Announcements

- Assignment 1 with extension is due now!
- Assignment 2 is due on Tuesday

Outline

- Structured Query Language (SQL)
- Extended relational algebra.
  - Bag semantics – needed for SQL.
- SQL Queries.
  - One-relation queries.
  - Multi-relation queries.

Bag Semantics

- A relation (in SQL, at least) is really a bag.
- It may contain the same tuple more than once, although there is no specified order (unlike a list).
- Example: \(\{1,2,1,3\}\) is a bag and not a set.
- Select, project, and join work for bags as well as sets.
  - Just work on a tuple-by-tuple basis, and don’t eliminate duplicates.

Bag Operations

- **Union**: sum the times an element appears in the two bags.
  - Example: \(\{1,2,1\} \cup \{1,2,3,3\} = \{1,1,1,2,2,3,3\}\).
- **Intersection**: take the minimum of the number of occurrences in each bag.
  - Example: \(\{1,2,1\} \cap \{1,2,3,3\} = \{1,2\}\)
- **Difference**: subtract the number of occurrences in the two bags.
  - Example: \(\{1,2,1\} - \{1,2,3,3\} = \{1\}\).

Different Laws for Bags

- Some familiar laws continue to hold for bags.
  - Examples: union and intersection are still commutative and associative.
- But other laws that hold for sets do **not** hold for bags!
Example

- \( R \cap (S \cup T) = (R \cap S) \cup (R \cap T) \) holds for sets but **not** for bags!
- Let \( R, S, \) and \( T \) each be the bag \( \{1\} \).
- Left side: \( S \cup T = \{1,1\}; R \cap (S \cup T) = \{1\} \).
- Right side: \( R \cap S = R \cap T = \{1\}; (R \cap S) \cup (R \cap T) = \{1\} \cup \{1\} = \{1,1\} \neq \{1\} \).

Extended Relational Algebra

- Adds features needed for SQL, bags.
- Duplicate-elimination operator \( \delta \).
- Extended projection.
- Sorting operator \( \tau \).

Duplicate Elimination

- \( \delta(R) = \) relation with one copy of each tuple that appears one or more times in \( R \).

<table>
<thead>
<tr>
<th>Beers</th>
<th>( \delta(\text{Beers}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>beer</td>
<td>price</td>
</tr>
<tr>
<td>Amstel</td>
<td>4</td>
</tr>
<tr>
<td>Guinness</td>
<td>6</td>
</tr>
<tr>
<td>Guinness</td>
<td>7</td>
</tr>
<tr>
<td>Guinness</td>
<td>7</td>
</tr>
<tr>
<td>Bud</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>beer</th>
<th>price</th>
<th>price-cost(Beers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amstel</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Guinness</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Guinness</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Guinness</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Bud</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Sorting

- \( \tau(L(R)) = \) list of tuples of \( R \), ordered according to attributes on list \( L \).
- Note that result type is outside the normal types (set or bag) for relational algebra.
- Consequence: \( \tau \) cannot be followed by other relational operators.

Extended Projection

- Allow the columns in the projection to be functions of one or more columns in the argument relation.

<table>
<thead>
<tr>
<th>Beers</th>
<th>( \pi_{\text{price},\text{price},\text{price-cost}(\text{Beers})}(\text{Beers}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>beer</td>
<td>cost</td>
</tr>
<tr>
<td>Amstel</td>
<td>4</td>
</tr>
<tr>
<td>Guinness</td>
<td>6</td>
</tr>
<tr>
<td>Guinness</td>
<td>7</td>
</tr>
<tr>
<td>Guinness</td>
<td>7</td>
</tr>
<tr>
<td>Guinness</td>
<td>8</td>
</tr>
<tr>
<td>Bud</td>
<td>5</td>
</tr>
</tbody>
</table>

SQL

- Structured Query Language (SQL)
  - The language of databases
  - Based on relational algebra
    - extended algebra operations
    - other extensions.
SQL Queries

- General form:

  \[
  \text{SELECT attributes you want} \\
  \text{FROM relations} \\
  \text{WHERE conditions about tuples from relations;} 
  \]

Running Example

- \textit{Beers(name, manf)}
- \textit{Bars(name, addr, license)}
- \textit{Drinkers(name, addr, phone)}
- \textit{Likes(drinker, beer)}
- \textit{Sells(bar, beer, price)}
- \textit{Frequents(drinker, bar)}

Example Query

- What beers are made by Anheuser-Busch?
- \textit{Beers(name, manf)}

  \[
  \text{SELECT name} \\
  \text{FROM Beers} \\
  \text{WHERE manf = 'Anheuser-Busch'}; 
  \]

  \begin{tabular}{|c|}
  \hline
  name \\
  BudLite \\
  Bud \\
  Michelob \\
  \hline
  \end{tabular}

  Result:

- \textit{Michelob}  
- \textit{Bud}  
- \textit{BudLite}

Formal Semantics of Single-Relation SQL Query

1. Start with the relation in the FROM clause.
2. Apply (bag) \(\sigma\), using condition in WHERE clause.
3. Apply (extended, bag) \(\pi\) using attributes in SELECT clause.

Equivalent Operational Semantics

- Imagine a \textit{tuple variable} ranging over all tuples of the relation. For each tuple:
  - Check if it satisfies the WHERE clause.
  - Print the values of terms in SELECT, if so.

Star as List of All Attributes

- \textit{Beers(name, manf)}

  \[
  \text{SELECT *} \\
  \text{FROM Beers} \\
  \text{WHERE manf = 'Anheuser-Busch'}; 
  \]

  \begin{tabular}{|c|c|}
  \hline
  name & manf \\
  BudLite & Anheuser-Busch \\
  Bud & Anheuser-Busch \\
  Michelob & Anheuser-Busch \\
  \hline
  \end{tabular}

  Result:
Renaming Columns

- Beers(name, manf)

  ```sql
  SELECT name AS beer
  FROM Beers
  WHERE manf = 'Anheuser-Busch';
  ```

  Result:
<table>
<thead>
<tr>
<th>beer</th>
</tr>
</thead>
<tbody>
<tr>
<td>BudLite</td>
</tr>
<tr>
<td>Bud</td>
</tr>
<tr>
<td>Michelob</td>
</tr>
</tbody>
</table>

Expressions as Values in Columns

- Sells(bar, beer, price)

  ```sql
  SELECT bar, beer, price*1.9 AS priceInLevs
  FROM Sells;
  ```

  Result:
<table>
<thead>
<tr>
<th>bar</th>
<th>beer</th>
<th>priceInLevs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spoon</td>
<td>Amstel</td>
<td>7.6</td>
</tr>
<tr>
<td>Spoon</td>
<td>Guinness</td>
<td>13.3</td>
</tr>
<tr>
<td>Whiskey</td>
<td>Guinness</td>
<td>13.3</td>
</tr>
<tr>
<td>Whiskey</td>
<td>Bud</td>
<td>9.5</td>
</tr>
</tbody>
</table>

  Note: no WHERE clause is OK.

Constant Values

- If you want an answer with a particular string in each row, use that constant as an expression.

  ```sql
  Likes(drinker, beer)
  SELECT drinker, 'connoisseur' AS status
  FROM Likes
  WHERE beer = 'Guinness';
  ```

  Result:
<table>
<thead>
<tr>
<th>drinker</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paul</td>
<td>connoisseur</td>
</tr>
<tr>
<td>Ryan</td>
<td>connoisseur</td>
</tr>
<tr>
<td>Paul</td>
<td>connoisseur</td>
</tr>
</tbody>
</table>

Example

- Find the price Spoon charges for Bud.

  ```sql
  SELLS(bar, beer, price)
  SELECT price
  FROM Sells
  WHERE bar = 'Spoon' AND beer = 'Bud';
  ```

  Conditions in WHERE clause can use logical operators
  AND, OR, NOT and parentheses in the usual way.

- SQL is case insensitive. Keywords like SELECT or AND
  can be written upper/lower case as you like.

- Only inside quoted strings does case matter.

String Patterns

- _ stands for any one character.

  "Attribute LIKE pattern" is a condition that
  is true if the string value of the attribute
  matches the pattern.

- Also NOT LIKE for negation.

Example

- Find drinkers whose phone has exchange 555.

  ```sql
  Drinkers(name, addr, phone)
  SELECT name
  FROM Drinkers
  WHERE phone LIKE '%555-_ _ _ _';
  ```

  Note patterns must be quoted, like strings
Nulls

- In place of a value in a tuple's component.
- Interpretation is not exactly *missing value*.
- There could be many reasons why no value is present, e.g., value inappropriate.

Comparing Nulls to Values

- 3rd truth value UNKNOWN.
- A query only produces tuples if the WHERE-condition evaluates to TRUE (UNKNOWN is not sufficient).

Example

```
SELECT bar
FROM Sells
WHERE price < 2.00 OR price >= 2.00;
```

UNKNOWN         UNKNOWN
UNKNOWN

- The result is empty, even though the WHERE condition is a tautology.

3-Valued Logic

- Think of true = 1; false = 0, and unknown = 1/2.
- Then:
  - AND = min.
  - OR = max.
  - NOT(\(x\)) = 1 – \(x\).

Some Key Laws Do Not Hold

- Example: Law of the excluded middle, i.e., \(p \text{ OR NOT } p = \text{TRUE}\)
- For 3-valued logic: if \(p = \text{unknown}\), then left side = \(\text{max}(1/2,(1-1/2)) = 1/2 \neq 1\).
- Like bag algebra, there is no way known to make 3-valued logic conform to all the laws we expect for sets/2-valued logic, respectively.

Multirelation Queries

- List of relations in FROM clause.
- Relation-dot-attribute disambiguates attributes from several relations.
- Example: Find the beers that the frequenters of Spoon like.

\[
\text{Likes(drinker, beer) Frequents(drinker, bar)}
\]

```
SELECT beer
FROM Frequent, Likes
WHERE bar = 'Spoon' AND Frequent.drinker = Likes.drinker;
```
Formal Semantics

- Same as for single relation, but start with the product of all the relations mentioned in the FROM clause:
  - Apply selection (for bags) – WHERE clause
  - Apply projection (extended) – SELECT clause

Operational Semantics

- Consider a tuple variable for each relation in the FROM.
- Imagine these tuple variables each pointing to a tuple of their relation, in all combinations (e.g., nested loops).
- If the current assignment of tuple-variables to tuples makes the WHERE true, then output the attributes of the SELECT.

Explicit Tuple Variables

- Sometimes we need to refer to two or more copies of a relation.
- Use tuple variables as aliases of the relations.
- Example: Find pairs of beers by the same manufacturer.

```
SELECT b1.name, b2.name
FROM Beers b1, Beers b2
WHERE b1.manf = b2.manf AND
  b1.name < b2.name;
```

Explicit Tuple Variables

- SQL permits AS between relation and its tuple variable; Oracle does not.
- Note that b1.name < b2.name is needed to avoid producing (Bud, Bud) and to avoid producing a pair in both orders.

Examples

- Find all bars that sell two different beers at the same price.
- Find all bars that sell three different beers at the same price.
- Find all drinkers that frequent a bar that serves their favorite beer.