Database Systems I

CMPT 354 Summer 2024

Zhengjie Miao

15 May 2024

Course Setup

Ask professor if he can post the due dates of the assignment

- OH slots:
  - Tue 2pm-3pm, Zhengjie@TASC 1 9407
  - Mon 11am-12pm, Xinyi@ASB9812
  - Wed 1pm-2pm, Obumneme@ASB9812
  - Thu 3pm-4pm, Zahra@ASB9812

- Sign up for Piazza!
  - The place to ask course-related questions
  - https://piazza.com/sfu.ca/summer2024/cmpt354d100
  - Access code: qc76kdzsz9l

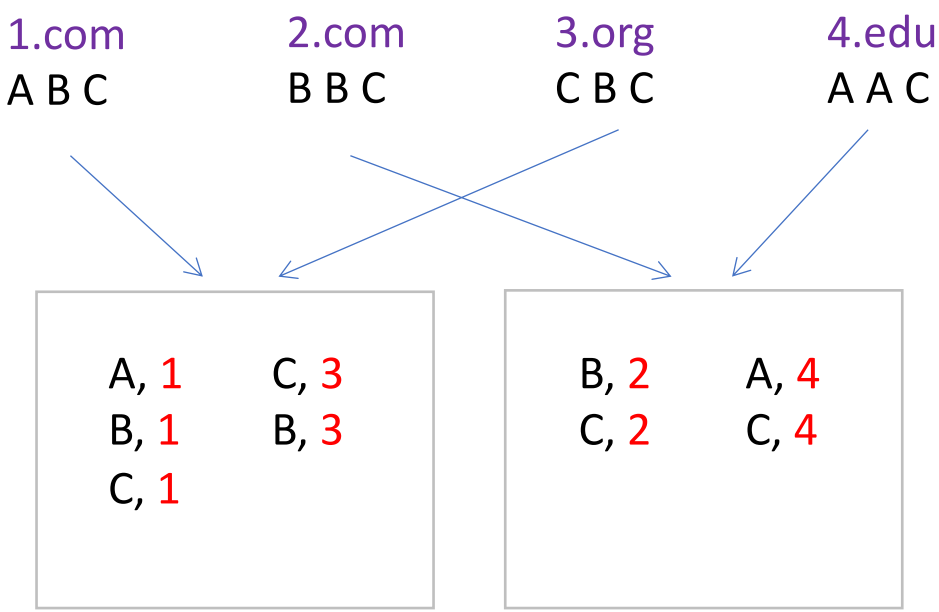
Recap: MapReduce

Online analytical processing

- Which design is better for OLAP?
  - MapReduce vs. Relational DBMS?
- Many problems can be processed in MapReduce pattern:
  - Given a lot of unsorted data Allows you to divide your data among multiple machines
  - Map: extract something of interest from each record
  - Shuffle: group the intermediate results in some way
  - Reduce: further process (e.g., aggregate, summarize, analyze, transform) each group and write final results

  (Customize map and reduce for problem at hand)

MR Example - Map



(Local disk)

key: document id
value: document contents

```
map(String key, String value):

  for each word w in value:
    Emit_Intermediate(w, document id);
```

MR Example - Shuffle & Reduce

A, 1
B, 1
C, 1
C, 3
B, 3

B, 2
C, 2
A, 4
C, 4

A, { 1, 4 }
C, { 3, 1, 2, 4 }

B, { 1, 3, 2 }

A, [1, 4]
B, [1, 2, 3]
C, [1, 2, 3, 4]

Values exchanged by shuffle process

key: a word

values: a list of doc id

Why is MapReduced criticized?
Proved to be inefficient in database systems.
Ask professor why it is inefficient.

The reduce function accepts all pairs for a given word, sorts the corresponding document IDs

MapReduce does not require any schema

Trends

Typically, you will need to maintain a transactional and analytical system, where you move data between them.

Ask him about why we would move our data frequently?

- 1 Hybrid Transactional and Analytical Processing

- 2. Cloud-Native Databases

Maintain database inside virtual machine
Cloud databases could allow infinite storage
disaggregating the storage and computation layer?
The scale of computation is independent from storage? Ask the professor what he means

- 3. Lakehouse = Data Lake + Data Warehouse

Search up Data Lake and Data Warehouse

Lakehouse wants to result in efficient queries

- 4. Specialized Systems
  - E.g. Time-series Database, GPU Acceleration, Vector Database

Outline

- An overview of data models
- Basics of the Relational Model
- Relational Algebra
  - Core operators & Derived operators
  - Extension
  - Relational calculus

Recap: Data Models

- Query: Who have accounts with 0 balance managed by a branch in Springfield?

Account

Id	Name	Balance	Branch
1000	Ned Flanders	0	234
1001	Homer Simpson	0	653
1002	Montgomery Burns	1000000000	781

Branch

Id	Location
234	Brockway

Id	Location
653	Springfield

Id	Location
781	Shelbyville

- Programmer controls “navigation”: accounts → branches
  - How about branches → accounts? (e.g., find branches managing accounts with 0 balance)

Data Model

- Data Model
  - mathematical formalism (or conceptual way) for describing the data

If you do not have a data model, what happens?
Ask professor

- The description generally consists of three parts:
  - Structure of the data
  - Operations on the data
  - Constraints on the data

Structure of the data

- Schema (e.g., table names, attribute names)
  - Describe the **conceptual** structure of the data
- Different from data structure (e.g., list, array)
  - Data structure can be seen as a **physical** data model

Operations on the data

- Query language (e.g., RA, SQL)
  - Describe what operations that can be performed on data

- Two kinds of operations
  - operations that retrieve information (“query” the data)
  - operations that change the database (“update” the data)

- Different from programming languages (e.g., C, Java)
  - Support a set of limited operations
  - Allow for query optimizations

Your query can be optimized if the system finds another query which is equivalent and more efficient

Programming languages can perform operations which the query language can not

Constraints on the data.

- Constraints (e.g., balance >= 0, account id is unique)
  - describe limitations on what the data can be.
- Different kinds of constraints
  - Domain constraints Ex: balance >= 0
  - Integrity constraints Ex: the user id is unique
- Why does it matter?
  - Ensure the correctness of data

Commonly Used Data Models

- Relational Data Model
- Key-Value Data Model
- Semi-structured Data Model (e.g., Json, XML)

The Relational Model in Brief

Id	Name	Balance	Branch
1000	Ned Flanders	0	234
1001	Homer Simpson	0	653
1002	Montgomery Burns	1000000000	781

Id	Location
234	Brockway
653	Springfield
781	Shelbyville

I thought SQL used RA, but that RA is not the query language

- Structure of the Data
  - Table structure
- Query language
  - RA, RC, SQL
- Constraints on the data
  - E.g., id is unique, balance >= 0, name is not NULL

The Key-Value Model in Brief

Key → Value
1000 → (Ned Flanders, 0, 234)
1001 → (Homer Simpson, 0, 653)
1002 → (Montgomery Burns, 1000000000, 781)

- Structure of the Data
  - (Key, Value) pairs
  - Key is an integer/string, value can be any object
- Query language
  - get(key), put(key, value)
- Constraints on the data
  - E.g., key is unique, value is not NULL

The Semistructured Model in Brief

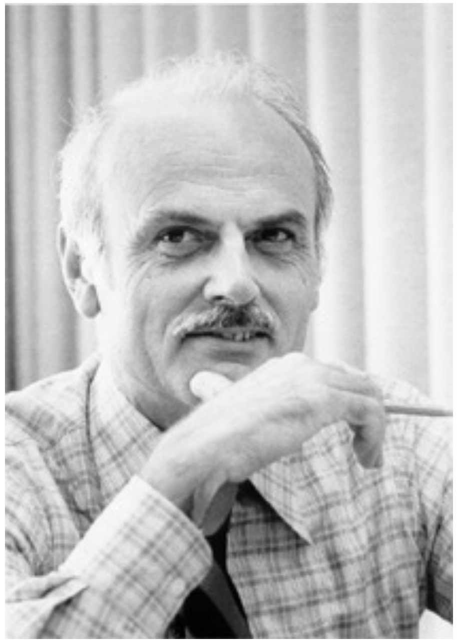
You cannot apply the same operation to each account?
Ask professor to clarify this

- Structure of the Data
  - Tree structure (e.g., XML, json)
- Query language
  - XQuery, MongoDB QL
- Constraints
  - E.g., each <Account> has a <Id> element and a <name> element nested within

Resembles the hierarchical model
“what goes around comes around”

```xml
<Accounts>
    <Account>
        <Id>1000</Id>
        <Name>Ned Flanders</Name>
        <Balance>0</Balance>
        <Branch>234</Branch>
    </Account>
    <Account>
        <Id>1001</Id>
        <Name>Homer Simpson</Name>
        <Balance>0</Balance>
        <Branch>653</Branch>
    </Account>
    <Account>
        <Id>1002</Id>
        <Name>Montgomery Burns</Name>
        <Balance>1000000000</Balance>
        <Branch>781</Branch>
    </Account>
</Accounts>
```

Edgar F. Codd (1923-2003)



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

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

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 name and a domain (or type)
  - Set-valued attributes are not allowed   Attributes must be atomic
- 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   Set semantics

- Simplicity is a virtue!

Example

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	Book Club
gov	Student Government
dps	Dead Putting Society
…	…

Member

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

Ordering of rows doesn't matter
(even though output is
always in some order)

Schema vs. instance

- Schema (metadata)
  - Specifies the logical structure of data
  - Is defined at setup time
  - Rarely changes

The schema rarely changes since we must update the change to the schema to all instances

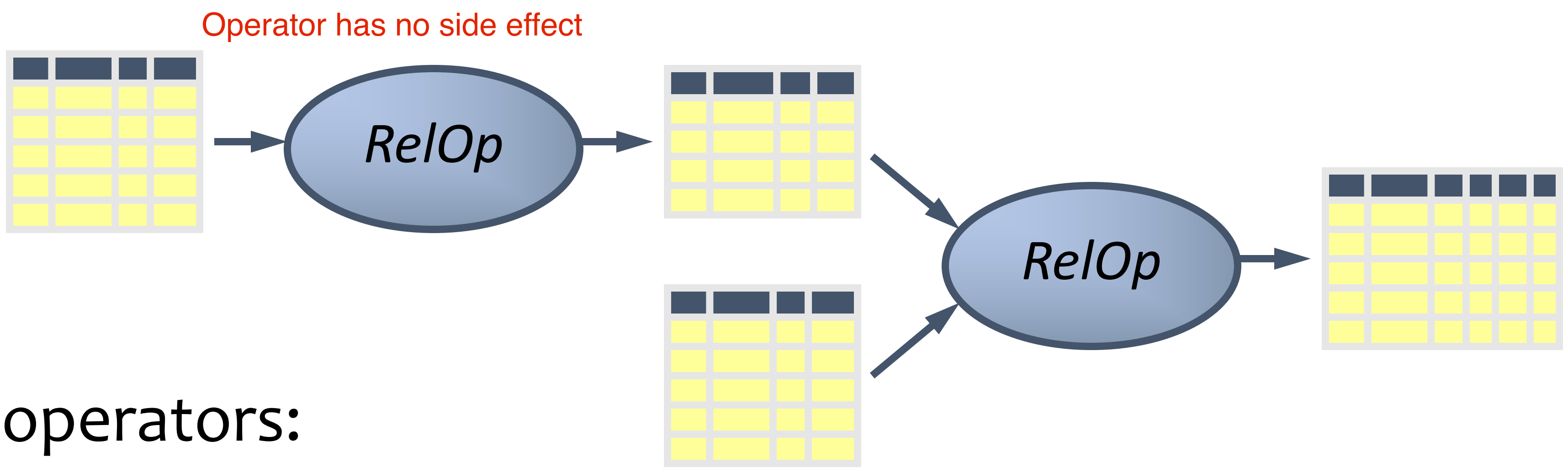
- Instance
  - Represents the data content
  - Changes rapidly, but always conforms to the schema
- Compare to types vs. collections of objects of these types in a programming language

Example

- Schema
  Ask the professor what he said regarding
  - *User* (*uid* int, *name* string, *age* int, *pop* float)
  - *Group* (*gid* string, *name* string)
  - *Member* (*uid* int, *gid* string)
- Instance
  - *User*: {⟨142, Bart, 10, 0.9⟩, ⟨857, Milhouse, 10, 0.2⟩, … }
  - *Group*: {⟨abc, Book Club⟩, ⟨gov, Student Government⟩, … }
  - *Member*: {⟨142, dps⟩, ⟨123, gov⟩, … }

Relational algebra

A language for querying relational data based on "operators"



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

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 or R that satisfy p

of

Selection example

- 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
…	…	…	…

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

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

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

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

- 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!

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

Projection 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
…	…	…	…

$$\pi_{uid,name}$$

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

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]
…

Composing Select and Project

- Names of users with popularity higher than 0.5

$$\pi_{name}(\sigma_{pop>0.5} User)$$

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

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

uid	name	age	pop
142	Bart	10	0.9
857	Lisa	8	0.7
…	…	…	…

$\xrightarrow{\pi_{name}}$

name
Bart
Lisa
…

- What about: $\sigma_{pop>0.5}(\pi_{name}(User))$
  - *Invalid types. Input to* $\sigma_{pop>0.5}$ *does not contain pop.*

Cross product

- Input: two tables R and S
- Natation: $R \times S$
- 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)

Cross product example

$User \times Member$

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
…	…	…	…	…	…

- How many rows in result?
  - |R|*|S|
- Schema compatability?
  - Not needed.
- Duplicates?
  - None generated.

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: join

(A.k.a. "theta-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 $\sigma_p(R \times S)$

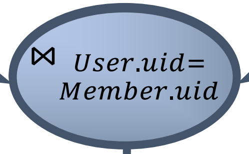
Join example

- 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: 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)

Natural join example

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

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

uid	gid
123	gov
857	abc
857	gov
…	…

$\bowtie$

uid	name	age	pop	uid	gid
123	Milhouse	10	0.2	[redacted]	gov
[redacted]					
857	Lisa	8	0.7	[redacted]	abc
857	Lisa	8	0.7	[redacted]	gov
…	…	…	…	[redacted]	…

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
857	gov

$=$

uid	*gid*
123	gov
857	abc
857	gov

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
857	gov

$=$

uid	*gid*
857	abc

Derived operator: 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?

Could we also use the natural join?

uid	*gid*
123	gov
857	abc

$\cap$

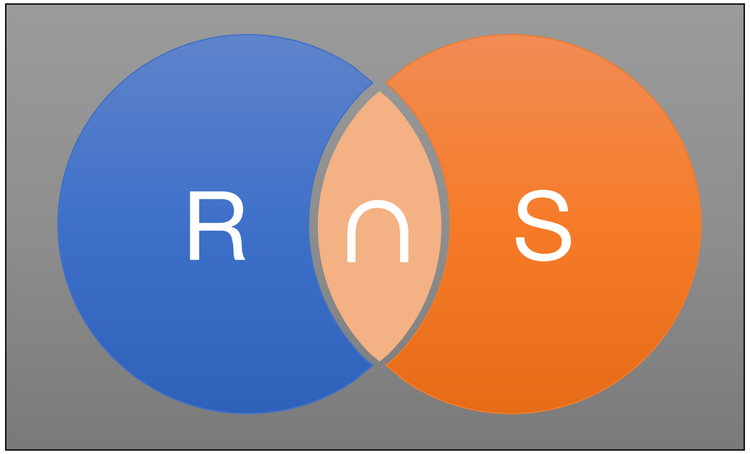
uid	*gid*
123	gov
857	gov

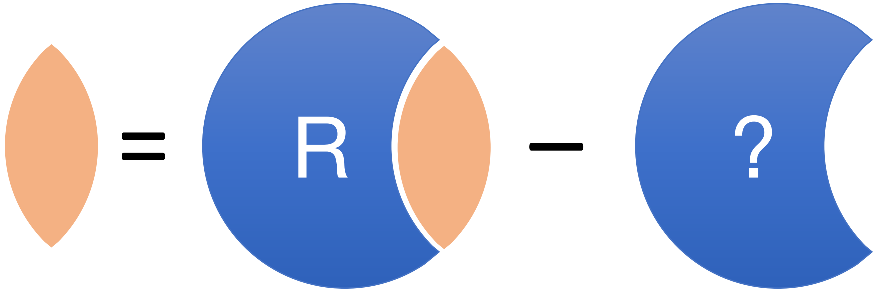
$=$

uid	*gid*
123	gov

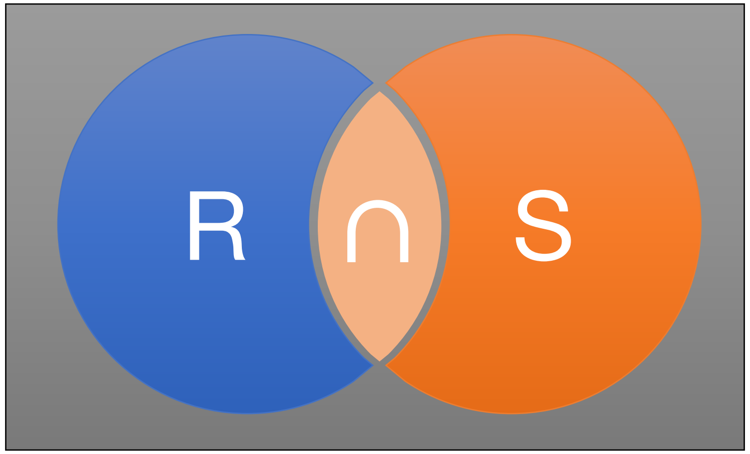
Intersection

- R ∩ S = R – ?

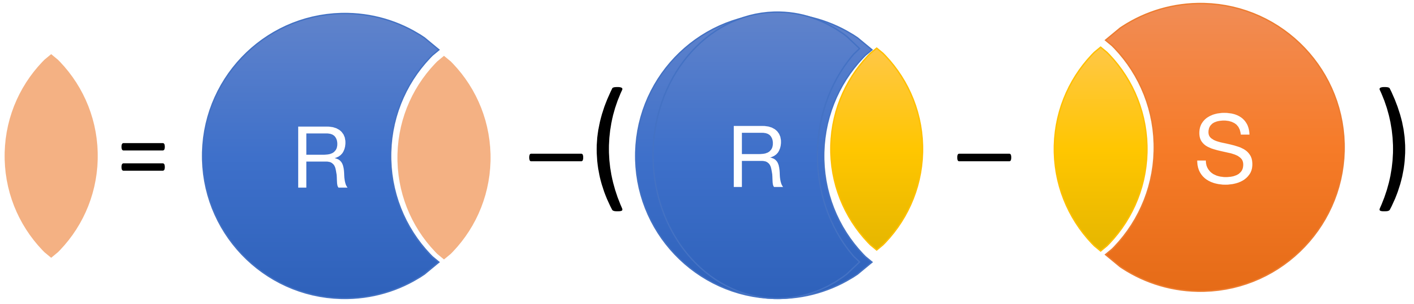




Intersection



- $R \cap S = R - (R - S)$
- Also $= S - (S - R)$



- How can you write it without –

$$R \bowtie S$$

Renaming

- Input: a table R
- Notation: $\rho_S\ R$, $\rho_{(A_1, A_2, \ldots)} R$, or $\rho_{S(A_1, A_2, \ldots)} R$
- Purpose: "rename" a table and/or its columns
- Output: a table with the same rows as R, but called differently

Renaming allows you to clarify which attributes belong to which instance

Member

uid	*gid*
123	gov
857	abc

$$\rho_{M1(uid1, gid1)} Member \quad =$$

M1

uid	*gid*
123	gov
857	gov

Should the columns not be renamed?
Professor: they should

Renaming

- Used to
  - Avoid confusion caused by identical column names
  - Create identical column names for natural joins
- As with all other relational operators, it doesn't modify the database
  - Think of the renamed table as a copy of the original

Member

uid	*gid*
123	gov
857	abc

$$\rho_{M1(uid1,gid1)} Member \quad =$$

M1

uid	*gid*
123	gov
857	gov

Note:
Learn to differentiate the natural join and regular join

Renaming example

- IDs of users who belong to at least two groups

$$Member \bowtie_{?} Member$$

uid	gid
123	gov
857	abc
857	gov

uid	gid
123	gov
857	abc
857	gov

$\bowtie_{?}$

uid	gid	uid	gid
~~123~~	~~gov~~	~~123~~	~~gov~~
~~123~~	~~gov~~	~~857~~	~~abc~~
~~123~~	~~gov~~	~~857~~	~~gov~~
~~857~~	~~abc~~	~~123~~	~~gov~~
~~857~~	~~abc~~	~~857~~	~~abc~~
857	abc	857	gov
~~857~~	~~gov~~	~~123~~	~~gov~~
857	gov	857	abc
~~857~~	~~gov~~	~~857~~	~~gov~~

Same uid

Different gids

Renaming example

- IDs of users who belong to at least two groups

$$Member \bowtie_{?} Member$$

First condition will always be true
Second condition will always be false

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

WRONG!

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

Expression tree notation

$$\pi_{uid_1}$$

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

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

$$Member \qquad Member$$

Summary of core operators

- Selection: $\sigma_p R$
- Projection: $\pi_L R$
- Cross product: $R \times S$
- Union: $R \cup S$
- Difference: $R - S$
- Renaming: $\rho_{S(A_1, A_2, \dots)} R$
    - Does not really add “processing” power

Summary of derived operators

- Join: $R \bowtie_p S$
- Natural join: $R \bowtie S$
- Intersection: $R \cap S$

- Many more
  - Semijoin, anti-semijoin, quotient, …

An exercise

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

Writing a query bottom-up:

Their names π_{name}

name
Milhouse
Lisa

$\bowtie$

Should we remove Lisa's name

gid	uid
abc	123
gov	857
gov	857

Users in Lisa's groups π_{uid}

$User$

uid	name	age	pop	*gid*
857	Lisa	8	0.7	abc
857	Lisa	8	0.7	gov

$\bowtie$

Lisa's groups π_{gid}

$Member$

uid	name	age	pop
857	Lisa	8	0.7

$\bowtie$

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

$Member$

uid	*gid*
123	gov
857	abc
857	gov
…	…

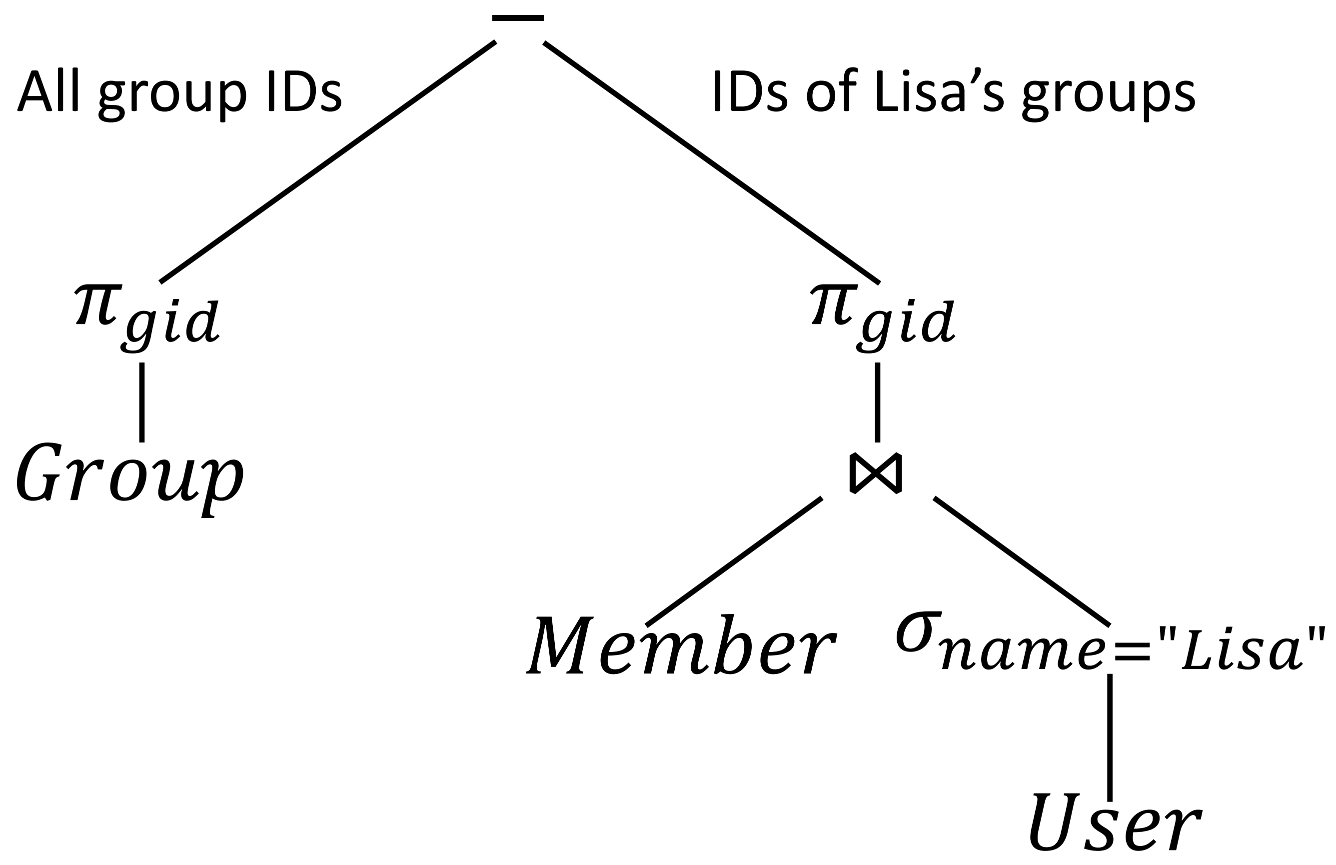
$User$

Another exercise

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

- IDs of groups that Lisa doesn't belong to

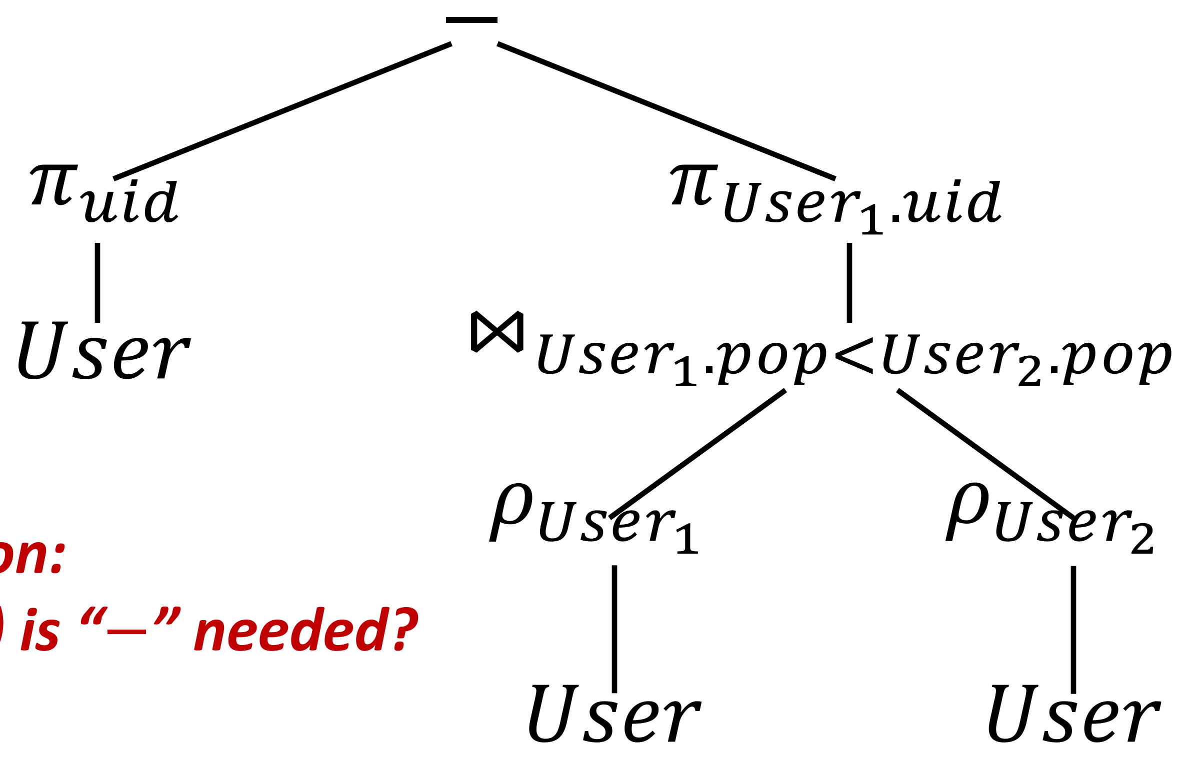
Writing a query top-down:



A trickier exercise

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

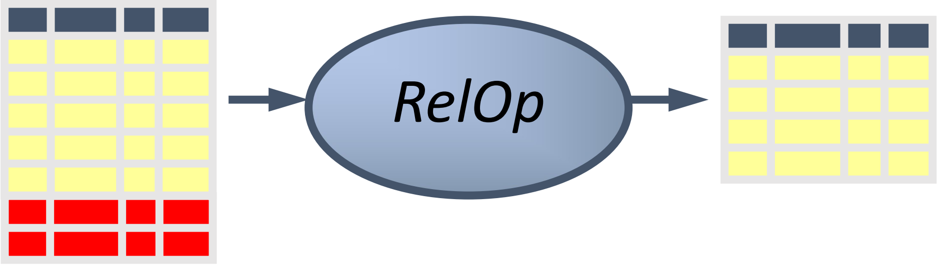
- Who are the most popular?
  - Who do NOT have the highest *pop* rating?
  - Whose *pop* is lower than somebody else's?



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

Monotone operators

Add more rows to the input…



What happens to the output?

Monotonicity:
Applying the relational operator on a new input will always output what you had earlier

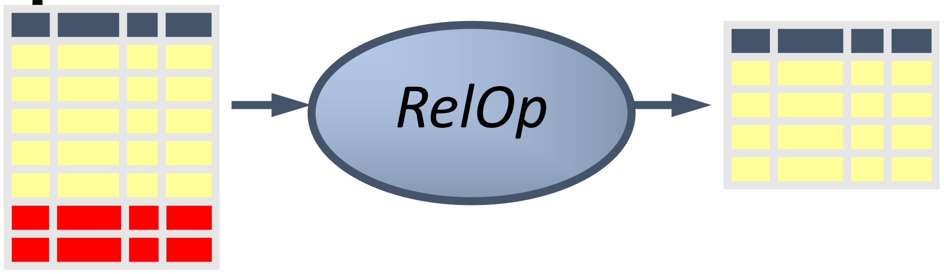
- If some old output rows may need to be removed
  - Then the operator is non-monotone
- Otherwise the operator is monotone
  - That is, old output rows always remain "correct" when more rows are added to the input

Does monotonicity hold when removing rows from the input?
Is there another property

- Formally, for a monotone operator op:
  $R \subseteq R'$ implies $op(R) \subseteq op(R')$ for any R, R'

Monotone operators

Add more rows to the input…



What happens to the output?

- If some old output rows may need to be removed
  - Then the operator is non-monotone
- Otherwise the operator is monotone
  - That is, old output rows always remain “correct” when more rows are added to the input

uid	gid
123	gov
857	abc

−

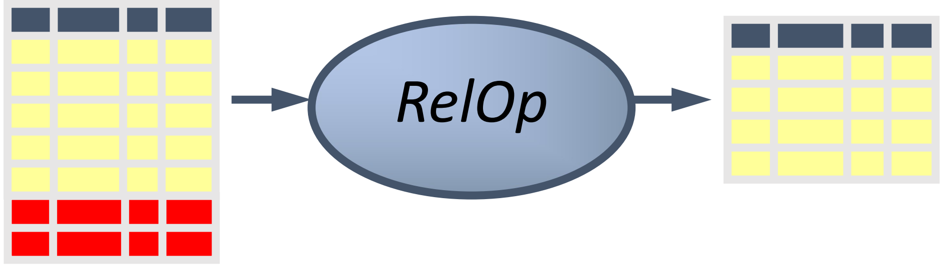
uid	gid
123	gov
857	gov
857	abc

=

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

Monotone operators

Add more rows to the input…



What happens to the output?

- If some old output rows may need to be removed
  - Then the operator is non-monotone
- Otherwise the operator is monotone
  - That is, old output rows always remain "correct" when more rows are added to the input

uid	gid
123	gov
857	abc
857	**gov**
693	**abc**

−

uid	gid
123	gov
857	gov

=

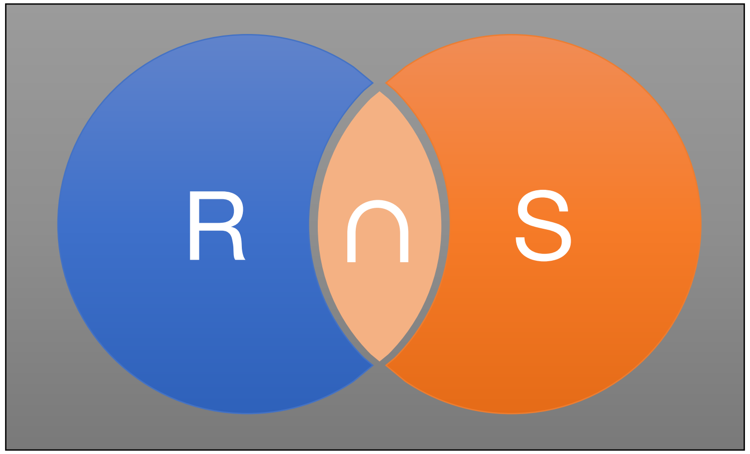
uid	gid
857	abc
693	**abc**

The old row is always "correct" no matter what is added to R

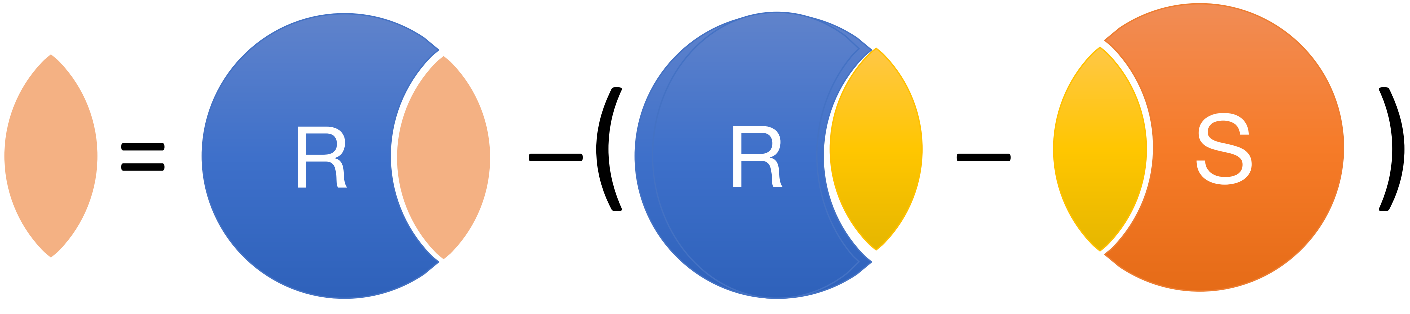
Classification of relational operators

- 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?

Is intersection monotonic?



- $R \cap S = R - (R - S)$



- Yes!

$$R_1 \subseteq R_2 \quad \Rightarrow \quad S \cap R_1 \subseteq S \cap R_2$$

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?
  - 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
  - The only operator that allows you to add rows?
  - A more rigorous argument?
- Selection?
  - Left as an exercise for the viewer

Extensions to relational algebra

- Duplicate handling ("bag algebra")
- Grouping and aggregation

- All these will come up when we talk about SQL
- But for now we will stick to standard relational algebra without these extensions

Why is RA a good query language?

- Simple
  - A small set of core operators
  - Semantics are easy to grasp
- Declarative?
  - Yes, compared with older languages like CODASYL
  - Though assembling operators into a query does feel somewhat "procedural"
- Complete?
  - With respect to what?

Relational Calculus

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

- $\{u.uid \mid u \in User \wedge$
  $\neg(\exists u' \in User{:}\, u.pop < u'.pop)\}$, or
- $\{u.uid \mid u \in User \wedge$
  $(\forall u' \in User{:}\, u.pop \geq u'.pop)\}$
- Relational 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
  - $\{u.name \mid \neg(u \in User)\}$
  - Cannot evaluate it just by looking at the database

Relational Calculus

- RA and RC form the basis for "real" query languages (e.g. SQL), and for implementation
  - Relational Algebra
    - Queries are expressed by applying a sequence of operations to relations
    - More operational/procedural, very useful for representing how query executes
  - Relational Calculus
    - Let's users describe WHAT they want in first-order logic
    - Rather than HOW to compute it
    - Non-operational, declarative

Codd's Theorem

- Established equivalence in expressivity between :
  - Relational Calculus
  - Relational Algebra

- Why an important result?
  - Connects declarative representation of queries with operational description
  - Constructive: we can "compile" SQL into relational algebra

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 RA is not as powerful as general-purpose languages (not Turing-complete)
- 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 😱)

What's Next

- Next class:
  - Database design in E/R model (recording)
- Sign up Piazza
  - https://piazza.com/sfu.ca/summer2024/cmpt354d100
  - Access code: qc76kdzsz9l
- Assignment 1 releasing soon
- Stay tuned for RA help tools