CSE 544
Principles of Database Management Systems

Fall 2016
Lecture 2 – Relational Algebra and SQL
Announcements

• Paper review
  – First paper review is posted now (due Wednesday 6pm)
  – Details on website

• Milestone 1 of the project was due
  – You don’t need to choose a project yet; more suggestions will continue to be posted on website
  – M2 Project Proposal due next Wednesday
Outline

Three topics today

• Relational algebra

• Crash course on SQL
Relational Operators

• **Selection**: $\sigma_{\text{condition}}(S)$
  - Condition is Boolean combination ($\wedge, \vee$) of terms
  - Term is: attr. op constant, attr. op attr.
  - Op is: $<$, $\leq$, $=$, $\neq$, $\geq$, or $>$

• **Projection**: $\pi_{\text{list-of-attributes}}(S)$

• **Union** ($\cup$), **Intersection** ($\cap$), **Set difference** ($-$),

• **Cross-product or cartesian product** ($\times$)

• **Join**: $R \bowtie_{\theta} S = \sigma_{\theta}(R \times S)$

• **Division**: $R/S$

• **Rename** $\rho(R(F), E)$
Join Galore

- **Theta-join**: \( R \bowtie_\theta S = \sigma_\theta(R \times S) \)
  - Join of \( R \) and \( S \) with a join condition \( \theta \)
  - Cross-product followed by selection \( \theta \)

- **Equijoin**: \( R \bowtie_\theta S = \pi_A (\sigma_\theta(R \times S)) \)
  - Join condition \( \theta \) consists only of equalities
  - Projection \( \pi_A \) drops all redundant attributes

- **Natural join**: \( R \bowtie S = \pi_A (\sigma_\theta(R \times S)) \)
  - aka Equijoin
  - Equality on **all** fields with same name in \( R \) and in \( S \)
  - Natural join **does** drop redundant attributes

*Alvin is wrong…*
Theta-Join Example

AnonPatient $P$

<table>
<thead>
<tr>
<th>age</th>
<th>zip</th>
<th>disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>98125</td>
<td>heart</td>
</tr>
<tr>
<td>19</td>
<td>98120</td>
<td>flu</td>
</tr>
</tbody>
</table>

Voters $V$

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>p2</td>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>

$P \bowtie_{P.zip = V.zip \text{ and } P.age \leq V.age + 1 \text{ and } P.age \geq V.age - 1} V$

<table>
<thead>
<tr>
<th>P.age</th>
<th>P.zip</th>
<th>disease</th>
<th>name</th>
<th>V.age</th>
<th>V.zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>98120</td>
<td>flu</td>
<td>p2</td>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>
# Equijoin Example

<table>
<thead>
<tr>
<th>AnonPatient P</th>
<th>Voters V</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td>zip</td>
</tr>
<tr>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>

\[
P \bowtie_{P.\text{age}=V.\text{age}} V
\]

<table>
<thead>
<tr>
<th>age</th>
<th>P.zip</th>
<th>disease</th>
<th>name</th>
<th>V.zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
<td>p1</td>
<td>98125</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
<td>p2</td>
<td>98120</td>
</tr>
</tbody>
</table>
### Natural Join Example

#### AnonPatient P

<table>
<thead>
<tr>
<th>age</th>
<th>zip</th>
<th>disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
</tr>
</tbody>
</table>

#### Voters V

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>p2</td>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>

#### P \Join V

<table>
<thead>
<tr>
<th>age</th>
<th>zip</th>
<th>disease</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
<td>p1</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
<td>p2</td>
</tr>
</tbody>
</table>
Even More Joins

• Outer join
  – Include tuples with no matches in the output
  – Use NULL values for missing attributes

• Variants
  – Left outer join
  – Right outer join
  – Full outer join
Outer Join Example

<table>
<thead>
<tr>
<th>age</th>
<th>zip</th>
<th>disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
</tr>
<tr>
<td>33</td>
<td>98120</td>
<td>lung</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>p2</td>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>

**AnonPatient P** \(\bowtie\) **Voters V**

<table>
<thead>
<tr>
<th>age</th>
<th>zip</th>
<th>disease</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
<td>p1</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
<td>p2</td>
</tr>
<tr>
<td>33</td>
<td>98120</td>
<td>lung</td>
<td>null</td>
</tr>
</tbody>
</table>
Extended Operators of Relational Algebra

• Duplicate elimination ($\delta$)
  – Since commercial DBMSs operate on multisets/bags not sets

• Aggregate operators ($\gamma$)
  – Useful in practice and requires bag semantics
  – Min, max, sum, average, count

• Grouping operators ($\gamma$)
  – Partitions tuples of a relation into “groups”
  – Aggregates can then be applied to groups

• Sort operator ($\tau$)
Relational Calculus

- Alternative to relational algebra
  - Declarative query language
  - Describe what we want NOT how to get it
- Tuple relational calculus query
  - \( \{ T \mid p(T) \} \)
  - Where \( T \) is a tuple variable
  - \( p(T) \) denotes a formula that describes \( T \)
  - Result: set of all tuples for which \( p(T) \) is true
  - Language for \( p(T) \) is subset of first-order logic

Q1: Names of patients who have heart disease
\[
\{ T \mid \exists P \in \text{AnonPatient} \exists V \in \text{Voter} \\
\quad (P.zip = V.zip \land P.age = V.age \land P.disease = 'heart' \land T.name = V.name) \}
\]
Example

• Show set division on white board…
Outline

Three topics today

• Wrap up relational algebra
• Crash course on SQL
• Brief overview of database design
Structured Query Language: SQL

• Influenced by relational calculus

• Declarative query language

• Multiple aspects of the language
  – Data definition language (DDL)
    • Statements to create, modify tables and views
  – Data manipulation language (DML)
    • Statements to issue queries, insert, delete data
  – More
Outline

• Today: crash course in SQL DML
  – Data Manipulation Language
  – SELECT-FROM-WHERE-GROUPBY
  – Study independently: INSERT/DELETE/MODIFY

• Study independently SQL DDL
  – Data Definition Language
  – CREATE TABLE, DROP TABLE, CREATE INDEX, CLUSTER, ALTER TABLE, …
  – E.g. google for the postgres manual, or type this in psql:
    `\h create`
    `\h create table`
    `\h cluster`
SQL Query

Basic form: (plus many many many many more bells and whistles)

```
SELECT <attributes>
FROM <one or more relations>
WHERE <conditions>
```
### Simple SQL Query

#### SQL Query

```sql
SELECT PName, Price, Manufacturer
FROM Product
WHERE Price > 100
```

#### Table - Product

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203.99</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

---

"selection" and "projection"
Eliminating Duplicates

SELECT DISTINCT category
FROM Product

Compare to:

SELECT category
FROM Product
Ordering the Results

```
SELECT  pname, price, manufacturer
FROM     Product
WHERE    category='gizmo' AND price > 50
ORDER BY price, pname
```

Ties are broken by the 2nd attribute on the ORDER BY list, etc.

Ordering is ascending, unless you specify the DESC keyword.

Can also request only top-k with LIMIT clause.
Joins

Product (pname, price, category, manufacturer)
Company (cname, stockPrice, country)

Find all products under $200 manufactured in Japan; return their names and prices.

```
SELECT P.pname, P.price
FROM Product P, Company C
WHERE P.manufacturer = C.cname AND C.country = 'Japan'
AND P.price <= 200
```

```
SELECT P.pname, P.price
FROM Product P JOIN Company C ON P.manufacturer = C.cname
WHERE C.country = 'Japan' AND P.price <= 200
```
Semantics of SQL Queries

\[
\text{SELECT } a_1, a_2, \ldots, a_k
\]
\[
\text{FROM } R_1 \text{ AS } x_1, R_2 \text{ AS } x_2, \ldots, R_n \text{ AS } x_n
\]
\[
\text{WHERE } \text{Conditions}
\]

\[
\text{Answer} = \{\}
\]
\[
\text{for } x_1 \text{ in } R_1 \text{ do}
\]
\[
\text{for } x_2 \text{ in } R_2 \text{ do}
\]
\[
\ldots
\]
\[
\text{for } x_n \text{ in } R_n \text{ do}
\]
\[
\text{if } \text{Conditions}
\]
\[
\text{then } \text{Answer} = \text{Answer} \cup \{(a_1,\ldots,a_k)\}
\]
\[
\text{return } \text{Answer}
\]
Aggregation

SQL supports several aggregation operations:

- sum, count, min, max, avg

Except count, all aggregations apply to a single attribute
Grouping and Aggregation

Purchase(product, price, quantity)

Find total quantities for all sales over $1, by product.

\[
\begin{align*}
\text{SELECT} & \quad \text{product, Sum(quantity) AS TotalSales} \\
\text{FROM} & \quad \text{Purchase} \\
\text{WHERE} & \quad \text{price > 1} \\
\text{GROUP BY} & \quad \text{product}
\end{align*}
\]

Let’s see what this means…
Grouping and Aggregation

1. Compute the FROM and WHERE clauses.

2. Group by the attributes in the GROUPBY

3. Compute the SELECT clause: grouped attributes and aggregates.
1&2. FROM-WHERE-GROUPBY

<table>
<thead>
<tr>
<th>Product</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>0.5</td>
<td>50</td>
</tr>
<tr>
<td>Banana</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Banana</td>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>

WHERE price > 1
3. SELECT

<table>
<thead>
<tr>
<th>Product</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>0.5</td>
<td>50</td>
</tr>
<tr>
<td>Banana</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Banana</td>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>

What can go in SELECT clause?
Will return ONE TUPLE per group

```sql
SELECT product, Sum(quantity) AS TotalSales
FROM Purchase
WHERE price > 1
GROUP BY product
```
HAVING Clause

Same query as earlier, except that we consider only products that had at least 30 sales.

```
SELECT product, sum(price*quantity)
FROM Purchase
WHERE price > 1
GROUP BY product
HAVING Sum(quantity) > 30
```

HAVING clause contains conditions on aggregates.
WHERE vs HAVING

• WHERE condition is applied to individual rows
  – The rows may or may not contribute to the aggregate
  – No aggregates allowed here

• HAVING condition is applied to the entire group
  – Entire group is returned, or not at all
  – May use aggregate functions in the group
General form of Grouping and Aggregation

\[
\text{SELECT } S \\
\text{FROM } R_1, \ldots, R_n \\
\text{WHERE } C_1 \\
\text{GROUP BY } a_1, \ldots, a_k \\
\text{HAVING } C_2
\]

S = may contain attributes \( a_1, \ldots, a_k \) and/or any aggregates but NO OTHER ATTRIBUTES
C1 = is any condition on the attributes in \( R_1, \ldots, R_n \)
C2 = is any condition on aggregate expressions and on attributes \( a_1, \ldots, a_k \)
Semantics of SQL With Group-By

Evaluation steps:
1. Evaluate FROM-WHERE using Nested Loop Semantics
2. Group by the attributes $a_1, \ldots, a_k$
3. Apply condition $C_2$ to each group (may have aggregates)
4. Compute aggregates in $S$ and return the result
Subqueries

- A subquery is a SQL query nested inside a larger query
- Such inner-outer queries are called nested queries
- A subquery may occur in:
  - A SELECT clause
  - A FROM clause
  - A WHERE clause

- Rule of thumb: avoid writing nested queries when possible; keep in mind that sometimes it’s impossible
Subqueries in WHERE

Product (pname, price, cid)
Company(cid, cname, city)

Find all companies that make some products with price < 200

Using EXISTS:

```
SELECT DISTINCT C.cname
FROM Company C
WHERE EXISTS (SELECT *
               FROM Product P
               WHERE C.cid = P.cid and P.price < 200)
```
Subqueries in WHERE

Product (pname, price, cid)
Company(cid, cname, city)

Find all companies that make some products with price < 200

Using **IN**

```
SELECT DISTINCT C.cname 
FROM Company C 
WHERE C.cid IN (SELECT P.cid 
FROM Product P 
WHERE P.price < 200)
```
Subqueries in WHERE

Find all companies that make some products with price < 200

Using **ANY**:

```
SELECT DISTINCT C.cname
FROM Company C
WHERE 200 > ANY (SELECT price
FROM Product P
WHERE P.cid = C.cid)
```
Find all companies that make some products with price < 200

Now let’s unnest it:

```sql
SELECT DISTINCT C.cname
FROM Company C, Product P
WHERE C.cid= P.cid and P.price < 200
```

Subqueries in WHERE

Product (pname, price, cid)
Company(cid, cname, city)
Subqueries in WHERE

Product (pname, price, cid)
Company(cid, cname, city)

Find all companies that make only products with price < 200

same as:
Find all companies whose products all have price < 200

Universal quantifiers are hard! 😞
Subqueries in WHERE

1. Find *the other* companies: i.e. s.t. **some** product ≥ 200

   ```sql
   SELECT DISTINCT C.cname
   FROM Company C
   WHERE C.cid IN (SELECT P.cid
                   FROM Product P
                   WHERE P.price >= 200)
   ```

2. Find all companies s.t. **all** their products have price < 200

   ```sql
   SELECT DISTINCT C.cname
   FROM Company C
   WHERE C.cid NOT IN (SELECT P.cid
                        FROM Product P
                        WHERE P.price >= 200)
   ```
Subqueries in WHERE

Product (pname, price, cid)  
Company(cid, cname, city)

Find all companies that make only products with price < 200

Using EXISTS:

```
SELECT DISTINCT C.cname
FROM Company C
WHERE NOT EXISTS (SELECT *
    FROM Product P
    WHERE P.cid = C.cid and P.price >= 200)
```
Subqueries in WHERE

Product (pname, price, cid)
Company(cid, cname, city)

Find all companies that make only products with price < 200

Using \textbf{ALL}:

\begin{verbatim}
SELECT DISTINCT  C.cname  
FROM    Company C  
WHERE  200 > ALL (SELECT price  
FROM  Product P  
WHERE  P.cid = C.cid)
\end{verbatim}
Can we unnest the *universal quantifier* query?

• A query Q is **monotone** if:
  – Whenever we add tuples to one or more of the tables…
  – … the answer to the query cannot contain fewer tuples

• **Fact:** all unnested queries are monotone
  – Proof: using the “nested for loops” semantics

• **Fact:** Query with universal quantifier is not monotone

• **Consequence:** we cannot unnest a query with a universal quantifier