Relational Model and Algebra

Introduction to Databases
CompSci 316 Spring 2020
Announcements (Tue. Jan. 14)

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

• HW1 has been posted, 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

Drinker **Likes** Beer

Bar **Serves** Beer At price “Y”

Bar Each has an address

Beer Each has a brewer

Drinker Each has an address
“Beers” as a Relational Database

What is an example of a

- Relation
- Attribute
- Tuple
- Schema
- Instance

What is

- Set semantic
  - in relational model
- Bag semantic
  - In SQL (why)

- Set semantic
  - No duplicates, Order of tuples does not matter
- Bag semantic
  - Duplicates allowed, for efficiency and flexibility
  - Do not want duplicates? Use SELECT DISTINCT ...
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”

<table>
<thead>
<tr>
<th>bar</th>
<th>beer</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Edge</td>
<td>Budweiser</td>
<td>2.50</td>
</tr>
<tr>
<td>The Edge</td>
<td>Corona</td>
<td>3.00</td>
</tr>
<tr>
<td>Satisfaction</td>
<td>Budweiser</td>
<td>2.25</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>bar</th>
<th>beer</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Edge</td>
<td>Budweiser</td>
<td>2.50</td>
</tr>
<tr>
<td>The Edge</td>
<td>Corona</td>
<td>3.00</td>
</tr>
<tr>
<td>Satisfaction</td>
<td>Budweiser</td>
<td>2.25</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>bar</th>
<th>beer</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Edge</td>
<td>Budweiser</td>
<td>2.50</td>
</tr>
<tr>
<td>The Edge</td>
<td>Corona</td>
<td>3.00</td>
</tr>
<tr>
<td>Satisfaction</td>
<td>Budweiser</td>
<td>2.25</td>
</tr>
</tbody>
</table>
Example: selecting few rows

- SELECT beer AS mybeer
  FROM Serves
  WHERE price < 2.75

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

- SELECT list can contain expressions
  Can also use built-in functions such as SUBSTR, ABS, etc.
- NOT EQUAL TO: Use <>
- LIKE matches a string against a pattern
  % matches any sequence of zero or more characters
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’

<table>
<thead>
<tr>
<th>name</th>
<th>address</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>

<table>
<thead>
<tr>
<th>drinker</th>
<th>bar</th>
<th>times_a_week</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ben</td>
<td>Satisfaction</td>
<td>2</td>
</tr>
<tr>
<td>Dan</td>
<td>The Edge</td>
<td>1</td>
</tr>
<tr>
<td>Dan</td>
<td>Satisfaction</td>
<td>2</td>
</tr>
</tbody>
</table>
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 $R_m$:

  If *condition* is true over $t_1, t_2, ..., t_m$:

  1. Apply “FROM”
     Form “cross-product” of $R_1, .., R_m$

  2. Apply “WHERE”
     Only consider satisfying rows

  Compute and output $E_1, E_2, ..., E_n$ as a row

  3. Apply “SELECT”
     Output the desired columns
Step 1: Illustration of Semantics of SFW

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

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

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>Satisfaction</td>
<td>2</td>
</tr>
<tr>
<td>Satisfaction</td>
<td>905 W. Main Street</td>
<td>Ben</td>
<td>Satisfaction</td>
<td>2</td>
</tr>
<tr>
<td>Satisfaction</td>
<td>905 W. Main Street</td>
<td>Dan</td>
<td>The Edge</td>
<td>1</td>
</tr>
<tr>
<td>Satisfaction</td>
<td>905 W. Main Street</td>
<td>Dan</td>
<td>Satisfaction</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!

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

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>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>Satisfaction</td>
<td>2</td>
</tr>
<tr>
<td>Satisfaction</td>
<td>905 W. Main Street</td>
<td>Ben</td>
<td>Satisfaction</td>
<td>2</td>
</tr>
<tr>
<td>Satisfaction</td>
<td>905 W. Main Street</td>
<td>Dan</td>
<td>The Edge</td>
<td>4</td>
</tr>
<tr>
<td>Satisfaction</td>
<td>905 W. Main Street</td>
<td>Dan</td>
<td>Satisfaction</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

### SQL Query

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

### Table

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

### Table

<table>
<thead>
<tr>
<th>Frequents</th>
<th>drinker</th>
<th>bar</th>
<th>times_a_week</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ben</td>
<td>Satisfaction</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Dan</td>
<td>The Edge</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Dan</td>
<td>Satisfaction</td>
<td>2</td>
<td></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

• SELECT B.address
  FROM Bar B, Frequents F
  WHERE B.name = F.bar
  AND F.drinker = ‘Dan’

<table>
<thead>
<tr>
<th>Bar</th>
<th>address</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>

<table>
<thead>
<tr>
<th>Frequents</th>
</tr>
</thead>
<tbody>
<tr>
<td>drinker</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>Ben</td>
</tr>
<tr>
<td>Dan</td>
</tr>
<tr>
<td>Dan</td>
</tr>
</tbody>
</table>
SQL vs. C++, Java, Python...
**SQL vs. C++, Java, Python...**

**SQL is declarative**

- Programmer specifies what answers a query should return,
- but not how the query is executed
- DBMS picks the best execution strategy based on availability of indexes, data/workload characteristics, etc.
- Not a “Procedural” or “Operational” language like C++, Java, Python
- There are several ways to write a query, but equivalent queries always provide the same (equivalent) results
- SQL (+ its execution and optimizations) is based on a strong foundation of “Relational Algebra”
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 $p$ (set!)

Example: Find beers with price < 2.75

<table>
<thead>
<tr>
<th>bar</th>
<th>beer</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Edge</td>
<td>Budweiser</td>
<td>2.50</td>
</tr>
<tr>
<td>The Edge</td>
<td>Corona</td>
<td>3.00</td>
</tr>
<tr>
<td>Satisfaction</td>
<td>Budweiser</td>
<td>2.25</td>
</tr>
</tbody>
</table>

$\sigma_{price<2.75} Serves$

<table>
<thead>
<tr>
<th>bar</th>
<th>beer</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Edge</td>
<td>Budweiser</td>
<td>2.50</td>
</tr>
<tr>
<td>Satisfaction</td>
<td>Budweiser</td>
<td>2.25</td>
</tr>
</tbody>
</table>

No actual deletion!

Equivalent SQL query?
More on selection

• Selection condition can include any column of \( R \), constants, comparison (=, ≤, etc.) and Boolean connectives (\( \land \): and, \( \lor \): or, \( \neg \): not)
  
  • Example: Serves tuples for “The Edge” or price >= 2.75

\[ \sigma_{bar='The Edge' \lor price \geq 2.75} \text{Serves} \]

• You must be able to evaluate the condition over each single row of the input table!
  
  • Example: the most expensive beer at any bar

\[ \sigma_{price \geq \text{every price in Serves}} \text{User} \text{WRONG!} \]

<table>
<thead>
<tr>
<th>bar</th>
<th>beer</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Edge</td>
<td>Budweiser</td>
<td>2.50</td>
</tr>
<tr>
<td>The Edge</td>
<td>Corona</td>
<td>3.00</td>
</tr>
<tr>
<td>Satisfaction</td>
<td>Budweiser</td>
<td>2.25</td>
</tr>
</tbody>
</table>
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!)

Example: Find all the prices for each beer

<table>
<thead>
<tr>
<th>bar</th>
<th>beer</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Edge</td>
<td>Budweiser</td>
<td>2.50</td>
</tr>
<tr>
<td>The Edge</td>
<td>Corona</td>
<td>3.00</td>
</tr>
<tr>
<td>Satisfaction</td>
<td>Budweiser</td>
<td>2.25</td>
</tr>
</tbody>
</table>

Output of $\pi_{beer} Serves$?

<table>
<thead>
<tr>
<th>beer</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Budweiser</td>
<td>2.50</td>
</tr>
<tr>
<td>Corona</td>
<td>3.00</td>
</tr>
<tr>
<td>Budweiser</td>
<td>2.25</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 $s$)

<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>Satisfaction</td>
<td>2</td>
</tr>
<tr>
<td>Satisfaction</td>
<td>905 W. Main Street</td>
<td>Ben</td>
<td>Satisfaction</td>
<td>2</td>
</tr>
<tr>
<td>Satisfaction</td>
<td>905 W. Main Street</td>
<td>Dan</td>
<td>The Edge</td>
<td>1</td>
</tr>
<tr>
<td>Satisfaction</td>
<td>905 W. Main Street</td>
<td>Dan</td>
<td>Satisfaction</td>
<td>2</td>
</tr>
</tbody>
</table>

Bar x Frequents

Note: ordering of columns does not matter, so $R \times S = S \times R$ (commutative)
Derived operator: join

(A.k.a. “theta-join”: most general joins)

• 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 $\sigma_p(R \times S)$

Predicate $p$ only has equality ($A = 5 \land B = 7$): equijoin

One of the most important operations!
Join example

• Extend Frequents relation with addresses of the bars

\[ \text{Frequents} \bowtie_{\text{bar} = \text{name}} \text{Bar} \]

<table>
<thead>
<tr>
<th>Bar</th>
<th>name</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Edge</td>
<td>108 Morris Street</td>
<td></td>
</tr>
<tr>
<td>Satisfaction</td>
<td>905 W. Main Street</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Frequents</th>
<th>name</th>
<th>address</th>
<th>drinker</th>
<th>bar</th>
<th>times_a_week</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ben</td>
<td>The Edge</td>
<td>108 Morris Street</td>
<td>Ben</td>
<td>Satisfaction</td>
<td>2</td>
</tr>
<tr>
<td>Dan</td>
<td>The Edge</td>
<td>108 Morris Street</td>
<td>Dan</td>
<td>The Edge</td>
<td>1</td>
</tr>
<tr>
<td>Dan</td>
<td>The Edge</td>
<td>108 Morris Street</td>
<td>Dan</td>
<td>Satisfaction</td>
<td>2</td>
</tr>
<tr>
<td>Ben</td>
<td>Satisfaction</td>
<td>905 W. Main Street</td>
<td>Ben</td>
<td>Satisfaction</td>
<td>2</td>
</tr>
<tr>
<td>Dan</td>
<td>The Edge</td>
<td>905 W. Main Street</td>
<td>Dan</td>
<td>The Edge</td>
<td>4</td>
</tr>
<tr>
<td>Dan</td>
<td>Satisfaction</td>
<td>905 W. Main Street</td>
<td>Dan</td>
<td>Satisfaction</td>
<td>2</td>
</tr>
</tbody>
</table>
(Lecture 3 -- contd)
Announcements (Thu. Jan. 16)  

• Reminder: HW1 due next Tuesday 01/21

• HW2 on RA to be posted next Tuesday 01/21, due on 01/28
  • HW2-Q1 on gradiance already open if you want to start early
  • Check gradiance code on Sakai announcements

• In-class lab on RA next Tuesday 01/21
  • Part of HW2 (~ 2 questions) in class to get the set up ready with TAs help
  • Last 30-40 mins of Tuesday’s lecture
  • You can work in groups of size 2 or 3, but would submit your own solution
  • You can submit by the next day -- 10% extra credit for finishing all questions correctly in class (last timestamp <= 4:20 pm)!

• In-class quiz next Thursday 01/23
  • You can work in groups of size 2 or 3, but would submit your own solution
  • 50% for attempt, 50% for correct answer
  • What if you miss a class? We would drop 25% (ceiling) of the lowest grades while calculating your final score for quiz, i.e. if we have 4 quizzes 1 dropped, 5-8 quizzes 2 dropped, ...

• Quiz or Lab -- you can submit while not being in the class too, but you would miss the fun of discussing with others (+ help from TAs for Labs)!
Join Types

• Theta Join

• Equi-Join

• Natural Join

• Later, (left/right) outer join, semi-join
Derived operator: natural join

• Input: two tables $R$ and $S$
• Notation: $R \bowtie S$ (i.e. no subscript)
• 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

$$Serves \bowtie Likes$$

$$= \pi_? (Serves \bowtie Likes)$$

$$= \pi_{\text{bar, beer, price, drinker}} (Serves \bowtie_{\text{Serves.beer = Likes.beer}} Likes)$$

<table>
<thead>
<tr>
<th>bar</th>
<th>beer</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Edge</td>
<td>Budweiser</td>
<td>2.50</td>
</tr>
<tr>
<td>The Edge</td>
<td>Corona</td>
<td>3.00</td>
</tr>
<tr>
<td>Satisfaction</td>
<td>Budweiser</td>
<td>2.25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>drinker</th>
<th>beer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amy</td>
<td>Corona</td>
</tr>
<tr>
<td>Dan</td>
<td>Budweiser</td>
</tr>
<tr>
<td>Dan</td>
<td>Corona</td>
</tr>
<tr>
<td>Ben</td>
<td>Budweiser</td>
</tr>
</tbody>
</table>

Natural Join is on beer

Only one column for beer in the output

What happens if the tables have two or more common columns?
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)

Example on board
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 $S$

Example on board
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?

• Shorthand for $R - (R - S)$
• Also equivalent to $S - (S - R)$
• And to $R \bowtie S$
Expression tree notation

- Find addresses of all bars that ‘Dan’ frequents

<table>
<thead>
<tr>
<th>name</th>
<th>address</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>

<table>
<thead>
<tr>
<th>drinker</th>
<th>bar</th>
<th>times_a_week</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ben</td>
<td>Satisfaction</td>
<td>2</td>
</tr>
<tr>
<td>Dan</td>
<td>The Edge</td>
<td>1</td>
</tr>
<tr>
<td>Dan</td>
<td>Satisfaction</td>
<td>2</td>
</tr>
</tbody>
</table>

Also called logical Plan tree

\[
\pi_{\text{address}} (\Join_{\text{drinker} = \text{name}} (\sigma_{\text{drinker} = 'Dan'} \text{Frequents}))
\]

Equivalent to \[\sigma_{\text{drinker} = 'Dan'} \text{Frequents}\]

What if you move \(\sigma\) to the top? Still correct? More or less efficient?
Using the same relation multiple times

- Find drinkers who frequent both “The Edge” and “Satisfaction”

\[
\pi_{\text{drinker}} \left( \text{Frequents} \Join \begin{array}{ll}
\text{bar} = \text{\textquotesingle The Edge\textquotesingle} & \land \\
\text{bar} = \text{\textquotesingle Satisfaction\textquotesingle} & \land \\
\text{drinker} = \text{drinker}
\end{array} \right)
\]

\[
\pi_{\text{uid}_1} \left( \rho_{(d_1,b_1,t_1)} \text{Frequents} \Join \begin{array}{ll}
\text{b}_1 = \text{\textquotesingle The Edge\textquotesingle} & \land \\
\text{b}_2 = \text{\textquotesingle Satisfaction\textquotesingle} & \land \\
\text{d}_1 = d_2
\end{array} \right) \rho_{(d_2,b_2,t_2)} \text{Frequents}
\]

**Wrong!**

**Rename!**
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
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, …
Exercise

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

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

\[ \rho_{\text{bar}} \]
\[ \pi_{\text{name}} \]
\[ \text{Bar} \]

Bars that the drinkers at this address frequent

\[ \pi_{\text{bar}} \]
\[ \text{drinker} = \text{name} \]
\[ \sigma_{\text{address} = "300 N. Duke Street"} \]

\[ \text{Frequents} \]
\[ \text{Drinker} \]
A trickier exercise

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

• For each bar, find the drinkers who frequent it max no. times a week
  • Who do NOT visit a bar max no. of times?
  • Whose times_of_week is lower than somebody else’s for a given bar

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 $R, R'$
Which operators are non-monotone?

• 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 price 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

• (Coming later)