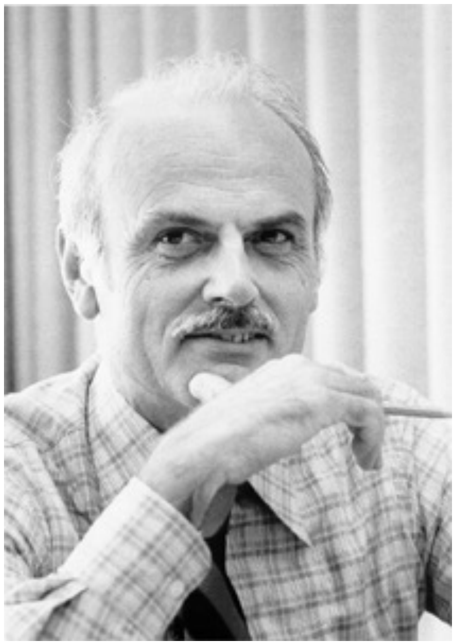
Intro to the Relational Model

CS348 Spring 2023

Instructor: Sujaya Maiyya

Sections: **002 and 004 only**

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

http://en.wikipedia.org/wiki/File:Edgar_F_Codd.jpg

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

User

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

Group

gid	*name*
abc	A Book Club
gov	Student Government
dps	Dead Putting Society
…	…

Member

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

relations (or tables)

Attributes

User

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

Group

gid	*name*
abc	A Book Club
gov	Student Government
dps	Dead Putting Society
…	…

Member

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

attributes (or columns)

Domain

Group

gid	*name*
abc	A Book Club
gov	Student Government
dps	Dead Putting Society
…	…

User

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

String Int Float

Member

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

domain (or type)

Tuples

User

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

Group

gid	*name*
abc	A Book Club
gov	Student Government
dps	Dead Putting Society
…	…

tuples (or rows)

Member

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

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

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

Set representation of tuples

User

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

Group

gid	*name*
abc	A Book Club
gov	Student Government
edu	Dead Putting Society
…	…

Member

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

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

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!

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 satisfies all the integrity constraints.

- Reasons to use constraints:
  - Address consistency challenges
    (last class: duplicate entry for Bob)
  - Ensure data entry/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 (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

Key (Candidate key)

uid	*name*	*age*	*pop*
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

- *Member (uid, gid)*

Member

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

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

☞ **A key can contain multiple attributes**

More examples of keys

- *Address* (*street_address*, *city*, *state*, *zip*)
  - Key 1: {*street_address*, *city*, *state*}
  - 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*, *state*, *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

User

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

Group

gid	name
abc	A Book Club
gov	Student Government
dps	Dead Putting Society
…	…

Member

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

"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

Group

gid	*name*
abc	A Book Club
gov	Student Government
dps	Dead Putting Society

Member

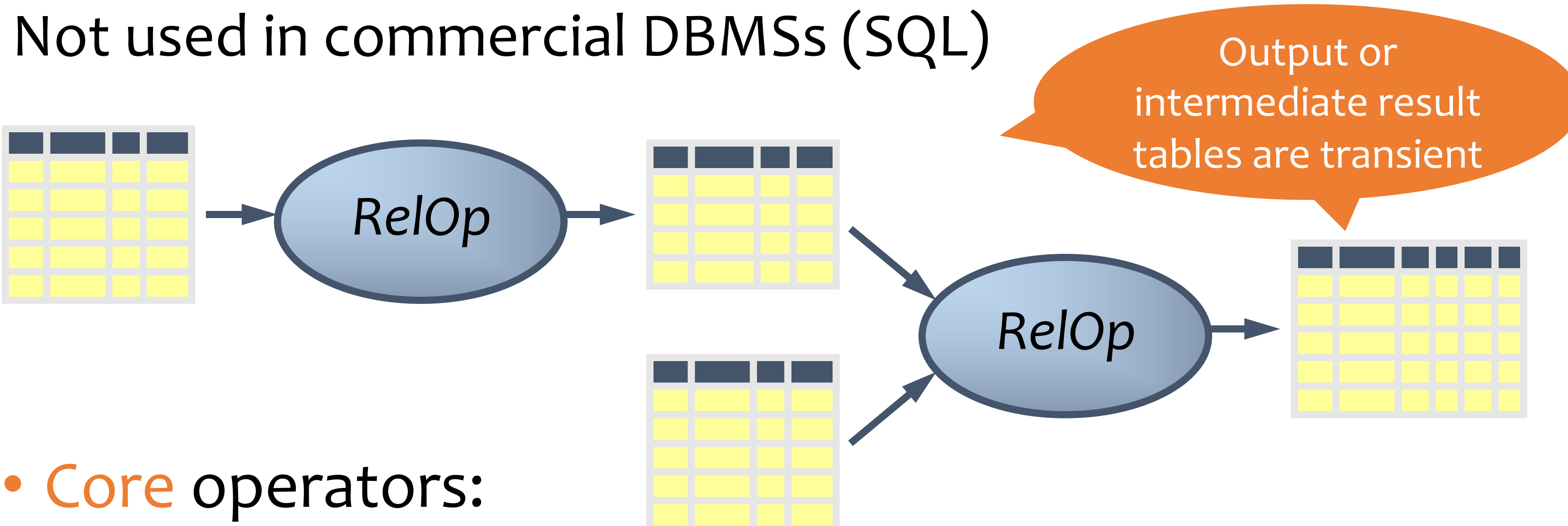
uid	*gid*
142	dps
123	gov
857	ON
857	gov
456	abc
456	gov
…	…

Outline

- Part 1: Relational data model
  - Data model
  - Database schema
  - Integrity constraints (keys)
  - Languages
    - Relational algebra (focus in this lecture)
    - SQL (next lecture)
    - 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)



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

Core operator 1: Selection σ

- Example query: Users with popularity higher than 0.5

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

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

→ $\sigma_{pop>0.5}$ →

uid	*name*	*age*	*pop*
142	Bart	10	0.9
[redacted]			
857	Lisa	8	0.7
[redacted]			
…	…	…	…

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 ($=$, $\leq$, etc.) and Boolean connectives (∧: and, ∨: or, ¬: 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

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

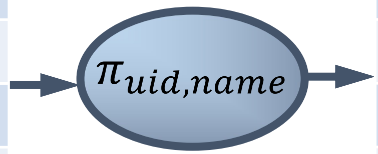
WRONG!

Core operator 2: Projection π

- Example: IDs and names of all users

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

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



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

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

$$\pi_{age}\ User$$

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

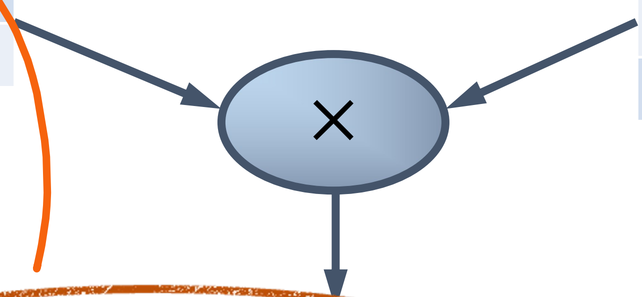
age
10
[redacted]
8
[redacted]
…

Core operator 3: Cross product ×

$$User \times Member$$

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
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

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

uid	*name*	*age*	*pop*	*uid*	*gid*
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
…	…	…	…	…	…

=

uid	*gid*	*uid*	*name*	*age*	*pop*
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

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

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
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
…	…	…	…	…	…

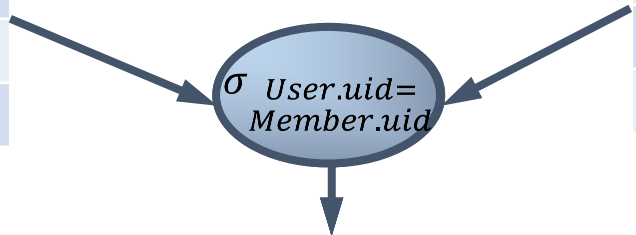
Derived operator 1: Join ⋈

- 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
[redacted]					
857	Lisa	8	0.7	857	abc
857	Lisa	8	0.7	857	gov
…	…	…	…	…	…

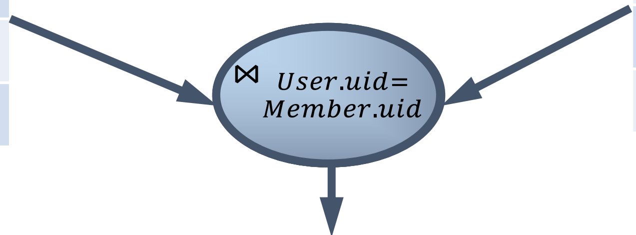
Derived operator 1: Join ⋈

- 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
[redacted]					
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

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

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*	[redacted]	*gid*
123	Milhouse	10	0.2	[redacted]	gov
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
857	Lisa	8	0.7	[redacted]	abc
857	Lisa	8	0.7	[redacted]	gov
…	…	…	…	[redacted]	…

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 $\pi_L\left(R \bowtie_p S\right)$, 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)

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)

uid	*gid*
123	gov
857	abc

$\cup$

uid	*gid*
123	gov
901	edf

=

uid	*gid*
123	gov
857	abc
901	edf

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

uid	*gid*
123	gov
857	abc

$-$

uid	*gid*
123	gov
901	edf

$=$

uid	*gid*
857	abc

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 to $R \bowtie S$

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

Core operator 6: Renaming

- Input: a table (or an expression) R
- Notation: $\rho_S\ R$, $\rho_{(A_1 \to A_1', \ldots)} R$, or $\rho_{S(A_1 \to A_1', \ldots)} R$
- Purpose: “rename” a table and/or its columns
- Output: a table with the same rows as R, but called differently

Member

uid	*gid*
123	gov
857	abc

$$\rho_{M1(uid \to uid_1, gid \to gid_1)} Member$$

M1

uid1	*gid1*
123	gov
857	abc

9. Core operator: 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: Avoid confusion caused by identical column names

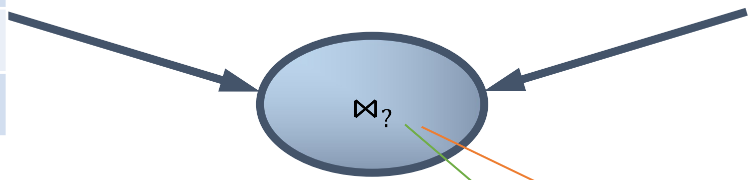
9. Core operator: Renaming

- IDs of users who belong to at least two groups

uid	gid
100	gov
100	abc
200	gov

$$Member \bowtie_{?} Member$$

uid	gid
100	gov
100	abc
200	gov



uid	gid	uid	gid
~~100~~	~~gov~~	~~100~~	~~gov~~
100	gov	100	abc
~~100~~	~~gov~~	~~200~~	~~gov~~
100	abc	100	gov
~~100~~	~~abc~~	~~100~~	~~abc~~
~~100~~	~~abc~~	~~200~~	~~gov~~
~~200~~	~~gov~~	~~100~~	~~gov~~
~~200~~	~~gov~~	~~100~~	~~abc~~
~~200~~	~~gov~~	~~200~~	~~gov~~

Condition 1: same uid

Condition 2: different gids

Renaming example

- IDs of users who belong to at least two groups

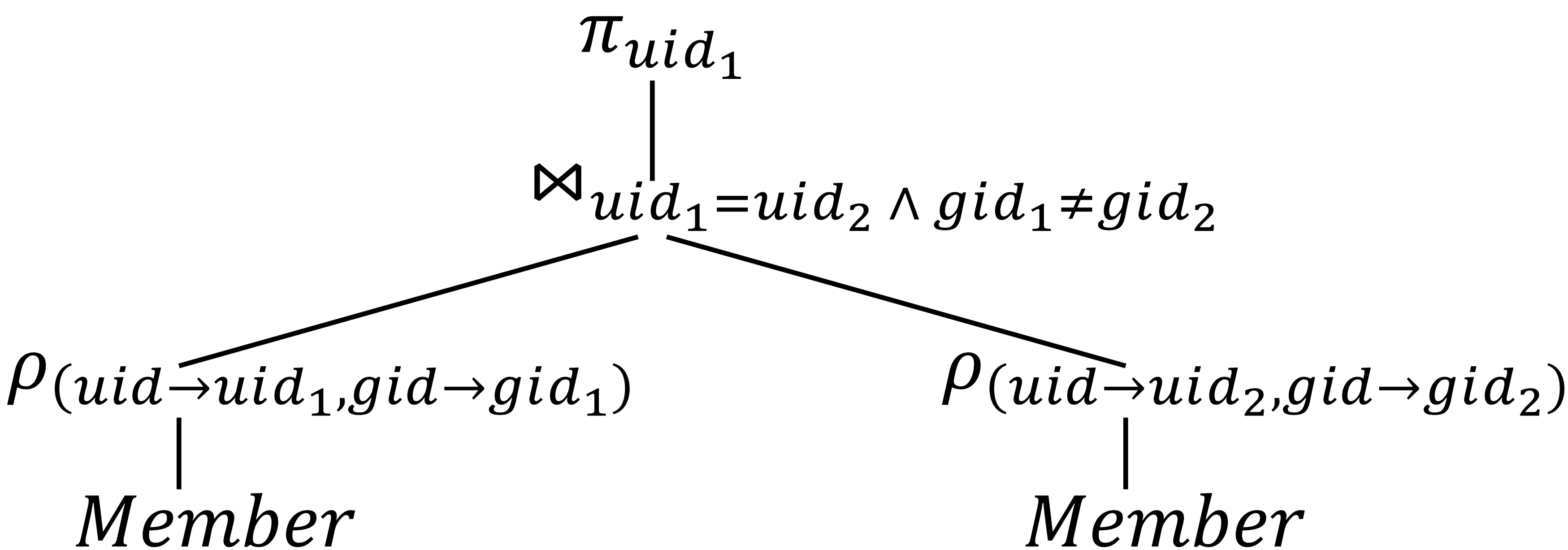
$$Member \bowtie_{?} Member$$

$$\pi_{uid}\left(Member \bowtie_{\substack{Member.uid=Member.uid\ \wedge \\ Member.gid \neq Member.gid}} Member\right)$$

WRONG!

$$\pi_{uid_1}\left(\begin{array}{c}\rho_{(uid \rightarrow uid_1, gid \rightarrow gid_1)} Member \\ \bowtie_{uid_1 = uid_2 \wedge gid_1 \neq gid_2} \\ \rho_{(uid \rightarrow uid_2, gid \rightarrow gid_2)} Member\end{array}\right)$$

Expression tree notation



Take-home Exercises

- Exercise 1: IDs of groups who have at least 2 users?
- Exercise 2: IDs of users who belong to at least three groups?

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

Derived Operators

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

More example

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

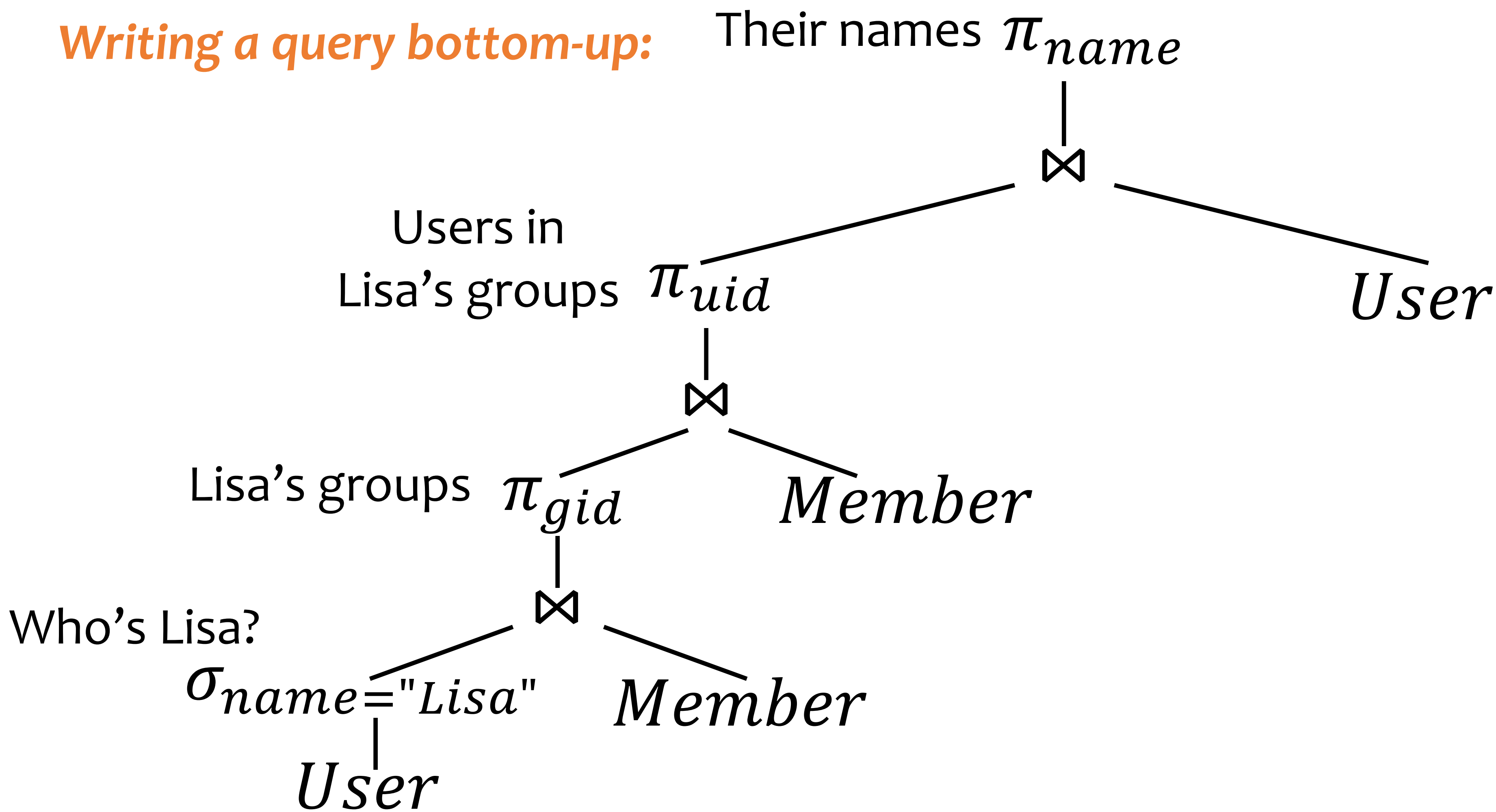
- Names of users in Lisa's groups

More 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)

- Names of users in Lisa's groups

Writing a query bottom-up:



More 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

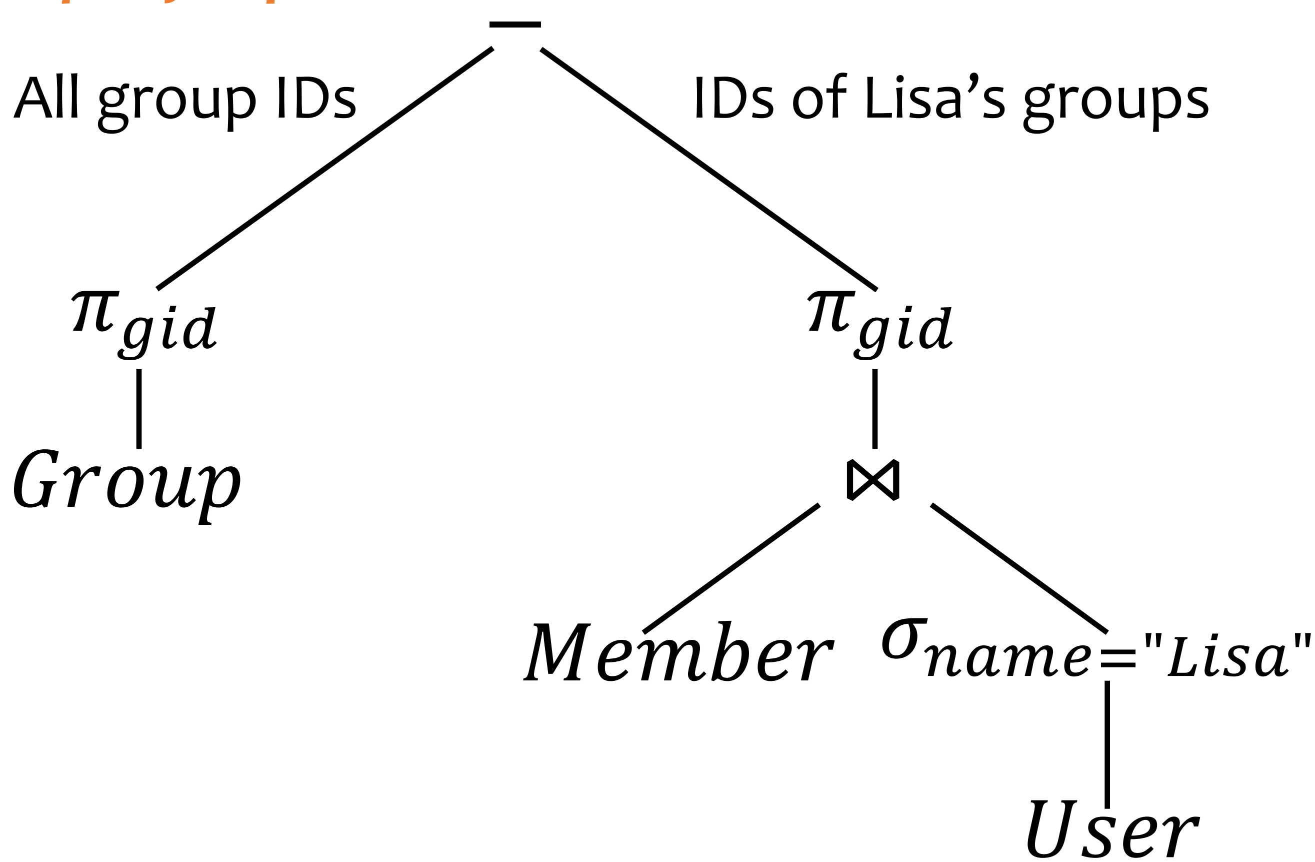
Writing a query top-down:

More 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

Writing a query top-down:



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)

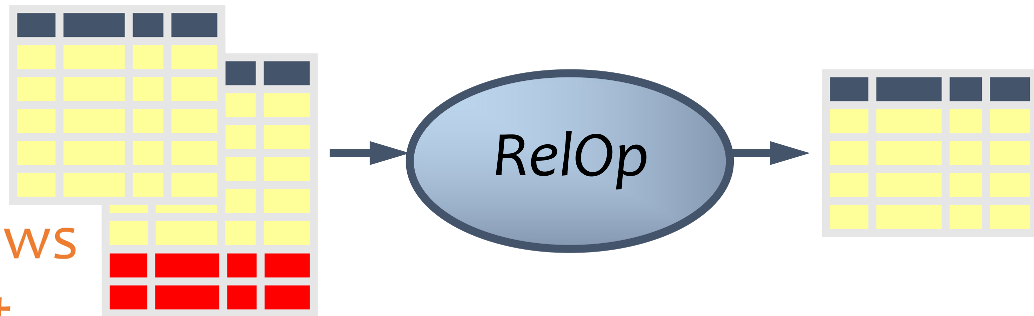
- Who are the most popular?
  - Who do NOT have the highest *pop* rating?
  - Whose *pop* is lower than somebody else's?

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

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

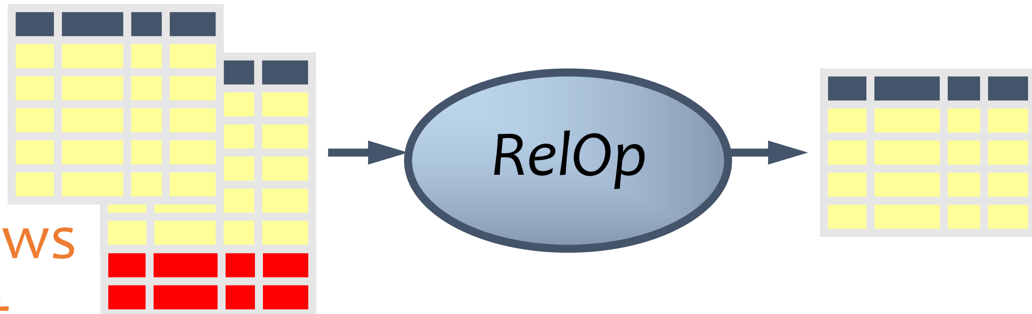
=

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

R

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

−

S

uid	gid
123	gov
901	edf

=

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

Why is "—" needed for "highest"?

- Composition of monotone operators produces a monotone query
  - Old output rows remain "correct" when more rows are added to the input
- Is the "highest" query monotone? (slide 50)
  - No!
  - Current highest *pop* is 0.9
  - Add another row with *pop* 0.91
  - Old answer is invalidated

☞So it must use difference!

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
  - ?
- Selection
  - ?

Extensions to relational algebra

- Duplicate handling (“bag algebra”)
- Grouping and aggregation
- “Extension” (or “extended projection”) to allow new column values to be computed

☞ All these will come up when we talk about SQL

☞ But for now we will stick to standard relational algebra without these extensions

Relational Calculus

- Relational Algebra: **procedural** language
  - An algebraic formalism in which queries are expressed by applying a sequence of operations to relations.
- Relational Calculus: **declarative** language
  - A logical formalism in which queries are expressed as formulas of first-order logic.
- Codd's Theorem: Relational Algebra and Relational Calculus are essentially equivalent in terms of expressive power.

Relational calculus

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)

- Use *first-order logic (FOL) formulae* to specify properties of the query answer

- Example: Who are the most popular?
  - $\{u.uid \mid u \in User \wedge \neg(\exists u' \in User: u.pop < u'.pop)\}$, or
  - $\{u.uid \mid u \in User \wedge (\forall u' \in User: u.pop \geq u'.pop)\}$

Relational calculus

- Relational algebra = "safe" relational calculus
  - Every query expressible as a safe relational calculus query is also expressible as a relational algebra query
  - And vice versa
- Example of an "unsafe" relational calculus query
  - $\{u.name \mid \neg(u \in User)\}$ → users not in the database
  - Cannot evaluate it just by looking at the database
- A query is *safe* if, for all database instances conforming to the schema, the query result can be computed using only constants appearing in the database instance or in the query itself.

Turing machine

How does relational algebra compare with a Turing machine?

- A conceptual device that can execute any computer algorithm
- Approximates what general-purpose programming languages can do
  - E.g., Python, Java, C++, …

Alan Turing (1912-1954)

http://en.wikipedia.org/wiki/File:Alan_Turing_photo.jpg

Limits of relational algebra

- Relational algebra has no recursion
  - Example: given relation *Friend*(*uid1*, *uid2*), who can Bart reach in his social network with any number of hops?
    - Writing this query in r.a. is impossible!
  - So r.a. is not as powerful as general-purpose languages

- But why not?
  - Optimization becomes undecidable
  - ☞ Simplicity is empowering
  - Besides, you can always implement it at the application level, and recursion is added to SQL nevertheless!

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 (how to write a query)
  - V.s. relational calculus
  - V.s. general programming language
- What's next?
  - SQL – query language used in practice (4 lectures)