Midterm Review

CS348 Spring 2025:
Introduction to Database Management

Instructor: **Xiao Hu**
Sections: 001, 002, 003

Announcements

- Appealing of Assignment 1
  - Check remark request guidelines on Piazza
  - Reach out to corresponding TA, IA (Guy), ISC (Sylvie)
  - Check sample solutions on Learn
- Milestone 1 of Group Project
  - Due on June 19
- Switch-type cut-off for assessment
  - Due on June 19
- Assignment 2
  - Coverage: Lecture 4 - Lecture 12
  - Check online office hours on Piazza
  - Due on June 24

Midterm Exam

- Logistics
  - Time & Date: 4:30 PM – 6:00 PM June 27
  - Location: M3 1006 and STC 0040 (check your room)
  - All contents in Lectures 1 – 12 (no optional parts)
  - Closed book, but a two-page reference sheet will be provided (already released on Learn)

- How to prepare (in addition to lectures)?
  - Sample questions released on Learn
  - Partial solutions to be released later on Learn
  - A1 (with partial solutions) and A2

Relational Model and Relational Algebra

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

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
    - E.g. *uid* should be unique in the *User* relation
  - Functional dependencies (Lecture 11)
- Database-level
  - Referential integrity – foreign key
    - *uid* in *Member* must refer to a row in *User* with the same *uid*

Key (Candidate Key)

A set of attributes K for a relation R is a key if

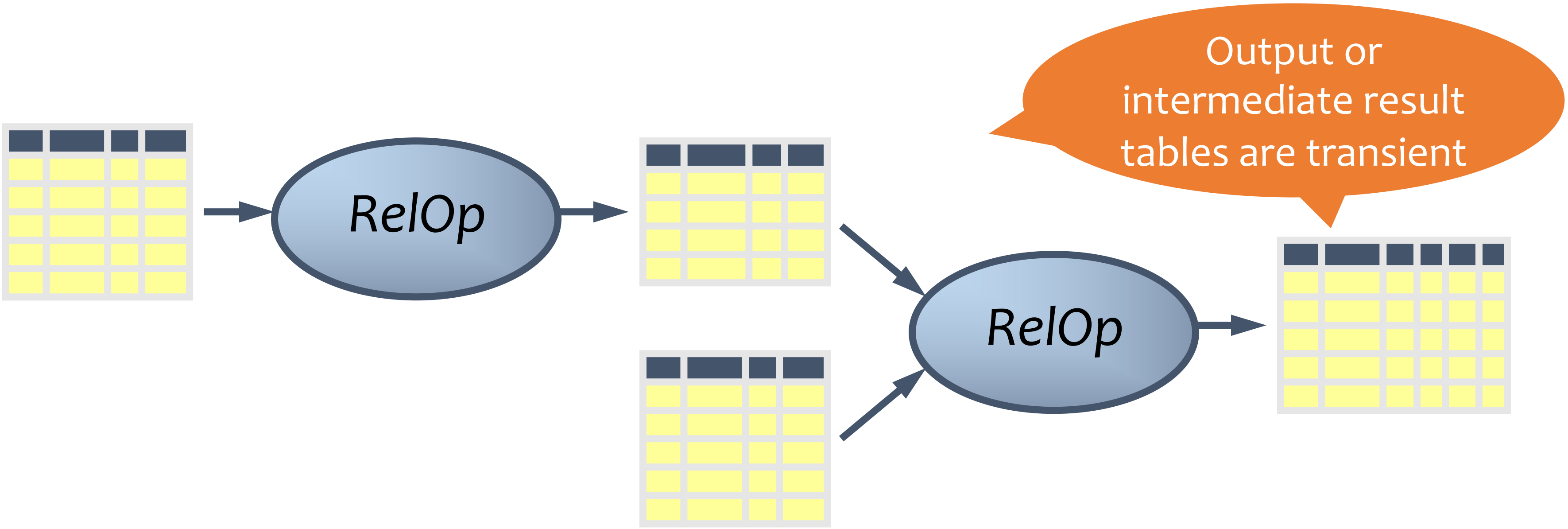
- Condition 1: In no instance of R will two different tuples agree on all attributes of K
  - That is, K can serve as a "tuple identifier"
- Condition 2: No proper subset of K satisfies the above condition
  - That is, K is minimal
- Example: *User* (*uid*, *name*, *age*, *pop*)
  - *uid* is a key of *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"
- Set semantics



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: $\rho_{S(A_1 \to A_1', A_2 \to A_2', \dots)} R$

Note: Only use these operators for assignments & exams

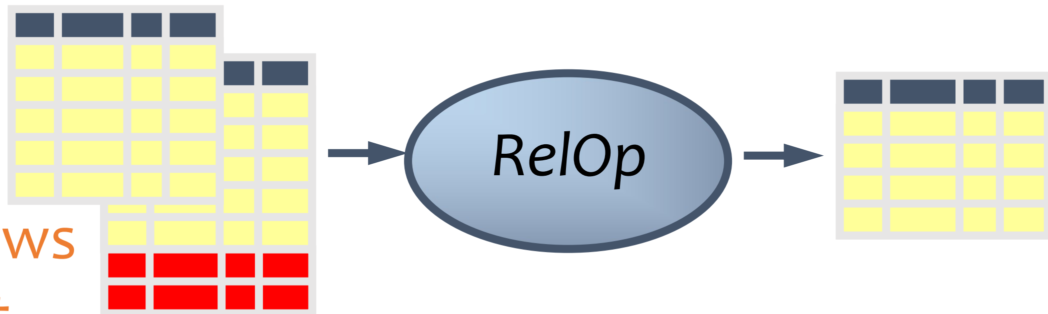
Note: Outerjoin is also allowed.

Derived Operators

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

Non-monotone operators

Add more rows to the input…



What happens to the output?

- If some old output rows may become invalid, and need to be removed → the operator is non-monotone
- Otherwise (old output rows always remain "correct") → the operator is monotone

uid	gid
123	gov
857	abc

—

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

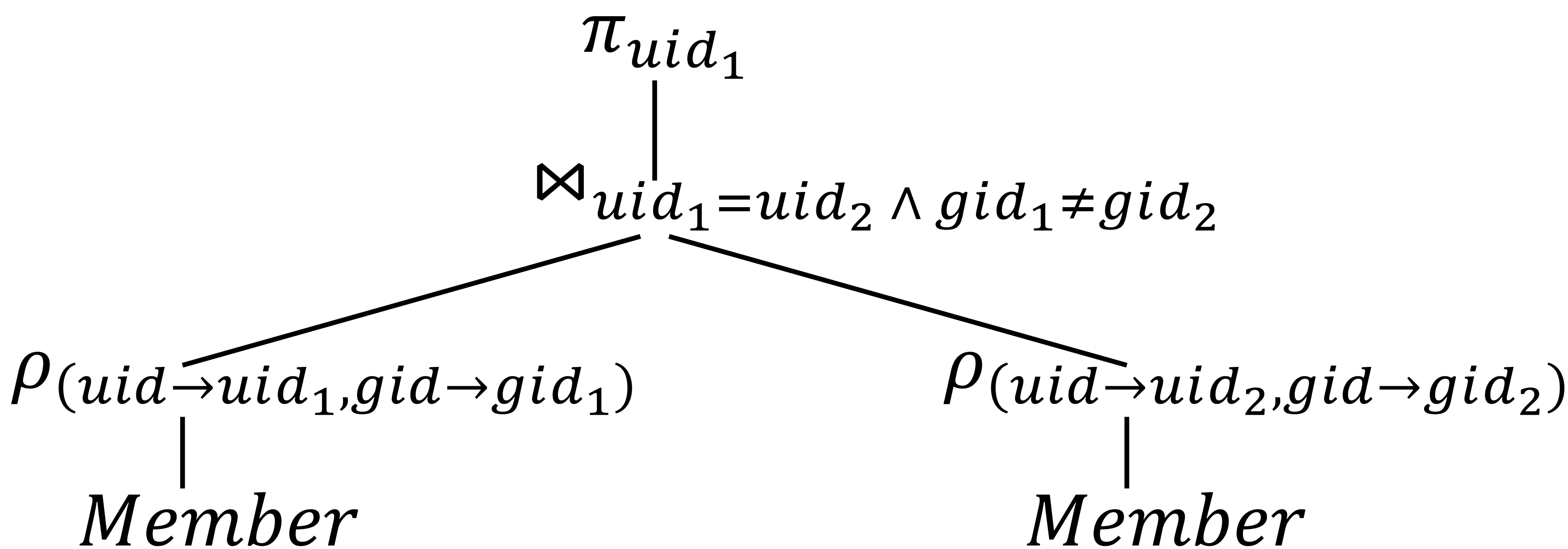
→

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

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

Expression tree notation

- IDs of users who belong to at least two groups

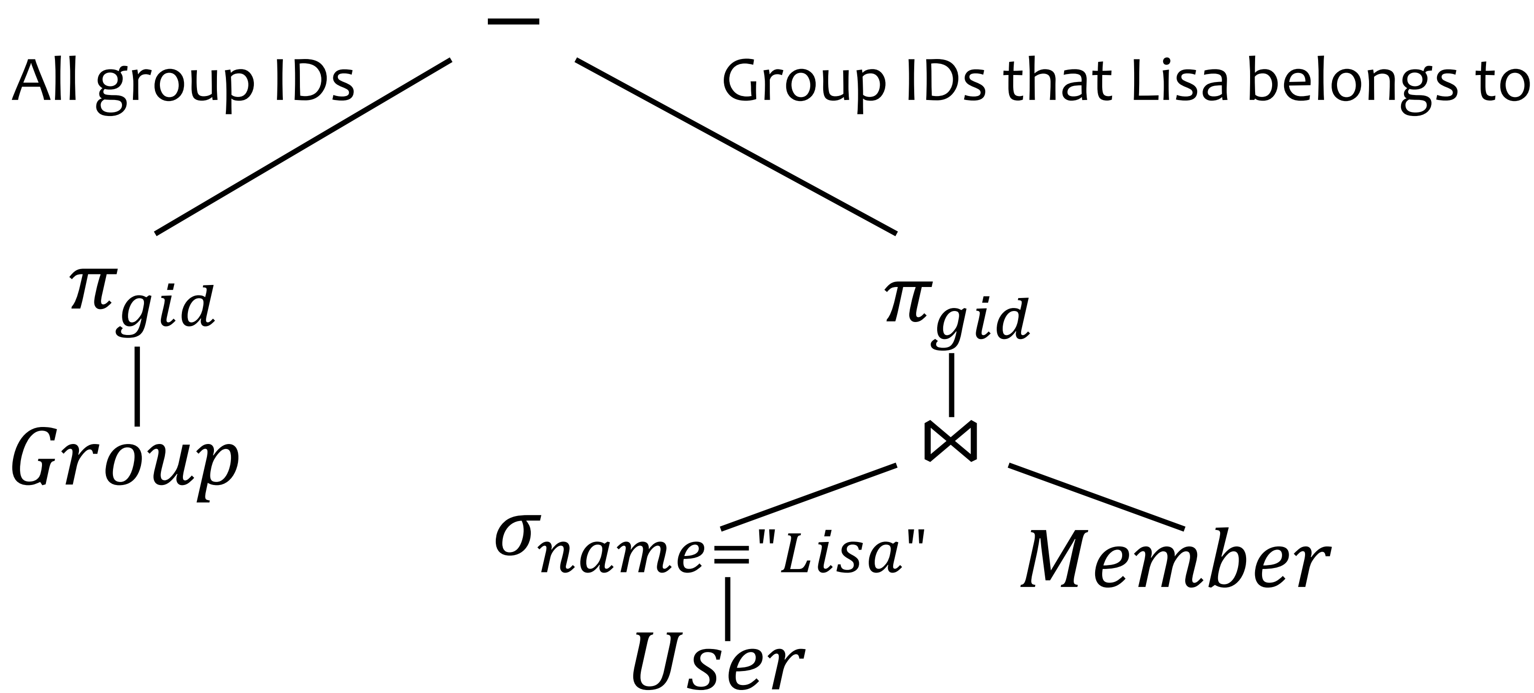


- IDs of groups that contain at least two users

An 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)

- IDs of groups that Lisa doesn't belong to

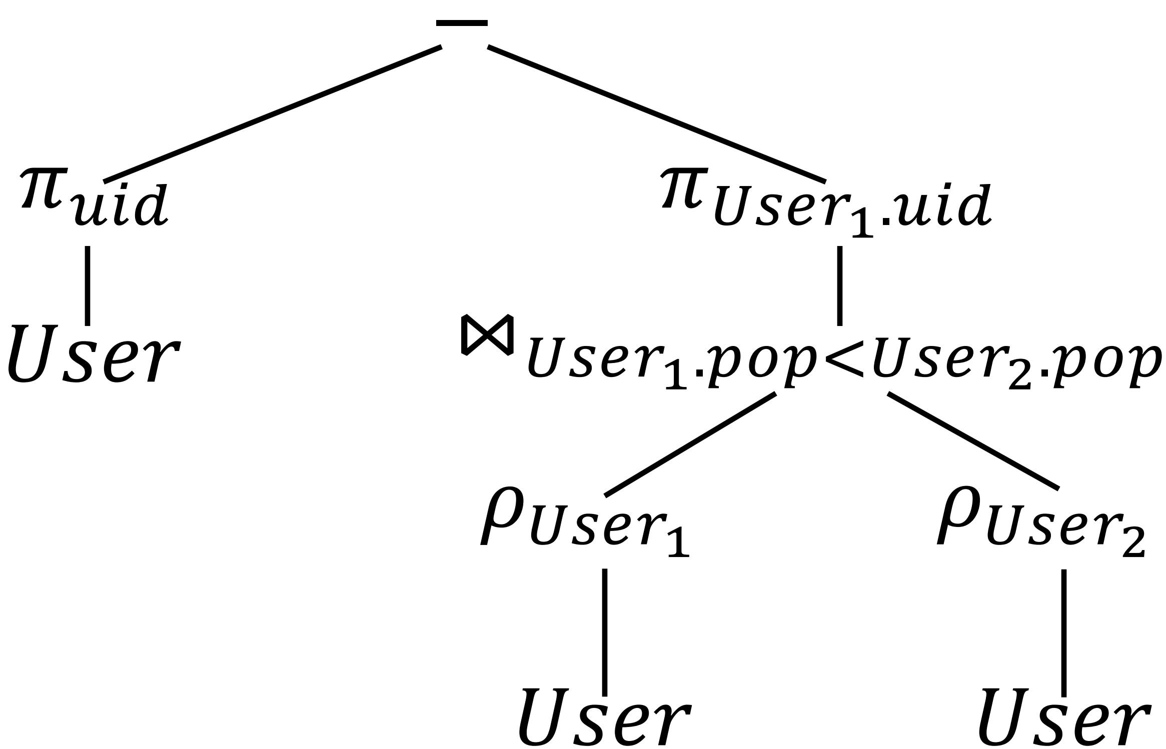


$$\left(\pi_{gid} Group\right) - \left(\pi_{gid}\left(\left(\sigma_{name="Lisa"} User\right) \bowtie Member\right)\right)$$

Most popular user

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)

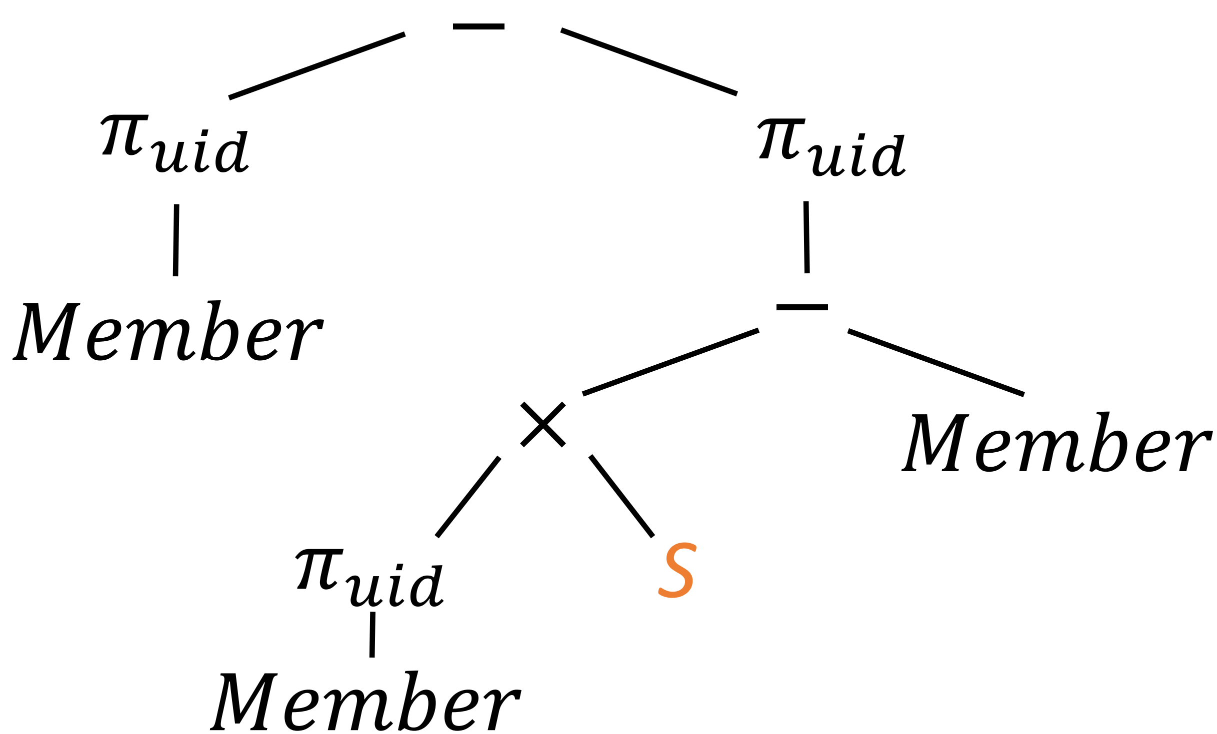
- Who are the most popular?
  - Who do NOT have the highest *pop* rating?
  - Whose *pop* is lower than somebody else's?



Division

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 joins all the groups that Lisa joins?
  - All groups (ids) that Lisa belongs to   Suppose as *S(gid)*
  - Who joins all groups in S?
  - *Who does not join some group in S?*



SQL

Summary of SQL

- Basic topics
  - Data-definition language (DDL): define/modify schemas, drop relations
  - Data-manipulation language (DML): query data
    - SELECT-FROM-WHERE
    - DISTINCT, UNION/EXCEPT/INTERSECT (ALL)
    - Table, Scalar, IN, EXISTS, ALL, ANY)
    - GROUP BY, HAVING
    - ORDER
    - NULL and JOIN

    and modify data (INSERT/DELETE/UPDATE)
  - Constraints (NOT NULL, UNIQUE, PRIMARY/FOREIGN KEY, CHECK, ASSERTION)
- Advanced topics
  - View, Triggers, Recursion, Index, Programming (optional)

DDL

User (uid int, *name* string, *age* int, *pop* float)
Group (gid string, *name* string)
Member (uid int, gid string)

- CREATE TABLE *table_name* (… , *name 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

- ALTER TABLE *table_name*;

```
ALTER TABLE Member ADD date;
ALTER TABLE Member RENAME date TO mdate;
ALTER TABLE Member DROP mdate;
```

Basic queries for DML: SFW statement

- SELECT (DISTINCT) $A_1, A_2, \ldots, A_n$
  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 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

Bucket1

fruit
apple
apple
orange

Bucket2

fruit
apple
orange
orange

```
(SELECT * FROM Bucket1)
UNION
(SELECT * FROM Bucket2);
```

fruit
apple
orange

```
(SELECT * FROM Bucket1)
EXCEPT
(SELECT * FROM Bucket2);
```

fruit

```
(SELECT * FROM Bucket1)
INTERSECT
(SELECT * FROM Bucket2);
```

fruit
apple
orange

SQL bag operations

- 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)

Bucket1

fruit
apple
apple
orange

apple: 2
orange:1

Bucket2

fruit
apple
orange
orange

apple: 1
orange:2

(SELECT * FROM Bucket1)
UNION ALL
(SELECT * FROM Bucket2);

sum up the two counts

(SELECT * FROM Bucket1)
EXCEPT ALL
(SELECT * FROM Bucket2);

proper-subtract the two counts

(SELECT * FROM Bucket1)
INTERSECT ALL
(SELECT * FROM Bucket2);

take the minimum of the two counts

Table subqueries

- Query result as a table that can be used in FROM, set/bag operations, etc.
  - Temporarily exist only in the duration of the outer query

- Example: names of users belonging to at least two groups

```
SELECT name
FROM User,
          (SELECT DISTINCT m1.uid
           FROM Member m1, Member m2
           WHERE m1.uid=m2.uid AND m1.gid != m2.gid) AS temp
WHERE User.uid = temp.uid;
```

WITH clause

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)

- Another way of defining a temporary table
  - Available only to the query in which the WITH clause occurs
- Example: names of users belonging to at least two groups

```
WITH temp AS (SELECT DISTINCT m1.uid
                        FROM Member m1, Member m2
                        WHERE m1.uid=m2.uid AND m1.gid != m2.gid)
SELECT name
FROM User, temp
WHERE User.uid = temp.uid;
```

Scalar subqueries

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)

- A query that returns a single row can be used as a value in SELECT, WHERE, etc.
- Example: users at the same age as Bart

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

- When can this query go wrong?
  - Return more than 1 row
  - Return no rows or NULL values

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

User (uid int, *name* string, *age* int, *pop* float)
Group (gid string, *name* string)
Member (uid int, gid string)

- EXISTS (*subquery*) checks if the result of *subquery* is non-empty
  - True if at least one row is returned by subquery
- Example: users that have the same age as (some) Bart

```
SELECT *
FROM User 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

Quantified subqueries

- Universal quantification (for all):
  - … WHERE x op ALL($subquery$) …
  - True if for all t in the $subquery$ result such that x op t is true

```
SELECT * FROM User
WHERE pop >= ALL (SELECT pop FROM User);
```

- Existential quantification (exists):
  - … WHERE x op ANY($subquery$) …
  - True if there exists some t in the $subquery$ result such that x op t is true

```
SELECT * FROM User
WHERE NOT (pop < ANY (SELECT pop FROM User));
```

Aggregates

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

- Example: number of users under 18, and their average popularity

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

COUNT(*)	AVG(pop)
6	0.625

- Aggregate functions do not appear in WHERE clause
- Aggregate with DISTINCT

```
SELECT COUNT(DISTINCT uid) FROM Member;
```

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

- 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;
```

```
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 and LIMIT

- List the top 3 users after sorting them by popularity (descending) and name (ascending)

```
SELECT uid, name, age, pop
FROM User
ORDER BY pop DESC, name
LIMIT   3;
```

  - ASC is the default option
  - The LIMIT clause specifies the number of rows to return

Three-valued logic - 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

Rules of Dealing with NULL

- 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
  - FALSE and UNKNOWN are not sufficient

- Aggregate functions ignore NULL, except COUNT(*)
  - SUM, AVG, MIN, MAX all ignore NULLs
  - COUNT(age) also ignores NULL
  - If all inputs are NULL, SUM, AVG, MIN, MAX all return NULL

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

Full Outerjoin

Group

gid	*gname*
abc	Book Club
gov	Student Government
dps	Dead Putting Society
spr	Sports Club

Member

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

Group ⟗ *Member*

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

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

Left/Right Outerjoin

Group

gid	*gname*
abc	Book Club
gov	Student Government
dps	Dead Putting Society
spr	Sports Club

Member

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

Group ⟕ *Member*

gid	*gname*	*uid*
abc	Book Club	857
gov	Student Government	123
gov	Student Government	857
dps	Dead Putting Society	142
spr	Sports Club	**NULL**

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

Group ⟖ *Member*

gid	*gname*	*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 in SQL

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

$\approx Group \bowtie_{Group.gid=Member.gid} Member$

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

$\approx Group \bowtie Member$

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

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

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

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

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

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

Modify Data

- Insert one row or the results of a query

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

```
INSERT INTO Member
(SELECT uid, ‘dps’ FROM User WHERE uid NOT IN
        (SELECT uid FROM Member WHERE gid = ‘dps’));
```

- Delete according to a WHERE condition

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

```
DELETE m, u  FROM Member m NATURAL JOIN User u
WHERE age > 18 AND gid = ‘abc';
```

- Update: User 142 changes name to “Barney”

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

```
UPDATE User SET pop = (SELECT AVG(pop) FROM User);
```

Types of SQL constraints

- NOT NULL
- Key
- Referential integrity
- Tuple- and attribute-based CHECK’s
- General assertion

Example of NOT NULL

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

```
CREATE TABLE Group
(gid CHAR(10) NOT NULL,
 name VARCHAR(100) NOT NULL);
```

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

Incorrect

```
INSERT INTO User (uid, age)
VALUES (389, 18);
```

Incorrect

```
INSERT INTO User VALUES (789,
'Nelson', NULL, NULL, NULL);
```

Examples of KEY

PRIMARY KEY should not contain NULLs

option 1

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

At most one primary key per table

Any number of UNIQUE keys per table

```
CREATE TABLE Group
(gid CHAR(10) NOT NULL,
 name VARCHAR(100) NOT NULL,
 PRIMARY KEY (gid));
```

option 2

Option 2 is required for multi-attribute keys

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

Incorrect!

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

Referential integrity in SQL

- Referenced column(s) must be PRIMARY KEY
  - Some systems allow both PRIMARY KEY and UNIQUE
- Referencing column(s) form a FOREIGN KEY
- Example

```
CREATE TABLE User (...
uid INT NOT NULL PRIMARY KEY);
```

```
CREATE TABLE Group (...
gid CHAR(10) NOT NULL PRIMARY KEY);
```

```
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));
```

option 1

option 2

Enforcing referential integrity

Example: *Member.uid references User.uid*

- Insert or update a *Member* row whose uid refers to a non-existent *uid in User*
  - Reject

User

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

Member

uid	*gid*
142	dps
123	gov
857	abc
857	gov
456	abc
456	gov
000	**gov**

Reject

Enforcing referential integrity

Example: *Member.uid references User.uid*

- Delete or update a *User* row whose *uid* is referenced by some *Member* row
  - Multiple Options (SQL)

Option 3: Set NULL
(set all references to NULL)

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

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~~
…	….

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

Tuple- and attribute-based CHECK

- Associated with a single table!
- Only checked when a tuple or an attribute is updated
  - Reject if condition evaluates to FALSE
  - TRUE and UNKNOWN are fine

(Recap) WHERE and HAVING clauses should evaluate to TRUE

- Examples: each user has age above 0 or NULL

```
CREATE TABLE User(...
 age INTEGER CHECK(age IS NULL OR age > 0), ...);
```

```
CREATE TABLE User(...
age INT,
CONSTRAINT minAge CHECK(age IS NULL OR age > 0), ...);
```

General assertion

- Can involve multiple tables!
- CREATE ASSERTION … CHECK *assertion_condition*
- Checked for any modification that could potentially violate it
  - Reject if condition evaluates to FALSE or UNKNOWN
  - TRUE is required
- Example: *Member.uid* references *User.uid*

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

Checked when Member or User is modified

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
    - NEW ROW and OLD ROW
  - FOR EACH STATEMENT that performs modification
    - NEW TABLE and OLD TABLE
- Certain triggers are only at statement-level

INSTEAD OF triggers for views

```
CREATE VIEW AveragePop(pop) AS
        SELECT AVG(pop) FROM User;
```

```
CREATE TRIGGER ModifyAveragePop
INSTEAD OF UPDATE ON AveragePop
REFERENCING OLD ROW AS old, NEW ROW AS new
FOR EACH ROW
        UPDATE User
        SET pop = pop + (new.pop - old.pop);
```

0.5

0.4

For each row to be updated in AveragePop

- What does this trigger do?

```
UPDATE AveragePop SET pop = 0.5;
```

User

...	pop	...
	0.4 +0.1	
	0.4 +0.1	
	0.5 +0.1	
	0.3 +0.1	

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');
```

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

```
DROP VIEW popGroup;
```

Modifying views

- Goal: modify base tables such that the modification would appear to have been done on the view

```
CREATE VIEW UserPop AS
SELECT uid, pop FROM User;
```

```
DELETE FROM UserPop WHERE uid = 123;
```

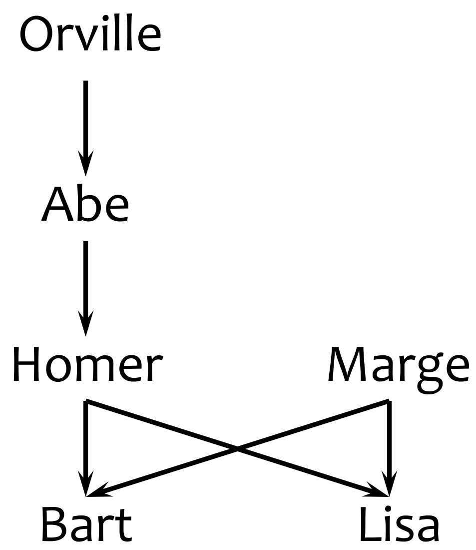
```
DELETE FROM User WHERE uid = 123;
```

- Given any DML that violate the view's filter
  - If WITH CHECK OPTION: reject
  - If WITH CHECK OPTION is not specified: it is possible to "sneak" valid rows into the base table through the view -- these rows simply won't appear in the view

Recursion Example

Parent (parent, child)

parent	*child*
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Orville	Abe



- Example: find Bart's ancestors
- "Ancestor" has a recursive definition
  - X is Y's ancestor if
    - X is Y's parent, or
    - X is Z$'$s ancestor and Z is Y's ancestor

Example of Ancestor Query

```
WITH RECURSIVE
Ancestor(anc, desc) AS

((SELECT parent, child FROM Parent)

 UNION

(SELECT a1.anc, a2.desc
 FROM Ancestor a1, Ancestor a2
 WHERE a1.desc = a2.anc))

SELECT anc
FROM Ancestor
WHERE desc = 'Bart';
```

base case

a1.anc (X) → a1.desc(Z)
a2.anc (Z) → a2.desc (Y)

recursion step

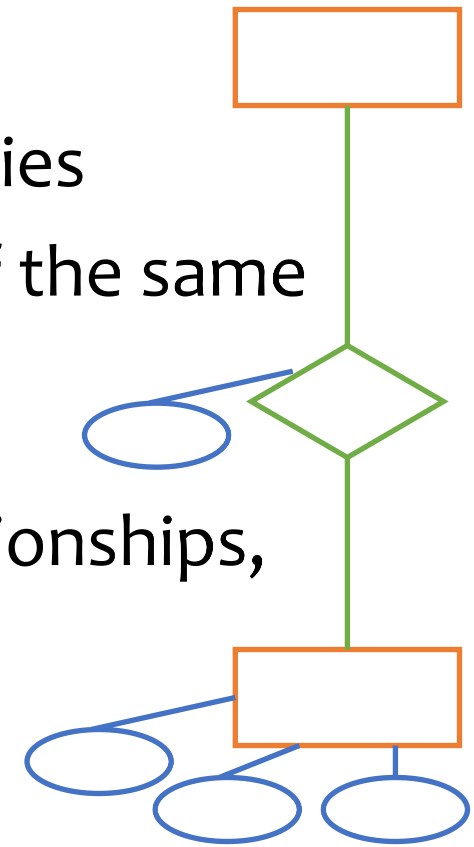
Define a relation recursively

Query using the relation defined in WITH clause

Database Design

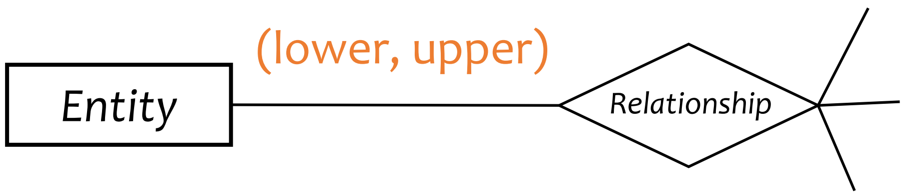
E/R basics

- 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

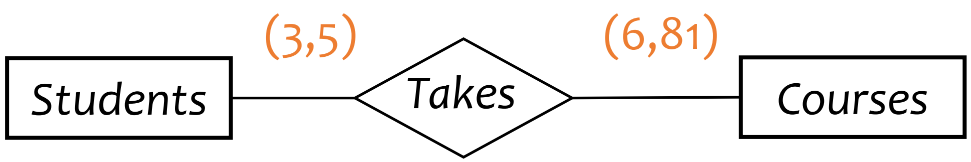


General cardinality constraints

- 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



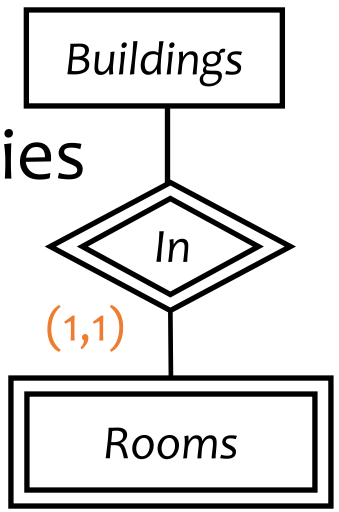
- Example:



- Total v.s. partial participation: (1,*) v.s. (0,*)

Weak entity sets

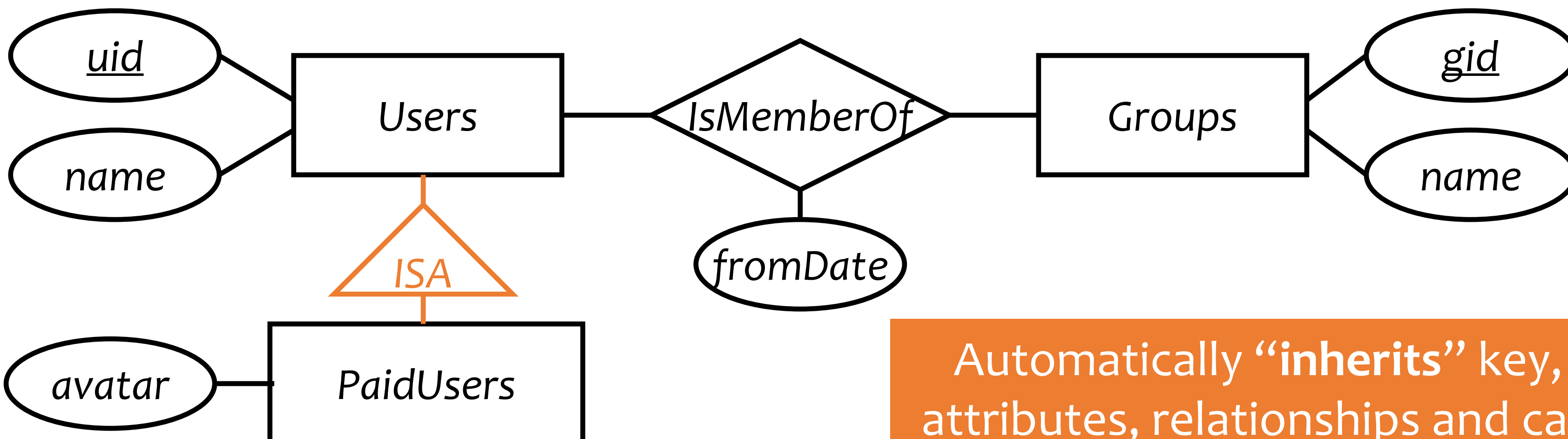
- If entity E's existence depends on entity F, then
  - F is a dominant entity
  - E is a subordinate entity
  - Example: *Rooms* inside *Buildings* are partly identified by *Buildings*' name



- 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 + total participation relationship to a distinct entity set

ISA relationships

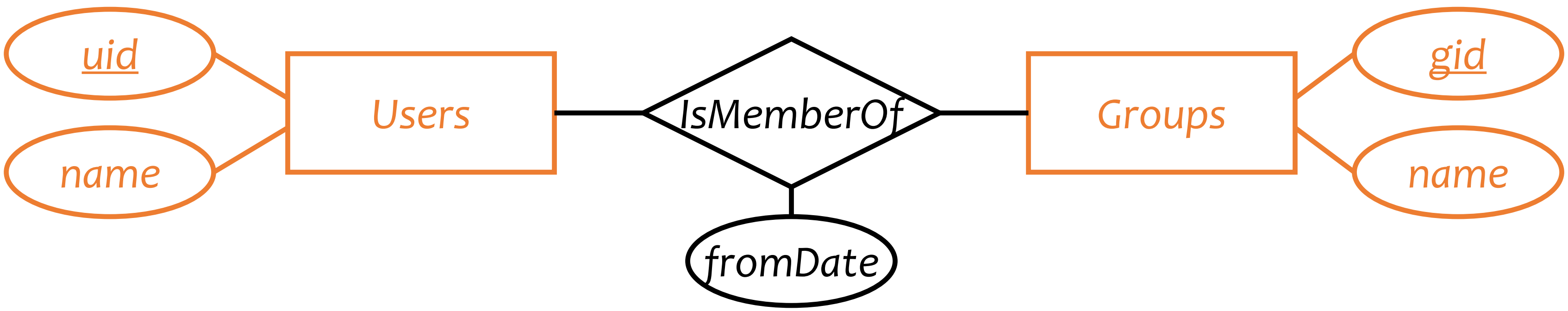
- Similar to the idea of subclasses in object-oriented programming: subclass = special case, fewer entities, and possibly more properties
  - Represented as a triangle (direction is important)
- Example: paid users are users, but they also get avatars (yay!)



Automatically "**inherits**" key, attributes, relationships and can participate in other relationships

E/R Translation

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

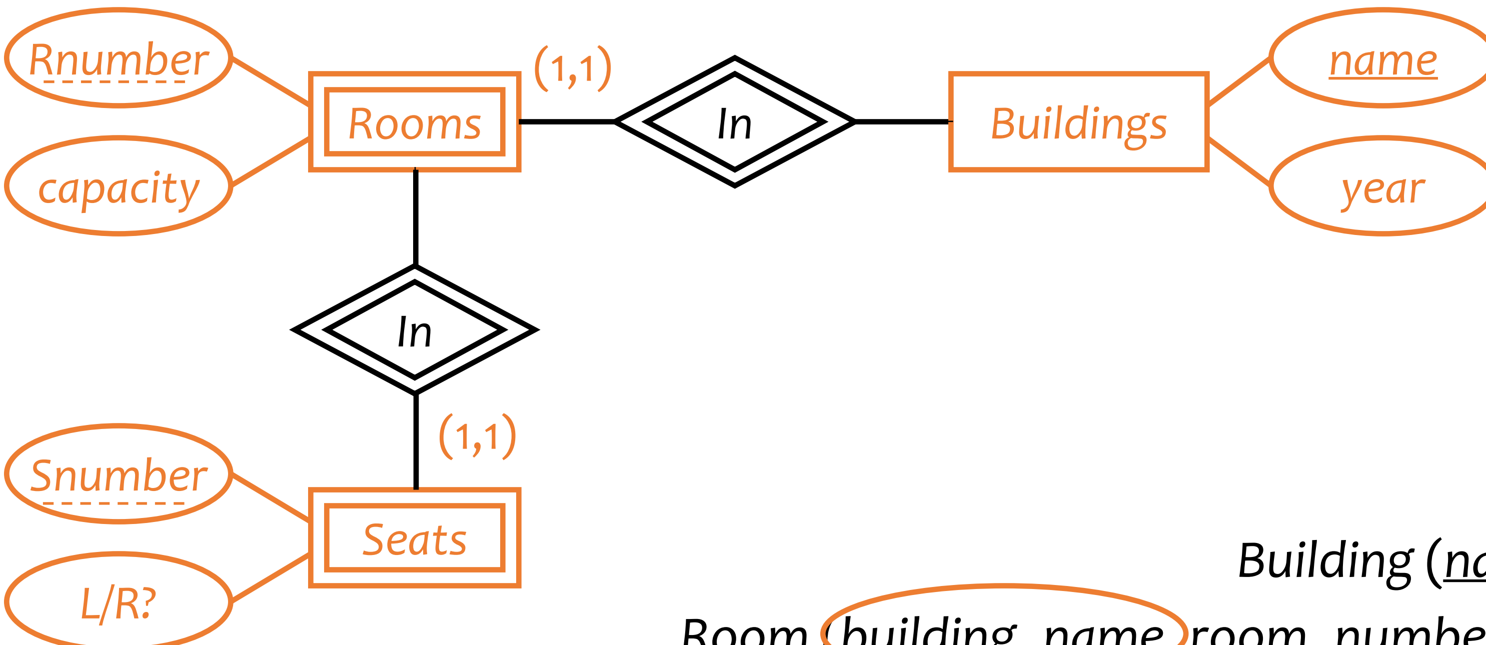


User (<u>uid</u>, name)

Group (<u>gid</u>, name)

Translating weak entity sets

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



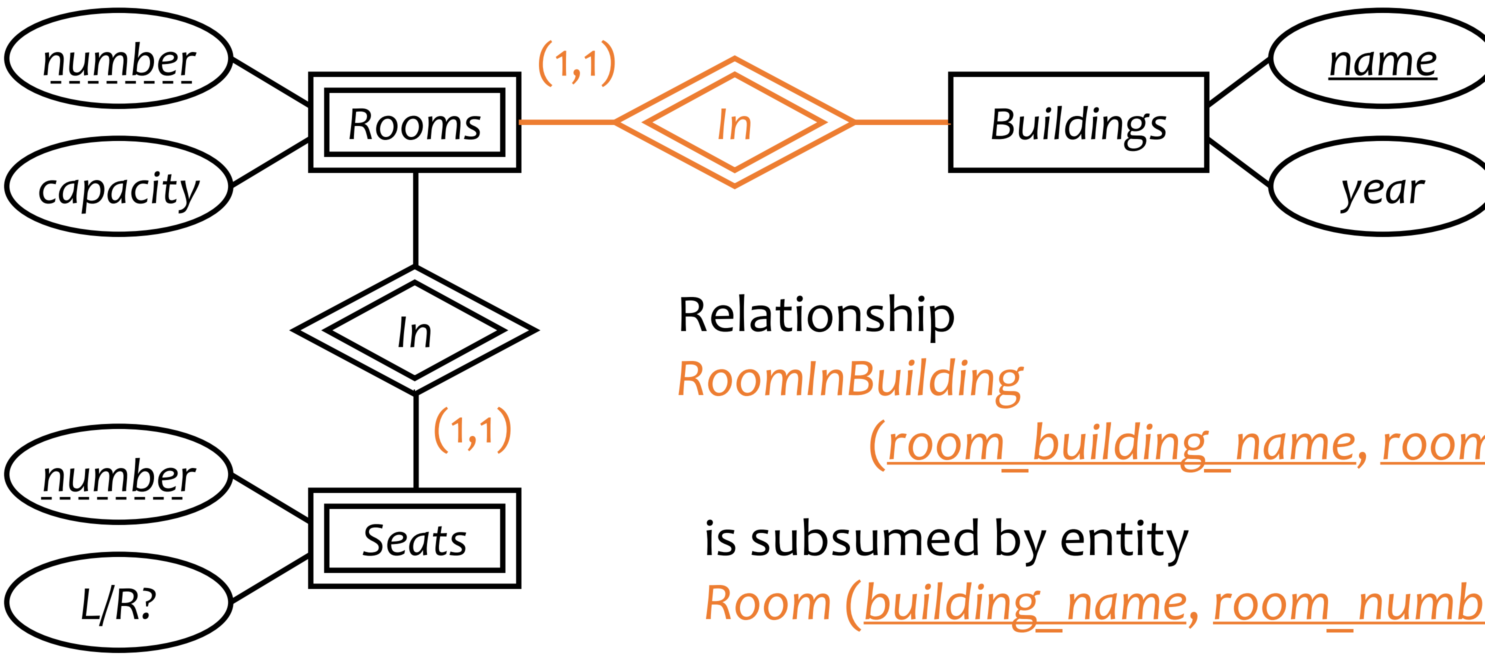
Building (name, year)

Room (building_name, room_number, capacity)

Seat (building_name, room_number, seat_number, left_or_right)

Translating double diamonds?

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

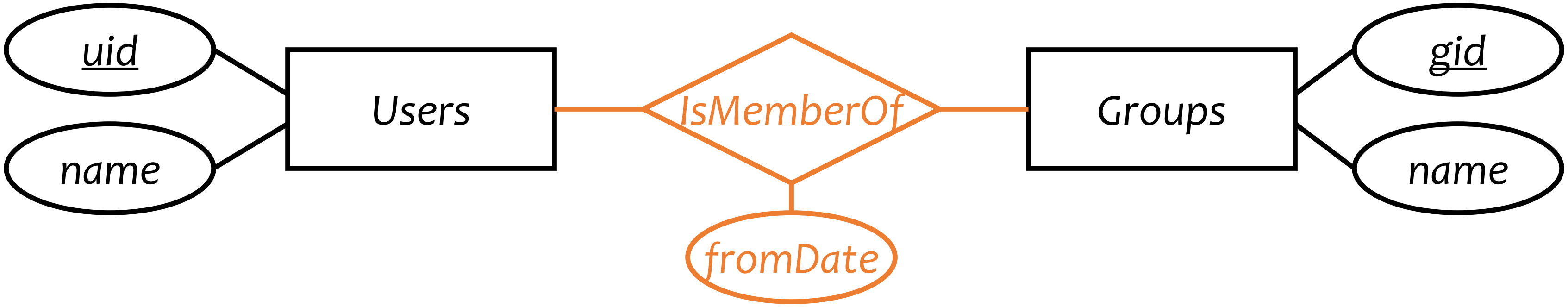


Relationship
RoomInBuilding
(room_building_name, room_number,)

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

Translating relationship sets

- A relationship set translates to a table
  - Keys of connected entity sets → columns
  - Attributes of the relationship set (if any) → columns
  - Multiplicity of the relationship set determines the key of the table

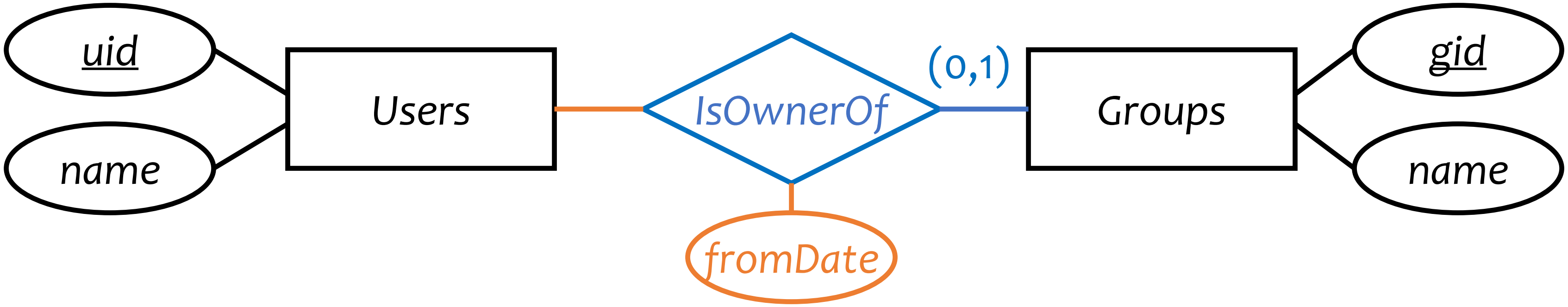


Member (<u>uid</u>, <u>gid</u>, fromDate)

- If we can deduce the general cardinality constraint (0,1) for a component entity set E, then take the primary key attributes for E
- Otherwise, choose primary key attributes of each component entity

Translating relationship sets

- A relationship set translates to a table
  - Keys of connected entity sets → columns
  - Attributes of the relationship set (if any) → columns
  - Multiplicity of the relationship set determines the key of the table



Owner (uid, gid, fromDate)

- If we can deduce the general cardinality constraint (0,1) for a component entity set E, then take the primary key attributes for E
- Otherwise, choose primary key attributes of each component entity

Translating subclasses & ISA

- Entity-in-all-superclasses
  - *User* (<u>*uid*</u>, *name*), *PaidUser* (<u>*uid*</u>, *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* (<u>*uid*</u>, *name*), *PaidUser* (<u>*uid*</u>, *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* (<u>*uid*</u>, [*type*], *name*, *avatar*)
  - Pro: Everything is in one table
  - Con: Lots of NULL's; complicated if class hierarchy is complex

Database Design theory

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
  - 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 $X \rightarrow R$ (all the attributes)

Armstrong's Axioms

- Reflexivity: if $Y \subseteq X$, then $X \rightarrow Y$
- Augmentation: if $X \rightarrow Y$, then $XZ \rightarrow YZ$
- Transitivity: if $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

Implications of Armstrong's Axioms

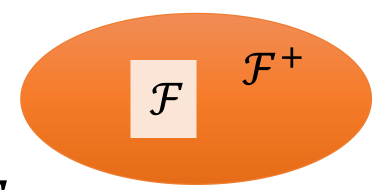
- Decomposition: If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$
- Union: If $X \rightarrow Y$ and $X \rightarrow Z$ then $X \rightarrow YZ$
- Pseudo-transitivity: If $X \rightarrow Y$ and $YZ \rightarrow T$ then $XZ \rightarrow T$
- Using Armstrong's Axioms, you can prove or disprove 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 $\{A_1, A_2, \dots\}$ functionally determined by Z (that is, Z $\rightarrow A_1 A_2 \dots$)

- 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

Using attribute closure

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

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

Lossless decomposition

- We should be able to reconstruct 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 if and only if the common attributes of R_1 and R_2 form a superkey for either schema, i.e., $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$

Dependency-preserving decomposition

- We should be able to (explicitly and implicitly) test all dependencies in each base table of the decomposition

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

Boyce-Codd Normal Form (BCNF)

- A relation R is in BCNF under $\mathcal{F}$ if each FD $X \rightarrow Y \in \mathcal{F}^+$ with $XY \subseteq R$ satisfies:
  - either $X \rightarrow Y$ is trivial, i.e., $Y \subseteq X$
  - or X is a super key of R, i.e., $X \rightarrow R$

$\mathcal{F}$ includes:
$A, B \rightarrow C$
$C \rightarrow B$

- Is $R = \{A, B, C\}$ under $\mathcal{F}$ in BCNF? NO!
  - $C \rightarrow B$ is a violation since C is not a super key of R

Compute BCNF decomposition

Repeat the following until all relations are in BCNF

- Step 1: Find a BCNF violation
  - A relation R
  - A non-trivial FD $X \rightarrow Y$ in $\mathcal{F}^+$ with $XY \subseteq R$, where X is not a super key of R
- Step 2: Decompose R into R_1 and R_2
  - 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

- BCNF is lossless!

Is BCNF dependency-preserving?

- NO!
- Consider a simple example R under $\mathcal{F}$:

$R = \{A, B, C\}$

BCNF violation: $C \rightarrow B$

$R_1 = \{A, C\}$

In BCNF since no FD in $\mathcal{F}^+$ only contain A, C

$R_2 = \{C, B\}$

In BCNF since $C \rightarrow B$

$\mathcal{F}$ includes:

$A, B \rightarrow C$

$C \rightarrow B$

$A, B \rightarrow C$ is cross-table and cannot be tested directly

Third normal form (3NF)

- A relation R is in 3NF under $\mathcal{F}$ if each FD $X \rightarrow Y \in \mathcal{F}^+$ with $XY \subseteq R$ satisfies:
  - 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,
  - or each attribute in $Y - X$ is in a key of R

BCNF only allows the first two cases, so 3NF is looser than BCNF

- Is $R = \{A, B, C\}$ under $\mathcal{F}$ in 3NF? YES!
  - $A, B \rightarrow C$ is satisfied since AB is a super key
  - $C \rightarrow B$ is satisfied since B is part of key $\{A, B\}$

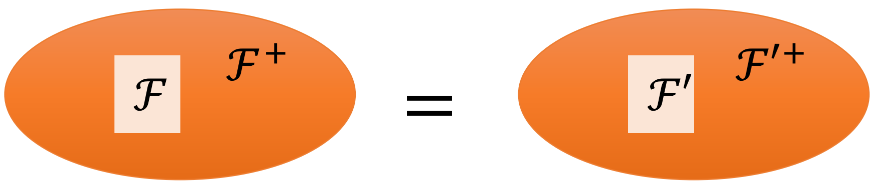
$\mathcal{F}$ includes:

$A, B \rightarrow C$

$C \rightarrow B$

Compute 3NF decomposition

- Step 1: Finding the minimal cover of the FD set $\mathcal{F}$



  - Given a set of FDs $\mathcal{F}$, we say $\mathcal{F}'$ is equivalent to $\mathcal{F}$ if their closures are the same, i.e., $\mathcal{F}^+ = \mathcal{F}'^+$.
  - The smallest equivalent set of FDs

- Step 2: Decompose based on the minimal cover

Minimal cover

A set of FDs $\mathcal{F}$ is minimal if

- every right-hand side of a FD in $\mathcal{F}$ is a single attribute
- there does not exist $X \rightarrow A$ *with* Z *as a proper subset of* X, such that $(\mathcal{F} - \{X \rightarrow A\}) \cup \{Z \rightarrow A\}$ is equivalent to $\mathcal{F}$
- there does not exist $X \rightarrow A$ in $\mathcal{F}$ such that $\mathcal{F} - \{X \rightarrow A\}$ equivalent to $\mathcal{F}$

Compute minimal cover of $\mathcal{F}$

Repeat the following steps until $\mathcal{F}$ does not change

- Step 1: Replace $X \rightarrow YZ$ with $X \rightarrow Y$ and $X \rightarrow Z$
- Step 2: Remove A from the LHS of $X \rightarrow B$ if B is in the attribute closure of $X - \{A\}$ until $\mathcal{F}$
- Step 3: Remove $X \rightarrow A$ if A is in the attribute closure of X under $\mathcal{F} - \{X \rightarrow A\}$

Compute 3NF decomposition

Given a relation R with a set $\mathcal{F}$ of FDs:

Step 1: Find a minimal cover $\mathcal{F}^*$ for $\mathcal{F}$

Step 2: For every $X \rightarrow Y$ in $\mathcal{F}^*$, add a relation {X, Y} to the decomposition

Step 3: If no relation contains a key for R, add a relation containing an arbitrary key for R to the decomposition

BCNF v.s. 3NF

- Both BCNF and 3NF are lossless
- BCNF is not necessarily dependency-preserving but 3NF is dependency-preserving

$\mathcal{F}$ includes:

$A, B \rightarrow C$

$C \rightarrow B$

$R = \{A, B, C\}$

BCNF violation: $C \rightarrow B$

$R_1 = \{A, C\}$

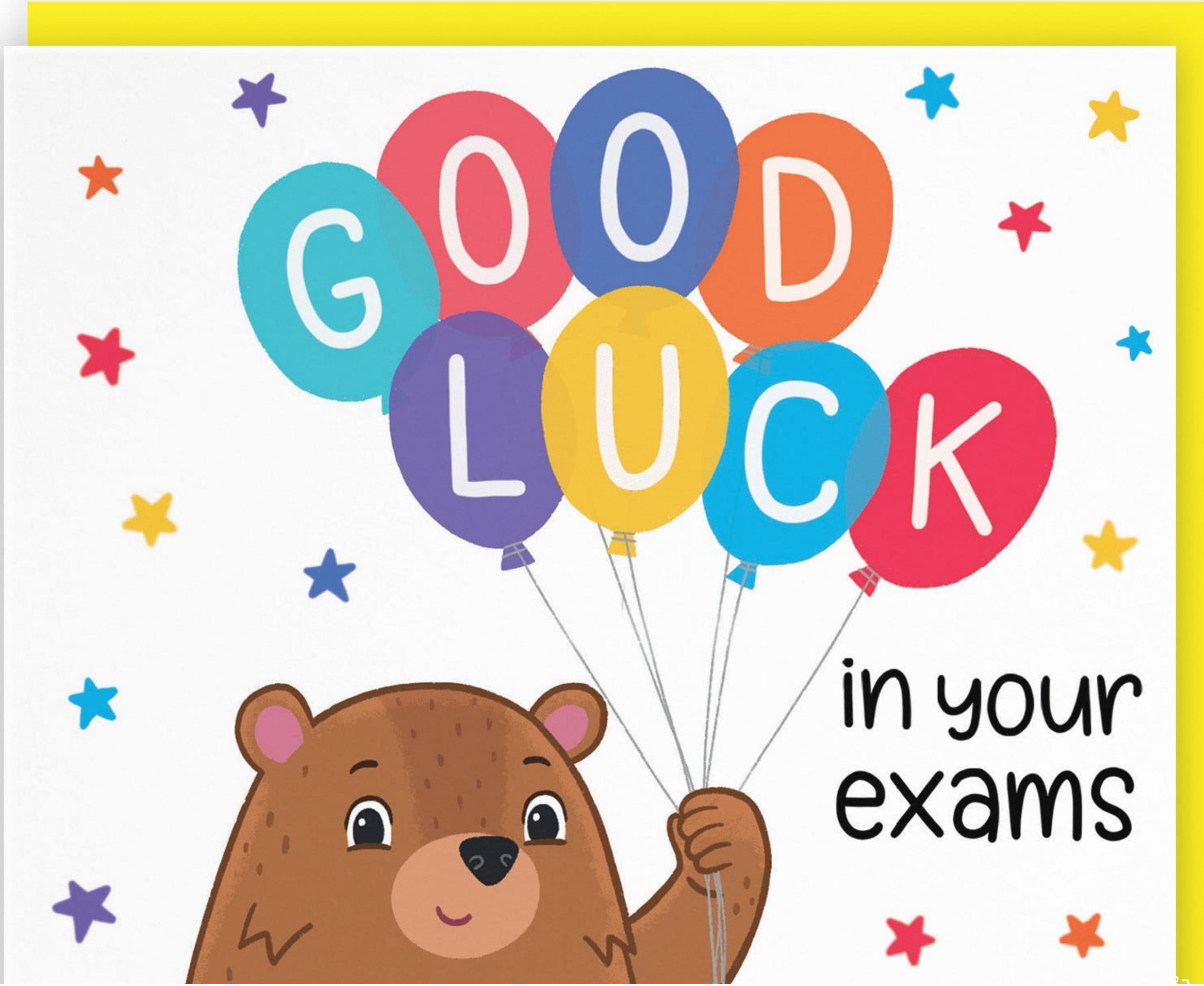
In BCNF since no FD in $\mathcal{F}^+$ only contain A, C

$R_2 = \{C, B\}$

In BCNF since $C \rightarrow B$

$A, B \rightarrow C$ is cross-table and cannot be tested directly

- 3NF contains possible more redundancy than BCNF


GOOD LUCK
in your exams