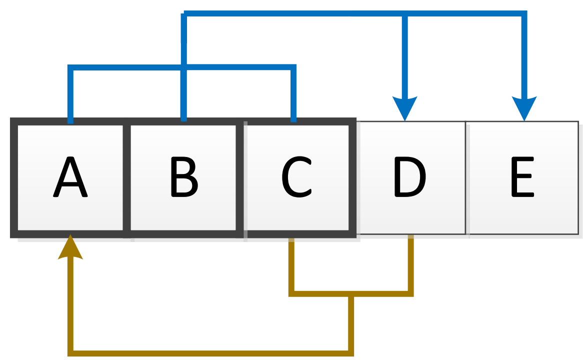
If Only One Key Then 3NF ⇒ BCNF

- Proof by contradiction (using example, but really general)
- Assume that a relation is in 3NF but not in BCNF and there is only one key
- Then we have a functional dependency that is permitted by 3NF but not permitted by BCNF, that is of the form

  (something not containing a key) → (attribute in a key)

- Example

  ABC is a key
  and CD → A holds



- Then we see that BCD is a key also, so we have more than one key
- So we proved: ***if*** 3NF and only one key ***then*** BCNF

Relative Power of FDS: Simplify RHS

- If attributes removed from RHS (right hand side), the functional dependency becomes weaker

- Changing from $ABCD \rightarrow EFG$ to $ABCD \rightarrow EF$ the dependency becomes weaker

- Intuitively, after the simplification, we start with the same assumptions and deduce fewer conclusions

Relative Power of FDS: Simplify LHS

- If attributes removed from LHS (left hand side), the functional dependency becomes stronger
- Changing from $ABCD \rightarrow EFG$ to $ABC \rightarrow EFG$ the dependency becomes stronger
- Intuitively, after the simplification, we start with fewer assumptions and deduce the same conclusions

A Typical Step in Computing Minimal Cover

- We have a set M of functional dependencies
- M contains two functional dependencies with the same left hand side, say

  $$X \to EFG$$
  $$X \to GH$$

- We replace these functional dependencies by one functional dependency

  $$X \to EFGH$$

- And we get a set N of functional dependencies

- N is equivalent to M

A Typical Step in Computing Minimal Cover

- We have a set M of functional dependencies.
- M contains a functional dependency with more than one attribute in the RHS, say

  $X \rightarrow EFG$

- We replace this functional dependency by

  $X \rightarrow EF$

- And we get a set N of functional dependencies

- N can only be weaker (in power) than M

- N is equivalent (in power) to M

  if and only if

  we can "prove the stronger functional dependency":

  X_N^+ contains EFG

A Typical Step in Computing Minimal Cover

- We have a set M of functional dependencies.
- M contains a functional dependency with more than one attribute in the LHS, say

  $ABCD \rightarrow Y$
- We replace this functional dependency by

  $ABC \rightarrow Y$
- And we get a set N of functional dependencies
- N can only be stronger (in power) than M
- N is equivalent to M

  if and only if

  we can "prove the stronger functional dependency":

  ABC_M^+ contains Y

The Goal

- Given a table R satisfying a set of FDs M, decompose it into tables: R_1 satisfying M_1, R_2 satisfying M_2, …, R_k satisfying M_k, such that

- The decomposition is lossless join: can recover R from R_1, R_2, …, R_k using natural join
- Dependencies are preserved: making sure that (after changes to the database) ***if*** R_1 satisfies M_1, R_2 satisfies M_2, …, R_k satisfies M_k, ***then*** that if we recover R it will satisfy M
- R_1, R_2, …, R_k are all in 3NF (and if we are lucky also in BCNF)

Sketch of The Procedure

- Compute a minimal cover N for M
- Create a table for each functional dependency in N
- Remove duplicate tables (really subsumed in the following)
- Remove a table if its set of columns is a subset of the set of columns of another table
- Check if at least one table contains a global key: just compute closure of its attributes using M (or N, likely faster) and see if you get all of R
- If no table contains a global key, find one global key (using heuristics or otherwise) and add a table whose columns are the attributes of the global key you found

Key Ideas

- 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

- Multivalued dependencies
- Fourth Normal Form: 4NF
- Minimal cover for a set of functional dependencies
- Algorithmic technique for finding keys
- Algorithmic technique for computing a minimal 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
- Denormalization after Normalization