

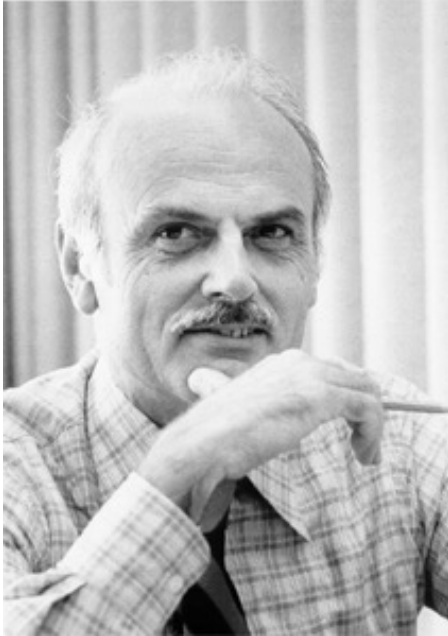
Lecture 2:

Relational Model

CS348 Spring 2025:
Introduction to Database Management

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

Edgar F. Codd (1923-2003)



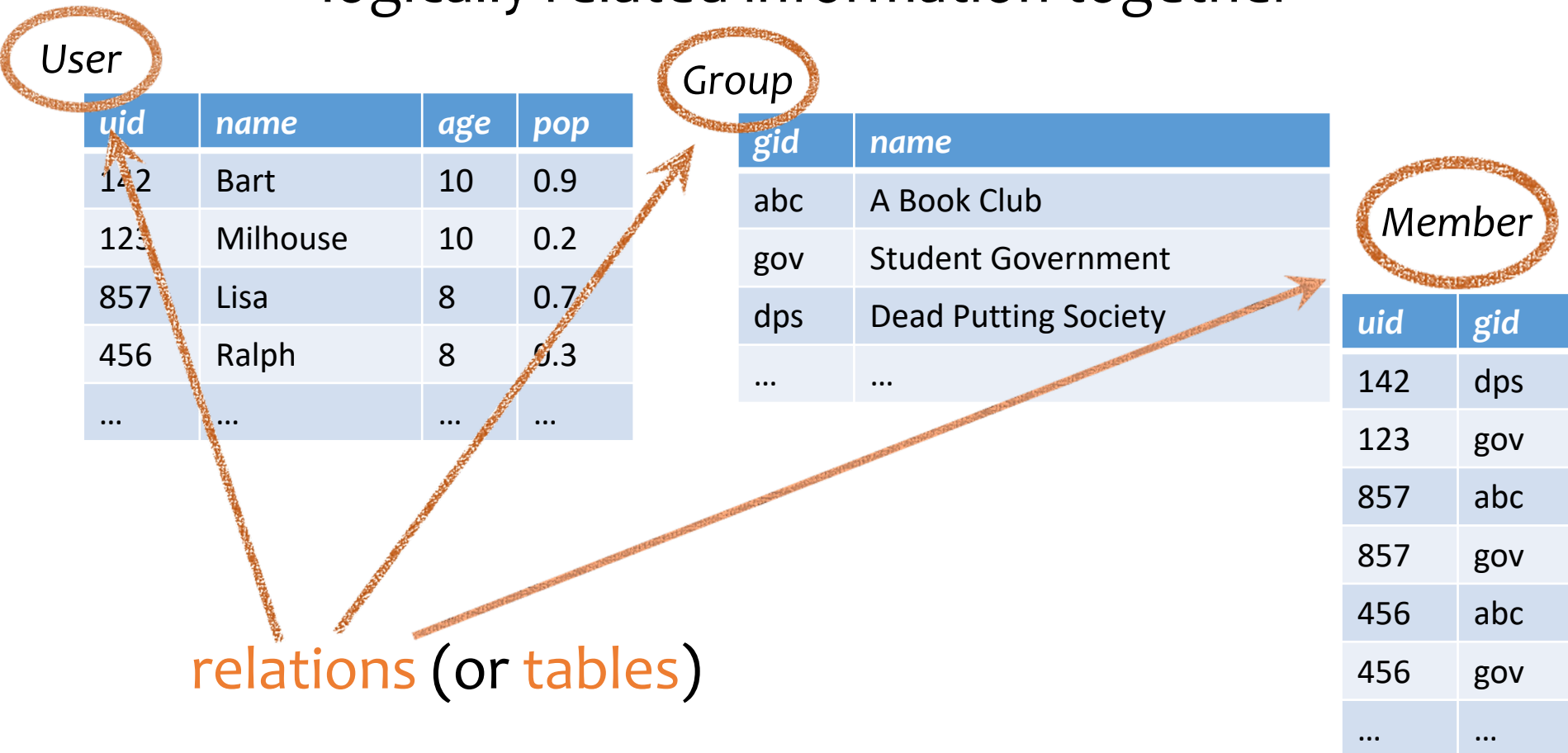
- Inventor of the relational model and algebra while at IBM
- Turing Award, 1981
- Pilot in the Royal Air Force in WW2

Outline

- Part 1: Relational data model
- Part 2: Relational algebra

Relational data model

Modeling data as **relations** or **tables**, each storing logically related information together



Attributes

User

<i>uid</i>	<i>name</i>	<i>age</i>	<i>pop</i>
142	Bart	10	0.9
123	Milhouse	10	0.2
857	Lisa	8	0.7
456	Ralph	8	0.3
...

Group

<i>gid</i>	<i>name</i>
abc	A Book Club
gov	Student Government
dps	Dead Putting Society
...	...

Member

<i>uid</i>	<i>gid</i>
142	dps
123	gov
857	abc
857	gov
456	abc
456	gov
...	...

attributes (or columns)

Domain

User

<i>uid</i>	<i>name</i>	<i>age</i>	<i>pop</i>
142	Bart	10	0.9
123	Milhouse	10	0.2
857	Lisa	8	0.7
456	Ralph	8	0.3
...

String

Int

Float

domain (or type)

Group

<i>gid</i>	<i>name</i>
abc	A Book Club
gov	Student Government
dps	Dead Putting Society
...	...

Member

<i>uid</i>	<i>gid</i>
142	dps
123	gov
857	abc
857	gov
456	abc
456	gov
...	...

Tuples

User

<i>uid</i>	<i>name</i>	<i>age</i>	<i>pop</i>
142	Bart	10	0.9
123	Milhouse	10	0.2
857	Lisa	8	0.7
456	Ralph	8	0.3
...

tuples (or rows)

Duplicates (all attr. have same val) are not allowed

Ordering of rows doesn't matter
(even though output can be ordered)

Group

<i>gid</i>	<i>name</i>
abc	A Book Club
gov	Student Government
dps	Dead Putting Society
...	...

Member

<i>uid</i>	<i>gid</i>
142	dps
123	gov
857	abc
857	gov
456	abc
456	gov
...	...

Set representation of tuples

User

<i>uid</i>	<i>name</i>	<i>age</i>	<i>pop</i>
142	Bart	10	0.9
123	Milhouse	10	0.2
857	Lisa	8	0.7
456	Ralph	8	0.3
...

Group

<i>gid</i>	<i>name</i>
abc	A Book Club
gov	Student Government
edu	Dead Putting Society
...	...

Member

<i>uid</i>	<i>gid</i>
142	dps
123	gov
857	abc
857	gov
456	abc
456	gov
...	...


User: $\{\langle 142, \text{Bart}, 10, 0.9 \rangle, \langle 857, \text{Milhouse}, 10, 0.2 \rangle, \dots\}$

Group: $\{\langle \text{abc}, \text{A Book Club} \rangle, \langle \text{gov}, \text{Student Government} \rangle, \dots\}$

Member: $\{\langle 142, \text{dps} \rangle, \langle 123, \text{gov} \rangle, \dots\}$

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**

👉 **Simplicity is a virtue!**

Schema vs. Instance

- **Schema (metadata)**

- Specifies the **logical structure** of data
- Is defined at setup time, rarely changes

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

- **Instance**

- Represents the data content
- Changes rapidly, but always **conforms** to the schema
- Typically has additional rules

User: {⟨142, Bart, 10, 0.9⟩, ⟨857, Milhouse, 10, 0.2⟩, ...}
Group: {⟨abc, A Book Club⟩, ⟨gov, Student Government⟩, ...}
Member: {⟨142, dps⟩, ⟨123, gov⟩, ...}

Integrity constraints

- A set of rules that database instances should follow
- Example:
 - *age* cannot be **negative**
 - *uid* should be **unique** in the *User* relation
 - *uid* in *Member* must **refer to** a row in *User*

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

Group (*gid* string, *name* string)

Member (*uid* int, *gid* string)

User: {⟨142, Bart, 10, 0.9⟩, ⟨857, Milhouse, 10, 0.2⟩, ...}

Group: {⟨abc, A Book Club⟩, ⟨gov, Student Government⟩, ...}

Member: {⟨142, dps⟩, ⟨857, gov⟩, ...}

Integrity constraints

- An instance is only **valid** if it follows the schema and satisfies **all** the integrity constraints.
- Reasons to use constraints:
 - Address consistency challenges
 - Ensure data modification respects to database design
 - Protect data from bugs in applications

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 (week 5/6)
- Database-level
 - Referential integrity – **foreign key** (focus in this lecture)
 - E.g. *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

Key (Candidate key)

<i>uid</i>	<i>name</i>	<i>age</i>	<i>pop</i>
142	Bart	10	0.9
123	Milhouse	10	0.2
857	Lisa	8	0.7
456	Ralph	8	0.3

- Is *name* a key of *User*?
 - Yes? Seems reasonable for this instance
 - No! User names are not unique **in general**
- Key declarations are part of the schema

More examples of keys

- Only **uid**?
 - No, because of repeated entries
- Only **gid**?
 - No, again due to repeated entries
- Use both! $\{uid, gid\}$

Member

uid	gid
142	dps
123	gov
857	abc
456	gov
857	dps
256	gov
...	...

👉 A key can contain multiple attributes!

More examples of keys

- Address (street_address, city, province, zip)
 - Key 1: {street_address, city, province}
 - Key 2: {street_address, zip}

👉 A relation can have multiple keys!

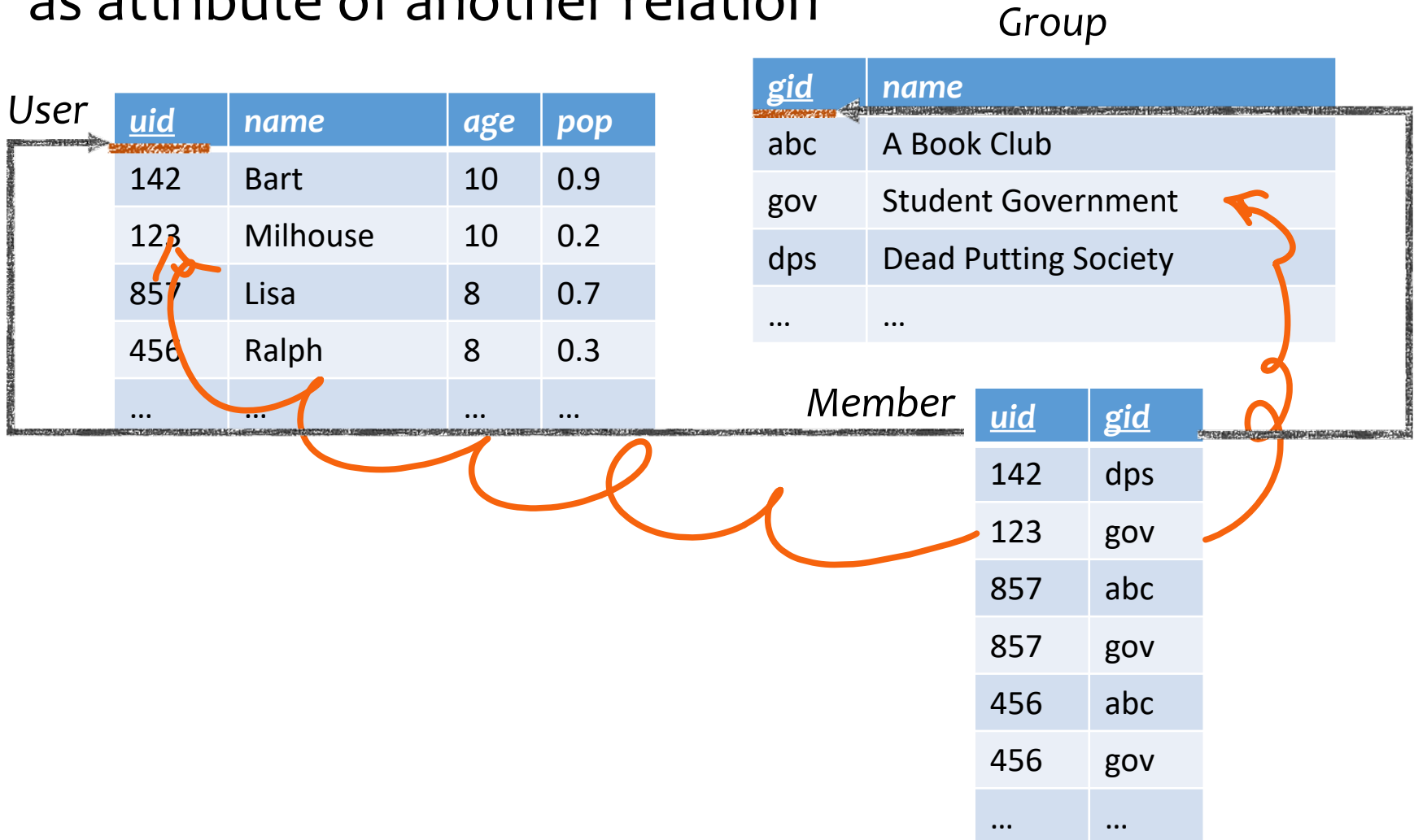
- **Primary key:** a **designated** candidate key in the schema declaration
 - Underline all its attributes
 - E.g., Address (street_address, city, province, zip)

Use of keys

- More constraints on data, fewer mistakes
- Look up a row by its key value
 - Many selection conditions are “key = value”
- “Pointers” to other rows (often across tables)

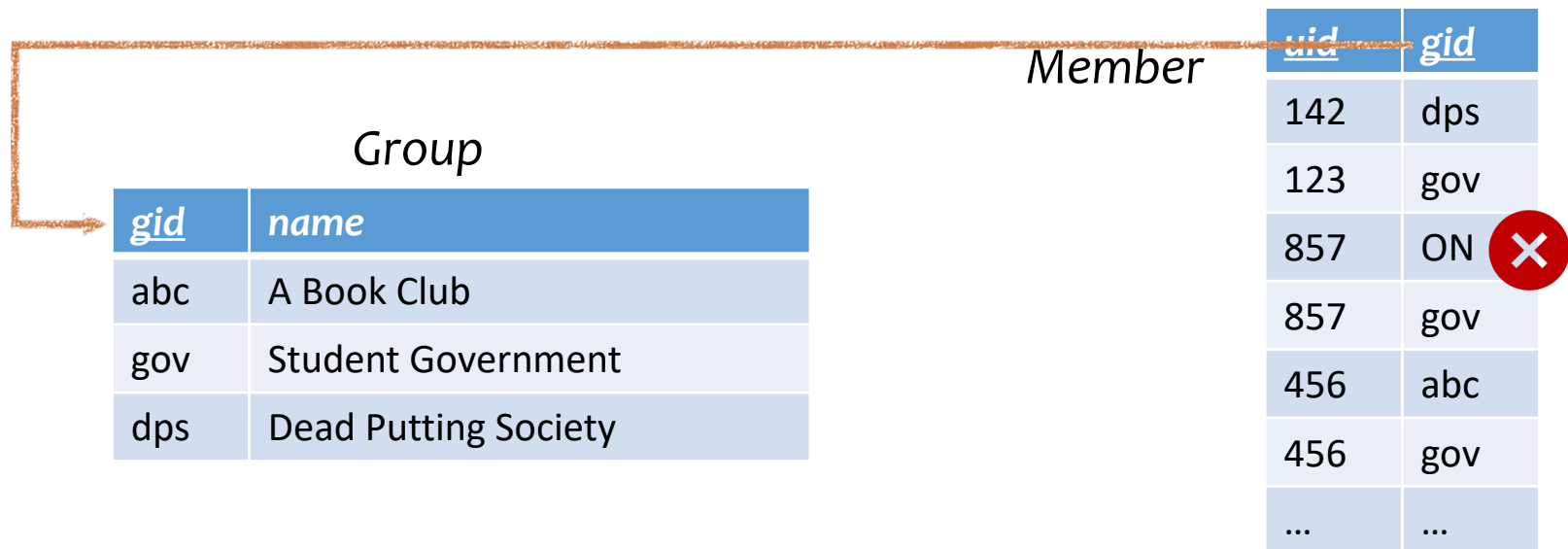
“Pointers” to other rows

- **Foreign key**: primary key of one relation appearing as attribute of another relation



“Pointers” to other rows

- **Referential integrity**: A tuple with a non-null value for a foreign key **must** match the primary key value of a tuple in the referenced relation



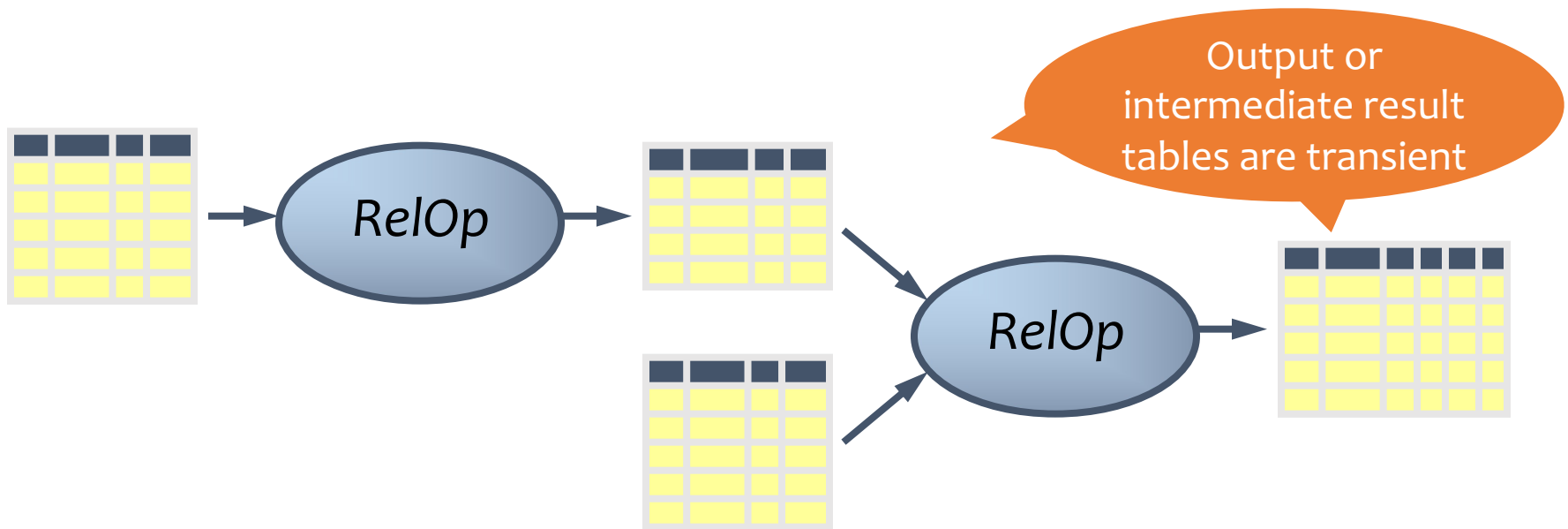
Referential integrity violation!

Outline

- Part 1: Relational data model
 - Data model
 - Database schema
 - Integrity constraints (keys)
 - Languages
 - Relational algebra (focus in this lecture)
 - SQL (next 6 lectures)
 - Relational calculus (textbook, Ch. 27)
- Part 2: Relational algebra

Relational algebra

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



Relational algebra

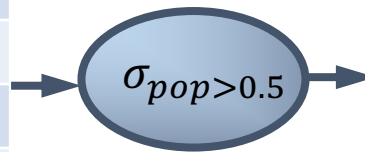
- Core operators:
 - Selection
 - Projection
 - Cross product
 - Union
 - Difference
 - Renaming
- Additional, derived operators:
 - Join, Natural join, Intersection, etc.
- Compose operators to make complex queries
 - Division

Core operator 1: Selection σ

- Example query: Users with popularity higher than 0.5

$$\sigma_{pop > 0.5} User$$

<i>uid</i>	<i>name</i>	<i>age</i>	<i>pop</i>
142	Bart	10	0.9
123	Milhouse	10	0.2
857	Lisa	8	0.7
456	Ralph	8	0.3
...



<i>uid</i>	<i>name</i>	<i>age</i>	<i>pop</i>
142	Bart	10	0.9
857	Lisa	8	0.7
...

Core operator 1: Selection

- Input: a table R
- Notation: $\sigma_p R$
 - p is called a **selection condition** (or **predicate**)
- Purpose: filter rows according to some criteria
- Output: same columns as R , but only rows of R that satisfy p

More on selection

- Selection condition can include any column of R , constants, comparison ($=, \neq, <, \leq, >, \geq$ etc.) and Boolean connectives (\wedge : and, \vee : or, \neg : not)
 - Example: users with popularity at least 0.9 and age under 10 or above 12

$$\sigma_{pop \geq 0.9 \wedge (age < 10 \vee age > 12)} User$$

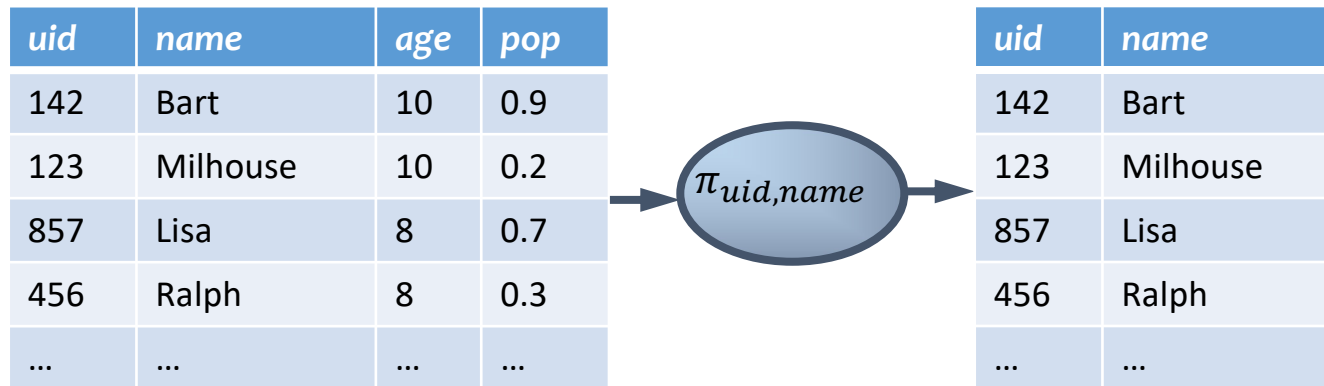
- You must be able to evaluate the condition over **each single row** of the input table!
 - Example: the most popular user

$$\sigma_{pop \geq \text{every pop in } User} User \text{ **WRONG!**}$$

Core operator 2: Projection π

- Example: IDs and names of all users

$$\pi_{uid, name} User$$



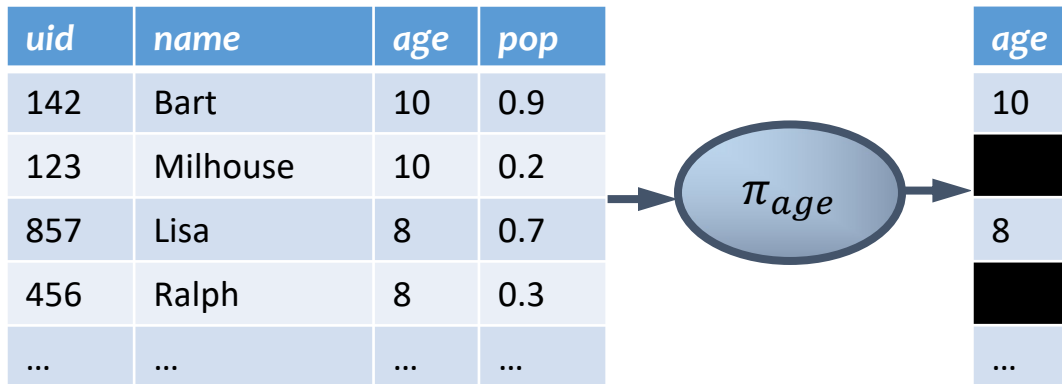
Core operator 2: Projection

- Input: a table R
- Notation: $\pi_L R$
 - L is a list of columns in R
- Purpose: output chosen columns
- Output: “same” rows, but only the columns in L

More on projection

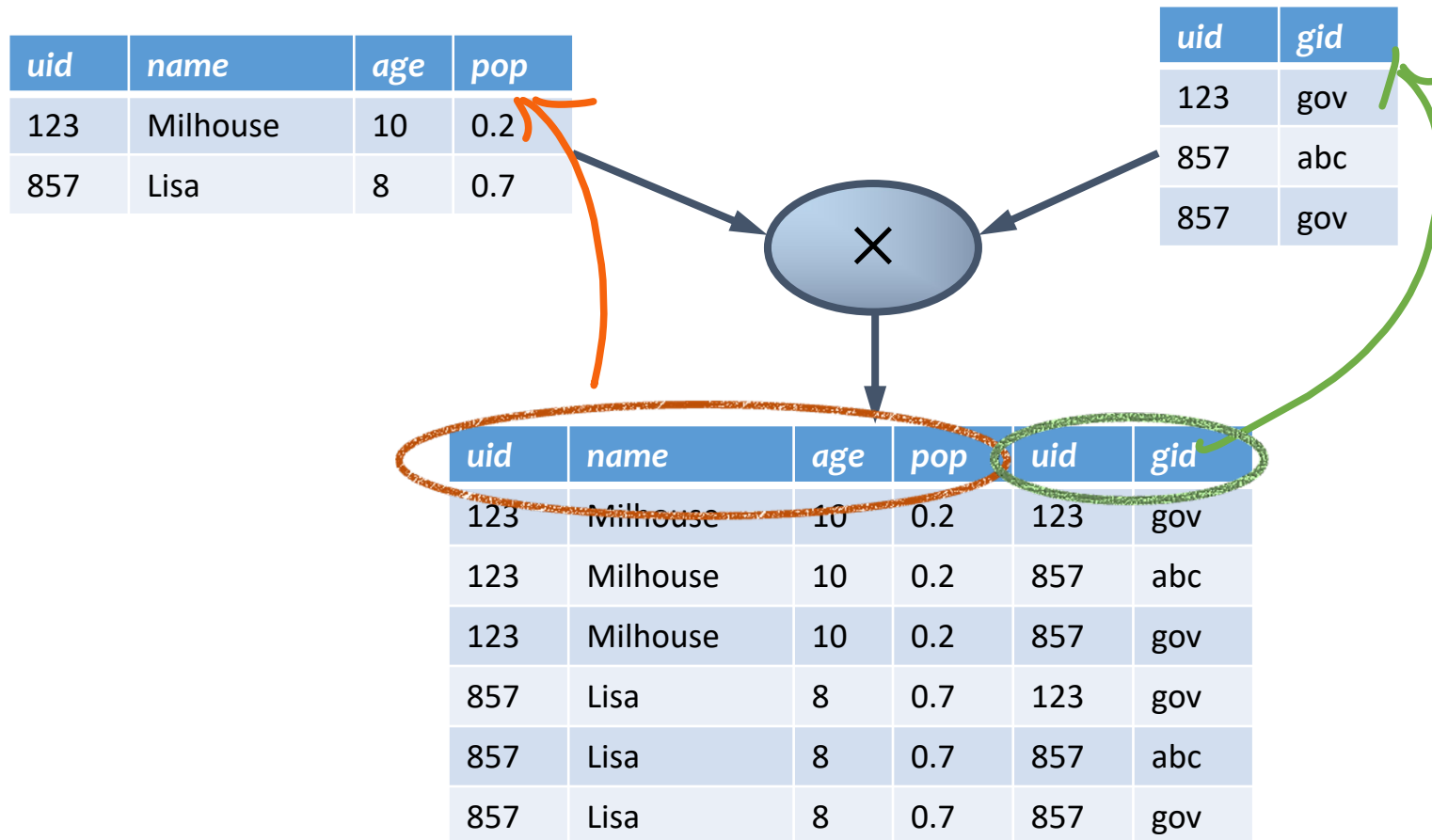
- Duplicate output rows are removed (by definition)
 - Example: user ages

$\pi_{age} User$



Core operator 3: Cross product \times

User \times *Member*



Core operator 3: Cross product

- Input: two tables R and S
- Notation: $R \times S$
- Purpose: pairs rows from two tables
- Output: for each row r in R and each s in S , output a row rs (concatenation of r and s)

A note on column ordering

- Ordering of columns is unimportant as far as contents are concerned

<i>uid</i>	<i>name</i>	<i>age</i>	<i>pop</i>	<i>uid</i>	<i>gid</i>
123	Milhouse	10	0.2	123	gov
123	Milhouse	10	0.2	857	abc
123	Milhouse	10	0.2	857	gov
857	Lisa	8	0.7	123	gov
857	Lisa	8	0.7	857	abc
857	Lisa	8	0.7	857	gov
...

=

<i>uid</i>	<i>gid</i>	<i>uid</i>	<i>name</i>	<i>age</i>	<i>pop</i>
123	gov	123	Milhouse	10	0.2
857	abc	123	Milhouse	10	0.2
857	gov	123	Milhouse	10	0.2
123	gov	857	Lisa	8	0.7
857	abc	857	Lisa	8	0.7
857	gov	857	Lisa	8	0.7
...

- So cross product is **commutative**, i.e., for any R and S , $R \times S = S \times R$ (up to the ordering of columns)

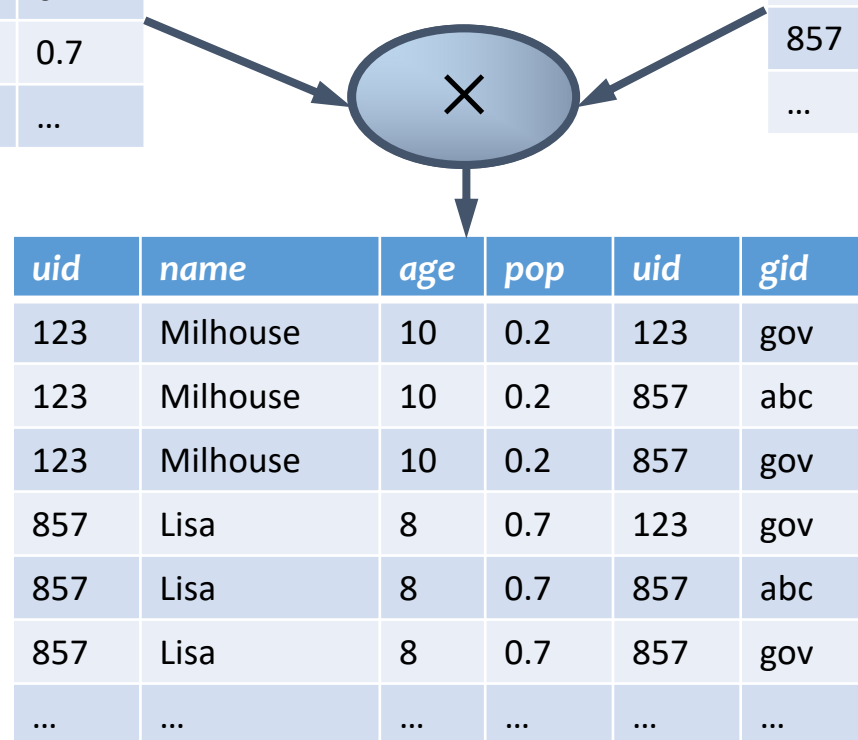
Derived operator 1: Join ⋈

- Info about users, plus IDs of their groups

$User \bowtie_{User.uid=Member.uid} Member$

<i>uid</i>	<i>name</i>	<i>age</i>	<i>pop</i>
123	Milhouse	10	0.2
857	Lisa	8	0.7
...

<i>uid</i>	<i>gid</i>
123	gov
857	abc
857	gov
...	...



Derived operator 1: Join ⋈

- Info about users, plus IDs of their groups

$User \bowtie_{User.uid=Member.uid} Member$

<i>uid</i>	<i>name</i>	<i>age</i>	<i>pop</i>
123	Milhouse	10	0.2
857	Lisa	8	0.7
...

<i>uid</i>	<i>gid</i>
123	gov
857	abc
857	gov
...	...



<i>uid</i>	<i>name</i>	<i>age</i>	<i>pop</i>	<i>uid</i>	<i>gid</i>
123	Milhouse	10	0.2	123	gov
857	Lisa	8	0.7	857	abc
857	Lisa	8	0.7	857	gov
...

Derived operator 1: Join ⋈

- Info about users, plus IDs of their groups

$User \bowtie_{User.uid=Member.uid} Member$

uid	name	age	pop
123	Milhouse	10	0.2
857	Lisa	8	0.7
...

uid	gid
123	gov
857	abc
857	gov
...	...



uid	name	age	pop	uid	gid
123	Milhouse	10	0.2	123	gov
857	Lisa	8	0.7	857	abc
857	Lisa	8	0.7	857	gov
...

Prefix a column reference with table name and “.” to disambiguate identically named columns from different tables

Derived operator 1: Join

- Input: two tables R and S
- Notation: $R \bowtie_p S$
 - p is called a **join condition** (or **predicate**)
- Purpose: relate rows from two tables according to some criteria
- Output: for each row r in R and each row s in S , output a row rs if r and s satisfy p
- Shorthand for $\sigma_p(R \times S)$
- (A.k.a. “**theta-join**”)

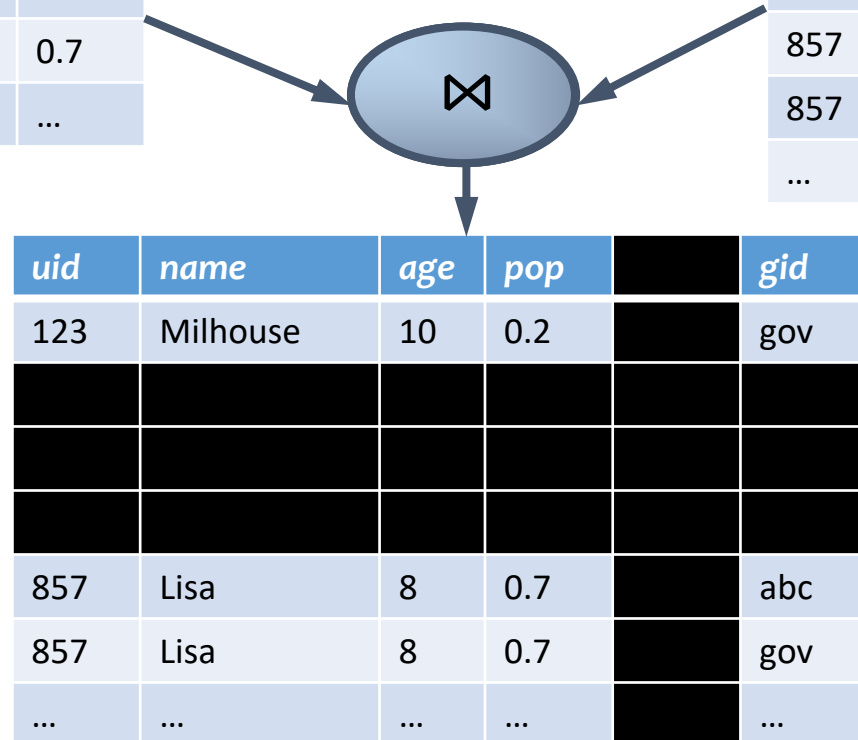
Derived operator 2: Natural join

$User \bowtie Member$

$= \pi_{uid, name, age, pop, gid} \left(User \bowtie_{\begin{smallmatrix} User.uid = \\ Member.uid \end{smallmatrix}} Member \right)$

uid	name	age	pop
123	Milhouse	10	0.2
857	Lisa	8	0.7
...




uid	gid
123	gov
857	abc
857	gov
...	...



Derived operator 2: Natural join

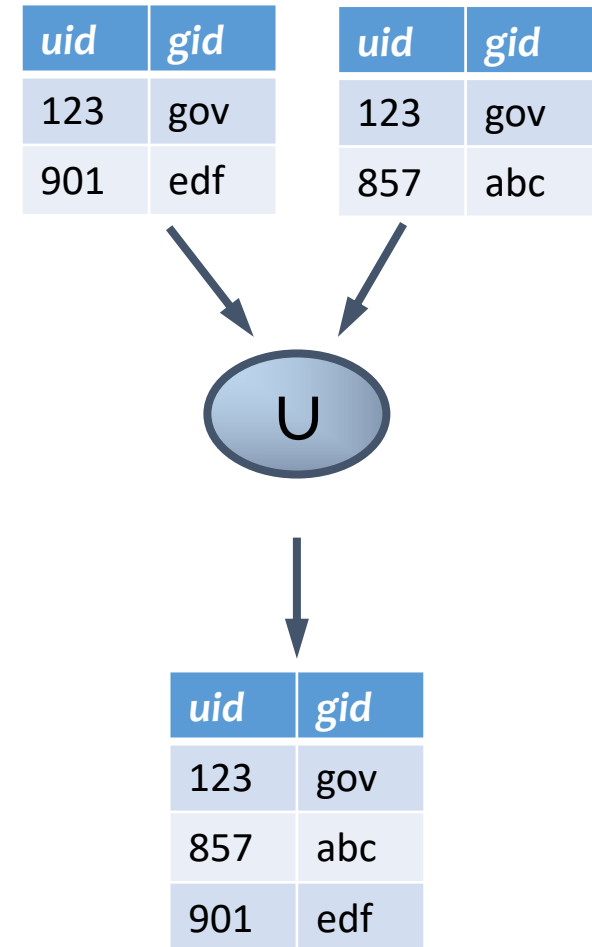
- Input: two tables R and S
- Notation: $R \bowtie S$
- Purpose: relate rows from two tables, and
 - Enforce equality between identically named columns
 - Eliminate one copy of identically named columns
- Shorthand for $\pi_L(R \bowtie_p S)$, where
 - p equates each pair of columns common to R and S
 - L is the union of column names from R and S (with duplicate columns removed)

Post-Lecture: Join v.s. Natural join

- Join: $R \bowtie_p S$
 - p is the **join condition** (with possible comparisons)
 - Output table contains all column names from R and S (**without** duplicate columns removed)
- Natural Join: $R \bowtie S$
 - Match all pairs of common columns between R and S
 - Output table contains the union of column names from R and S (**with** duplicate columns removed)
- What is the joining condition?
 - $R(A, B, C) \bowtie_{R.C \leq S.D} S(A, B, D)$  Only $R.C \leq S.D$
 - $R(A, B, C) \bowtie_{R.C \leq S.D \wedge R.A = S.A} S(A, B, D)$  $R.C \leq S.D$ and $R.A = S.A$
 - $R(A, B, C) \bowtie S(A, B, D)$  $R.A = S.A$ and $R.B = S.B$

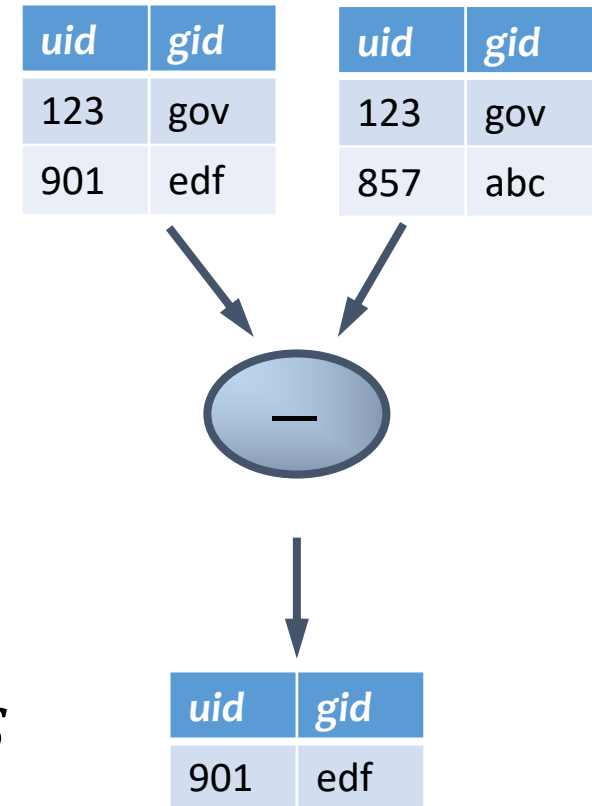
Core operator 4: Union

- Input: two tables R and S
- Notation: $R \cup S$
 - R and S must have identical schema
- Output:
 - Has the same schema as R and S
 - Contains all rows in R and all rows in S (with duplicate rows removed)



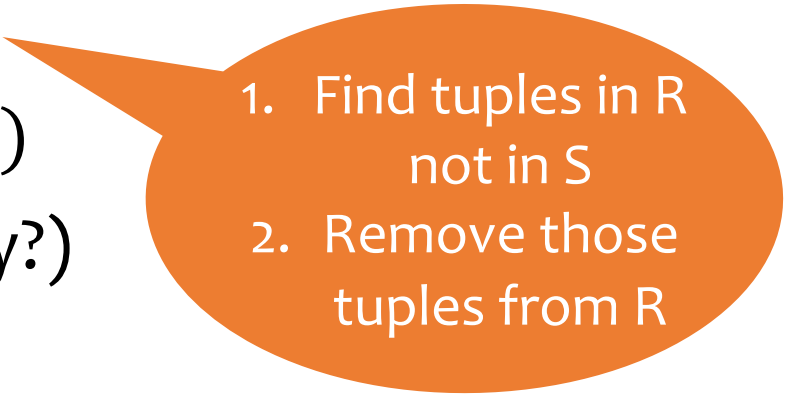
Core operator 5: Difference

- Input: two tables R and S
- Notation: $R - S$
 - R and S must have identical schema
- Output:
 - Has the same schema as R and S
 - Contains all rows in R that are not in S



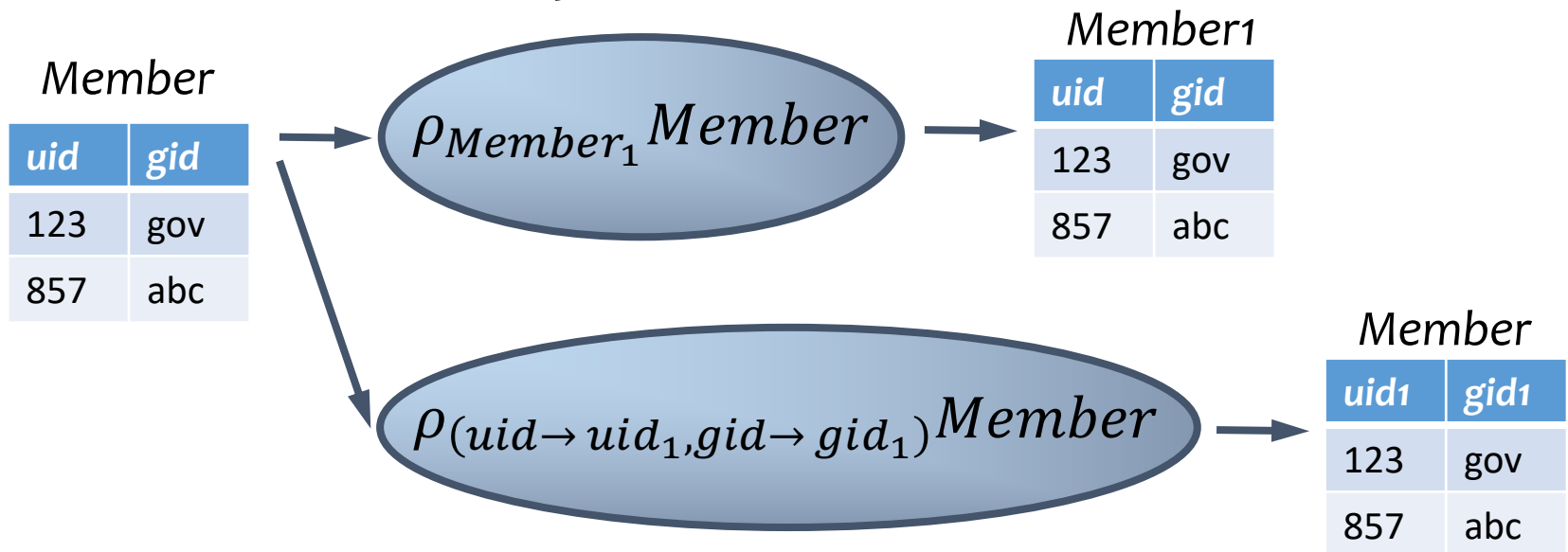
Derived operator 3: Intersection

- Input: two tables R and S
- Notation: $R \cap S$
 - R and S must have identical schema
- Output:
 - Has the same schema as R and S
 - Contains all rows that are in both R and S
- Shorthand for $R - (R - S)$
- Also equivalent to $S - (S - R)$
- And equivalent to $R \bowtie S$ (why?)

- 
1. Find tuples in R not in S
 2. Remove those tuples from R

Core operator 6: Renaming

- Input: a table $R(A_1, A_2, \dots, A_n)$
- Notation: $\rho_S R$, $\rho_{(A_1 \rightarrow A'_1, \dots)} R$, or $\rho_{S(A_1 \rightarrow A'_1, \dots)} R$
- Output: a table with the same rows as R , but called columns differently



Core operator 6: Renaming

- As with all other relational operators, it doesn't modify the database
 - Think of the renamed table as a copy of the original
- Used to
 - Create identical column names for natural joins
 - Example: $R(rid, \dots), S(sid, \dots)$
 - $R \bowtie_{rid=sid} S = (\rho_{(rid \rightarrow id)} R) \bowtie (\rho_{(sid \rightarrow id)} S)$
 - Avoid confusion caused by identical column name

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: $\rho_{S(A_1 \rightarrow A'_1, A_2 \rightarrow A'_2, \dots)} R$

Note: **Only** use these operators for assignments & exams

Derived Operators

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

Example

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

- All groups (ids) that Lisa belongs to

Example

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

- All groups (ids) that Lisa (a user's name) belongs to

Writing a query bottom-up:

uid	name	age	pop
857	Lisa	8	0.7

Who's Lisa? $\sigma_{name="Lisa"}$

User

uid	name	age	pop
123	Milhouse	10	0.2
857	Lisa	8	0.7
...

Member

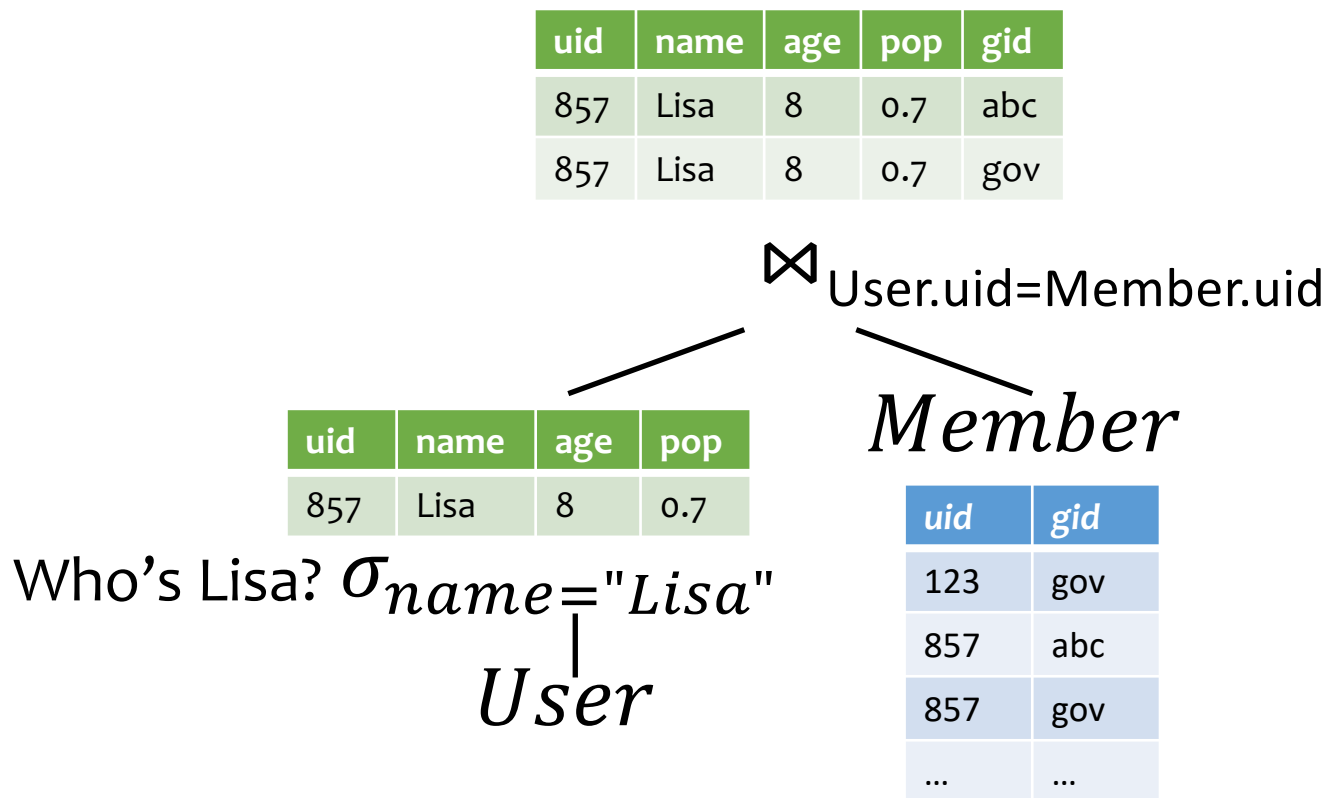
uid	gid
123	gov
857	abc
857	gov
...	...

Example

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

- All groups (ids) that Lisa belongs to

Writing a query bottom-up:

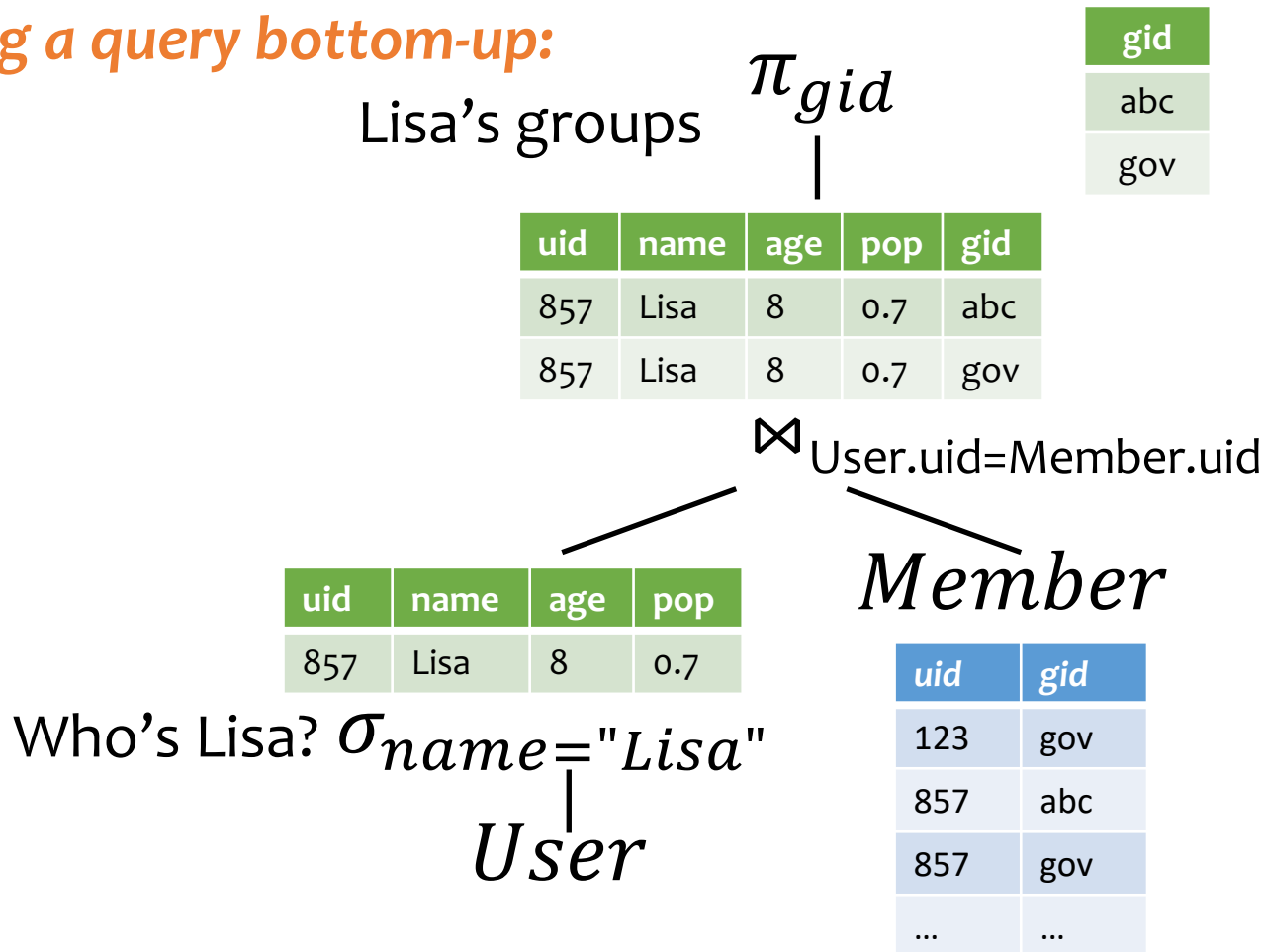


Example

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

- All groups (ids) that Lisa belongs to

Writing a query bottom-up:



Post-Lecture

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

- All groups (ids) that Lisa belongs to

$$\pi_{gid} \left(\begin{array}{c} (\sigma_{name="Lisa"} User) \\ \bowtie_{User.uid=Member.uid} Member \end{array} \right)$$

- There could be many different and equivalent ways to write the same query. We will discuss “efficiency” issue in Query Optimization!

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

$$\pi_{gid} \sigma_{name="Lisa" \wedge User.uid=Member.uid} User \times Member$$

Take home exercise

- Names of users in Lisa's group

Summary

- Part 1: Relational data model
 - Data model
 - Database schema
 - Integrity constraints (**keys**)
 - Languages (relational algebra, relational calculus, SQL)
- Part 2: Relational algebra – basic language
 - Core operators & derived operators
 - **Write a relational algebra query**
- What's next?
 - More example queries in relational algebra
 - Relational calculus (optional)
 - SQL (basic)