Schedule

• Today (TH) Bags and SQL Queries.
  ◆ Read Sections 6.1-6.3. Project Part 2 due.

• Oct. 16 (T) Duplicates, Aggregation, Modifications.
  ◆ Read Sections 5.3-5.4, 6.4-6.5. Assignment 3 due.

• Oct. 18 (TH) Schemas, Views.
  ◆ Read Sections 6.6-6.7. Project Part 3 due.
Review: Weak Entities

- Consider a relationship, Ordered, between two entity sets, Buyer and Product

- How can we add Shipments to the mix?

- is wrong. Why?
• Solution: make **Ordered** into a weak entity set.

• And then add **Shipment**.

**Part-of** is many-many and not a weak relationship!
Bag Semantics

A relation (in SQL, at least) is really a bag or multiset.

• 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 Union

Sum the times an element appears in the two bags.
• Example: \( \{1,2,1\} \cup \{1,2,3,3\} = \{1,1,1,2,2,3,3\} \).

Bag Intersection

Take the minimum of the number of occurrences in each bag.
• Example: \( \{1,2,1\} \cap \{1,2,3,3\} = \{1,2\} \).

Bag Difference

Proper-subtract the number of occurrences in the two bags.
• Example: \( \{1,2,1\} – \{1,2,3,3\} = \{1\} \).
Laws for Bags Differ From Laws for Sets

• 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

\[ R \cap (S \cup T) \equiv (R \cap S) \cup (R \cap T) \] holds for sets.

• 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 ("Nonclassical") Relational Algebra

Adds features needed for SQL, bags.

1. Duplicate-elimination operator $\delta$.
2. Extended projection.
3. Sorting operator $\tau$.
4. Grouping-and-aggregation operator $\gamma$.
5. Outerjoin operator $\bowtie$. 
Duplicate Elimination

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

Example

\[
R = \begin{array}{cc}
A & B \\
1 & 2 \\
3 & 4 \\
1 & 2 \\
\end{array}
\]

\[
\delta(R) = \begin{array}{cc}
A & B \\
1 & 2 \\
3 & 4 \\
\end{array}
\]
Sorting

• $\tau_L(R) = \text{list of tuples of } R, \text{ ordered according to attributes on list } L.$

• Note that result type is outside the normal types (set or bag) for relational algebra.
  ✦ Consequence: $\tau$ cannot be followed by other relational operators.

Example

$R = \begin{array}{cc}
A & B \\
1 & 3 \\
3 & 4 \\
5 & 2 \\
\end{array}$

$\tau_B(R) = [(5,2), (1,3), (3,4)].$
Extended Projection

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

Example

$$R = \begin{array}{c|c}
A & B \\
1 & 2 \\
3 & 4
\end{array}$$

$$\pi_{A+B,A,A}(R) = \begin{array}{c|c|c}
A+B & A1 & A2 \\
3 & 1 & 1 \\
7 & 3 & 3
\end{array}$$
Aggregation Operators

• These are not relational operators; rather they summarize a column in some way.

• Five standard operators: Sum, Average, Count, Min, and Max.
Grouping Operator

$\gamma_L(R)$, where $L$ is a list of elements that are either
a) Individual (grouping) attributes or
b) Of the form $\theta(A)$, where $\theta$ is an aggregation operator and $A$ the attribute to which it is applied,
is computed by:
1. Group $R$ according to all the grouping attributes on list $L$.
2. Within each group, compute $\theta(A)$, for each element $\theta(A)$ on list $L$.
3. Result is the relation whose columns consist of one tuple for each group. The components of that tuple are the values associated with each element of $L$ for that group.
Example

Let $R =$

<table>
<thead>
<tr>
<th>bar</th>
<th>beer</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joe's</td>
<td>Bud</td>
<td>2.00</td>
</tr>
<tr>
<td>Joe's</td>
<td>Miller</td>
<td>2.75</td>
</tr>
<tr>
<td>Sue's</td>
<td>Bud</td>
<td>2.50</td>
</tr>
<tr>
<td>Sue's</td>
<td>Coors</td>
<td>3.00</td>
</tr>
<tr>
<td>Mel's</td>
<td>Miller</td>
<td>3.25</td>
</tr>
</tbody>
</table>

Compute $\gamma_{\text{beer,AVG(price)}}(R)$.

1. Group by the grouping attribute(s), beer in this case:

<table>
<thead>
<tr>
<th>bar</th>
<th>beer</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joe's</td>
<td>Bud</td>
<td>2.00</td>
</tr>
<tr>
<td>Sue's</td>
<td>Bud</td>
<td>2.50</td>
</tr>
<tr>
<td>Joe's</td>
<td>Miller</td>
<td>2.75</td>
</tr>
<tr>
<td>Mel's</td>
<td>Miller</td>
<td>3.25</td>
</tr>
<tr>
<td>Sue's</td>
<td>Coors</td>
<td>3.00</td>
</tr>
</tbody>
</table>
2. Compute average of price within groups:

<table>
<thead>
<tr>
<th>beer</th>
<th>AVG(price)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bud</td>
<td>2.25</td>
</tr>
<tr>
<td>Miller</td>
<td>3.00</td>
</tr>
<tr>
<td>Coors</td>
<td>3.00</td>
</tr>
</tbody>
</table>
Outerjoin

The normal join can “lose” information, because a tuple that doesn’t join with any from the other relation (*dangles*) has no vestage in the join result.

- The *null value* \(\perp\) can be used to “pad” dangling tuples so they appear in the join.
- Gives us the *outerjoin* operator \(\bowtie\).
- Variations: theta-outerjoin, left- and right-outerjoin (pad only dangling tuples from the left (resp., right)).
Example

\[ R = \begin{array}{c|c}
1 & 2 \\
3 & 4 \\
\end{array} \]

\[ S = \begin{array}{c|c}
B & C \\
4 & 5 \\
6 & 7 \\
\end{array} \]

\[ R \bowtie S = \begin{array}{c|c|c}
A & B & C \\
3 & 4 & 5 \\
1 & 2 & \bot \\
\bot & 6 & 7 \\
\end{array} \]
SQL Queries

• Principal form:
  SELECT desired attributes
  FROM tuple variables — range over relations
  WHERE condition about tuple variables;

Running example relation schema:
  Beers(name, manf)
  Bars(name, addr, license)
  Drinkers(name, addr, phone)
  Likes(drinker, beer)
  Sells(bar, beer, price)
  Frequents(drinker, bar)
Example

What beers are made by Anheuser-Busch?

Beers(name, manf)

SELECT name
FROM Beers
WHERE manf = 'Anheuser-Busch';

• Note single quotes for strings.

  name
  Bud
  Bud Lite
  Michelob
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 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

Beers (name, manf)

SELECT *
FROM Beers
WHERE manf = 'Anheuser-Busch';

<table>
<thead>
<tr>
<th>name</th>
<th>manf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bud</td>
<td>Anheuser-Busch</td>
</tr>
<tr>
<td>Bud Lite</td>
<td>Anheuser-Busch</td>
</tr>
<tr>
<td>Michelob</td>
<td>Anheuser-Busch</td>
</tr>
</tbody>
</table>
Renaming columns

Beers(name, manf)

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

beer
Bud
Bud Lite
Michelob
Expressions as Values in Columns

\[ \text{Sells(bar, beer, price)} \]

\[
\text{SELECT bar, beer, price}\times120 \text{ AS priceInYen}
\]

\[
\text{FROM Sells;}
\]

<table>
<thead>
<tr>
<th>bar</th>
<th>beer</th>
<th>priceInYen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joe’s</td>
<td>Bud</td>
<td>300</td>
</tr>
<tr>
<td>Sue’s</td>
<td>Miller</td>
<td>360</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

- Note: no \text{WHERE} clause is OK.
• Trick: If you want an answer with a particular string in each row, use that constant as an expression.

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

\[
\text{SELECT} \ \text{drinker,}
\]
\[
\text{'likes Bud' AS whoLikesBud}
\]

\[
\text{FROM Likes}
\]

\[
\text{WHERE beer = 'Bud'};
\]

<table>
<thead>
<tr>
<th>drinker</th>
<th>whoLikesBud</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sally</td>
<td>likes Bud</td>
</tr>
<tr>
<td>Fred</td>
<td>likes Bud</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Example

• Find the price Joe's Bar charges for Bud.

\[
\text{Sells}(\text{bar}, \text{beer}, \text{price})
\]

\[
\begin{align*}
\text{SELECT} & \quad \text{price} \\
\text{FROM} & \quad \text{Sells} \\
\text{WHERE} & \quad \text{bar} = \text{'Joe''s Bar'} \ \text{AND} \\
& \quad \text{beer} = \text{'Bud'};
\end{align*}
\]

• Note: two single-quotes in a character string represent one single quote.

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

• Remember: SQL is \textit{case insensitive}. Keywords like SELECT or AND can be written upper/lower case as you like.
  • Only inside quoted strings does case matter.
Patterns

• % stands for any string.
• _ 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.

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

<table>
<thead>
<tr>
<th>bar</th>
<th>beer</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joe's bar</td>
<td>Bud</td>
<td>NULL</td>
</tr>
</tbody>
</table>

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

• Joe's Bar is not produced, 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 Fail to Hold

Example: Law of the excluded middle, i.e.,

\[ p \lor \neg p = \text{TRUE} \]

- For 3-valued logic: if p = unknown, then left side = \( \max(1/2,(1-1/2)) = 1/2 \neq 1 \).
- 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 Joe's Bar like.

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

SELECT beer
FROM Frequents, Likes
WHERE bar = 'Joe''s Bar' AND
    Frequents.drinker = Likes.drinker;
Formal Semantics of Multirelation Queries

Same as for single relation, but start with the product of all the relations mentioned in the FROM 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.
Frequents

 Likes

Sally

Joe's

drinker  bar

Sally

beer

1

drinker  beer
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.

\[
\text{Beers}(\text{name}, \text{manf})
\]

\[
\begin{align*}
\text{SELECT } & b1.\text{name}, b2.\text{name} \\
\text{FROM } & \text{Beers } b1, \text{Beers } b2 \\
\text{WHERE } & b1.\text{manf} = b2.\text{manf} \text{ AND } \\
& \quad b1.\text{name} < b2.\text{name};
\end{align*}
\]

• SQL permits AS between relation and its tuple variable; Oracle does not.

• Note that \(b1.\text{name} < b2.\text{name}\) is needed to avoid producing (Bud, Bud) and to avoid producing a pair in both orders.
Subqueries

Result of a select-from-where query can be used in the where-clause of another query.

Simplest Case: Subquery Returns a Single, Unary Tuple

Find bars that serve Miller at the same price Joe charges for Bud.

\[
\text{SELECT bar} \\
\text{FROM Sells} \\
\text{WHERE beer = 'Miller' AND price =} \\
\quad (\text{SELECT price} \\
\quad \text{FROM Sells}) \\
\quad \text{WHERE bar = 'Joe's Bar' AND} \\
\quad \quad \text{beer = 'Bud');}
\]

- Notice the *scoping rule*: an attribute refers to the most closely nested relation with that attribute.
- Parentheses around subquery are essential.
The \textbf{IN} Operator

“Tuple \textbf{IN} relation” is true iff the tuple is in the relation.

Example

Find the name and manufacturer of beers that Fred likes.

\begin{verbatim}
Beers(name, manf)
Likes(drinker, beer)

SELECT *
FROM Beers
WHERE name IN
    (SELECT beer
     FROM Likes
     WHERE drinker = 'Fred');
\end{verbatim}

\textbf{• Also: NOT IN.}
EXISTS

“EXISTS(relation)” is true iff the relation is nonempty.

Example

Find the beers that are the unique beer by their manufacturer.

```sql
Beers(name, manf)

SELECT name
FROM Beers b1
WHERE NOT EXISTS
  (SELECT *
   FROM Beers
   WHERE manf = b1.manf AND
   name <> b1.name);
```

- Note scoping rule: to refer to outer `Beers` in the inner subquery, we need to give the outer a tuple variable, `b1` in this example.
- A subquery that refers to values from a surrounding query is called a *correlated subquery*. 
Quantifiers

ANY and ALL behave as existential and universal quantifiers, respectively.

• Beware: in common parlance, “any” and “all” seem to be synonyms, e.g., “I am fatter than any of you” vs. “I am fatter than all of you.” But in SQL:

Example

Find the beer(s) sold for the highest price.

\[
\text{Sells}(\text{bar, beer, price})
\]

\[
\begin{align*}
\text{SELECT beer} \\
\text{FROM Sells} \\
\text{WHERE price} \geq \text{ALL(} \\
\text{SELECT price} \\
\text{FROM Sells)};
\end{align*}
\]

Class Problem

Find the beer(s) not sold for the lowest price.