Announcements (Tue. Jan. 14)

• You should be on Piazza and Gradescope
  • Otherwise, let the instructor know after class

• HW1 will be posted after class, due next Tuesday 11:59 pm
  • Instant feedback, multiple submissions allowed until correct!
  • 5% / hour late submission penalty
  • Use pgweb from course website to try your queries on small Beers dataset
  • If you join the class after Tuesday 01/14, let the instructor know

• Office hours posted on course website
  • There is at least one everyday except Saturday

Today’s plan

• Revisit relational model
• Revisit simple SQL queries and its semantic
• Start relational algebra

The famous “Beers” database

- Drinker Frequents Bars “X” times a week
- Bar Serves Beer at price “Y
- Drinker Each has an address
- Bar Each has an address
- Beer Each has a brewer

“Beers” as a Relational Database

<table>
<thead>
<tr>
<th>Bar</th>
<th>Brewer</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Edge</td>
<td>Budweiser</td>
<td>The Edge</td>
<td>108 Morris Street</td>
</tr>
<tr>
<td>The Edge</td>
<td>Corona</td>
<td>Grupo Modelo</td>
<td>905 W. Main Street</td>
</tr>
<tr>
<td>Dixie</td>
<td>Dixie Brewing</td>
<td>Dixie</td>
<td>300 N. Duke Street</td>
</tr>
<tr>
<td>The Edge</td>
<td>Budweiser</td>
<td>Anheuser-Busch Inc.</td>
<td></td>
</tr>
<tr>
<td>The Edge</td>
<td>Corona</td>
<td>Grupo Modelo</td>
<td></td>
</tr>
<tr>
<td>Dixie</td>
<td>Dixie Brewing</td>
<td>Dixie Brewing Company</td>
<td></td>
</tr>
<tr>
<td>The Edge</td>
<td>Budweiser</td>
<td>Anheuser-Busch Inc.</td>
<td></td>
</tr>
</tbody>
</table>

What is an example of a
• Relation
• Attribute
• Tuple
• Schema
• Instance

What is
• Set semantic
• in relational model
• Bag semantic
• in SQL (why)

Basic queries: SFW statement

- SELECT $A_1, A_2, \ldots, A_n$
  FROM $R_1, R_2, \ldots, R_m$
  WHERE condition

  In HW1, you can only use SFW

• SELECT, FROM, WHERE are often referred to as
SELECT, FROM, WHERE “clauses”

• Each query must have a SELECT and a FROM

• WHERE is optional
Example: reading a table

- SELECT *
  FROM Serves

- Single-table query
- * is a shorthand for “all columns”

---

Example: ORDER BY

- SELECT *
  FROM Serves
  ORDER BY beer

- Equivalent to “ORDER BY beer asc” (asc is default option)
- For descending order, use “desc”
- Can combine multiple orders
- What does this return?
  - ORDER BY beer asc, price desc

---

Example: some columns and DISTINCT

- SELECT beer
  FROM Serves
  Returns a bag

- Only want unique values? Use DISTINCT

- SELECT DISTINCT beer
  FROM Serves
  Returns a set

---

Example: selecting few rows

- SELECT beer AS mybeer
  FROM Serves
  WHERE price < 2.75

- SELECT S.beer
  FROM Serves S
  WHERE bar = 'The Edge'

What does these return?

---

Example: Join

- Find addresses of all bars that ‘Dan’ frequents

- SELECT B.address
  FROM Bar B, Frequents F
  WHERE B.name = F.bar
  AND F.drinker = 'Dan'

---

Semantics of SFW

- SELECT $E_1$, $E_2$, ..., $E_n$
  FROM $R_1$, $R_2$, ..., $R_m$
  WHERE condition

  - For each $t_1$ in $R_1$:
    - For each $t_2$ in $R_2$:
      - For each $t_m$ in $R_m$:
        1. Apply “FROM”
           Form “cross-product” of $R_1$, .., $R_m$
        2. Apply “WHERE”
           Only consider satisfying rows
        3. Apply “SELECT”
           Output the desired columns

- NOT EQUAL TO: Use <>
- LIKE matches a string against a pattern
  - % matches any sequence of zero or more characters

- Can also use built-in functions such as SUBSTR, ABS, etc.
Step 1: Illustration of Semantics of SFW

- NOTE: This is "NOT HOW" the DBMS outputs the result, but "WHAT" it outputs!

**Form a "Cross product" of two relations**

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Drinker</th>
<th>Bar</th>
<th>Times_a_week</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Edge</td>
<td>108 Morris Street</td>
<td>Ben</td>
<td>Satisfaction</td>
<td>2</td>
</tr>
<tr>
<td>The Edge</td>
<td>108 Morris Street</td>
<td>Dan</td>
<td>The Edge</td>
<td>1</td>
</tr>
<tr>
<td>The Edge</td>
<td>108 Morris Street</td>
<td>Dan</td>
<td>The Edge</td>
<td>2</td>
</tr>
<tr>
<td>The Edge</td>
<td>905 W. Main Street</td>
<td>Ben</td>
<td>Satisfaction</td>
<td>2</td>
</tr>
<tr>
<td>The Edge</td>
<td>905 W. Main Street</td>
<td>Dan</td>
<td>The Edge</td>
<td>1</td>
</tr>
<tr>
<td>The Edge</td>
<td>905 W. Main Street</td>
<td>Dan</td>
<td>The Edge</td>
<td>2</td>
</tr>
</tbody>
</table>

Step 2: Illustration of Semantics of SFW

- NOTE: This is "NOT HOW" the DBMS outputs the result, but "WHAT" it outputs!

**Discard rows that do not satisfy WHERE condition**

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Drinker</th>
<th>Bar</th>
<th>Times_a_week</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Edge</td>
<td>108 Morris Street</td>
<td>Dan</td>
<td>Satisfaction</td>
<td>2</td>
</tr>
<tr>
<td>The Edge</td>
<td>905 W. Main Street</td>
<td>Dan</td>
<td>The Edge</td>
<td>2</td>
</tr>
<tr>
<td>The Edge</td>
<td>905 W. Main Street</td>
<td>Dan</td>
<td>The Edge</td>
<td>2</td>
</tr>
</tbody>
</table>

Step 3: Illustration of Semantics of SFW

- NOTE: This is "NOT HOW" the DBMS outputs the result, but "WHAT" it outputs!

**Output the "address" output of rows that survived**

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Drinker</th>
<th>Bar</th>
<th>Times_a_week</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Edge</td>
<td>108 Morris Street</td>
<td>Dan</td>
<td>Satisfaction</td>
<td>2</td>
</tr>
</tbody>
</table>

Final output: Illustration of Semantics of SFW

- NOTE: This is "NOT HOW" the DBMS outputs the result, but "WHAT" it outputs!

**Output the "address" output of rows that survived**

- SQL vs. C++, Java, Python...
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

More on selection
• Selection condition can include any column of $R$, constants, comparison (≤, ≥, etc.) and Boolean connectives (A: and, V: or, ~: not)
  • Example: Serves tuples for “The Edge” or price ≥ 2.75
    \[ \sigma_{\text{bar} = \text{The Edge} \lor \text{price} \geq 2.75} \text{Serves} \]
  • You must be able to evaluate the condition over each single row of the input table!
  • Example: the most expensive beer at any bar
    \[ \sigma_{\text{price} = \text{every price in Serves User}} \text{WRONG!} \]

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 $r \times s$ (concatenation of $r$ and $s$)

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 $p$ (set!)
  \[ \sigma_{\text{price} \leq 2.75} \text{Serves} \]
  No actual deletion!
  Equivalent SQL query?

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 $L$ (set!)
  \[ \pi_{\text{beer, price}} \text{Serves} \]

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 $r \times s$ if $r$ and $s$ satisfy $p$
  \[ \sigma_p (R \times S) \]
Join example

- Extend Frequent relation with addresses of the bars

\[
\text{Frequent} \bowtie \text{Bar}
\]

<table>
<thead>
<tr>
<th>Frequent</th>
<th>Bar</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Edge</td>
<td>108 Morris Street</td>
</tr>
<tr>
<td>Satisfaction</td>
<td>905 W. Main Street</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 \( \pi_x(R \bowtie_p S) \), 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

\[
\text{Serves} \bowtie \text{Likes} = \pi_{\text{bar,beer,price,drinker}} \left( \text{Serves} \bowtie \text{beer,price} \right)
\]

<table>
<thead>
<tr>
<th>Serves</th>
<th>Likes</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Edge</td>
<td>Budweiser 2.50 Dan</td>
</tr>
<tr>
<td>The Edge</td>
<td>Corona 3.00 Amy</td>
</tr>
<tr>
<td>Satisfaction</td>
<td>Budweiser 2.25 Dan</td>
</tr>
<tr>
<td>The Edge</td>
<td>Corona 3.00 Amy</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 \( S \)
  - How can you write it using other operators?
Expression tree notation
• Find addresses of all bars that ’Dan’ frequents

Also called logical Plan tree

Using the same relation multiple times
• Find drinkers who frequent both “The Edge” and “Satisfaction”

Renaming
• Input: a table $R$
• Notation: $\rho_{X} R$, $\rho_{(A_{1}, A_{2}, \ldots)} R$, or $\rho_{(A_{1}, A_{2}, \ldots)} R$
• 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

Summary of core operators
• Selection: $\sigma_{P} R$
• Projection: $\pi_{I} R$
• Cross product: $R \times S$
• Union: $R \cup S$
• Difference: $R - S$
• Renaming: $\rho_{X} R$
  • Does not really add “processing” power

Summary of derived operators
• Join: $R \bowtie_{P} S$
• Natural join: $R \bowtie S$
• Intersection: $R \cap S$

Exercise
• Bars that drinkers in address “300 N. Duke Street” do not frequent
A trickier exercise

For each bar, find the drinkers who frequent it max no. times a week

A deeper question:
When (and why) is “−” needed?

Monotone operators

Add more rows to the input...

• 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: $\sigma_R$ 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 max price is 3.0
  • Add another row with price 3.01
  • Old answer is invalidated
  • So it must use difference!

Extensions to relational algebra

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