Review of lectures 1-10

CS348 Spring 2023

Instructor: Sujaya Maiyya

Sections: **002 and 004 only**

Announcements

- Assignment 2: Due on June 20th
  - Late policy: 5% penalty per 24 hours

- Project Milestone 1: Due on June 22nd
  - Late policy: 25% penalty per 24 hours

- Midterm: On June 26th
  - Everything until lecture 10 (except lecture 6 on advance SQL)

Relational data model

- A database is a collection of relations (or tables)
- Each relation has a set of attributes (or columns)
- Each attribute has a unique name and a domain (or type)
  - The domains are required to be **atomic**

Single, indivisible piece of information

- Each relation contains a set of tuples (or rows)
  - Each tuple has a value for each attribute of the relation
  - Duplicate tuples are not allowed
    - Two tuples are duplicates if they agree on all attributes

☞Simplicity is a virtue!

Types of integrity constraints

- Tuple-level
  - Domain restrictions, attribute comparisons, etc.
    - E.g. *age* cannot be negative
    - E.g. for flights table, arrival time > take off time
- Relation-level
  - Key constraints (focus in this lecture)
    - E.g. *uid* should be unique in the *User* relation
  - Functional dependencies (Textbook, Ch. 7)
- Database-level
  - Referential integrity – foreign key (focus in this lecture)
    - *uid* in *Member* must refer to a row in *User* with the same *uid*

Key (Candidate Key)

Def: A set of attributes K for a relation R if

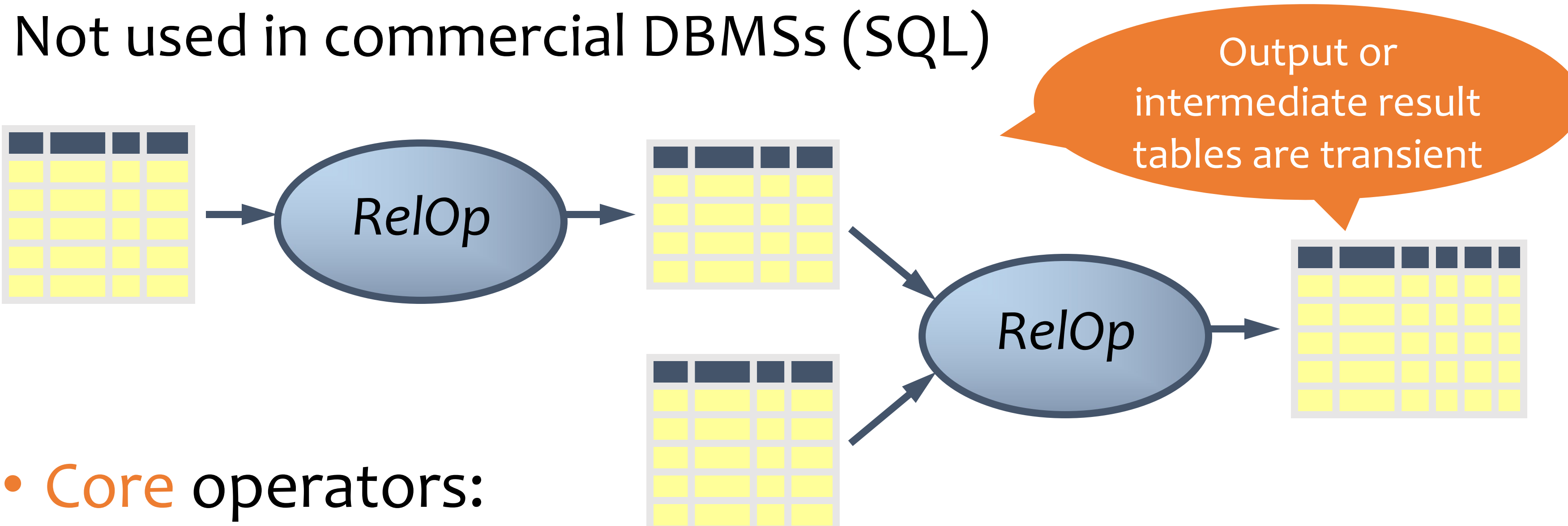
- Condition 1: In no instance of R will two different tuples agree on all attributes of K
  - That is, K can serve as a "tuple identifier"
- Condition 2: No proper subset of K satisfies the above condition
  - That is, K is minimal
- Example: *User* (*uid*, *name*, *age*, *pop*)
  - *uid* is a key of *User*
  - *age* is not a key (not an identifier)
  - {*uid*, *name*} is not a key (not minimal), but a superkey

Satisfies only Condition 1

- One candidate key is assigned to be primary key

Relational algebra

- A language for querying relational data based on “operators”
- Not used in commercial DBMSs (SQL)



- Core operators:
  - Selection, projection, cross product, union, difference, and renaming
- Additional, derived operators:
  - Join, natural join, intersection, etc.
- Compose operators to make complex queries

Operators can only be applied one row at a time

- You must be able to evaluate the condition over each single row of the input table!
  - Example: the most popular user

$$\sigma_{pop \geq every\ pop\ in\ User}\ User$$

WRONG!

Summary of operators

Core Operators

1. Selection: $\sigma_p R$
2. Projection: $\pi_L R$
3. Cross product: $R \times S$
4. Union: $R \cup S$
5. Difference: $R - S$
6. Renaming: $\rho_{S(A_1 \to A_1', A_2 \to A_2', \ldots)} R$

Note: Only use these operators for assignments & exams

Derived Operators

1. Join: $R \bowtie_p S$
2. Natural join: $R \bowtie S$
3. Intersection: $R \cap S$

Why do we need core operator X?

- Difference
  - The only non-monotone operator
- Projection
  - The only operator that removes columns
- Cross product
  - The only operator that adds columns
- Union
  - The only operator that adds rows
- Selection
  - The only operator that conditionally removes rows

Expression tree notation

IDs of users who belong to at least two groups

$$\pi_{uid_1}$$

$$\bowtie_{uid_1 = uid_2 \land gid_1 \neq gid_2}$$

$$\rho_{(uid \to uid_1, gid \to gid_1)} \qquad \rho_{(uid \to uid_2, gid \to gid_2)}$$

$$Member \qquad Member$$

A trickier example

User (<u>*uid*</u> int, *name* string, *age* int, *pop* float)
Group (<u>*gid*</u> string, *name* string)
Member (<u>*uid*</u> int, <u>*gid*</u> string)

- Who are the most popular?
  - Who do NOT have the highest *pop* rating?
  - Whose *pop* is lower than somebody else's?

A trickier example

User (<u>*uid*</u> int, *name* string, *age* int, *pop* float)
Group (<u>*gid*</u> string, *name* string)
Member (<u>*uid*</u> int, <u>*gid*</u> string)

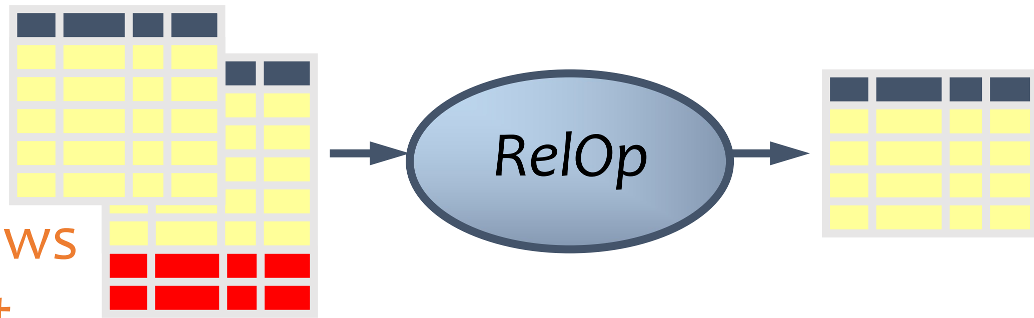
- Who are the most popular?
  - Who do NOT have the highest *pop* rating?
  - Whose *pop* is lower than somebody else's?

$$\pi_{uid}\, User \;-\; \pi_{User_1.uid}\left(\rho_{User_1} User \bowtie_{User_1.pop<User_2.pop} \rho_{User_2} User\right)$$

A deeper question:
When (and why) is "−" needed?

Non-monotone operators

Add more rows to the input…



What happens to the output?

- If some old output rows may become invalid → the operator is non-monotone

- Example: difference operator $R - S$

uid	gid
123	gov
857	abc

R

−

uid	gid
123	gov
901	edf
857	**abc**

S

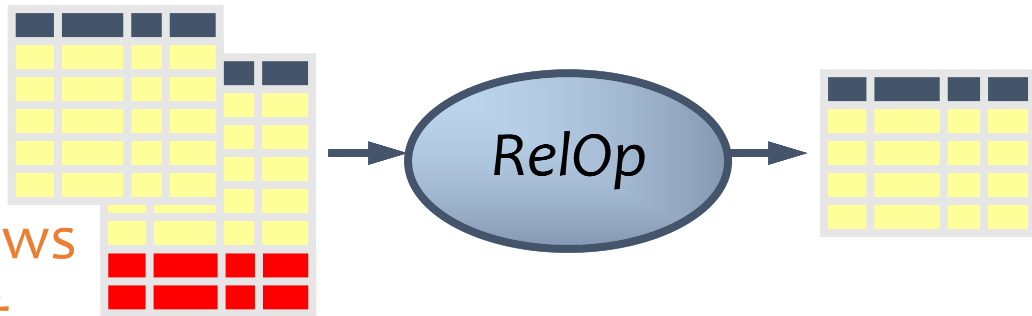
=

uid	gid
~~857~~	~~abc~~

This old row becomes invalid because the new row added to S

Non-monotone operators

Add more rows to the input…



What happens to the output?

- If some old output rows may become invalid (causing some row removal) → the operator is non-monotone
- Otherwise (old output rows always remain "correct") → the operator is monotone

uid	*gid*
123	gov
857	abc
189	**abc**

R

−

uid	*gid*
123	gov
901	edf

S

=

uid	*gid*
857	abc
189	**abc**

This old row is always valid no matter what rows are added to R

Classification of relational operators

• Selection: $\sigma_p R$	Monotone
• Projection: $\pi_L R$	Monotone
• Cross product: $R \times S$	Monotone
• Join: $R \bowtie_p S$	Monotone
• Natural join: $R \bowtie S$	Monotone
• Union: $R \cup S$	Monotone
• Difference: $R - S$	Monotone w.r.t. R; non-monotone w.r.t S
• Intersection: $R \cap S$	Monotone

SQL (lectures 3-6)

DDL

User (<u>*uid*</u> int, *name* string, *age* int, *pop* float)
Group (<u>*gid*</u> string, *name* string)
Member (<u>*uid*</u> int, <u>*gid*</u> string)

- CREATE TABLE *table_name* (..., *column_name column_type*, ...);

```
CREATE TABLE User(uid INT, name VARCHAR(30), age INT, pop DECIMAL(3,2));
CREATE TABLE Group (gid CHAR(10), name VARCHAR(100));
CREATE TABLE Member (uid INT, gid CHAR(10));
```

- DROP TABLE *table_name*;

```
DROP TABLE User;
DROP TABLE Group;
DROP TABLE Member;
```

Drastic action: deletes ALL info about the table, not just the contents

Basic queries for DML: SFW statement

- SELECT $A_1, A_2, \ldots, A_n$
  FROM $R_1, R_2, \ldots, R_m$
  WHERE $condition$;

- Also called an SPJ (select-project-join) query

- Corresponds to (but not really equivalent to) relational algebra query:
  $$\pi_{A_1, A_2, \ldots, A_n}\left(\sigma_{condition}(R_1 \times R_2 \times \cdots \times R_m)\right)$$

SQL set and bag operations

- Set: UNION, EXCEPT, INTERSECT
  - Exactly like set ∪, −, and ∩ in relational algebra
  - Duplicates in input tables, if any, are first eliminated
  - Duplicates in result are also eliminated (for UNION)

Bag1

fruit
apple
apple
orange

Bag2

fruit
orange
orange
orange

```
(SELECT * FROM Bag1)
UNION
(SELECT * FROM Bag2);
```

fruit
apple
orange

```
(SELECT * FROM Bag1)
EXCEPT
(SELECT * FROM Bag2);
```

fruit
apple

```
(SELECT * FROM Bag1)
INTERSECT
(SELECT * FROM Bag2);
```

fruit
orange

SQL set and bag operations

- Set: UNION, EXCEPT, INTERSECT
  - Exactly like set ∪, −, and ∩ in relational algebra
- Bag: UNION ALL, EXCEPT ALL, INTERSECT ALL
  - Think of each row as having an implicit count (the number of times it appears in the table)

Bag1

fruit
apple
apple
orange

apple: 2
orange:1

Bag2

fruit
apple
orange
orange

apple: 1
orange:2

```
(SELECT * FROM Bag1)
EXCEPT ALL
(SELECT * FROM Bag2);
```

fruit
apple

apple: 1
orange:0

proper-subtract the two counts

Set versus bag operations

Poke (uid1, uid2, timestamp)

- uid1 poked uid2 at timestamp

Question: How do these two queries differ?

```
Q1:
(SELECT uid1 FROM Poke)
EXCEPT
(SELECT uid2 FROM Poke);
```

```
Q2:
(SELECT uid1 FROM Poke)
EXCEPT ALL
(SELECT uid2 FROM Poke);
```

Users who poked others but never got poked by others

Users who poked others more than others poked them

Table subqueries

- Use query result as a table
  - In set and bag operations, FROM clauses, etc.

- Example: names of users who poked others more than others poked them

```
SELECT DISTINCT name
FROM User,
     (SELECT uid1 as uid FROM Poke)
     EXCEPT ALL
     (SELECT uid2 as uid FROM Poke) AS T
WHERE User.uid = T.uid;
```

WITH clause

- The WITH clause provides a way of defining a temporary relation whose definition is available only to the query in which the with clause occurs

```
WITH max_pop(popVal) AS (SELECT
max(pop) FROM user)
SELECT uid, name FROM user, max_pop
WHERE user.pop = max_pop.popVal
```

```
WITH max_pop AS (SELECT max(pop) AS
popVal FROM user)
SELECT uid, name FROM user, max_pop
WHERE user.pop = max_pop.popVal
```

- Supported by many but not all DBMSs
- Can be written using subqueries

IN subqueries

- x IN ($subquery$) checks if x is in the result of $subquery$

- Example: users at the same age as (some) Bart

```
SELECT *
FROM User,
WHERE age IN (SELECT age
              FROM User
              WHERE name = 'Bart');
```

EXISTS subqueries

- EXISTS (*subquery*) checks if the result of *subquery* is non-empty
- Example: users at the same age as (some) Bart

```
SELECT *
FROM User AS u,
WHERE EXISTS (SELECT * FROM User
              WHERE name = 'Bart'
                AND age = u.age);
```

  - This happens to be a correlated subquery—a subquery that references tuple variables in surrounding queries

Aggregates

- Standard SQL aggregate functions: COUNT, SUM, AVG, MIN, MAX

- Example: number of users under 18, and their average popularity
  - COUNT(*) counts the number of rows

```
SELECT COUNT(*), AVG(pop)
FROM User
WHERE age <18;
```

COUNT (*)	*AVG (pop)*
6	0.625

Grouping

- SELECT … FROM … WHERE …
  GROUP BY *list_of_columns*;

- Example: compute average popularity for each age group

```
SELECT age, AVG(pop)
FROM User
GROUP BY age;
```

Example of computing GROUP BY

```
SELECT age, AVG(pop) FROM User GROUP BY age;
```

uid	*name*	*age*	*pop*
142	Bart	10	0.9
857	Lisa	8	0.7
123	Milhouse	10	0.2
456	Ralph	8	0.3

Compute GROUP BY: group rows according to the values of GROUP BY columns

uid	*name*	*age*	*pop*
142	Bart	10	0.9
123	Milhouse	10	0.2
857	Lisa	8	0.7
456	Ralph	8	0.3

Compute SELECT for each group

age	*avg_pop*
10	0.55
8	0.50

HAVING examples

- Used to filter groups based on the group properties (e.g., aggregate values, GROUP BY column values)
- List the average popularity for each age group with more than a hundred users

```
SELECT age, AVG(pop)
FROM User
GROUP BY age
HAVING COUNT(*)>100;
```

  - Can be written using WHERE and table subqueries

```
SELECT T.age, T.apop
FROM (SELECT age, AVG(pop) AS apop, COUNT(*) AS gsize
        FROM User GROUP BY age) AS T
WHERE T.gsize>100;
```

ORDER BY example

- List all users, sort them by popularity (descending) and name (ascending)

```
SELECT uid, name, age, pop
FROM User
ORDER BY pop DESC, name;
```

  - ASC is the default option
  - Strictly speaking, only output columns can appear in ORDER BY clause (although some DBMS support more)

Three-valued logic to handle NULL

TRUE = 1, FALSE = 0, UNKNOWN = 0.5

x AND y = $\min(x, y)$

x OR y = $\max(x, y)$

NOT x = $1 - x$

x	y	x AND y	x OR y	NOT x
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	UNKNOWN	UNKNOWN	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
UNKNOWN	TRUE	UNKNOWN	TRUE	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN
UNKNOWN	FALSE	FALSE	UNKNOWN	UNKNOWN
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	UNKNOWN	FALSE	UNKNOWN	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

- Comparing a NULL with another value (including another NULL) using =, >, etc., the result is NULL
- WHERE and HAVING clauses only select rows for output if the condition evaluates to TRUE
  - NULL is not enough
- Aggregate functions ignore NULL, except COUNT(*)

Unfortunate consequences

- Q1a = Q1b?

```
Q1a. SELECT AVG(pop) FROM User;
```

```
Q1b. SELECT SUM(pop)/COUNT(*) FROM User;
```

- Q2a = Q2b?

```
Q2a. SELECT * FROM User;
```

```
Q2b SELECT * FROM User WHERE pop=pop;
```

- Be careful: NULL breaks many equivalences
- Use IS NULL or NOT NULL for null comparisons

Outerjoin examples

Group ⟗ *Member*

gid	*name*	*uid*
abc	Book Club	857
gov	Student Government	123
gov	Student Government	857
dps	Dead Putting Society	142
nuk	**United Nuclear Workers**	**NULL**
foo	**NULL**	**789**

Group

gid	*name*
abc	Book Club
gov	Student Government
dps	Dead Putting Society
nuk	United Nuclear Workers

Member

uid	*gid*
142	dps
123	gov
857	abc
857	gov
789	foo

A full outerjoin between R and S:

- All rows in the result of $R \bowtie S$, plus
- "Dangling" R rows (those that do not join with any S rows) padded with NULL's for S's columns
- "Dangling" S rows (those that do not join with any R rows) padded with NULL's for R's columns

Outerjoin examples

Group

gid	*name*
abc	Book Club
gov	Student Government
dps	Dead Putting Society
nuk	United Nuclear Workers

Member

uid	*gid*
142	dps
123	gov
857	abc
857	gov
789	foo

Group ⟕ *Member*

gid	*name*	*uid*
abc	Book Club	857
gov	Student Government	123
gov	Student Government	857
dps	Dead Putting Society	142
nuk	United Nuclear Workers	NULL

- A left outerjoin ($R ⟕ S$) includes rows in $R \bowtie S$ plus dangling R rows padded with NULL's

Group ⟖ *Member*

gid	*name*	*uid*
abc	Book Club	857
gov	Student Government	123
gov	Student Government	857
dps	Dead Putting Society	142
foo	NULL	789

- A right outerjoin ($R ⟖ S$) includes rows in $R \bowtie S$ plus dangling S rows padded with NULL's

Outerjoin syntax

```
SELECT * FROM Group LEFT OUTER JOIN Member
        ON Group.gid = Member.gid;
```

$\approx Group \mathbin{⟕}_{Group.gid=Member.gid} Member$

```
SELECT * FROM Group RIGHT OUTER JOIN Member
        ON Group.gid = Member.gid;
```

$\approx Group \mathbin{⟖}_{Group.gid=Member.gid} Member$

```
SELECT * FROM Group FULL OUTER JOIN Member
        ON Group.gid = Member.gid;
```

$\approx Group \mathbin{⟗}_{Group.gid=Member.gid} Member$

☞A similar construct exists for regular ("inner") joins:

```
SELECT * FROM Group JOIN Member ON Group.gid = Member.gid;
```

Theta join: gid is repeated

Natural join: gid appears once

☞For natural joins, add keyword NATURAL; don't use ON

```
SELECT * FROM Group NATURAL JOIN Member;
```

Insert/Delete/Update

- Insert one row
  - User 789 joins Dead Putting Society

```
INSERT INTO Member VALUES (789, 'dps');
```

- Delete everything from a table

```
DELETE FROM Member;
```

- Delete according to a WHERE condition
  - Example: User 789 leaves Dead Putting Society

```
DELETE FROM Member WHERE uid=789 AND gid=‘dps’;
```

- Update: User 142 changes name to “Barney”

```
UPDATE User
SET name = 'Barney’
WHERE uid = 142;
```

Types of SQL constraints

- NOT NULL
- Key
- Referential integrity (foreign key)
- General assertion
- Tuple- and attribute-based CHECK’s

NOT NULL & Key constraint examples

```
CREATE TABLE User
(uid INT NOT NULL,
 name VARCHAR(30) NOT NULL,
 twitterid VARCHAR(15) NOT NULL,
 age INT,
 pop DECIMAL(3,2));
```

```
CREATE TABLE User
(uid INT NOT NULL PRIMARY KEY,
 name VARCHAR(30) NOT NULL,
 twitterid VARCHAR(15) NOT NULL UNIQUE,
 age INT,
 pop DECIMAL(3,2));
```

At most one primary key per table

Any number of UNIQUE keys per table

```
CREATE TABLE Member
(uid INT NOT NULL,
 gid CHAR(10) NOT NULL,
PRIMARY KEY(uid,gid));
```

This form is required for multi-attribute keys

Referential integrity in SQL

- Referenced column(s) must be PRIMARY KEY
- Referencing column(s) form a FOREIGN KEY
- Example

Some system allow them to be non-PK but must be UNIQUE

```
CREATE TABLE Member
(uid INT NOT NULL REFERENCES User(uid),
 gid CHAR(10) NOT NULL,
PRIMARY KEY(uid,gid),
FOREIGN KEY (gid) REFERENCES Group(gid));
```

This form is required for multi-attribute foreign keys

```
CREATE TABLE MemberBenefits
(.....
FOREIGN KEY (uid,gid) REFERENCES Member(uid,gid));
```

Enforcing referential integrity

Example: *Member.uid references User.uid*

- Delete or update a *User* row whose *uid* is referenced by some *Member* row
  - Reject or ON DELETE CASCADE or ON DELETE SET NULL

User

uid	*name*	*…*
142	Bart	…
123	Milhouse	…
857	Lisa	…
~~456~~	~~Ralph~~	…
789	Nelson	…
…	…	…

Option 1: Reject

Member

uid	*gid*
142	dps
123	gov
857	abc
857	gov
~~456~~	~~abc~~
~~456~~	~~gov~~
…	….

```
CREATE TABLE Member
(uid INT NOT NULL
REFERENCES User(uid)
ON DELETE CASCADE,
…..);
```

Option 2: Cascade (ripple changes to all referring rows)

General assertion

- CREATE ASSERTION *assertion_name* CHECK *assertion_condition*;
- *assertion_condition* is checked for each modification that could potentially violate it
- Example: *Member.uid* references *User.uid*

```
CREATE ASSERTION MemberUserRefIntegrity
CHECK (NOT EXISTS
      (SELECT * FROM Member
       WHERE uid NOT IN
       (SELECT uid FROM User)));
```

Can include multiple tables

Assertions are statements that must always be true

Triggers

- A trigger is an event-condition-action (ECA) rule
  - When event occurs, test condition; if condition is satisfied, execute action

```
CREATE TRIGGER PickyPopGroup
AFTER UPDATE OF pop ON User
REFERENCING NEW ROW AS newUser
FOR EACH ROW
        WHEN (newUser.pop < 0.5)
              AND (newUser.uid IN (SELECT uid
                                   FROM Member
                                   WHERE gid = 'popgroup'))
              DELETE FROM Member
              WHERE uid = newUser.uid AND gid = 'popgroup';
```

Event

Transition variable

Condition

Action

Trigger options

- Possible events include:
  - INSERT ON *table*; DELETE ON *table*; UPDATE [OF *column*] ON *table*
- Timing—action can be executed:
  - AFTER or BEFORE the triggering event
  - INSTEAD OF the triggering event on views (lecture 5)
- Granularity—trigger can be activated:
  - FOR EACH ROW modified
  - FOR EACH STATEMENT that performs modification

Transition variables/tables

- OLD ROW: the modified row before the triggering event
- NEW ROW: the modified row after the triggering event
- OLD TABLE: a read-only table containing all old rows modified by the triggering event
- NEW TABLE: a table containing all modified rows after the triggering event

Event	Row	Statement
Delete	old r; old t	old t
Insert	new r; new t	new t
Update	old/new r; old/new t	old/new t

AFTER Trigger

Event	Row	Statement
Update	old/new r	-
Insert	new r	-
Delete	old r	-

BEFORE Trigger

Certain triggers are only possible at statement level

```
CREATE TRIGGER MaintainAvgPop
AFTER UPDATE OF pop ON User
REFERENCING NEW TABLE AS newUsers
                   OLD TBALE AS oldUsers
FOR EACH STATEMENT
         WHEN (0.5 > (SELECT AVG(pop) from User)
         BEGIN
                   DELETE FROM User WHERE uid IN (SELECT uid
                   FROM newUsers)
                   INSERT INTO User (SELECT * FROM oldUsers)
         END
```

Event

Transition tables

Condition

Action

Views

- A view is like a "virtual" table
  - Defined by a query, which describes how to compute the view contents on the fly
  - Stored as a query by DBMS instead of query contents
  - Can be used in queries just like a regular table

```
CREATE VIEW PopGroup AS
    SELECT * FROM User
    WHERE uid IN (SELECT uid
                  FROM Member
                  WHERE gid = 'popgroup');
```

Base tables

```
SELECT AVG(pop)
FROM (SELECT * FROM User
      WHERE uid IN
      (SELECT uid FROM Member
       WHERE gid = 'popgroup'))
       AS popGroup;
```

```
SELECT AVG(pop) FROM PopGroup;
```

```
SELECT MIN(pop) FROM PopGroup;
```

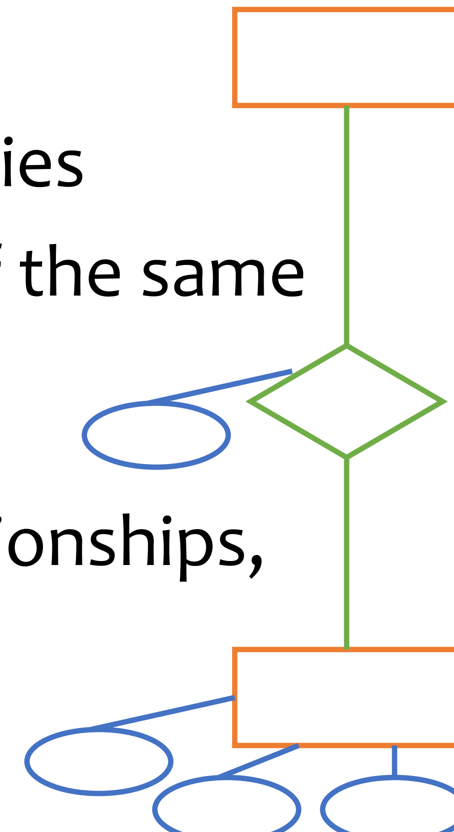
```
SELECT … FROM PopGroup;
```

```
DROP VIEW popGroup;
```

DB Design (lectures 7-10):
E/R models
Design theory

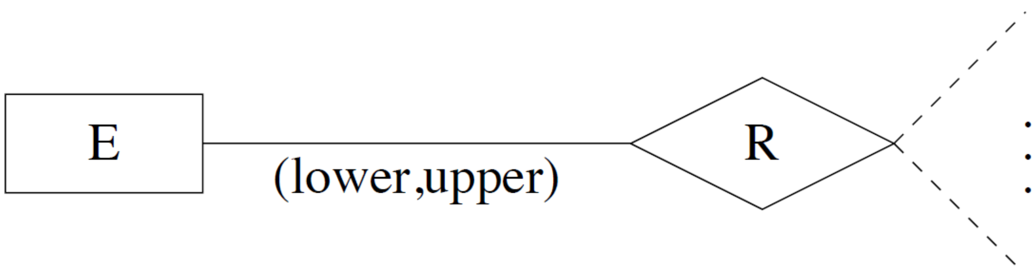
E/R basics

- Entity: a "thing," like an object
- Entity set: a collection of things of the same type, like a relation of tuples or a class of objects
  - Represented as a rectangle
- Relationship: an association among entities
- Relationship set: a set of relationships of the same type (among same entity sets)
  - Represented as a diamond
- Attributes: properties of entities or relationships, like attributes of tuples or objects
  - Represented as ovals

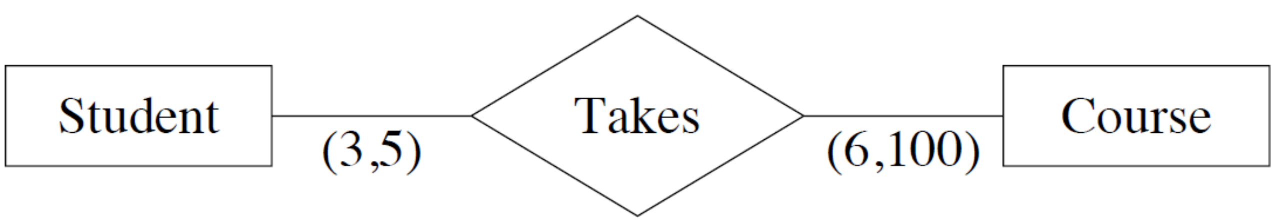


General cardinality constraints

- General cardinality constraints determine lower and upper bounds on the number of relationships of a given relationship set in which a component entity may participate

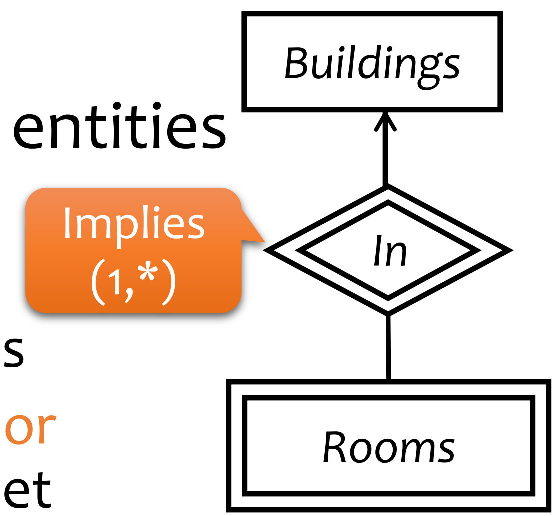


- Example:



Weak entity sets

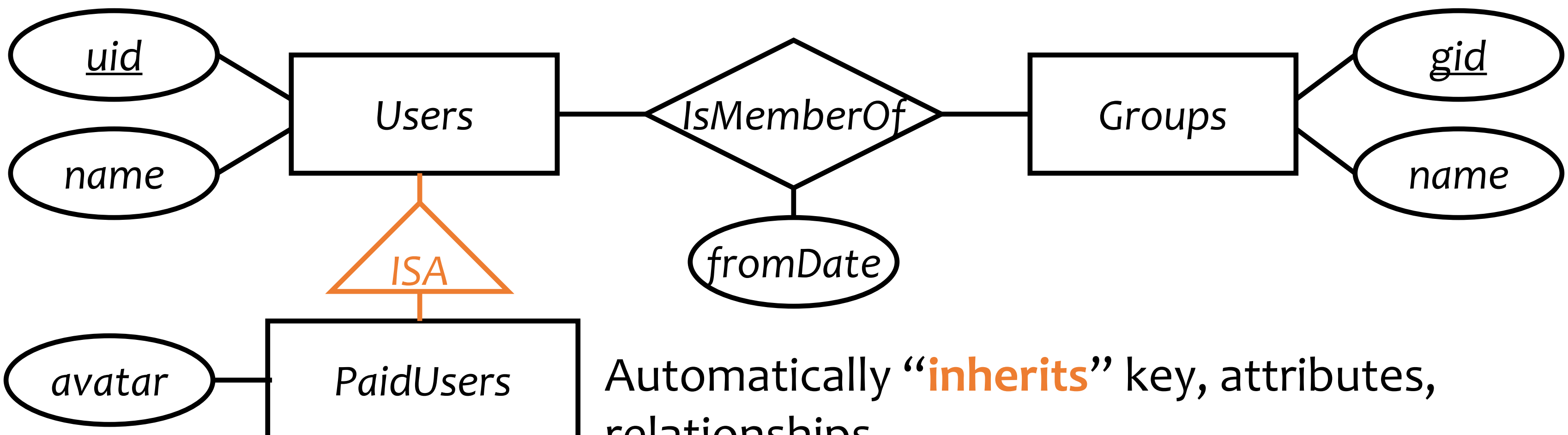
- If entity E's existence depends on entity F, then
  - F is a dominant entity
  - E is a subordinate entity
  - Example: *Rooms* inside *Buildings* are partly identified by *Buildings*' name
- Weak entity set: containing subordinate entities
  - Drawn as a double rectangle
  - The relationship sets are called supporting relationship sets, drawn as double diamonds
  - A weak entity set must have a many-to-one or one-to-one relationship to a distinct entity set



- Strong entity set: containing no subordinate entities

Specialization or ISA relationships

- Similar to the idea of subclasses in object-oriented programming: subclass = special case, fewer entities, and possibly more properties
  - Represented as a triangle (direction is important)
- Example: paid users are users, but they also get avatars (yay!)

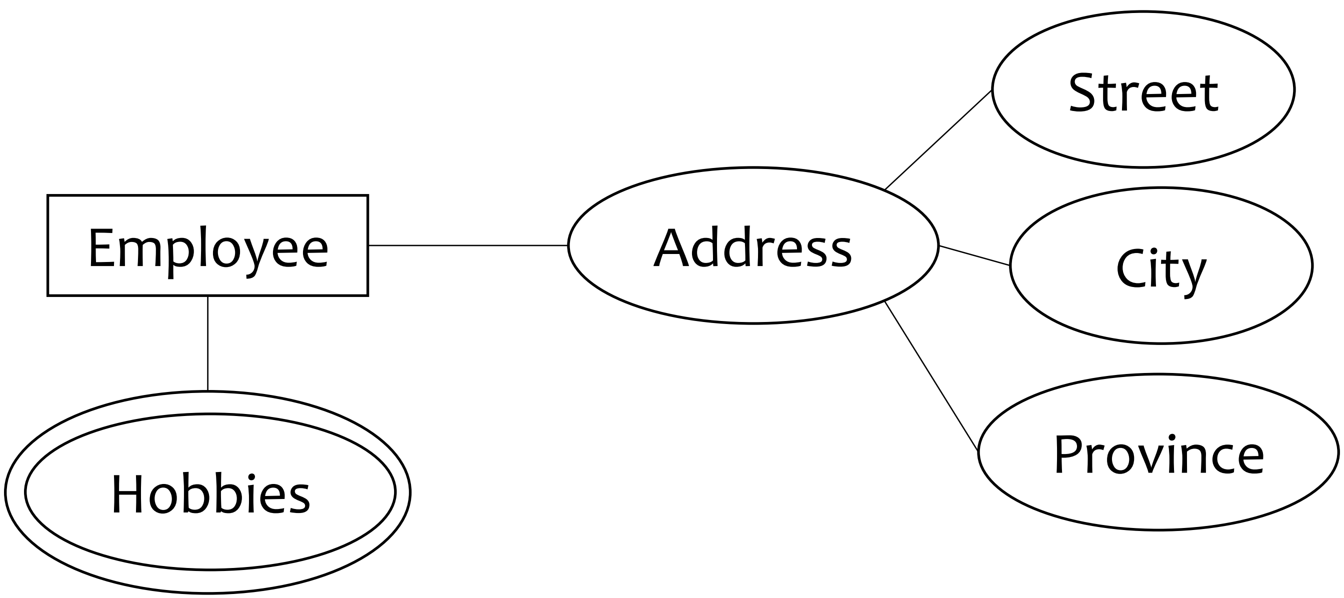


Automatically "**inherits**" key, attributes, relationships

Can participate in other relationships

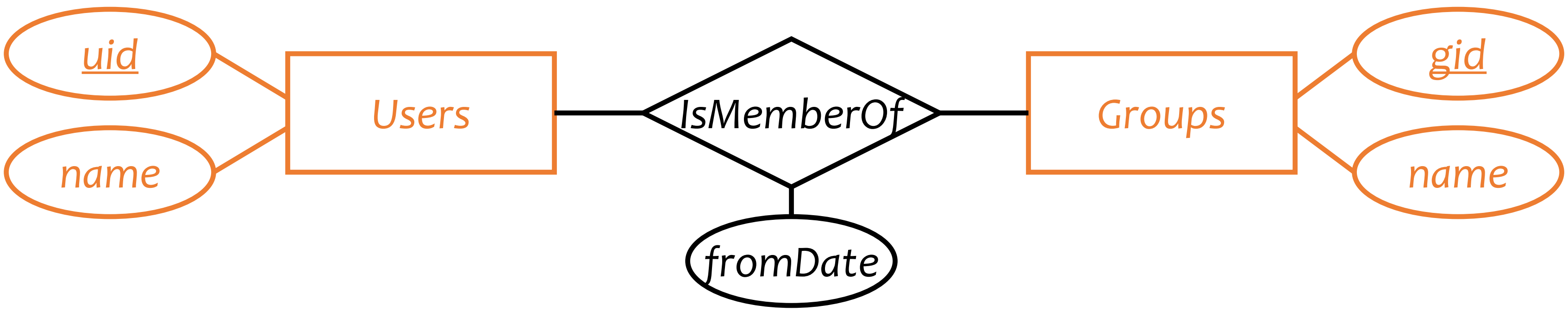
Composite and multi-valued attributes

- Composite attributes: composed of fixed number of other attributes
  - E.g. Address
- Multi-valued attributes: attributes that are set-valued
  - e.g. Hobbies (double edges)



Translating entity sets

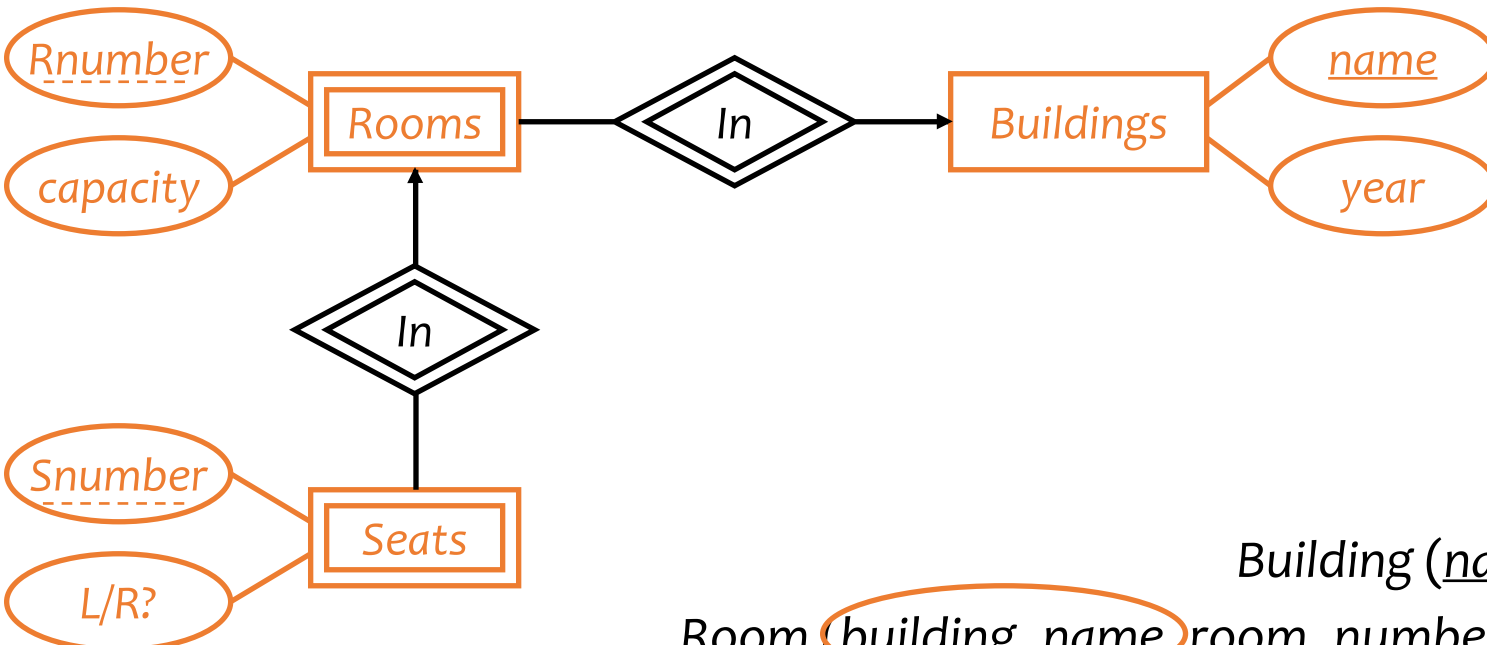
- An entity set translates directly to a table
  - Attributes → columns
  - Key attributes → key columns



User (<u>uid</u>, name)   *Group (<u>gid</u>, name)*

Translating weak entity sets

- Remember the "borrowed" key attributes
- Watch out for attribute name conflicts



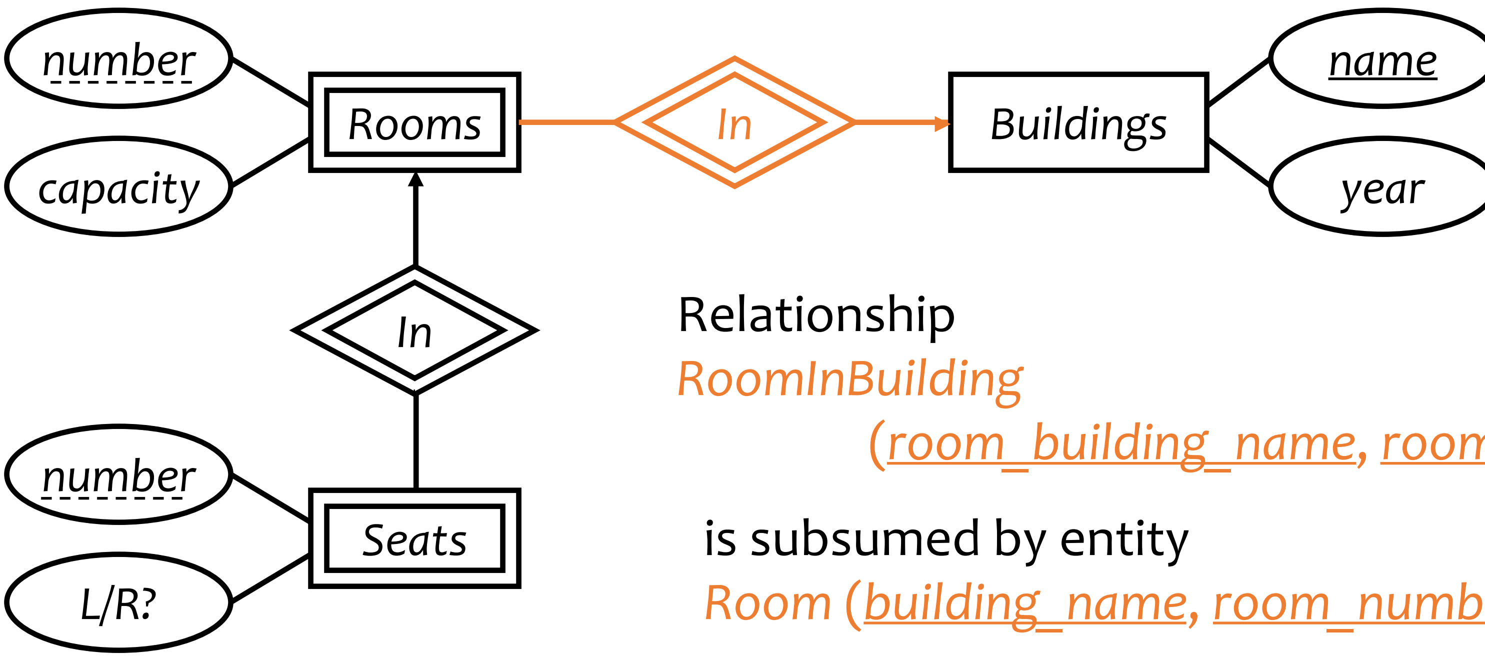
Building (<u>name</u>, *year*)

Room (<u>*building_name*</u>, <u>*room_number*</u>, *capacity*)

Seat (<u>*building_name*</u>, <u>*room_number*</u>, <u>*seat_number*</u>, *left_or_right*)

Translating double diamonds?

- No need to translate because the relationship is implicit in the weak entity set's translation



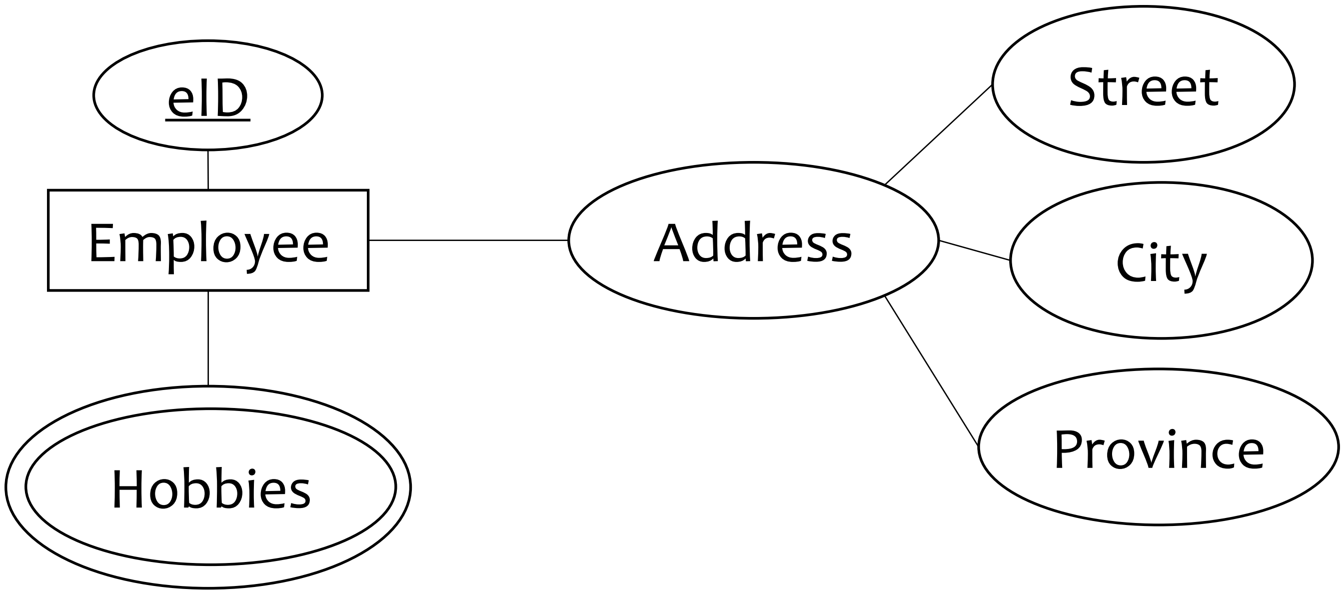
Relationship
RoomInBuilding
(room_building_name, room_number,)

is subsumed by entity
Room (building_name, room_number, capacity)

Comparison of three approaches of translating subclasses & ISA

- Entity-in-all-superclasses
  - *User* (*uid*, *name*), *PaidUser* (*uid*, *avatar*)
  - Pro: All users are found in one table
  - Con: Attributes of paid users are scattered in different tables
- Entity-in-most-specific-class
  - *User* (*uid*, *name*), *PaidUser* (*uid*, *name*, *avatar*)
  - Pro: All attributes of paid users are found in one table
  - Con: Users are scattered in different tables
- All-entities-in-one-table
  - *User* (*uid*, [*type*,]*name*, *avatar*)
  - Pro: Everything is in one table
  - Con: Lots of NULL's; complicated if class hierarchy is complex

Translating composite and multi-valued attributes



Composite:

Employee(eId,…,Street, City, Province,..)

Multi-valued:

EmployeeHobbies(<u>eID</u>, <u>hobby</u>)

Foreign key: *eId references Employee*

Employee join EmployeeHobbies to get all info

Functional dependencies

- A functional dependency (FD) is a constraint between two sets of attributes in a relation
- FD has the form $X \rightarrow Y$, where X and Y are sets of attributes in a relation R
- $X \rightarrow Y$ means that whenever two tuples in R agree on all the attributes in X, they must also agree on all attributes in Y

X	Y	Z
a	b	c
a	b	?
…	…	…

Must be b

Could be anything

  - If X is a superkey of R , then X → R (all the attributes)

Implied FDs: Armstrong's Axioms

- A set of fds can imply other fds via 3 intuitive rules: Armstrong's Axioms

1. Reflexivity: If $Y \subseteq X$, then $X \rightarrow Y$ (trivially)
   - iID, name $\rightarrow$ iID
   - English: Each iID and name value determine a unique iID value
2. Augmentation: if $X \rightarrow Y$, then $XZ \rightarrow YZ$ (trivially)
   - If iID $\rightarrow$ salary then iID, name $\rightarrow$ salary, name
   - English: if each iID determines a unique salary value, then each (iID, name) value pair determines a unique (salary, name) value

InstDep					
<u>iID</u>	name	salary	depName	bldng	budget
111	Alice	5000	CS	DC	20000
222	Bob	4000	Physics	PHY	30000
333	Carl	5200	CS	DC	20000
...	...	...	...	...	...

Implied FDs: Armstrong's Axioms

3. Transitivity: if $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

- Suppose each instructor can be in a single department and each dep has a single budget
- FD1: iID → depName FD2: depName → budget, then

  iID → budget
- English: If each iID value determines a unique depName value, which in turn determines a unique budget value, then each iID value determines a unique budget value.

InstDep					
iID	name	salary	depName	bldng	budget
111	Alice	5000	CS	DC	20000
222	Bob	4000	Physics	PHY	30000
333	Carl	5200	CS	DC	20000
…	…	…	…	…	…

Other Rules Implied by Armstrong's Axioms

1. Decomposition: If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$

   Proof:

   i. $X \rightarrow YZ$

   ii. $YZ \rightarrow Y$ (by reflexivity); $YZ \rightarrow Z$ (by reflexivity)

   iii. $X \rightarrow Y$ (by transitivity); $X \rightarrow Z$ (by transitivity)

2. Union: If $X \rightarrow Y$ and $X \rightarrow Z$ then $X \rightarrow YZ$ (Prove as exercise)
3. Pseudo-transitivity: If $X \rightarrow Y$ and $YZ \rightarrow T$ then $XZ \rightarrow T$ (Prove as exercise)

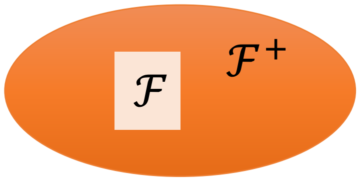
Using these rules, you can prove or disprove a (derived) FD given a set of (base) FDs

Closure of FD sets: $\mathcal{F}^+$

- How do we know what additional FDs hold in a schema?

- A set of FDs $\mathcal{F}$ logically implies a FD $X \rightarrow Y$ if $X \rightarrow Y$ holds in all instances of R that satisfy $\mathcal{F}$

- The closure of a FD set $\mathcal{F}$ (denoted $\mathcal{F}^+$):
  - The set of all FDs that are logically implied by $\mathcal{F}$
  - Informally, $\mathcal{F}^+$ includes all of the FDs in $\mathcal{F}$, i.e., $\mathcal{F} \subseteq F^+$, plus any dependencies they imply.



Attribute closure

- The closure of attributes Z in a relation R (denoted Z^+) with respect to a set of FDs, $\mathcal{F}$, is the set of all attributes $\{A_1, A_2, \ldots\}$ functionally determined by Z (that is, Z $\rightarrow A_1 A_2 \ldots$)

- Algorithm for computing the closure Compute$Z^+(Z, \mathcal{F})$:
  - Start with closure $= Z$
  - If $X \rightarrow Y$ is in $\mathcal{F}$ and X is already in the closure, then also add Y to the closure
  - Repeat until no new attributes can be added

Example for computing attribute closure

Given relation R(ABCDEFG)

Compute$Z^+(\{B, F\}, \mathcal{F})$:

$\mathcal{F}$ includes:

$A, B \to F$

$A \to C$

$B \to E, D$

$D, F \to G$

FD	Z^+
initial	B, F
$B \to E, D$	B, F, E, D
$D, F \to G$	B, F, E, D, G

$$B, F \to E, D, G$$

Using attribute closure

Given a relation R and set of FD's $\mathcal{F}$

- Does another FD $X \to Y$ follow from $\mathcal{F}$?
  - Compute X^+ with respect to $\mathcal{F}$
  - If $Y \subseteq X^+$, then $X \to Y$ follows from $\mathcal{F}$

- Is K a key of R?
  - Compute K^+ with respect to $\mathcal{F}$
  - If K^+ contains all the attributes of R, K is a super key
  - Still need to verify that K is *minimal* (how?)
    - Hint: check the attribute closure of its proper subset.
    - i.e., Check that for no set X formed by removing attributes from K is $K^+$the set of all attributes

"Good" Schema Decomposition

- Lossless-join decompositions
  - We should be able to construct the instance of the original table from the instances of the tables in the decomposition

A decomposition $\{R_1, R_2\}$ of R is lossless iff the common attributes of R_1 and R_2 form a superkey for either schema,

$$R_1 \cap R_2 \rightarrow R_1 \text{ or } R_1 \cap R_2 \rightarrow R_2$$

**If X is a superkey of R , then $X \rightarrow R$ (all the attributes) [last lecture]*

“Good” Schema Decomposition

- Lossless-join decompositions
- Dependency-preserving decompositions

Given a schema R and a set of FDs $\mathcal{F}$,
decomposition of R is dependency preserving
if there is an equivalent set of FDs $\mathcal{F}'$,
none of which is interrelational in the decomposition.

- Next, how to obtain such decompositions?
  - BCNF → guaranteed to be a lossless join decomposition!

Boyce-Codd Normal Form (BCNF)

- A relation R is in BCNF iff whenever $(X \rightarrow Y) \in \mathcal{F}^+$ and $XY \subseteq R$, then either
  - $(X \rightarrow Y)$ is trivial (i.e., $Y \subseteq X$), or
  - X is a super key of R (i.e., $X \rightarrow R$)
    - That is, all non-trivial FDs follow from "key → other attributes"

- Example: $R = \{Sno, Sname, City, Pno, Pname, Price\}$

$\mathcal{F}$ includes:

$\boxed{\text{FD1: } Sno \rightarrow Sname, City}$ FD2: $Pno \rightarrow Pname$ FD3: $Sno, Pno \rightarrow Price$

  - The schema is not in BCNF because, for example, *Sno* determines *Sname,City*, is non-trivial but is not a superkey of R

BCNF decomposition algorithm

- Find a BCNF violation
  - That is, a non-trivial FD $X \rightarrow Y$ in $\mathcal{F}^+$ of R where X is not a super key of R
    - Example: $R = \{Sno, Sname, City, Pno, Pname, Price\}$

$\mathcal{F}$ includes:

$\boxed{\text{FD1: } Sno \rightarrow Sname, City}$ FD2: $Pno \rightarrow Pname$ FD3: $Sno, Pno \rightarrow Price$

- Decompose R into R_1 and R_2, where
  - R_1 has attributes $X \cup Y$;
  - R_2 has attributes $X \cup Z$, where Z contains all attributes of R that are in neither X nor Y

$R = \{Sno, Sname, City, Pno, Pname, Price\}$

BCNF violation: $Sno \rightarrow Sname, City$

R2{Sno,Pno,Pname,Price} R1{Sno,Sname,City}

- Repeat (till all are in BCNF)

BCNF decomposition example

- $R = \{Sno, Sname, City, Pno, Pname, Price\}$

$\mathcal{F}$ includes:

FD1: $Sno \rightarrow Sname, City$ FD2: $Pno \rightarrow Pname$ FD3: $Sno, Pno \rightarrow Price$

{Sno,Sname,City,Pno,Pname,Price}

BCNF violation: $Sno \rightarrow Sname, City$

R2{Sno,Pno,Pname,Price}

$Pno \rightarrow Pname$ $Sno, Pno \rightarrow Price$

BCNF violation: $Pno \rightarrow Pname$

R2b{Sno,Pno,Price}

BCNF: $Sno, Pno \rightarrow Price$

R2a{Pno,Pname}

BCNF: $Pno \rightarrow Pname$

R1{Sno,Sname,City}

BCNF: $Sno \rightarrow Sname, City$

$\{SNo\}^{+}$={Sno, Sname, City}
→ a superkey of R1

"Good" Schema Decomposition

- Lossless-join decompositions
- Dependency-preserving decompositions
- BCNF → guaranteed to be a lossless join decomposition!
  - Depend on the on the sequence of FDs for decomposition
  - Not necessarily dependency preserving

Example: consider R={A, B, C}

$\mathcal{F}$ includes: FD1: $AB \rightarrow C$ FD2: C $\rightarrow B$

BCNF violation: C $\rightarrow B$

{A, C} {C, B}

$AB \rightarrow C$ is interrelational and cannot be tested directly

“Good” Schema Decomposition

- Lossless-join decompositions
- Dependency-preserving decompositions
- BCNF → guaranteed to be a lossless join decomposition!
    - Depend on the on the sequence of FDs for decomposition
    - Not necessarily dependency preserving
- 3NF → both lossless join and dependency preserving

Third normal form (3NF)

- A relation R is in 3NF iff whenever $(X \rightarrow Y) \in \mathcal{F}^+$ and $XY \subseteq R$, then either
  - $(X \rightarrow Y)$ is trivial (i.e., $Y \subseteq X$), or
  - X is a super key of R (i.e., $X \rightarrow R$) or
  - **Each attribute in $\mathbf{Y} - \boldsymbol{X}$ is contained in a candidate key of $\boldsymbol{R}$**

  - Example: consider R={A, B, C}
    - Satisfies 3NF, but not BCNF

$\mathcal{F}$ includes: FD1: $AB \rightarrow C$ FD2: $C \rightarrow B$

{B}-{C} = {B} is part of the key {AB}

- 3NF is looser than BCNF → Allows more redundancy

Finding minimal cover

- A minimal cover for $\mathcal{F}$ can be computed in 3 steps.
  1. Replace $X \rightarrow YZ$ with the pair $X \rightarrow Y$ and $X \rightarrow Z$
  2. Remove A from the left-hand side of $X \rightarrow B$ in $\mathcal{F}$ if B $\in$ $computeX^{+}(X - \{A\}, \mathcal{F})$
  3. Remove $X \rightarrow A$ from $\mathcal{F}$ if $A \in computeX^{+}(X, \mathcal{F} - \{X \rightarrow A\})$
  - Note that each step must be repeated until it no longer succeeds in updating $\mathcal{F}$.

- Example: $R = \{$Sno,Sname,City,Pno,Pname,Price, $PType$ $\}$

$\mathcal{F}$: FD1: $Sno \rightarrow Sname, City$
FD2: $Pno \rightarrow Pname$
FD3: $Sno, Pno \rightarrow Price$
FD4: $Sno, Pname \rightarrow Price$
FD5: $Pno, Pname \rightarrow Ptype$

$Sno \rightarrow Sname,$
$Sno \rightarrow City$

Remove FD3

$Pno \rightarrow Ptype$

Computing 3NF decomposition

Efficient algorithm for computing a 3NF decomposition of R with FDs $\mathcal{F}$:

1. Initialize the decomposition with empty set
2. Find a minimal cover for $\mathcal{F}$, let it be $\mathcal{F}^*$
3. For every $(\mathrm{X} \rightarrow \mathrm{Y}) \in \mathcal{F}^*$, add a relation {XY} to the decomposition
4. If no relation contains a candidate key for R, then compute a candidate key K for R, and add relation {K} to the decomposition.

Example for 3NF decomposition

- $R = \{Sno, Sname, City, Pno, Pname, Price\}$

$\mathcal{F}$: FD1: $Sno \rightarrow Sname, City$
FD2: $Pno \rightarrow Pname$
FD3: $Sno, Pno \rightarrow Price$
FD4: $Sno, Pname \rightarrow Price$

- Minimal cover $\mathcal{F}^*$

Exercise

$\mathcal{F}^*$: FD1a: $Sno \rightarrow Sname$
FD1b: $Sno \rightarrow City$
FD2: $Pno \rightarrow Pname$
FD4: $Sno, Pname \rightarrow Price$

R1a(Sno, Sname)
R1b(Sno, City)
R2(Pno, Pname)
R4(Sno,Pname,Price)

R5(Sno,Pno)

Exercise

- Add relation for candidate key
- Optimization for this example: combine relations R1a and R1b

Next lecture

- DB Architecture Overview & Physical Data Organization