Relational Model and Algebra

Introduction to Databases

CompSci 316 Fall 2018
Announcements (Thu. Aug. 30)

• Sign up for Piazza, NOW!
• Homework #1 to be posted today; due in 2½ weeks
  • Sign up for Gradiance
  • Gradescope not ready yet; please wait for my announcement
• Set up VM
  • Instructions on course website
  • Google Cloud coupon email sent
    • Check Sakai email archive for any missed announcements
  • Help sessions planned for next week
• My office hours are cancelled today—talk to me right after class if there’s something urgent
• TA/UTA office hours to be posted soon
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

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>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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

Group

<table>
<thead>
<tr>
<th>gid</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>Book Club</td>
</tr>
<tr>
<td>gov</td>
<td>Student Government</td>
</tr>
<tr>
<td>dps</td>
<td>Dead Putting Society</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Member

<table>
<thead>
<tr>
<th>uid</th>
<th>gid</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>dps</td>
</tr>
<tr>
<td>123</td>
<td>gov</td>
</tr>
<tr>
<td>857</td>
<td>abc</td>
</tr>
<tr>
<td>456</td>
<td>abc</td>
</tr>
<tr>
<td>456</td>
<td>gov</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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 \((uid \text{ int}, \text{name string}, \text{age int}, \text{pop float})\)
  • Group \((gid \text{ string}, \text{name string})\)
  • Member \((uid \text{ int}, gid \text{ string})\)

• Instance
  • User: \(\{\langle 142, \text{Bart}, 10, 0.9 \rangle, \langle 857, \text{Milhouse}, 10, 0.2 \rangle, \ldots \}\)
  • Group: \(\{\langle \text{abc}, \text{Book Club} \rangle, \langle \text{gov}, \text{Student Government} \rangle, \ldots \}\)
  • Member: \(\{\langle 142, \text{dps} \rangle, \langle 123, \text{gov} \rangle, \ldots \}\)
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 $p$
Selection example

• Users with popularity higher than 0.5

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

<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>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<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>857</td>
<td>Lisa</td>
<td>8</td>
<td>0.7</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</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
    
    $\sigma_{\text{pop} \geq 0.9 \land (\text{age} < 10 \lor \text{age} > 12)} \text{User}$
  - You must be able to evaluate the condition over each single row of the input table!
    - Example: the most popular user
      
      $\sigma_{\text{pop} \geq \text{every pop in User}} \text{User}$
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

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

<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>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>uid</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>Bart</td>
</tr>
<tr>
<td>123</td>
<td>Milhouse</td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
</tr>
<tr>
<td>456</td>
<td>Ralph</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
More on projection

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

\[ \pi_{age} \text{User} \]

<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>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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

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

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

\[ \text{User} \bowtie_{\text{User.uid} = \text{Member.uid}} \text{ Member} \]

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

<table>
<thead>
<tr>
<th>uid</th>
<th>name</th>
<th>age</th>
<th>pop</th>
</tr>
</thead>
<tbody>
<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>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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

$User \bowtie Member = \pi_？(User \bowtie_？ Member) = \pi_{uid,name,age,pop,gid} (User \bowtie_{User.uid= Member.uid})$
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 $R \setminus (R \setminus S)$
• Also equivalent to $S \setminus (S \setminus R)$
• And to $R \bowtie S$
Renaming

• Input: a table $R$

• Notation: $\rho_S R$, $\rho_{(A_1,A_2,...)} R$, or $\rho_{S(A_1,A_2,...)} R$

• 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
  \[ \text{Member} \bowtie \text{Member} \]

\[
\pi_{\text{uid}} \left( \text{Member} \bowtie_{\text{Member.uid=Member.uid} \land \text{Member.gid} \neq \text{Member.gid}} \text{Member} \right)
\]

\[
\pi_{\text{uid}_1} \left( \rho_{(\text{uid}_1, \text{gid}_1)} \text{Member} \bowtie_{\text{uid}_1=\text{uid}_2 \land \text{gid}_1 \neq \text{gid}_2} \rho_{(\text{uid}_2, \text{gid}_2)} \text{Member} \right)
\]

WRONG!
Expression tree notation
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: \( \rho_{S(A_1,A_2,...)} R \)
  • 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:

Who’s Lisa?

$\sigma_{name=\text{"Lisa"}}$

User

Lisa’s groups $\pi_{gid}$

Member

Users in Lisa’s groups $\pi_{uid}$

Their names $\pi_{name}$

User
Another exercise

• IDs of groups that Lisa doesn’t belong to

Writing a query top-down:
A trickier exercise

• 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

• 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: $\sigma_p R$  
  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?
  • No!
  • Current highest \( \text{pop} \) is 0.9
  • Add another row with \( \text{pop} \) 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?**
  - 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 operators do look somewhat “procedural”

• Complete?
  • With respect to what?
Relational calculus

- \{u.uid | u \in User \land 
  \neg(\exists u' \in User: u.pop < u'.pop)\}, or
- \{u.uid | u \in User \land 
  (\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 | \neg(u \in 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!