CP363: Database I

by Scott King

Dr. Siu-Cheung Chau

Chatper 1: Introduction

1 Database Management Systems (DBMS)

A *DBMS* is a collection of software to enable a user to create, maintain and utilize a DB (Ex. Oracle, MySQL, etc).

1.1 Advantages of DBMS

- Efficient data access
- Data integrity and security
- Data administration
- Data independence
- Concurrent processing crash recovery
- Reduce application development time and cost

Concurrent Processing

{figure out diagram}

2 Logical Schema of a Database

- Schema is the data structureIBM had the hierarchical model, other companies created the Network and Lodeysol models
- Logical (conceptual) scheme is the logical data structure

3 Physical Schema

- How data is stored physically: tree, heap, etc
- View on the logical schema, different ways of viewing data; sometimes public facing

4 Logical Data Independence

- Changing to logical schema will not effect the view on it

5 Physical Data Independence

- Changing to physical schema will not effect the logical schema

5.1 Lingo

{figure out diagram}

6 Data Models

- IBM had the hierarchical model, other companies created the Network and Lodeysol models

Chapter 2: Introduction to the Relational model

1 Relational DB Model

- Data is stored using tables, also known as relations
- Each column has a name, AKA attribute
- Each row is record

1.1 Schema

- Defining the structure of a relation

Example 1.

```
student_table(Id int(8),
              Name varchar(8),
              Major varchar(8),
              GPA float);
```

- Tables are defined as such: $R(A_1, ..., A_n)$
- An *instance* is a record in a table

A **database instance** is all the data in the DB. **Database schema** contains schemas for all tables in the form $S = R_1, ...R_m$.

The table must satisfy four constraints:

1. Domain constraint
2. Key integrity constraint
3. Entity integrity constraint
4. Referential integrity constraint

Example 2.

Eid	Name	Dependent
1234	*Tom*	Mary
1234	*Tom*	Peter
2345	*John*	Tim
2345	*John*	Jane
3456	David	Jim

Remember **NO** duplicates. Instead change those records to:

Eid	Name	Dependent
1234	*Tom*	Mary,Peter
2345	*John*	Tim,Jane
3456	David	Jim

Each attribute can only contain *atomic* values. Ex. Cannot contain a set of values. FFS, do this:

Eid	Name	Dependent 1	Dependent 2
1234	*Tom*	Mary	Peter
2345	*John*	Tim	Jane
3456	David	Jim	--

This is also **NOT** ideal.

R_1:

Eid	Name
1234	*Tom*
2345	*John*
3456	David

R_2:

Eid	Dependent
1234	Mary
1234	Peter
2345	Tim
2345	Jane
3456	Jim

1. Key Integrity Constraint

- Observation: No tuples in a table are the same; in other words, every tuple in a table needs to be unique..SO DONT STORE RECORDS TWICE
- $\Rightarrow$ Each record, tuple, can be <u>uniquely identified</u> by a certain set of attributes
- The *super key* is a set of attributes that can uniquely identify a record (tuple)
- The trivial super key is the one with all the attributes
- A minimal super key is called a *candidate key*
- A candidate key be picked as the *primary key* $\rightarrow$ Refer to next Example 3
- Every relation must have a primary key `student_table(Id,Name,Major,GPA)` where `Id` is the primary key

Example 3. `id,dept_name` are super keys but not candidate keys. But two candidate keys are `id,Name` as they are unique. Then pick on candidate to be the primary, this `Id` will be picked as the primary key.

Example 4. {table}

We have the minimal super key = candidate key. $\{A, D\}$ is not a super key. $\{B, C\}$ is a super key. Is it a candidate key? No, thus, C is a super key $\Rightarrow$ C is a candidate key $\Rightarrow$ C is a primary key

2. Entity Integrity Constraint

- The primary key cannot be `null`; in our example, that means `Id` cannot be null
- The rest of the attribute can be `null`

3. Referential Integrity Constraint

- A *foreign key* must either contain a `null` value or a value in the **referenced key**

Example 5. Back to the example with R_1 and R_2. We have R_1`(Eid,Name)` and R_2`(Eid,Dependent)`.

Thus, `Eid`, is an foreign key in R_2 because `Eid` is the primary key in R_1. `Eid` in R_1 is the referenced key.

Chapter 7: DB Design

1. Requirement Analysis

- Data to be stored
- Applications
- Performance analysis

Example 6. Registration system

```
student records
class records
professor records
```

Actions:

- To register a student in the class
- Print class list
- Print professor schedule

2. Conceptual Database Design

- Entity-Relationship model (ER model)
- Entity is equivalent to record type
- The relationship between entities

3. Logical Database Design

- Convert the ER model into a relational model (ie. convert to tables - relations)

4. Scheme Refinement

- Theory of relational database - 1^{st}, 2^{nd}, 3^{rd}, 4^{th} BC (Boyce-Codd) normal forms

5. "Physical" Database Design

- Add index files to speed up certain applications

6. Application and Security Design

1 Entity-Relationship Model

Entity $\equiv$ record

- An entity is an object in the real word that is distinguishable from other objects

- A **diamond** indicates a relationship between two entities
- A **triangle** indicates a subclass of
- A **circle** represents an attribute
- A **rectangle** represents an entity

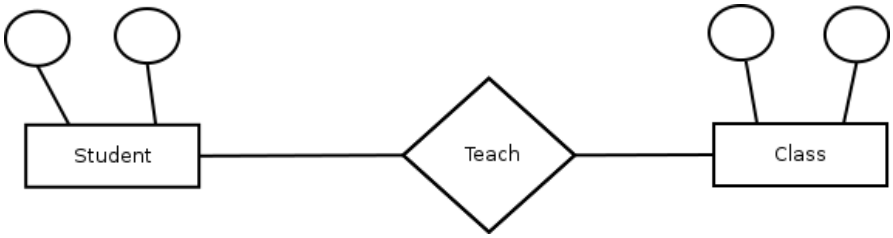
Example 7. A student record

These are attributes

`Id` `Name` `GPA` `Major`

- An entity is a collection of entities of the same type
- All entities in the same set have the same attributes (cicles)
- *Student* is the entity and *{Id, Name, GPA, Major}* are the attributes (circles)
- Each entity has a primary key
- The key should depend on a real life situation and not the current set of data

2 Relationship Between Entities



2.1 Many to Many Relationship

1. A student can take many courses
2. A class can have many students

2.2 1 to Many Relationship

1. A professor can teach many classes
2. A class is taught by 1 professor

2.3 1 to 1 Relationship

1. A department has only one chair
2. A professor is the chair of only one department

Note: Every entity has a key and every relationship has a key.

A relationship does not have to be binary.

2.4 Total Participation

- Everyone is involved ⇒ total participation (**thick line**)

2.5 Recursive Relationship

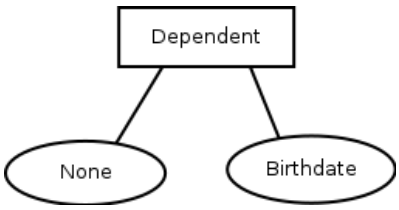
- An entity can only appear once in a design
- The key for the relationship (Id) - in our example

2.6 Weak Entity

- Entity without superkey

Note: Thick [box] lines implies weak entity.

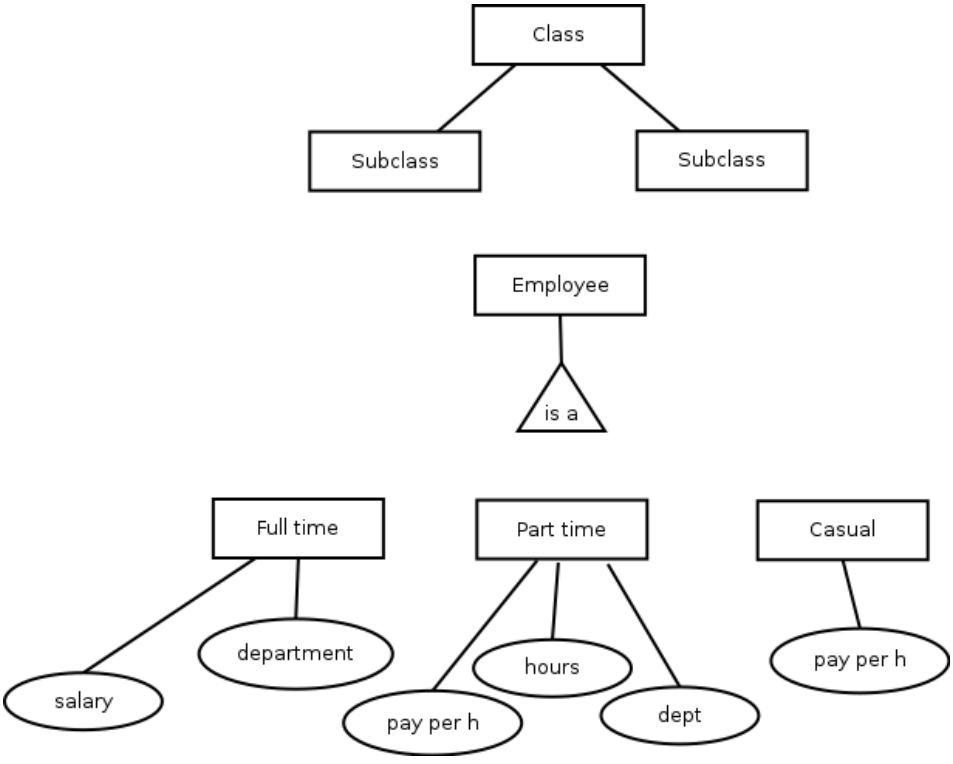
Example 8.



2.7 Strong Entity

- Entity with a super key (primary key)
- Everything so far is a strong entity

3 Class Hierarchies



The subclasses are disjoint.

4 Aggregation

- Abstraction to group relationships

Refer to customer - borrower - loan.

5 Issues in Conceptual Design Using the ER Model

1. Should a concept be modeled as an attribute or as an entity

    - Refer to employee one
    - Employees can have multiple addresses (sub address with employee to create new entity with attributes)
    - If we wanted to conduct searches in the address - it would be easier with address as an entity (ex. search for a city within a country)
    - Refer to picture - it is wrong, solution: look to next picture (make from and to as entity instead of attribute)

6 ER to Relational Database Mapping

Step 1

For each strong entity set, E, create a table (relation), R, that includes all attributes of E. The key for the entity will be the key for the table (relation).

$$\text{strong entity} \Rightarrow \text{table}$$

Step 2

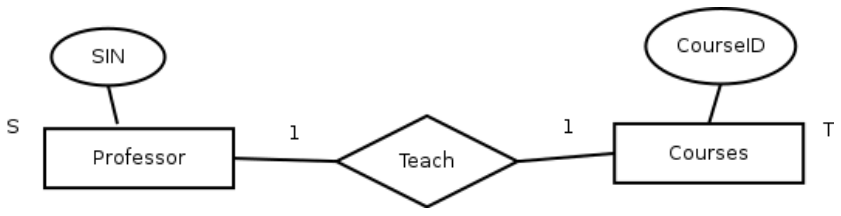
For each weak entity set, W, create a table, R, that includes all attributes of W and include it as a foreign key, the primary key of the owner entity. The key for the relation is the partial key and the key of the own entity.

{table for employee-dependents}

Step 3

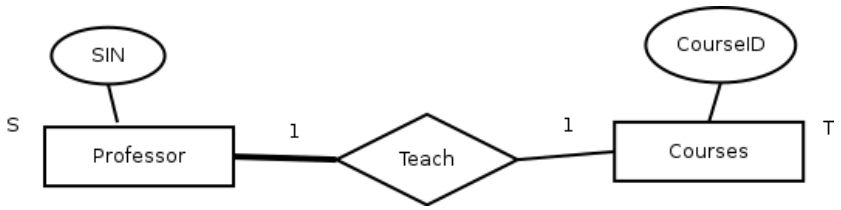
For each 1:1 relationship between two strong entitie, S and T. **No** new table (relation) is required. Just add the primary key of S as a foreign key in T.

Example 9.




```
professor(SIN, ..., ...)
courses(courseID, ..., ..., SIN)
```

If total participation exists in $\underline{S}$, we should add the primary key of T as a foreign key to $\underline{S}$.



Every professor much teach one course and the course cannot be taught by more than one professor.

Step 4

For each 1:n relationship between two strong entities S and T. No new table is required. Just add the primary key of S as a foreign key in T.

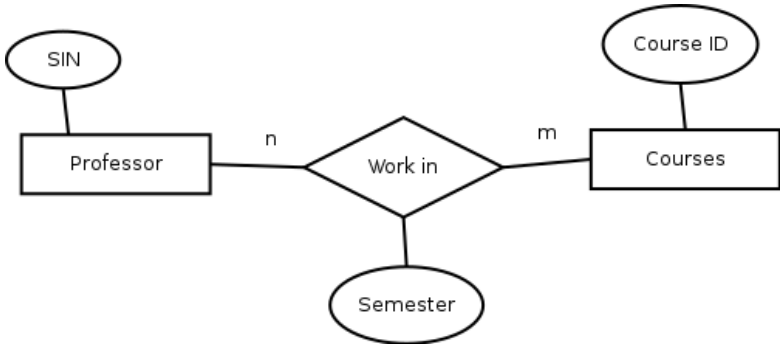
Example 10.

`Employee(id, ..., ..., departmentID)`
`-------------`

Step 5

For each many to many relatonship between S and T, create a new table (relation) that contains the primary key of S and T. The key for the new table is the primary key of both S and T together.

Example 11.

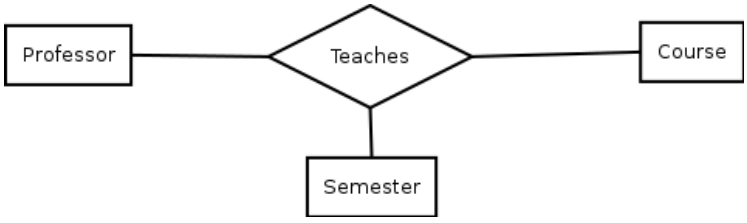


`Teaches(SIN, CourseID, semester)`

Step 6

n-ary relationships.

Example 12.

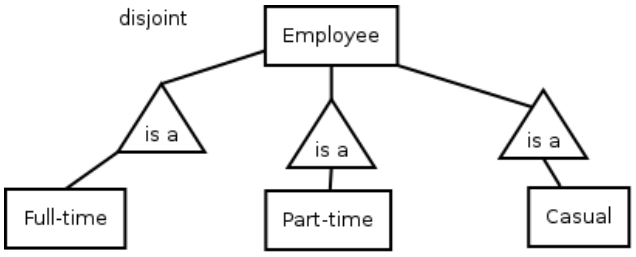


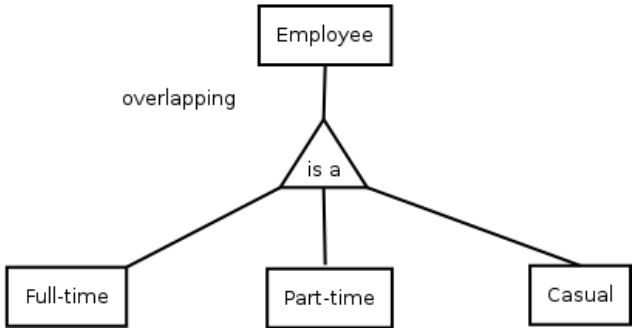
Teaches(SIN, semester_id, course_no)

Create a new table that contains the primary key of all the entities. The keys for the table are the primary keys of all entities together.

Step 7

Translating class hierarchies.





Disjoint implies that we create a table for each disjoint entity and it is necessary to create one for the parent. **Overlapping** implies that we create a table for all overlapping entities and create a table for the parent; only put the key of the parent in table of the subclass.

Step 8

Aggregates.

Create a table for an aggregate that contains the key for the aggregate and the key for the entity.

Example 13.

Supervise(SIN,Graduate_SIN, p_no) all with dotted underlines

7 Data Definition Language (DDL) - part of SQL

Example 14. Create a table in SQL

Employee(Eid,name,address,salary,Supervisor) - last one is dotted
Supervisor(Eid, S_ID) both dotted

Here's the code:

```
CREATE TABLE   Employee
```

```
(Eid            integer(9),      // or int(9)
 Name           varchar(30),
 address        varchar(50),
 salary         integer(5) unique,    // CANNOT BE null
 Sid            integer(9),
 primary key    (Eid)
 foreign key    (SupervisorID) reference (Employee))

CREATE TABLE    Supervisor
(Eid            int(9),
 Sid            int(9),
 primary key    (Eid,Sid),
 foreign key    (Eid) reference (Employee),
 foreign key    (Sid) reference (Employee))
```

7.1 Delete a Table

```
drop table Employee
```

7.2 Update a Table

Using `Employee` table, we want to add a new attribute.

```
Alter table Employee
    add status varchar(6)
```

Now we want to delete an attribute:

```
Alter table Employee
    drop name
```

We now want to delete a record with: `Employee id=1234`. `Employee` with `id=1234` may be a supervisor of other `Employee`'s. We need to update every other `Employee`'s `Supervisor`.

Chapter 7: Relation Algebra

1 Basic Operation

Select	ϖ (lowercase sigma)
Project	Π
Union	$\cup$
Set different	-
Cartesian product	$\times$
Rename	e

The first 3 +last are unary operations and the last three are binary operations.

Example 15. `Employee(Id,name,address,dept,salary)`

$\texttt{Temp} \leftarrow \Pi\ \texttt{Employee}_{\texttt{Id;name}}$

Just choose a number of attributes from the table.

Select all records with a salary of $\geqslant$80000

$\texttt{R} \leftarrow \varpi\ \texttt{Employee}_{\texttt{salary}\geqslant 80000}$

Π $\texttt{Employee}_{\texttt{Id;none}}$ (Select $\texttt{Employee}_{\texttt{salary} \geqslant 80000}$)

```
æ          Id,name
From       Employee
Where      salary ⩾ 80000
```

Union is the same as set union. Set difference: only operands (relations) with the same attributes can be involved.

1.1 Cartesian Product

```
Employee(Id,name,dept,...)
Dept(dept_no,name,...)

Employee
1   1234    Tom    2   ...
2   2345    Mary   1   ...
3   3456    Jim    1   ...
4   4567    Kate   2   ...

Dept
a   1       Sales      ...
b   2       Accounting ...
```

Where this will produce a new table.

The universal relation contains all attributes. In order to avoid duplicates, we break it into many smaller relationships. We can use the cartesian product to join them back together. Unforunately, joining them back together doesn't always work 100%.

1.2 Join Product

- Natural join
- Equivalent join
- Conditional join

Natural join: $\ltimes\rtimes$. Example, `Employee` $\ltimes\rtimes$ `Department`

1. `Employee` $\times$ `Department`
2. $\sigma_{\texttt{Employee;Dnum}} = \texttt{Dnum}$

Equi-Join

$$\begin{aligned} & \texttt{Employee}_{\texttt{dnum}} \ltimes\rtimes \texttt{Department} \\ \equiv & \sigma_{\texttt{Dnum}}\, \texttt{Employee} \times \texttt{Department} \end{aligned}$$

Conditional Join

$$\begin{aligned} & \texttt{Employee} \ltimes\rtimes \texttt{Department} \\ & E_1\,\texttt{salary} > E_2\,\texttt{salary} \\ \equiv & \sigma_{E_1\,\texttt{salary} > E_2\,\texttt{salary}} \quad E_1 \times E_2 \end{aligned}$$

Example 16.

```
Sailors(Sid, Sname, age)
Boat(Bid, Bname, colour)
Reserves(Sid, Bid, Date)
```

Query 1

Find the name of the sailors who have reserved boat with `Bid=103`.

$$\begin{aligned} T_1 &\leftarrow \texttt{Sailor} \ltimes\rtimes \texttt{Reserves} \\ T_2 &\leftarrow \sigma_{\texttt{Bid=103}}\, T_1 \\ T_3 &\leftarrow \Pi_{\texttt{Sname}}\, T_2 \\ T_3 &\leftarrow \Pi_{\texttt{Sname}}(\texttt{Sailor} \ltimes\rtimes (\sigma_{\texttt{Bid=103}}\, \texttt{Reserves})) \end{aligned}$$

Query 2

Find the names of the sailors who have reserved at least one **red** boat.

$$\begin{aligned} T_1 &\leftarrow \texttt{Sailor} \ltimes\rtimes \texttt{Reserves} \ltimes\rtimes \texttt{Boat} \\ T_2 &\leftarrow \sigma_{\texttt{colour=red}}\, T_1 \\ T_3 &\leftarrow \Pi_{\texttt{Sname}}\, T_2 \\ T_1 &\leftarrow \Pi_{\texttt{Sname}}(\texttt{Sailor} \ltimes\rtimes \texttt{Reserves} \ltimes\rtimes (\sigma_{\texttt{colour=red}}\, \texttt{Boat})) \end{aligned}$$

Query 3

Find the names of the sailors who have reserved at least a **red** or a **green** boat.

Note: You can use red or green in one line, but it only works for *or*. You cannot do a similar operation with *and*. But it is preferred to split it up.

$$\begin{aligned} \text{Red} &\leftarrow \Pi_{\texttt{Sname}}(\texttt{Sailor} \ltimes\rtimes \texttt{Reserves} \ltimes\rtimes (\sigma_{\texttt{colour=red}}\, \texttt{Boat})) \\ \text{Green} &\leftarrow \Pi_{\texttt{Sname}}(\texttt{Sailor} \ltimes\rtimes \texttt{Reserves} \ltimes\rtimes (\sigma_{\texttt{colour=green}}\, \texttt{Boat})) \\ \text{Answer} &\leftarrow \text{Red} \cup \text{Green} \end{aligned}$$

Query 4

Find the names of the sailors who have reserved at least a **red** *and* a **green** boat.

$$\text{Answer} \leftarrow \text{Red} \cap \text{Green}$$

Query 5

Find the name of sailors who have reserved **all** red boats.

$$\begin{aligned} \text{All the red boats} &\leftarrow \sigma_{\texttt{colour=red}}\, \texttt{Boats} \\ T &\leftarrow \Pi_{\texttt{Sname}}(\texttt{Sailor} \ltimes \rtimes \texttt{Reserves} \,/\, \texttt{All red}) \\ &\text{or} \\ T_1 &\leftarrow \Pi_{\texttt{Sid,Bid}}(\texttt{Sailors} \times \texttt{All red}) \\ T_2 &\leftarrow T_1 - \Pi_{\texttt{Sid,Bid}}(\texttt{Reserves}) \\ T_3 &\leftarrow \Pi_{\texttt{Sid}}(\texttt{Sailors}) - \Pi_{\texttt{Sid}}(T_2) \end{aligned}$$

Division

Note: This is a not a basic operation, shame.

A	x	y
	α	1
	α	2
	α	3
	β	1
	γ	1
	θ	1
	θ	3
	θ	4
	ϵ	1
	ϵ	2
	κ	1
	κ	2
	κ	3

Where A and B are the same attribute.

B	y
	1
	2

$$\frac{A}{B} = \begin{array}{c} x \\ \alpha \\ \epsilon \\ \kappa \end{array}$$

First test: chapter 1 (1.1-1.5), chapter 2 (2.1-2.4), chapter 3 (3.2), chapter 7, chapter 6 (6.1).

Example 17. `Employee(id, name, dept, ..., salary)`

We need to find the total salary of all employees.

1. Retrieve all employee records
2. Write a program to sum all salaries

Denote $\mathcal{G}$ as an aggregate function (sum, average, max, min, count)

Thus, grab all salaries of employees: $\mathcal{G}_{\texttt{sum(salary)}}\texttt{Employee}$.

Find the number of records per `Employee`: $\mathcal{G}_{\texttt{count(*)}}\texttt{Employee}$

Example 18. `Employee(id, name, dept, ..., salary)`

We want to find the total salary for each department.

$$\texttt{dept}\mathcal{G}_{\texttt{sum(salary)}}\texttt{Employee}$$

Maximum salary for each dept:

$$\texttt{dept}\mathcal{G}_{\texttt{sum(salary)}}\texttt{Employee}$$

Find the average salary:

$$\texttt{dept}\mathcal{G}_{\texttt{avg(salary) as average}}\texttt{Employee}$$

Note: We created an alias for `avg`, that is `average`. Thus calling it will invoke `avg`.

Count the number of employees in each dept:

$$\texttt{dept}\mathcal{G}_{\texttt{count(id)}}\texttt{Employee}$$

Outer Join

There are 3 types:

1. Full outer join *bowtie with both top and bottom lines extended*
2. Left outer join *bowtie with left top and bottom lines extended
3. Right outer join *bowtie with right top and bottom lines extended*

Project	Π
Selection	σ

In SQL:

```
Select   <attribute1 > ... < attribute n >
  From   <relation1 > ... < relation n >
```

Example 19.

$$\Pi_{\texttt{a}_2,\texttt{a}_3}(R_1{}_{R_1{}^{a}1=R_2{}^{a}2} \ltimes \rtimes R_2{}_{R_2{}^{a}4=R_3{}^{a}4} \ltimes \rtimes R_3)$$

Translation:

```
Select a2,a3
From   R1, R2, R3
where  R1a1 = R2a2 and R2a4 = R3a4
```

$$\mathcal{G}_{\texttt{sum(a}_1\texttt{)}}R_1$$

Translation:

```
Select sum(a1)
From   R1
```

Find the sum for each dept:

$${}_{\text{dept}}\mathcal{G}_{\texttt{sum(salary)}}R_1$$

Translation:

```
Select sum(salary) as dept_total
From   R1
```

```
Group  by dept
```

You can also group would qualify having {some condition}.

1.3 Set Operations

- Union
- Intersect
- Minus/except

We **cannot** check if something is or isn't in a set (set membership test). We also **cannot** check if something exists or not (existence test).

Example 20. `Employee(Id, name, dept, dept_location, salary)`

```
Select    sum(salary) as dept_total
From      Employee
Group     by dept
```

$$\equiv {}_{\texttt{dept}}\mathcal{G}_{\texttt{sum(salary)}}\text{Employee}$$

Example 21. Given these three tables:

```
Sailors(Sid, name, rating, age)
Boats(Bid, name, colour)
Reserves(foreign: Sid, foreign: Bid, date)
```

Find the name of the sailor who has reserved boat with `Bid`=103.

```
Select     name
From       Sailors S, Reserved R
where      Bid=103 and S.sid=R.sid
```

Find the `Sid`'s of sailors who reserved a red boat.

```
Select     Sid
From       Boats B, Reserved R
where      B.bid=R.bid and colour='red'
```

```
Select     name
From       Sailor S, Boats B, Reserved R
where      S.sid=R.sid and B.bid=R.bid and colour='red'
```

Find the `colour`'s of boats reserved by Tom.

```
Select     code
From       Sailor S, Boats B, Reserved R
where      S.sid=R.sid and B.bid=R.bid and S.name='Tom'
```

Find the `color` of boats reserved by a sailor where `name` starts with a 'T'. We use something like this:

```
S.name     like 'T%'
```

where % is 0 to any arbitrary characters - a wildcard.

Find the name of the of sailors who have reserved a red boat but NOT a green boat.

```
SELECT     name
FROM       Sailors S, Boats B, Reserved R
WHERE      S.sid=R.bid and
           B.bid=R.bid and
           colour='red'

minus (or except)

SELECT     name
FROM       Sailors S, Boats B, Reserved R
WHERE      S.sid=R.bid and
           B.bid=R.bid and
           colour='green'
```

Find the name of a sailor who has reserved a boat with bid=103.

```
SELECT     name
FROM       Sailor S
WHERE      S.sid  IN
               SELECT     Sid
               FROM       Reserved
               WHERE      bid=103
```

Set Comparison Operators:

- Any
- ALL
- Some
- Every

To be used with $>$, $<$, $\leq$, $\geq$, $\neq$.

Example 22. Find the same of sailors whose rating is greater than some sailor called Tom.

```
SELECT      name
FROM        Sailor S
WHERE       S.rating > Any
                ( SELECT     S2.rating
                  FROM       Sailor S2
                  WHERE      S2.name = 'Tom' ) // also S2.name like 'T%'
```

Example 23. Find sailors who have reserved all the boats.

```
SELECT      B.bid
FROM        Boats B

SELECT      R.bid
FROM        Reserves R
```

```
WHERE       R.sid=S.sid
```

Example 24. Find the name of the oldest sailor:

- This is 2 steps:
    - Find the oldest sailor
    - Fnd the name of the oldest sailor

```
SELECT      name
FROM        Sailor

WHERE       age=(SELECT    max(age)
                 FROM      Sailor)
```

1.4 Null Values

- Aggregate functions → ignore null values
- Null ⇒ unknown

Three Value Logic:

- `True`, `False`, `unknown`
- `OR`
    - `T or unknown ⇒ True`
    - `False or unknown ⇒ unknown`
    - `unknown or unknown ⇒ unknown`
- `AND`
    - `T and unknown ⇒ unknown`
    - `False and unknown ⇒ False`
    - `unknown and unknown ⇒ unknown`
- `NOT`
    - `Not(unknown) ⇒ unknown`

Example 25. Full, left, right outer joins

Sailors

Sid	name	rating	age
22	Tom	7	22
31	John	10	15
58	Peter	7	40

Reserves

Sid	bid	date
22	30	10/10/15
31	40	10/11/15
32	50	12/11/15

Then:

```
Select      S.sid, R.bid
```

```
From       Sailors S {natural left outer join} Reserves R
```

The result:

```
22   30
31   40
22   50
58   null
```

1.5 Updating Tables

Example 26.

```
Update      Employee
Set         name='Tom'
Where       eid=1234
```

Options:

1. Restrict → default
2. Cascade
3. Set default
4. Set null

How to determine a good database design?

The main goal is this:

- **NO DUPLICATES!!!**

Problem with duplicates:

- Redundant storage
- Update, delete and insert anomalies

Functional Dependencies (FD)

Let R be a relation and let X, Y be non-empty sets of attributes in R.

FD $X \to Y$ is satisfied if for every tuple t_i and t_k in R if

$$t_i X = t_k X \Longrightarrow t_i Y = t_k Y$$

(Constraints on a legal set of relations)

Armstrong's Axiom

Reflexivity

$$\begin{array}{c} A \ \to \ A \\ AB \to A \quad \text{or} \quad AB \to B \end{array}$$

Augmentation

$$\begin{aligned} A &\rightarrow B \\ x\,A &\rightarrow x\,B \end{aligned}$$

Transitive

$$\begin{aligned} A &\rightarrow B \\ B &\rightarrow C \\ A &\rightarrow C \end{aligned}$$

Union

$$\begin{aligned} X \rightarrow Y \quad , \quad X \rightarrow Z \\ X \rightarrow YZ \end{aligned}$$

Decomposition

$$\begin{aligned} X &\rightarrow YZ \\ X \rightarrow Y \quad , \quad X \rightarrow Z \end{aligned}$$

Functional dependencies are used for:

- Testing relations to see if they are legal under a given set of functional dependencies
- Specify constraints on set of legal relations

Closures of FD's

- Denoted by FD^+
- The closure is all FD that can be generated
- Ex. $R = (A, B, C)$, where $F = \{A \rightarrow B, B \rightarrow C\} \longrightarrow$ Find F^+
- Union: if $a \rightarrow b$, and $a \rightarrow c$, then $a \rightarrow b\,c$
- Decomposition: if $a \rightarrow b\,c$, then $a \rightarrow b$ and $a \rightarrow c$
- Pseudotransitivity: if $a \rightarrow b$ and $b\,c \rightarrow d$, then $a\,c \rightarrow d$

Let F be the set of FD's. The closure of F is denoted as F^+.

Note: All FD's implied by F is F^+.

Example 27. If $R(A, B, C, D)$, we can break it up into $R(A, B, C)$ and $R(C, D)$, we can break it up into

$$\begin{aligned} F_1 &= \\ F_1^+ &= \\ F_2 &= \\ F_2^+ &= \end{aligned}$$

Where $F_1^+ \equiv F_2^+$. The decomposition **preserves** all FD.

Attribute Closure

Example 28. $F = \{A \to B, B \to C\}$. Find the attribute closure of A, B and C.

$$\begin{aligned} A^+ &= A \\ &= A\,B \rightsquigarrow \text{because } A \to B \\ &= A\,B\,C \rightsquigarrow \text{because } B \to C \\ B^+ &= B \\ &= B\,C \rightsquigarrow \text{because } B \to C \\ C^+ &= C \end{aligned}$$

From the attribute closure, we can determine the key. The key is A because $A^+ = R$.

To check if a given FD $B \to A$ is in F^+, check if A is in B^+.

Example 29. $R = (A, B, C, G, H, I)$ with $F = \{A \to B, A \to C, C\,G \to H, C\,G \to I\}$. Find a key in R (primary).

$$\begin{aligned} A^+ &= A \\ &= A\,B \rightsquigarrow \because A \to B \\ &= A\,B\,H \rightsquigarrow \because A \to H \\ &= A\,B\,C\,H \rightsquigarrow \because A \to C \\ G^+ &= G \\ I^+ &= I \\ A\,G^+ &= A\,G \\ &= A\,B\,G \\ &= A\,B\,G\,H \\ &= A\,B\,C\,G\,H \\ &= A\,B\,C\,G\,H\,I \\ \therefore A\,G^+ &= R \end{aligned}$$

Normal Forms

1st Normal Form → 1NF

- Every attribute contains only atomic values

2nd Normal Forms → 2NF

- In 1NF and every $n\,m$-key attribute must be fully functionally dependent of the entire key

Boyce-Codd Normal Form → BCNF

- R is in BCNF if and only if for every $X \to A$ in F, where $X \to F$ is a trivial FD *or* X is a superkey of R

Example 30. If $R(A, B, C)$ and $F = \{A \to B, B \to C\}$.

Is R in BCNF? $A \to B$. Is A a superkey of R? Yes, because $A \to B$ and $B \to C \Rightarrow A \to C$.

$B \to C$

$$\begin{aligned} B^+ &= B \\ &= BC \rightsquigarrow \therefore B \to C \\ B^+ &\neq R \end{aligned}$$

This is because B is not a superkey, $\therefore$ R is not in BCNF.

Example 31. Based on a DB of some people:

$$\begin{aligned} \text{FD} &= \text{id} \to \text{name, rank, hours worked} \\ &\quad\ \text{rank} \to \text{hourly wage} \end{aligned}$$

We have E(id, name, rank, hours worked, hourly wage) and then R_1(id, name, rank, hours worked) and R_2(rank, hourly wage).

BCNF has **NO** data redundancy.

Example 32. Decompose a relation into relations in BCNF. $R = (C, S, J, D, P, Q, V)$. C is the key. $SD \to P$, $J \to S$.

R is in 2NF, but not BCNF.

Chapter 8: Relational Database Design

Database System Concepts, 6th Ed.

' Silberschatz, Korth and Sudarshan

Chapter 8: Relational Database Design

- Features of Good Relational Design
- Atomic Domains and First Normal Form
- Decomposition Using Functional Dependencies
- Functional Dependency Theory
- Algorithms for Functional Dependencies
- Decomposition Using Multivalued Dependencies
- More Normal Form
- Database-Design Process
- Modeling Temporal Data

Combine Schemas?

- Suppose we combine *instructor* and *department* into *inst_dept*
  - *(No connection to relationship set inst_dept)*
- Result is possible repetition of information

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

A Combined Schema Without Repetition

- Consider combining relations
  - *sec_class(sec_id, building, room_number)* and
  - *section(course_id, sec_id, semester, year)*

  into one relation
  - *section(course_id, sec_id, semester, year, building, room_number)*
- No repetition in this case

What About Smaller Schemas?

- Suppose we had started with *inst_dept.* How would we know to split up (**decompose**) it into *instructor* and *department*?
- Write a rule "if there were a schema (*dept_name, building, budget*), then *dept_name* would be a candidate key"
- Denote as a **functional dependency**:

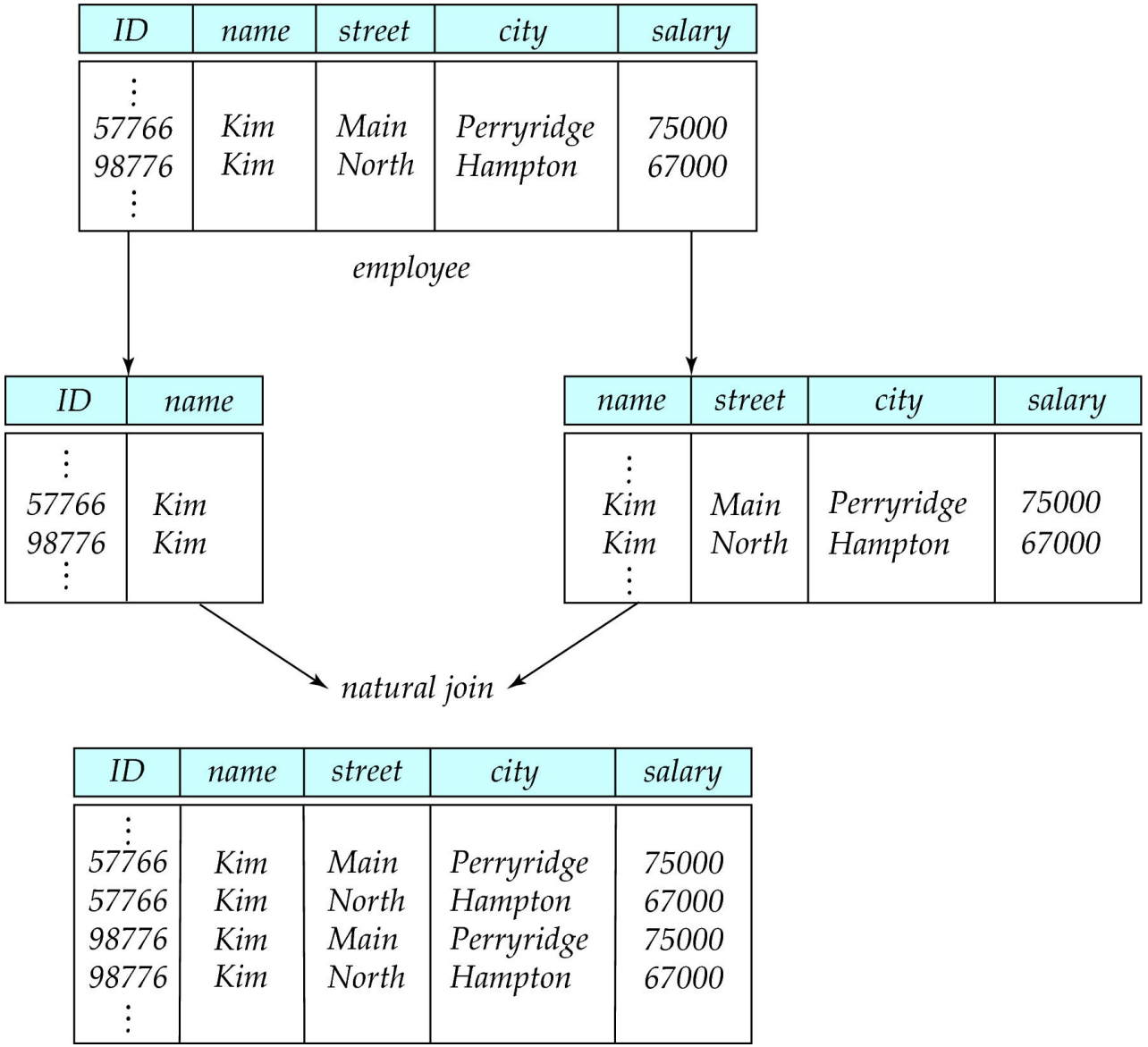
  dept_name → *building, budget*

- In *inst_dept*, because *dept_name* is not a candidate key, the building and budget of a department may have to be repeated.
  - This indicates the need to decompose *inst_dept*
- Not all decompositions are good. Suppose we decompose *employee(ID, name, street, city, salary)* into

  employee1 (*ID*, *name*)

  employee2 (*name*, *street, city, salary*)
- The next slide shows how we lose information -- we cannot reconstruct the original *employee* relation -- and so, this is a **lossy decomposition**.

A Lossy Decomposition



Example of Lossless-Join Decomposition

- **Lossless join decomposition**
- Decomposition of $R = (A, B, C)$

  $R_1 = (A, B)$ $\quad$ $R_2 = (B, C)$

A	B	C
α	1	A
β	2	B

r

A	B
α	1
β	2

$\Pi_{A,B}(r)$

B	C
1	A
2	B

$\Pi_{B,C}(r)$

$\Pi_A (r) \bowtie \Pi_B (r)$

A	B	C
α	1	A
β	2	B

First Normal Form

- Domain is **atomic** if its elements are considered to be indivisible units
  - Examples of non-atomic domains:
    - Set of names, composite attributes
    - Identification numbers like CS101 that can be broken up into parts
- A relational schema R is in **first normal form** if the domains of all attributes of R are atomic
- Non-atomic values complicate storage and encourage redundant (repeated) storage of data
  - Example: Set of accounts stored with each customer, and set of owners stored with each account
  - We assume all relations are in first normal form (and revisit this in Chapter 22: Object Based Databases)

First Normal Form (Cont'd)

- Atomicity is actually a property of how the elements of the domain are used.
  - Example: Strings would normally be considered indivisible
  - Suppose that students are given roll numbers which are strings of the form *CS0012* or *EE1127*
  - If the first two characters are extracted to find the department, the domain of roll numbers is not atomic.
  - Doing so is a bad idea: leads to encoding of information in application program rather than in the database.

Goal — Devise a Theory for the Following

- Decide whether a particular relation R is in "good" form.
- In the case that a relation R is not in "good" form, decompose it into a set of relations $\{R_1, R_2, ..., R_n\}$ such that
  - each relation is in good form
  - the decomposition is a lossless-join decomposition
- Our theory is based on:
  - functional dependencies
  - multivalued dependencies

Functional Dependencies

- Constraints on the set of legal relations.
- Require that the value for a certain set of attributes determines uniquely the value for another set of attributes.
- A functional dependency is a generalization of the notion of a *key.*

Functional Dependencies (Cont.)

- Let R be a relation schema

$$\alpha \subseteq R \quad and \quad \beta \subseteq R$$

- The **functional dependency**

$$\alpha \rightarrow \beta$$

holds on R if and only if for any legal relations r(R), whenever any two tuples t_1 and t_2 of r agree on the attributes α, they also agree on the attributes β. That is,

$$t_1[\alpha] = t_2[\alpha] \quad \Rightarrow \quad t_1[\beta] = t_2[\beta]$$

- Example: Consider r(A,B) with the following instance of r.

1	4
1	5
3	7

- On this instance, $A \rightarrow B$ does **NOT** hold, but $B \rightarrow A$ does hold.

Functional Dependencies (Cont.)

- K is a superkey for relation schema R if and only if $K \rightarrow R$
- K is a candidate key for R if and only if
  - $K \rightarrow R$, and
  - for no $\alpha \subset K$, $\alpha \rightarrow R$
- Functional dependencies allow us to express constraints that cannot be expressed using superkeys. Consider the schema:

  inst_dept (*ID, name, salary, dept_name, building, budget*).

  We expect these functional dependencies to hold:

  dept_name → *building*

  and *ID* → *building*

  but would not expect the following to hold:

  dept_name → *salary*

Use of Functional Dependencies

- We use functional dependencies to:
  - test relations to see if they are legal under a given set of functional dependencies.
    - If a relation *r* is legal under a set *F* of functional dependencies, we say that *r* **satisfies** *F.*
  - specify constraints on the set of legal relations
    - We say that *F* **holds on** *R* if all legal relations on *R* satisfy the set of functional dependencies *F.*
- Note: A specific instance of a relation schema may satisfy a functional dependency even if the functional dependency does not hold on all legal instances.
  - For example, a specific instance of *instructor* may, by chance, satisfy
    name $\rightarrow$ *ID.*

Functional Dependencies (Cont.)

- *A* functional dependency is **trivial** if it is satisfied by all instances of a relation
  - Example*:*
    - *ID, name* → *ID*
    - *name* → *name*
  - In general, $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$

Closure of a Set of Functional Dependencies

- Given a set F of functional dependencies, there are certain other functional dependencies that are logically implied by F.
  - For example: If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
- The set of **all** functional dependencies logically implied by F is the **closure** of F.
- We denote the *closure* of F by $\mathbf{F^+}$.
- F^+ is a superset of F.

Boyce-Codd Normal Form

A relation schema R is in BCNF with respect to a set F of functional dependencies if for all functional dependencies in F^+ of the form

$$\alpha \rightarrow \beta$$

where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
- α is a superkey for R

Example schema *not* in BCNF:

instr_dept (*ID, name, salary, dept_name, building, budget*)

because *dept_name* → *building, budget*
holds on *instr_dept,* but *dept_name* is not a superkey

Decomposing a Schema into BCNF

- Suppose we have a schema *R* and a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF.

  We decompose *R* into:

  - $(\alpha \cup \beta)$
  - $(R - (\beta - \alpha))$

- In our example,

  - α = *dept_name*
  - β = *building, budget*

  and *inst_dept* is replaced by

  - $(\alpha \cup \beta)$ = (*dept_name, building, budget*)
  - $(R - (\beta - \alpha))$ = (*ID, name, salary, dept_name*)

BCNF and Dependency Preservation

- Constraints, including functional dependencies, are costly to check in practice unless they pertain to only one relation
- If it is sufficient to test only those dependencies on each individual relation of a decomposition in order to ensure that *all* functional dependencies hold, then that decomposition is *dependency preserving.*
- Because it is not always possible to achieve both BCNF and dependency preservation, we consider a weaker normal form, known as *third normal form.*

Third Normal Form

- A relation schema R is in **third normal form (3NF)** if for all:

  $$\alpha \rightarrow \beta \text{ in } F^+$$

  at least one of the following holds:
  - $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \in \alpha$)
  - α is a superkey for R
  - Each attribute A in $\beta - \alpha$ is contained in a candidate key for R.

    (**NOTE**: each attribute may be in a different candidate key)
- If a relation is in BCNF it is in 3NF (since in BCNF one of the first two conditions above must hold).
- Third condition is a minimal relaxation of BCNF to ensure dependency preservation (will see why later).

Goals of Normalization

- Let R be a relation scheme with a set F of functional dependencies.
- Decide whether a relation scheme R is in "good" form.
- In the case that a relation scheme R is not in "good" form, decompose it into a set of relation scheme $\{R_1, R_2, ..., R_n\}$ such that
  - each relation scheme is in good form
  - the decomposition is a lossless-join decomposition
  - Preferably, the decomposition should be dependency preserving.

How good is BCNF?

- There are database schemas in BCNF that do not seem to be sufficiently normalized
- Consider a relation

  inst_info (ID, child_name, phone)

  - where an instructor may have more than one phone and can have multiple children

ID	*child_name*	*phone*
99999 99999 99999 99999	David David William Willian	512-555-1234 512-555-4321 512-555-1234 512-555-4321

inst_info

How good is BCNF? (Cont.)

- There are no non-trivial functional dependencies and therefore the relation is in BCNF
- Insertion anomalies – i.e., if we add a phone 981-992-3443 to 99999, we need to add two tuples

(99999, David, 981-992-3443)
(99999, William, 981-992-3443)

How good is BCNF? (Cont.)

- Therefore, it is better to decompose *inst_info* into:

inst_child

ID	*child_name*
99999	David
99999	David
99999	William
99999	Willian

inst_phone

ID	*phone*
99999	512-555-1234
99999	512-555-4321
99999	512-555-1234
99999	512-555-4321

This suggests the need for higher normal forms, such as Fourth Normal Form (4NF), which we shall see later.

Functional-Dependency Theory

- We now consider the formal theory that tells us which functional dependencies are implied logically by a given set of functional dependencies.
- We then develop algorithms to generate lossless decompositions into BCNF and 3NF
- We then develop algorithms to test if a decomposition is dependency-preserving

Closure of a Set of Functional Dependencies

- Given a set F set of functional dependencies, there are certain other functional dependencies that are logically implied by F.
  - For e.g.: If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
- The set of **all** functional dependencies logically implied by F is the **closure** of F.
- We denote the *closure* of F by F^+.

Closure of a Set of Functional Dependencies

- We can find F^+, the closure of F, by repeatedly applying **Armstrong's Axioms:**
  - if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ **(reflexivity)**
  - if $\alpha \rightarrow \beta$, then $\gamma\ \alpha \rightarrow \gamma\ \beta$ **(augmentation)**
  - if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ **(transitivity)**
- These rules are
  - **sound** (generate only functional dependencies that actually hold), and
  - **complete** (generate all functional dependencies that hold).

Example

- $R = (A, B, C, G, H, I)$
  $F = \{ A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- some members of F^+
  - $A \rightarrow H$
    - by transitivity from $A \rightarrow B$ *and* $B \rightarrow H$
  - $AG \rightarrow I$
    - by augmenting $A \rightarrow C$ with G, to get $AG \rightarrow CG$
      and then transitivity with $CG \rightarrow I$
  - $CG \rightarrow HI$
    - by augmenting $CG \rightarrow I$ to infer $CG \rightarrow CGI$,
      and augmenting of $CG \rightarrow H$ to infer $CGI \rightarrow HI$,
      and then transitivity

Procedure for Computing F^+

- To compute the closure of a set of functional dependencies F:

$F^+ = F$
repeat
  for each functional dependency f in F^+
    apply reflexivity and augmentation rules on f
    add the resulting functional dependencies to F^+
  for each pair of functional dependencies f_1 and f_2 in F^+
    if f_1 and f_2 can be combined using transitivity
      then add the resulting functional dependency to F^+
until F^+ does not change any further

NOTE: We shall see an alternative procedure for this task later

Closure of Functional Dependencies (Cont.)

- Additional rules:
  - If $\alpha \rightarrow \beta$ holds *a*nd $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\,\gamma$ holds **(union)**
  - If $\alpha \rightarrow \beta\,\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds **(decomposition)**
  - If $\alpha \rightarrow \beta$ holds *a*nd $\gamma\ \beta \rightarrow \delta$ holds, then $\alpha\ \gamma \rightarrow \delta$ holds **(pseudotransitivity)**

  The above rules can be inferred from Armstrong's axioms.

Closure of Attribute Sets

- Given a set of attributes α, define the ***closure*** of α **under** *F* (denoted by α^+) as the set of attributes that are functionally determined by α under *F*

- Algorithm to compute α^+, the closure of α under *F*

```
result := α;
while (changes to result) do
        for each β → γ in F do
            begin
                if β ⊆ result then  result := result ∪ γ
            end
```

Example of Attribute Set Closure

- $R = (A, B, C, G, H, I)$
- $F = \{A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- $(AG)^+$
  1. $result = AG$
  2. $result = ABCG$ $(A \rightarrow C$ and $A \rightarrow B)$
  3. $result = ABCGH$ $(CG \rightarrow H$ and $CG \subseteq AGBC)$
  4. $result = ABCGHI$ $(CG \rightarrow I$ and $CG \subseteq AGBCH)$
- Is AG a candidate key?
  1. Is AG a super key?
     1. Does $AG \rightarrow R$? == Is $(AG)^+ \supseteq R$
  2. Is any subset of AG a superkey?
     1. Does $A \rightarrow R$? == Is $(A)^+ \supseteq R$
     2. Does $G \rightarrow R$? == Is $(G)^+ \supseteq R$

Uses of Attribute Closure

There are several uses of the attribute closure algorithm:

- Testing for superkey:
  - To test if α is a superkey, we compute α^+, and check if α^+ contains all attributes of *R*.
- Testing functional dependencies
  - To check if a functional dependency $\alpha \rightarrow \beta$ holds (or, in other words, is in F^+), just check if $\beta \subseteq \alpha^+$.
  - That is, we compute α^+ by using attribute closure, and then check if it contains β.
  - Is a simple and cheap test, and very useful
- Computing closure of F
  - For each $\gamma \subseteq R$, we find the closure γ^+, and for each $S \subseteq \gamma^+$, we output a functional dependency $\gamma \rightarrow S$.

Canonical Cover

- Sets of functional dependencies may have redundant dependencies that can be inferred from the others
  - For example: $A \rightarrow C$ is redundant in: $\{A \rightarrow B, \quad B \rightarrow C, A \rightarrow C\}$
  - Parts of a functional dependency may be redundant
    - E.g.: on RHS: $\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow CD\}$ can be simplified to

      $\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow D\}$
    - E.g.: on LHS: $\{A \rightarrow B, \quad B \rightarrow C, \quad AC \rightarrow D\}$ can be simplified to

      $\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow D\}$
- Intuitively, a canonical cover of F is a “minimal” set of functional dependencies equivalent to F, having no redundant dependencies or redundant parts of dependencies

Extraneous Attributes

- Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F.
  - Attribute A is **extraneous** in α if $A \in \alpha$
    and F logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
  - Attribute A is **extraneous** in β if $A \in \beta$
    and the set of functional dependencies
    $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies F.
- *Note:* implication in the opposite direction is trivial in each of the cases above, since a "stronger" functional dependency always implies a weaker one
- Example: Given $F = \{A \rightarrow C, AB \rightarrow C\}$
  - B is extraneous in $AB \rightarrow C$ because $\{A \rightarrow C, AB \rightarrow C\}$ logically implies $A \rightarrow C$ (I.e. the result of dropping B from $AB \rightarrow C$).
- Example: Given $F = \{A \rightarrow C, AB \rightarrow CD\}$
  - C is extraneous in $AB \rightarrow CD$ since $AB \rightarrow C$ can be inferred even after deleting C

Testing if an Attribute is Extraneous

- Consider a set *F* of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in *F*.
- To test if attribute $A \in \alpha$ is extraneous in α
  1. compute $(\{\alpha\} - A)^+$ using the dependencies in *F*
  2. check that $(\{\alpha\} - A)^+$ contains β; if it does, *A* is extraneous in α
- To test if attribute $A \in \beta$ is extraneous in β
  1. compute α^+ using only the dependencies in
     $F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$,
  2. check that α^+ contains *A;* if it does, *A* is extraneous in β

Canonical Cover

- A **canonical cover** for F is a set of dependencies F_c such that
  - F logically implies all dependencies in F_c, and
  - F_c logically implies all dependencies in F, and
  - No functional dependency in F_c contains an extraneous attribute, and
  - Each left side of functional dependency in F_c is unique.
- To compute a canonical cover for F:

  repeat

  Use the union rule to replace any dependencies in F

  $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1 \beta_2$

  Find a functional dependency $\alpha \rightarrow \beta$ with an

  extraneous attribute either in α or in β

  /* Note: test for extraneous attributes done using F_c, not F*/

  If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$

  until F does not change
- Note: Union rule may become applicable after some extraneous attributes have been deleted, so it has to be re-applied

Computing a Canonical Cover

- $R = (A, B, C)$
  $F = \{A \rightarrow BC$
  $B \rightarrow C$
  $A \rightarrow B$
  $AB \rightarrow C\}$
- Combine $A \rightarrow BC$ and $A \rightarrow B$ into $A \rightarrow BC$
  - Set is now $\{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$
- A is extraneous in $AB \rightarrow C$
  - Check if the result of deleting A from $AB \rightarrow C$ is implied by the other dependencies
    - Yes: in fact, $B \rightarrow C$ is already present!
  - Set is now $\{A \rightarrow BC, B \rightarrow C\}$
- C is extraneous in $A \rightarrow BC$
  - Check if $A \rightarrow C$ is logically implied by $A \rightarrow B$ and the other dependencies
    - Yes: using transitivity on $A \rightarrow B$ *and* $B \rightarrow$ C.
      - Can use attribute closure of A in more complex cases
- The canonical cover is: $A \rightarrow B$
  $B \rightarrow C$

Lossless-join Decomposition

- For the case of $R = (R_1, R_2)$, we require that for all possible relations r on schema R

$$r = \prod_{R1}(r) \bowtie \prod_{R2}(r)$$

- A decomposition of R into R_1 and R_2 is lossless join if at least one of the following dependencies is in F^+:
  - $R_1 \cap R_2 \rightarrow R_1$
  - $R_1 \cap R_2 \rightarrow R_2$
- The above functional dependencies are a sufficient condition for lossless join decomposition; the dependencies are a necessary condition only if all constraints are functional dependencies

Example

- $R = (A, B, C)$
  $F = \{A \rightarrow B, B \rightarrow C)$
  - Can be decomposed in two different ways
- $R_1 = (A, B), \quad R_2 = (B, C)$
  - Lossless-join decomposition:

    $$R_1 \cap R_2 = \{B\} \text{ and } B \rightarrow BC$$

  - Dependency preserving
- $R_1 = (A, B), \quad R_2 = (A, C)$
  - Lossless-join decomposition:

    $$R_1 \cap R_2 = \{A\} \text{ and } A \rightarrow AB$$

  - Not dependency preserving
    (cannot check $B \rightarrow C$ without computing $R_1 \bowtie R_2$)

Dependency Preservation

- Let F_i be the set of dependencies F^+ that include only attributes in R_i.
  - A decomposition is **dependency preserving**, if

    $(F_1 \cup F_2 \cup \ldots \cup F_n)^+ = F^+$
  - If it is not, then checking updates for violation of functional dependencies may require computing joins, which is expensive.

Testing for Dependency Preservation

- To check if a dependency $\alpha \rightarrow \beta$ is preserved in a decomposition of R into R_1, R_2, …, R_n we apply the following test (with attribute closure done with respect to F)
  - *result* = α
    while (changes to *result*) do
      for each R_i in the decomposition
        $t = (result \cap R_i)^+ \cap R_i$
        $result = result \cup t$
  - If *result* contains all attributes in β, then the functional dependency
    $\alpha \rightarrow \beta$ is preserved.
- We apply the test on all dependencies in F to check if a decomposition is dependency preserving
- This procedure takes polynomial time, instead of the exponential time required to compute F^+ and $(F_1 \cup F_2 \cup \ldots \cup F_n)^+$

Example

- $R = (A, B, C)$
  $F = \{A \rightarrow B$
  $B \rightarrow C\}$
  Key = $\{A\}$
- R is not in BCNF
- Decomposition $R_1 = (A, B)$, $R_2 = (B, C)$
  - R_1 and R_2 in BCNF
  - Lossless-join decomposition
  - Dependency preserving

Testing for BCNF

- To check if a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF
  1. compute α^+ (the attribute closure of α), and
  2. verify that it includes all attributes of R, that is, it is a superkey of R.
- **Simplified test**: To check if a relation schema R is in BCNF, it suffices to check only the dependencies in the given set F for violation of BCNF, rather than checking all dependencies in F^+.
  - If none of the dependencies in F causes a violation of BCNF, then none of the dependencies in F^+ will cause a violation of BCNF either.
- However, **simplified test using only F is incorrect when testing a relation in a decomposition of R**
  - Consider $R = (A, B, C, D, E)$, with $F = \{ A \rightarrow B, BC \rightarrow D\}$
    - Decompose R into $R_1 = (A,B)$ and $R_2 = (A,C,D, E)$
    - Neither of the dependencies in F contain only attributes from (A,C,D,E) so we might be mislead into thinking R_2 satisfies BCNF.
    - In fact, dependency $AC \rightarrow D$ in F^+ shows R_2 is not in BCNF.

Testing Decomposition for BCNF

- To check if a relation R_i in a decomposition of R is in BCNF,
  - Either test R_i for BCNF with respect to the **restriction** of F to R_i (that is, all FDs in F^+ that contain only attributes from R_i)
  - or use the original set of dependencies F that hold on R, but with the following test:
    - for every set of attributes $\alpha \subseteq R_i$, check that α^+ (the attribute closure of α) either includes no attribute of $R_i - \alpha$, or includes all attributes of R_i.
    - If the condition is violated by some $\alpha \rightarrow \beta$ in F, the dependency
      $\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i$
      can be shown to hold on R_i, and R_i violates BCNF.
    - We use above dependency to decompose R_i

BCNF Decomposition Algorithm

result := {*R* };
done := false;
compute F^+;
while (not *done)* **do**
  if (there is a schema R_i in *result* that is not in BCNF)
    then begin
      let $\alpha \rightarrow \beta$ be a nontrivial functional dependency that
        holds on R_i such that $\alpha \rightarrow R_i$ is not in F^+,
          and $\alpha \cap \beta = \varnothing$;
        result := (*result* – R_i) $\cup$ (R_i – β) $\cup$ (α, β);
      end
    else *done* := **true;**

Note: each R_i is in BCNF, and decomposition is lossless-join.

Example of BCNF Decomposition

- $R = (A, B, C)$
  $F = \{A \rightarrow B$
  $B \rightarrow C\}$
  Key = $\{A\}$
- R is not in BCNF ($B \rightarrow C$ but B is not superkey)
- Decomposition
  - $R_1 = (B, C)$
  - $R_2 = (A, B)$

Example of BCNF Decomposition

- *class* (*course_id*, *title*, *dept_name*, *credits*, *sec_id*, *semester*, *year*, *building*, *room_number*, *capacity*, *time_slot_id*)
- Functional dependencies:
  - *course_id*→ *title*, *dept_name*, *credits*
  - *building*, *room_number*→*capacity*
  - *course_id*, *sec_id*, *semester*, *year*→*building*, *room_number*, *time_slot_id*
- A candidate key {*course_id*, *sec_id*, *semester*, *year*}.
- BCNF Decomposition:
  - *course_id*→ *title*, *dept_name*, *credits* holds
    - but *course_id* is not a superkey.
  - We replace *class* by:
    - *course*(*course_id*, *title*, *dept_name*, *credits*)
    - *class-1* (*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number, capacity*, *time_slot_id*)

BCNF Decomposition (Cont.)

- *course* is in BCNF
  - How do we know this?
- *building*, *room_number*→*capacity* holds on *class-1*
  - but {*building*, *room_number*} is not a superkey for *class-1*.
  - We replace *class-1* by:
    - *classroom* (*building*, *room_number*, *capacity*)
    - *section* (*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, *time_slot_id*)
- *classroom* and *section* are in BCNF.

BCNF and Dependency Preservation

It is not always possible to get a BCNF decomposition that is dependency preserving

- $R = (J, K, L)$
  $F = \{JK \rightarrow L$
  $L \rightarrow K\}$
  Two candidate keys = JK and JL
- R is not in BCNF
- Any decomposition of R will fail to preserve

  $$JK \rightarrow L$$

  This implies that testing for $JK \rightarrow L$ requires a join

Third Normal Form: Motivation

- There are some situations where
  - BCNF is not dependency preserving, and
  - efficient checking for FD violation on updates is important
- Solution: define a weaker normal form, called Third Normal Form (3NF)
  - Allows some redundancy (with resultant problems; we will see examples later)
  - But functional dependencies can be checked on individual relations without computing a join.
  - There is always a lossless-join, dependency-preserving decomposition into 3NF.

3NF Example

- Relation *dept_advisor*:
  - *dept_advisor* (*s_ID, i_ID, dept_name)*
    F = {*s_ID, dept_name* → *i_ID*, *i_ID* → *dept_name*}
  - Two candidate keys: *s_ID, dept_name,* and *i_ID, s_ID*
  - *R* is in 3NF
    - *s_ID, dept_name* → *i_ID* *s_ID*
      - *dept_name* is a superkey
    - *i_ID* → *dept_name*
      - *dept_name* is contained in a candidate key

Redundancy in 3NF

- There is some redundancy in this schema
- Example of problems due to redundancy in 3NF
  - $R = (J, K, L)$
    $F = \{JK \rightarrow L, L \rightarrow K\}$

J	L	K
j_1	l_1	k_1
j_2	l_1	k_1
j_3	l_1	k_1
null	l_2	k_2

- repetition of information (e.g., the relationship l_1, k_1)
  - (*i_ID, dept_name)*
- need to use null values (e.g., to represent the relationship l_2, k_2 where there is no corresponding value for J).
  - (*i_ID, dept_nameI)* if there is no separate relation mapping instructors to departments

Testing for 3NF

- Optimization: Need to check only FDs in F, need not check all FDs in F^+.
- Use attribute closure to check for each dependency $\alpha \rightarrow \beta$, if α is a superkey.
- If α is not a superkey, we have to verify if each attribute in β is contained in a candidate key of R
  - this test is rather more expensive, since it involve finding candidate keys
  - testing for 3NF has been shown to be NP-hard
  - Interestingly, decomposition into third normal form (described shortly) can be done in polynomial time

3NF Decomposition Algorithm

Let F_c be a canonical cover for F;
$i := 0$;
for each functional dependency $\alpha \rightarrow \beta$ in F_c **do**
if none of the schemas R_j, $1 \leq j \leq i$ contains $\alpha \beta$
then begin
$i := i + 1$;
$R_i := \alpha \beta$
end
if none of the schemas R_j, $1 \leq j \leq i$ contains a candidate key for R
then begin
$i := i + 1$;
R_i := any candidate key for R;
end
/* Optionally, remove redundant relations */
repeat
if any schema R_j is contained in another schema R_k
then /* delete R_j */
$R_j = R$;;
$i=i-1$;
return $(R_1, R_2, ..., R_i)$

3NF Decomposition Algorithm (Cont.)

- Above algorithm ensures:
  - each relation schema R_i is in 3NF
  - decomposition is dependency preserving and lossless-join
  - Proof of correctness is at end of this presentation (click here)

3NF Decomposition: An Example

- Relation schema:

  cust_banker_branch = (*customer_id, employee_id, branch_name, type*)

- The functional dependencies for this relation schema are:
  1. *customer_id, employee_id* → *branch_name, type*
  2. *employee_id* → *branch_name*
  3. *customer_id, branch_name* → *employee_id*
- We first compute a canonical cover
  - *branch_name* is extraneous in the r.h.s. of the 1st dependency
  - No other attribute is extraneous, so we get F_C =

    customer_id, employee_id → *type*
    employee_id → *branch_name*
    customer_id, branch_name → *employee_id*

3NF Decompsition Example (Cont.)

- The **for** loop generates following 3NF schema:

  (*customer_id, employee_id, type*)

  (*employee_id, branch_name*)

  (*customer_id, branch_name, employee_id)*

  - Observe that (*customer_id, employee_id, type*) contains a candidate key of the original schema, so no further relation schema needs be added
- At end of for loop, detect and delete schemas, such as (*employee_id, branch_name*), which are subsets of other schemas
  - result will not depend on the order in which FDs are considered
- The resultant simplified 3NF schema is:

  (*customer_id, employee_id, type*)

  (*customer_id, branch_name, employee_id)*

Comparison of BCNF and 3NF

- It is always possible to decompose a relation into a set of relations that are in 3NF such that:
  - the decomposition is lossless
  - the dependencies are preserved
- It is always possible to decompose a relation into a set of relations that are in BCNF such that:
  - the decomposition is lossless
  - it may not be possible to preserve dependencies.

Design Goals

- Goal for a relational database design is:
  - BCNF.
  - Lossless join.
  - Dependency preservation.
- If we cannot achieve this, we accept one of
  - Lack of dependency preservation
  - Redundancy due to use of 3NF
- Interestingly, SQL does not provide a direct way of specifying functional dependencies other than superkeys.

  Can specify FDs using assertions, but they are expensive to test, (and currently not supported by any of the widely used databases!)
- Even if we had a dependency preserving decomposition, using SQL we would not be able to efficiently test a functional dependency whose left hand side is not a key.

Multivalued Dependencies

- Suppose we record names of children, and phone numbers for instructors:
  - *inst_child*(*ID*, *child_name*)
  - *inst_phone*(*ID*, *phone_number*)
- If we were to combine these schemas to get
  - *inst_info*(*ID*, *child_name*, *phone_number*)
  - Example data:
    (99999, David, 512-555-1234)
    (99999, David, 512-555-4321)
    (99999, William, 512-555-1234)
    (99999, William, 512-555-4321)
- This relation is in BCNF
  - Why?

Multivalued Dependencies (MVDs)

- Let R be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$. The **multivalued dependency**

$$\alpha \twoheadrightarrow \beta$$

holds on R if in any legal relation $r(R)$, for all pairs for tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, there exist tuples t_3 and t_4 in r such that:

$$\begin{aligned}
t_1[\alpha] &= t_2[\alpha] = t_3[\alpha] = t_4[\alpha] \\
t_3[\beta] &= t_1[\beta] \\
t_3[R - \beta] &= t_2[R - \beta] \\
t_4[\beta] &= t_2[\beta] \\
t_4[R - \beta] &= t_1[R - \beta]
\end{aligned}$$

MVD (Cont.)

- Tabular representation of $\alpha \twoheadrightarrow \beta$

	α	β	$R - \alpha - \beta$
t_1	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$a_{j+1} \ldots a_n$
t_2	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$b_{j+1} \ldots b_n$
t_3	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$b_{j+1} \ldots b_n$
t_4	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$a_{j+1} \ldots a_n$

Example

- Let R be a relation schema with a set of attributes that are partitioned into 3 nonempty subsets.

  $$Y, Z, W$$

- We say that $Y \twoheadrightarrow Z$ (Y **multidetermines** Z) if and only if for all possible relations $r(R)$

  $$< y_1, z_1, w_1 > \in r \text{ and } < y_1, z_2, w_2 > \in r$$

  then

  $$< y_1, z_1, w_2 > \in r \text{ and } < y_1, z_2, w_1 > \in r$$

- Note that since the behavior of Z and W are identical it follows that $Y \twoheadrightarrow Z$ if $Y \twoheadrightarrow W$

Example (Cont.)

- In our example:

$$ID \twoheadrightarrow child\_name$$
$$ID \twoheadrightarrow phone\_number$$

- The above formal definition is supposed to formalize the notion that given a particular value of Y (ID) it has associated with it a set of values of Z *(child_name)* and a set of values of W *(phone_number)*, and these two sets are in some sense independent of each other.
- Note:
  - If $Y \rightarrow Z$ then $Y \twoheadrightarrow Z$
  - Indeed we have (in above notation) $Z_1 = Z_2$
    The claim follows.

Use of Multivalued Dependencies

- We use multivalued dependencies in two ways:
  1. To test relations to **determine** whether they are legal under a given set of functional and multivalued dependencies
  2. To specify **constraints** on the set of legal relations. We shall thus concern ourselves *only* with relations that satisfy a given set of functional and multivalued dependencies.
- If a relation *r* fails to satisfy a given multivalued dependency, we can construct a relations *r* ´ that does satisfy the multivalued dependency by adding tuples to *r.*

Theory of MVDs

- From the definition of multivalued dependency, we can derive the following rule:
  - If $\alpha \rightarrow \beta$, then $\alpha \twoheadrightarrow \beta$

  That is, every functional dependency is also a multivalued dependency
- The **closure** D^+ of D is the set of all functional and multivalued dependencies logically implied by D.
  - We can compute D^+ from D, using the formal definitions of functional dependencies and multivalued dependencies.
  - We can manage with such reasoning for very simple multivalued dependencies, which seem to be most common in practice
  - For complex dependencies, it is better to reason about sets of dependencies using a system of inference rules (see Appendix C).

Fourth Normal Form

- A relation schema R is in **4NF** with respect to a set D of functional and multivalued dependencies if for all multivalued dependencies in D^+ of the form $\alpha \rightarrow\!\!\!\rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following hold:
  - $\alpha \rightarrow\!\!\!\rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$ or $\alpha \cup \beta = R$)
  - α is a superkey for schema R
- If a relation is in 4NF it is in BCNF

Restriction of Multivalued Dependencies

- The restriction of D to R_i is the set D_i consisting of
  - All functional dependencies in D^+ that include only attributes of R_i
  - All multivalued dependencies of the form

    $\alpha \rightarrow\!\!\rightarrow (\beta \cap R_i)$

    where $\alpha \subseteq R_i$ and $\alpha \rightarrow\!\!\rightarrow \beta$ is in D^+

4NF Decomposition Algorithm

result: = {*R*};
done := false;
compute D+;
Let D_i denote the restriction of D+ to R_i

while (**not** *done*)
  if (there is a schema $\mathbf{R}_i$ in *result* that is not in 4NF) **then**
    begin
      let $\alpha \rightarrow\!\!\!\!\rightarrow \beta$ be a nontrivial multivalued dependency that holds on R_i such that $\alpha \rightarrow R_i$ is not in D_i, and $\alpha \cap \beta = \phi$;
      result := (*result* - R_i) $\cup$ (R_i - β) $\cup$ (α, β);
    end
  else *done*:= true;

Note: each R_i is in 4NF, and decomposition is lossless-join

Example

- $R = (A, B, C, G, H, I)$

  $F = \{ A \twoheadrightarrow B$

  $B \twoheadrightarrow HI$

  $CG \twoheadrightarrow H \}$
- R is not in 4NF since $A \twoheadrightarrow B$ and A is not a superkey for R
- Decomposition

  a) $R_1 = (A, B)$ (R_1 is in 4NF)

  b) $R_2 = (A, C, G, H, I)$ (R_2 is not in 4NF, decompose into R_3 and R_4)

  c) $R_3 = (C, G, H)$ (R_3 is in 4NF)

  d) $R_4 = (A, C, G, I)$ (R_4 is not in 4NF, decompose into R_5 and R_6)
  - $A \twoheadrightarrow B$ and $B \twoheadrightarrow HI$ ➔ $A \twoheadrightarrow HI$, (MVD transitivity), and
  - and hence $A \twoheadrightarrow I$ *(MVD restriction to R_4)*

  e) $R_5 = (A, I)$ (R_5 is in 4NF)

  f) $R_6 = (A, C, G)$ (R_6 is in 4NF)

Further Normal Forms

- **Join dependencies** generalize multivalued dependencies
  - lead to **project-join normal form (PJNF)** (also called **fifth normal form**)
- A class of even more general constraints, leads to a normal form called **domain-key normal form**.
- Problem with these generalized constraints: are hard to reason with, and no set of sound and complete set of inference rules exists.
- Hence rarely used

Overall Database Design Process

- We have assumed schema *R* is given
  - *R* could have been generated when converting E-R diagram to a set of tables.
  - *R* could have been a single relation containing *all* attributes that are of interest (called **universal relation**).
  - Normalization breaks *R* into smaller relations.
  - *R* could have been the result of some ad hoc design of relations, which we then test/convert to normal form.

ER Model and Normalization

- When an E-R diagram is carefully designed, identifying all entities correctly, the tables generated from the E-R diagram should not need further normalization.
- However, in a real (imperfect) design, there can be functional dependencies from non-key attributes of an entity to other attributes of the entity
  - Example: an *employee* entity with attributes
    department_name and *building*,
    and a functional dependency
    department_name → *building*
  - Good design would have made department an entity
- Functional dependencies from non-key attributes of a relationship set possible, but rare --- most relationships are binary

Denormalization for Performance

- May want to use non-normalized schema for performance
- For example, displaying *prereqs* along with *course_id,* and *title* requires join of *course* with *prereq*
- Alternative 1: Use denormalized relation containing attributes of *course* as well as *prereq* with all above attributes
  - faster lookup
  - extra space and extra execution time for updates
  - extra coding work for programmer and possibility of error in extra code
- Alternative 2: use a materialized view defined as
  course ⋈ *prereq*
  - Benefits and drawbacks same as above, except no extra coding work for programmer and avoids possible errors

Other Design Issues

- Some aspects of database design are not caught by normalization
- Examples of bad database design, to be avoided:

  Instead of *earnings* (*company_id, year, amount*), use
  - *earnings_2004, earnings_2005, earnings_2006*, etc., all on the schema (*company_id, earnings*).
    - Above are in BCNF, but make querying across years difficult and needs new table each year
  - *company_year* (*company_id, earnings_2004, earnings_2005, earnings_2006*)
    - Also in BCNF, but also makes querying across years difficult and requires new attribute each year.
    - Is an example of a **crosstab**, where values for one attribute become column names
    - Used in spreadsheets, and in data analysis tools

Modeling Temporal Data

- **Temporal data** have an association time interval during which the data are *valid.*
- A **snapshot** is the value of the data at a particular point in time
- Several proposals to extend ER model by adding valid time to
  - attributes, e.g., address of an instructor at different points in time
  - entities, e.g., time duration when a student entity exists
  - relationships, e.g., time during which an instructor was associated with a student as an advisor.
- But no accepted standard
- Adding a temporal component results in functional dependencies like

  $ID \rightarrow street, city$

  not to hold, because the address varies over time
- A **temporal functional dependency** X → Y holds on schema *R* if the functional dependency X → Y holds on all snapshots for all legal instances r (*R*).

Modeling Temporal Data (Cont.)

- In practice, database designers may add start and end time attributes to relations
  - E.g., *course*(*course_id, course_title*) is replaced by

    course(*course_id, course_title, start, end*)
    - Constraint: no two tuples can have overlapping valid times
      - Hard to enforce efficiently
- Foreign key references may be to current version of data, or to data at a point in time
  - E.g., student transcript should refer to course information at the time the course was taken

End of Chapter

Proof of Correctness of 3NF Decomposition Algorithm

Correctness of 3NF Decomposition Algorithm

- 3NF decomposition algorithm is dependency preserving (since there is a relation for every FD in F_c)
- Decomposition is lossless
  - A candidate key (C) is in one of the relations R_i in decomposition
  - Closure of candidate key under F_c must contain all attributes in R.
  - Follow the steps of attribute closure algorithm to show there is only one tuple in the join result for each tuple in R_i

Correctness of 3NF Decomposition Algorithm (Cont'd.)

Claim: if a relation R_i is in the decomposition generated by the above algorithm, then R_i satisfies 3NF.

- Let R_i be generated from the dependency $\alpha \rightarrow \beta$
- Let $\gamma \rightarrow B$ be any non-trivial functional dependency on R_i. (We need only consider FDs whose right-hand side is a single attribute.)
- Now, B can be in either β or α but not in both. Consider each case separately.

Correctness of 3NF Decomposition (Cont'd.)

- Case 1: If B in β:
  - If γ is a superkey, the 2nd condition of 3NF is satisfied
  - Otherwise α must contain some attribute not in γ
  - Since $\gamma \rightarrow B$ is in F^+ it must be derivable from F_c, by using attribute closure on γ.
  - Attribute closure not have used $\alpha \rightarrow \beta$. If it had been used, α must be contained in the attribute closure of γ, which is not possible, since we assumed γ is not a superkey.
  - Now, using $\alpha \rightarrow (\beta - \{B\})$ and $\gamma \rightarrow B$, we can derive $\alpha \rightarrow B$

    (since $\gamma \subseteq \alpha\ \beta$, and $B \notin \gamma$ since $\gamma \rightarrow B$ is non-trivial)
  - Then, B is extraneous in the right-hand side of $\alpha \rightarrow \beta$; which is not possible since $\alpha \rightarrow \beta$ is in F_c.
  - Thus, if B is in β then γ must be a superkey, and the second condition of 3NF must be satisfied.

Correctness of 3NF Decomposition (Cont'd.)

- Case 2: *B* is in α.
  - Since α is a candidate key, the third alternative in the definition of 3NF is trivially satisfied.
  - In fact, we cannot show that γ is a superkey.
  - This shows exactly why the third alternative is present in the definition of 3NF.

Q.E.D.

Figure 8.02

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

Figure 8.03

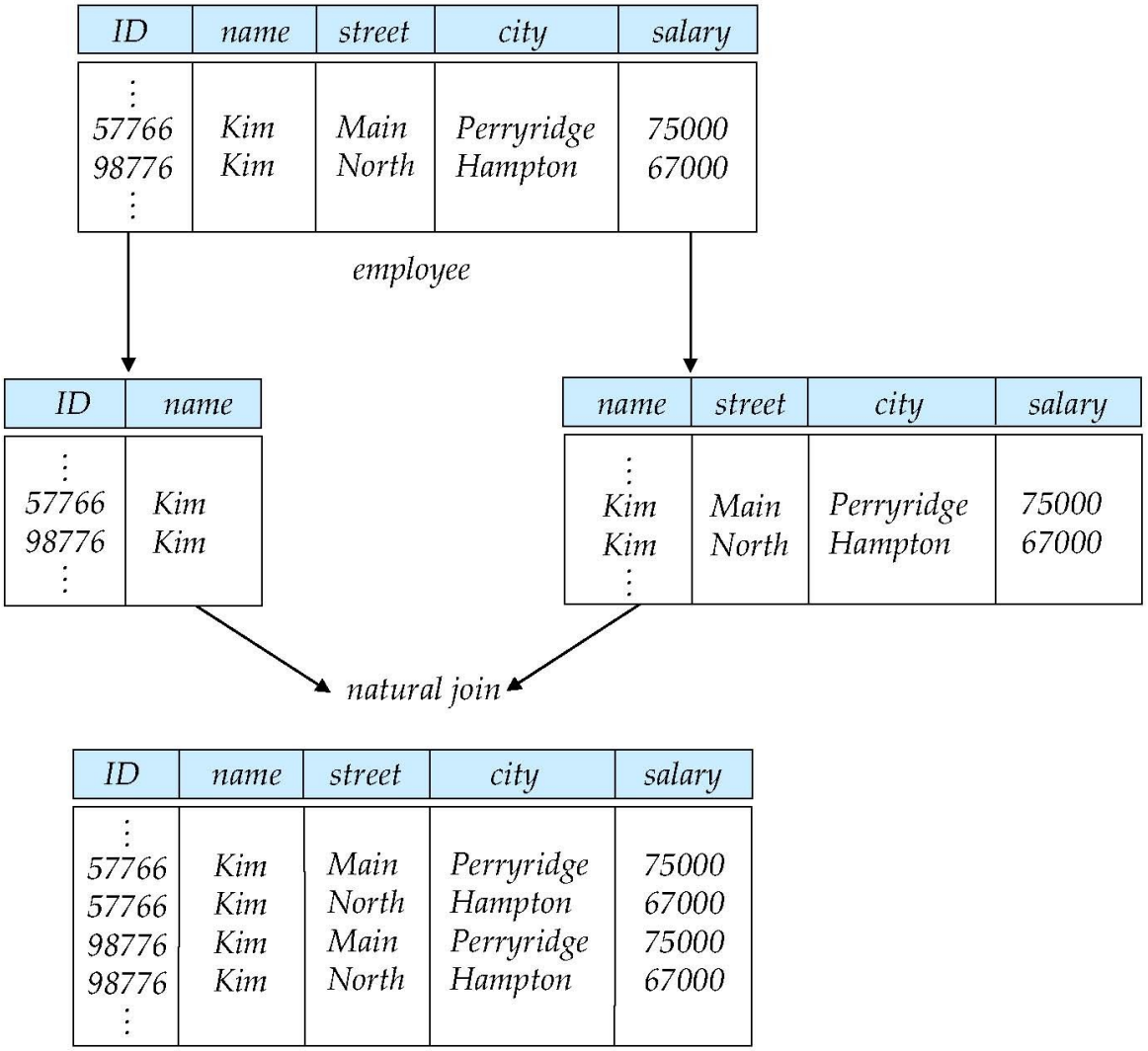

ID
name
street
city
salary
57766
98776
Kim
Kim
Main
North
Perryridge
Hampton
75000
67000
employee
ID
name
57766
98776
Kim
Kim
name
street
city
salary
Kim
Kim
Main
North
Perryridge
Hampton
75000
67000
natural join
ID
name
street
city
salary
57766
57766
98776
98776
Kim
Kim
Kim
Kim
Main
North
Main
North
Perryridge
Hampton
Perryridge
Hampton
75000
67000
75000
67000

Figure 8.04

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_1	d_2
a_2	b_2	c_2	d_2
a_2	b_3	c_2	d_3
a_3	b_3	c_2	d_4

Figure 8.05

building	*room_number*	*capacity*
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

Figure 8.06

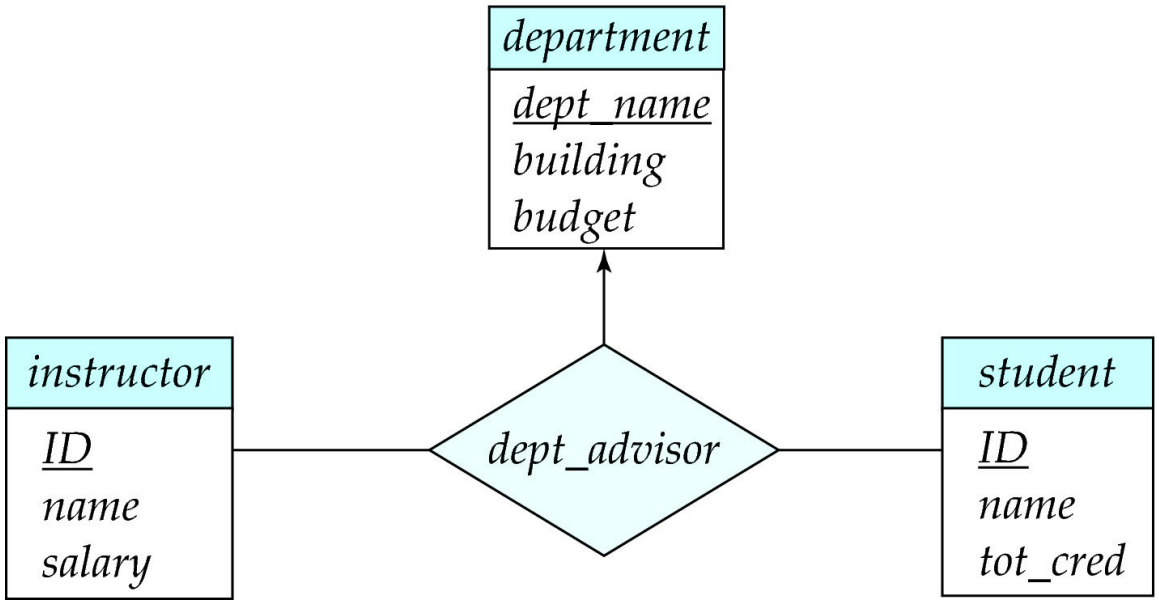

department
dept_name
building
budget
instructor
ID
name
salary
dept_advisor
student
ID
name
tot_cred

Figure 8.14

dept_name	*ID*	*street*	*city*
Physics	22222	North	Rye
Physics	22222	Main	Manchester
Finance	12121	Lake	Horseneck

Figure 8.15

dept_name	*ID*	*street*	*city*
Physics	22222	North	Rye
Math	22222	Main	Manchester

Figure 8.17

A	B	C
a_1	b_1	c_1
a_1	b_1	c_2
a_2	b_1	c_1
a_2	b_1	c_3