Announcements (Thu. Aug. 26)

- Start following Ed, NOW!
- Gradiance RA Exercise assigned; due in a week
  - See “Help/Getting Started with Gradiance” of the course website
- Homework 1 will be posted tonight; due in 2½ weeks
  - See “Help/Submitting Non-Gradiance Work” for instructions on Gradescope
- Set up VM (virtual machine)
  - See “Help/VM-related” for instructions
  - Google Cloud coupon email sent
- Check Sakai email archive for any missed announcements
- Office hours: North 232
  - Instructor/GTA office hours are starting this week
  - UTA office hours will be posted this weekend

Announcements (Thu. Aug. 26)

- Our current team of UTAs, so far
  - Paige Bartusak
  - Kevin Day
  - Anesh Gupta
  - Himanshu Jain
  - Tharun Raj
  - Scott Mani Raj
  - Jay Don Scott
  - Jeevan Tewari
  - Caleb Woo
  - Joshua Guo
  - Florence Liu
  - Tharun Raj
  - Jade Raj

- As announced by email, I will update the enrollment situation tonight
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

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
- 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!

Example

<table>
<thead>
<tr>
<th>User</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>uid</td>
<td>gid</td>
</tr>
<tr>
<td>142</td>
<td>abc</td>
</tr>
<tr>
<td>123</td>
<td>gov</td>
</tr>
<tr>
<td>857</td>
<td>abc</td>
</tr>
<tr>
<td>456</td>
<td>gov</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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

- **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**
  - 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 of $R$ that satisfy $p$

Selection example

- Users with popularity higher than 0.5

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

<table>
<thead>
<tr>
<th>uid</th>
<th>name</th>
<th>age</th>
<th>pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>Bart</td>
<td>10</td>
<td>0.9</td>
</tr>
<tr>
<td>123</td>
<td>Milhouse</td>
<td>10</td>
<td>0.2</td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
<td>8</td>
<td>0.7</td>
</tr>
<tr>
<td>456</td>
<td>Ralph</td>
<td>8</td>
<td>0.3</td>
</tr>
</tbody>
</table>

More on selection

- Selection condition can include any column of $R$, constants, comparison ($=, \leq, \text{etc.}$) and Boolean connectives ($\land$: and, $\lor$: or, $\neg$: not)
- Example: users with popularity at least 0.9 and age under 10 or above 12

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

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

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

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
  \( \pi_{\text{uid}, \text{name}} \text{User} \)

<table>
<thead>
<tr>
<th>uid</th>
<th>name</th>
<th>age</th>
<th>pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>Bart</td>
<td>10</td>
<td>0.9</td>
</tr>
<tr>
<td>123</td>
<td>Milhouse</td>
<td>10</td>
<td>0.2</td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
<td>8</td>
<td>0.7</td>
</tr>
<tr>
<td>456</td>
<td>Ralph</td>
<td>8</td>
<td>0.3</td>
</tr>
</tbody>
</table>

More on projection

- Duplicate output rows are removed (by definition)
- Example: user ages
  \( \pi_{\text{age}} \text{User} \)

<table>
<thead>
<tr>
<th>uid</th>
<th>name</th>
<th>age</th>
<th>pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>Bart</td>
<td>10</td>
<td>0.9</td>
</tr>
<tr>
<td>123</td>
<td>Milhouse</td>
<td>10</td>
<td>0.2</td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
<td>8</td>
<td>0.7</td>
</tr>
<tr>
<td>456</td>
<td>Ralph</td>
<td>8</td>
<td>0.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>-</td>
</tr>
</tbody>
</table>
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 $s$)

Cross product example

**User x Member**

```
<table>
<thead>
<tr>
<th>uid</th>
<th>name</th>
<th>age</th>
<th>pop</th>
<th>gid</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Milhouse</td>
<td>10</td>
<td>0.2</td>
<td>gov</td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
<td>8</td>
<td>0.7</td>
<td>abc</td>
</tr>
</tbody>
</table>

uid | name  | age | pop | gid |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Milhouse</td>
<td>10</td>
<td>0.2</td>
<td>gov</td>
</tr>
<tr>
<td>123</td>
<td>Milhouse</td>
<td>10</td>
<td>0.2</td>
<td>857</td>
</tr>
<tr>
<td>123</td>
<td>Milhouse</td>
<td>10</td>
<td>0.2</td>
<td>857</td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
<td>8</td>
<td>0.7</td>
<td>123</td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
<td>8</td>
<td>0.7</td>
<td>857</td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
<td>8</td>
<td>0.7</td>
<td>857</td>
</tr>
</tbody>
</table>
```

A note on column ordering

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

<table>
<thead>
<tr>
<th>User</th>
<th>Member</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Milhouse</td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

\[
\text{User} \bowtie \text{Member} = \pi_{\text{uid, name, age, pop, gid}} \left( \text{User} \bowtie \pi_{\text{uid}} \text{Member} \right)
\]

<table>
<thead>
<tr>
<th>uid</th>
<th>name</th>
<th>age</th>
<th>pop</th>
<th>gid</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Milhouse</td>
<td>10</td>
<td>0.2</td>
<td>gov</td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
<td>8</td>
<td>0.7</td>
<td>abc</td>
</tr>
</tbody>
</table>

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)

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: 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 _______
  - Also equivalent to _______
  - And to _______

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
- 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

Renaming example

- IDs of users who belong to at least two groups

$$\pi_{\text{uid}} \left( \sigma_{\text{Member.uid} = \text{Member.uid} \land \text{Member}} \left( \rho_{\text{uid} = \text{uid}_1 \land \text{gid} = \text{gid}_1} \left( \pi_{\text{uid}_1} \left( \rho_{\text{uid}_2, \text{gid}_2} \text{Member} \left( \sigma_{\text{uid}_2 = \text{uid}_1 \land \text{gid}_2 = \text{gid}_1} \pi_{\text{uid}_1} \right) \right) \right) \right) \right)$$
Summary of core operators

- Selection: \( \sigma_p R \)
- Projection: \( \pi_x R \)
- Cross product: \( R \times S \)
- Union: \( R \cup S \)
- Difference: \( R - S \)
- Renaming: \( \rho_{(A_1, A_2, \ldots)} R \)
  - 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

• Names of users in Lisa’s groups

  Writing a query bottom-up: Their names

  Users in Lisa’s groups

  Lisa’s groups

  Who’s Lisa?

  \[ \sigma_{\text{name} = “Lisa”} \]

  Member

  User

Another exercise

• IDs of groups that Lisa doesn’t belong to

  Writing a query top-down:

  All group IDs

  ID of Lisa’s groups

  Group

  Member

  User

A trickier exercise

• Who are the most popular?
  • Who do NOT have the highest pop rating?

  A deeper question:
  When (and why) is “…” needed?
Monotone operators

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

Classification of relational operators

- Selection: \( \sigma_p R \)
- Projection: \( \pi_k R \)
- Cross product: \( R \times S \)
- Join: \( R \bowtie S \)
- Natural join: \( R \bowtie S \)
- Union: \( R \cup S \)
- Difference: \( R - S \)
- Intersection: \( R \cap S \)

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?
Why do we need core operator $X$?
- Difference
  - The only non-monotone operator
- Projection
  - The only operator that _______
- Cross product
  - The only operator that _______
- Union
  - The only operator that allows you to _____?
  - A more rigorous argument?
- Selection?
  - Homework problem

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

Why is r.a. 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

- \{ u.uid | u \in \text{User} \land \neg (\exists u' \in \text{User}: u.pop < u'.pop) \}, or
- \{ u.uid | u \in \text{User} \land (\forall u' \in \text{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 | \neg (u \in \text{User}) \}
- Cannot evaluate it just by looking at the database

Turing machine

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

- So how does relational algebra compare with a Turing machine?

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 😱)