Principles of Database Systems
CSE 544

Lecture #2
SQL – The Complete Story
Announcements

• Paper assignment
  – Review was due last night
  – Discussion on Thursday

• We need to schedule a makeup lecture
  – Doodle: http://doodle.com/avg2rngq8zkwdek9
  – Please submit preferences by Wednesday night

• Find partners (0 or more) for the project
  – Project groups due by Friday, 4/12 (email)
  – You don’t need to choose a project yet; more suggestions will continue to be posted

• Start working on Homework 1 now!
  – Due Monday, 4/22/2013
Outline

• Today: crash course in **SQL DML**
  – Data Manipulation Language
  – **SELECT-FROM-WHERE-GROUPBY**
  – Also: NULLs, nested queries, lots of tricks
  – 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
    ...

• Practice the examples on the slides by running them on postgres
• Study independently whatever we don’t have time to cover today
Selections in SQL

```
SELECT * 
FROM Product 
WHERE category='Gadgets'
```

```
SELECT * 
FROM Product 
WHERE category > 'Gadgets'
```

```
SELECT * 
FROM Product 
WHERE category LIKE 'Ga%'
```

```
SELECT * 
FROM Product 
WHERE category LIKE '%dg%'
```
Projections (and Selections) in SQL

```
SELECT  pname
FROM    Product
WHERE   category='Gadgets'
```

```
SELECT  category
FROM    Product
```

```
SELECT DISTINCT  category
FROM    Product
```

Need DISTINCT (why?)
“DISTINCT”, “ORDER BY”, “LIMIT”

```
SELECT DISTINCT category
FROM Product

SELECT pname, price, manufacturer
FROM Product
WHERE category='gizmo' AND price > 50
ORDER BY price, pname
LIMIT 20
```
Keys and Foreign Keys

**Company**

<table>
<thead>
<tr>
<th>CName</th>
<th>StockPrice</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>25</td>
<td>USA</td>
</tr>
<tr>
<td>Canon</td>
<td>65</td>
<td>Japan</td>
</tr>
<tr>
<td>Hitachi</td>
<td>15</td>
<td>Japan</td>
</tr>
</tbody>
</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>
Joins

Product (PName, Price, Category, Manufacturer)
Company (CName, stockPrice, Country)

Find all products under $200 manufactured in Japan:

```
SELECT x.PName, x.Price
FROM Product x, Company y
WHERE x.Manufacturer=y.CName
    AND y.Country='Japan'
    AND x.Price <= 200
```
Semantics of SQL Queries

```
SELECT a_1, a_2, ..., a_k
FROM R_1 AS x_1, R_2 AS x_2, ..., R_n AS x_n
WHERE Conditions
```

```
Answer = {}
for x_1 in R_1 do
    for x_2 in R_2 do
        .....  
        for x_n in R_n do
            if Conditions
                then Answer = Answer ∪ {(a_1, ..., a_k)}

return Answer
```
Subqueries

• A subquery or a nested query is another SQL query nested inside a larger query

• A subquery may occur in:

```
SELECT
FROM
WHERE
```

Examples at the end of the lecture

Examples on following slides

Avoid writing nested queries when possible; keep in mind that sometimes it’s impossible
Running Example

Run this in postgres, then try the examples on the following slides.

```
cREATE TABLE company (cname text PRIMARY KEY, city text);
cREATE TABLE product (pname text PRIMARY KEY, price int, company text REFERENCES company);

INSERT INTO company VALUES('abc', 'seattle');
INSERT INTO company VALUES('cde', 'seattle');
INSERT INTO company VALUES('fgh', 'portland');
INSERT INTO company VALUES('klm', 'portland');

INSERT INTO product VALUES('p1', 10, 'abc');
INSERT INTO product VALUES('p2', 200, 'abc');
INSERT INTO product VALUES('p3', 10, 'cde');
INSERT INTO product VALUES('p4', 20, 'cde');

INSERT INTO product VALUES('p5', 10, 'fgh');
INSERT INTO product VALUES('p6', 200, 'fgh');
INSERT INTO product VALUES('p7', 10, 'klm');
INSERT INTO product VALUES('p8', 220, 'klm');
```
Existential Quantifiers

Find cities that have a company that manufacture *some* product with price $< 100$
Existential Quantifiers

Find cities that have a company that manufacture some product with price < 100

```
SELECT DISTINCT c.city
FROM Company c, Product p
WHERE c.cname = p.company
    and p.price < 100
```

Existential quantifiers are easy! 😊
Universal Quantifiers

Find cities that have a company such that all its products have price < 100

Universal quantifiers are hard! 😞
Universal Quantifiers

Find cities that have a company such that all its products have price < 100

Relational Calculus (a.k.a. First Order Logic) – next week

\[ q(y) = \exists x. \text{Company}(x,y) \land (\forall z. \forall p. \text{Product}(z,p,x) \rightarrow p < 100) \]
Universal Quantifiers

De Morgan’s Laws:

\[
\neg (A \land B) = \neg A \lor \neg B \\
\neg (A \lor B) = \neg A \land \neg B \\
\neg \forall x. P(x) = \exists x. \neg P(x) \\
\neg \exists x. P(x) = \forall x. \neg P(x)
\]

\[
\neg (A \to B) = A \land \neg B
\]
Universal Quantifiers

De Morgan’s Laws:

\[
\neg(A \land B) = \neg A \lor \neg B \\
\neg(A \lor B) = \neg A \land \neg B \\
\neg \forall x. P(x) = \exists x. \neg P(x) \\
\neg \exists x. P(x) = \forall x. \neg P(x)
\]

\[q(y) = \exists x. \text{Company}(x,y) \land (\forall z. \forall p. \text{Product}(z,p,x) \rightarrow p < 100)\]

= 

\[q(y) = \exists x. \text{Company}(x,y) \land \neg(\exists z \exists p. \text{Product}(z,p,x) \land p \geq 100)\]
Universal Quantifiers

De Morgan’s Laws:

\[ \neg(A \land B) = \neg A \lor \neg B \]
\[ \neg(A \lor B) = \neg A \land \neg B \]
\[ \neg \forall x. P(x) = \exists x. \neg P(x) \]
\[ \neg \exists x. P(x) = \forall x. \neg P(x) \]

\[ q(y) = \exists x. \text{Company}(x,y) \land (\forall z. \forall p. \text{Product}(z,p,x) \rightarrow p < 100) \]

\[ q(y) = \exists x. \text{Company}(x,y) \land \neg(\exists z \exists p. \text{Product}(z,p,x) \land p \geq 100) \]

\[ \text{theOtherCompanies}(x) = \exists z \exists p. \text{Product}(z,p,x) \land p \geq 100 \]
\[ q(y) = \exists x. \text{Company}(x,y) \land \neg \text{theOtherCompanies}(x) \]
Universal Quantifiers: NOT IN

\[
\text{theOtherCompanies}(x) = \exists z \exists p. \text{Product}(z,p,x) \land p \geq 100
\]
\[
q(y) = \exists x. \text{Company}(x,y) \land \neg \text{theOtherCompanies}(x)
\]

\[
\text{SELECT DISTINCT c.city}
\text{FROM Company c}
\text{WHERE c.cname NOT IN (SELECT p.company}
\text{FROM Product p}
\text{WHERE p.price \geq 100)}
\]
Universal Quantifiers: \( \text{NOT EXISTS} \)

theOtherCompanies(x) = \( \exists z \exists p. \text{Product}(z,p,x) \land p \geq 100 \)

q(y) = \( \exists x. \text{Company}(x,y) \land \neg \text{theOtherCompanies}(x) \)

\[
\begin{align*}
\text{SELECT DISTINCT} & \quad \text{c.city} \\
\text{FROM} & \quad \text{Company } c \\
\text{WHERE} & \quad \text{NOT EXISTS } (\text{SELECT } p.\text{company} \\
& \quad \text{FROM Product } p \\
& \quad \text{WHERE } c.\text{cname} = p.\text{company } \land \text{AND } p.\text{price } \geq 100)
\end{align*}
\]
Universal Quantifiers: **ALL**

```sql
SELECT DISTINCT c.city
FROM Company c
WHERE 100 > ALL (SELECT p.price
FROM Product p
WHERE p.company = c.cname)
```
Question for Database Fans and their Friends

• Can we unnest this query?

Find cities that have a company such that all its products have price < 100
Monotone Queries

- Definition A query $Q$ is **monotone** if:
  - Whenever we add tuples to one or more input tables, the answer to the query will not lose any existing tuples
Monotone Queries

- Definition: A query $Q$ is **monotone** if:
  - Whenever we add tuples to one or more input tables, the answer to the query will not lose any existing tuples.

### Product

<table>
<thead>
<tr>
<th>pname</th>
<th>price</th>
<th>cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>19.99</td>
<td>c001</td>
</tr>
<tr>
<td>Gadget</td>
<td>999.99</td>
<td>c003</td>
</tr>
<tr>
<td>Camera</td>
<td>149.99</td>
<td>c001</td>
</tr>
</tbody>
</table>

### Company

<table>
<thead>
<tr>
<th>cid</th>
<th>cname</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>c001</td>
<td>Sunworks</td>
<td>Bonn</td>
</tr>
<tr>
<td>c002</td>
<td>DB Inc.</td>
<td>Lyon</td>
</tr>
<tr>
<td>c003</td>
<td>Builder</td>
<td>Lodtz</td>
</tr>
</tbody>
</table>

Is the mystery query monotone?
Monotone Queries

**Theorem:** If Q is a SELECT-FROM-WHERE query that does not have subqueries, and no aggregates, then it is monotone.

```
SELECT a_1, a_2, ..., a_k
FROM   R_1 as x_1, R_2 as x_2, ..., R_n as x_n
WHERE  Conditions
```

**Proof.** We use the nested loop semantics: if we insert a tuple in a relation $R_i$, then $x_i$ will take all the old values, in addition to the new value.

```
for x_1 in R_1 do
  for x_2 in R_2 do
    ....
    for x_n in R_n do
      if Conditions
        output (a_1, ..., a_k)
```
Monotone Queries

This query is not monotone:

Find cities that have a company such that all its products have price < 100

Consequence: we cannot write it as a SELECT-FROM-WHERE query without nested subqueries
NULLS in SQL

• Whenever we don’t have a value, we can put a NULL

• Can mean many things:
  – Value does not exists
  – Value exists but is unknown
  – Value not applicable
  – Etc.

• The schema specifies for each attribute if can be null (nullable attribute) or not
Null Values

Person(name, age, height, weight)

INSERT INTO Person VALUES('Joe', 20, NULL, 200)

Rules for computing with NULLs

- If x is NULL then x+7 is still NULL
- If x is 2 then x>5 is FALSE
- If x is NULL then x>5 is UNKNOWN
- If x is 10 then x>5 is TRUE

<table>
<thead>
<tr>
<th>Boolean Value</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>0</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>0.5</td>
</tr>
<tr>
<td>TRUE</td>
<td>1</td>
</tr>
</tbody>
</table>
Null Values

- $C_1 \text{ AND } C_2 = \min(C_1, C_2)$
- $C_1 \text{ OR } C_2 = \max(C_1, C_2)$
- $\neg C_1 = 1 - C_1$

**SELECT** *

**FROM** Person

**WHERE** (age < 25) \text{ AND } (height > 6 \text{ OR } weight > 190)

Rule in SQL: result includes only tuples that yield **TRUE**
Null Values

Unexpected behavior:

```
SELECT *  
FROM Person 
WHERE age < 25 OR age >= 25
```

Some Persons not included!
Null Values

Can test for NULL explicitly:

- IS NULL
- IS NOT NULL

```sql
SELECT * 
FROM   Person 
WHERE  age < 25 OR age >= 25 OR age IS NULL
```

Now all Person are included
Detour into DB Research

Imielinski&Libski, *Incomplete Databases*, 1986

- **Database** = is in one of several states, or *possible worlds*
  - Number of possible worlds is exponential in size of db
- **Query semantics** = return the *certain answers*

Very influential paper:

- Incomplete DBs used in probabilistic databases, *what-if* scenarios, data cleaning, data exchange

In SQL, NULLs are the simplest form of incomplete database:

- **Database** = a NULL takes independently any possible value
- **Query semantics** = not exactly certain answers (why?)
Outerjoins

An “inner join”:

```
SELECT x.name, y.store
FROM Product x, Purchase y
WHERE x.name = y.prodName
```

Same as:

```
SELECT x.name, y.store
FROM Product x JOIN Purchase y ON
    x.name = y.prodName
```

But Products that never sold will be lost
If we want the never-sold products, need a “left outer join”:

```
SELECT x.name, y.store
FROM   Product x LEFT OUTER JOIN Purchase y ON
       x.name = y.prodName
```
Product(name, category)
Purchase(prodName, store)

### Product

<table>
<thead>
<tr>
<th>name</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>gadget</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
</tr>
</tbody>
</table>

### Purchase

<table>
<thead>
<tr>
<th>prodName</th>
<th>store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Wiz</td>
</tr>
<tr>
<td>Camera</td>
<td>Ritz</td>
</tr>
<tr>
<td>Camera</td>
<td>Wiz</td>
</tr>
<tr>
<td>OneClick</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Outer Joins

• **Left outer join:**
  – Include the left tuple even if there’s no match

• **Right outer join:**
  – Include the right tuple even if there’s no match

• **Full outer join:**
  – Include both left and right tuples even if there’s no match
Aggregations

Five basic aggregate operations in SQL

• count
• sum
• avg
• max
• min
COUNT applies to duplicates, unless otherwise stated:

```
SELECT count(product)  
FROM Purchase  
WHERE price>3.99
```

Same as `count(*)`

Except if some product is NULL

We probably want:

```
SELECT count(DISTINCT product)  
FROM Purchase  
WHERE price>3.99
```
Find total quantities for all sales over $1, by product.

```
SELECT product, sum(quantity) AS TotalSales
FROM Purchase
WHERE price > 1
GROUP BY product
```

What is the answer?

<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>
Grouping and Aggregation

1. Compute the **FROM** and **WHERE** clauses.

2. Group by the attributes in the **GROUP BY**

3. Compute the **SELECT** clause: group attrs 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>

```sql
SELECT product, sum(quantity) AS TotalSales
FROM Purchase
WHERE price > 1
GROUP BY product
```
3. SELECT:
Each Group → One Answer

<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>

```
SELECT product, sum(quantity) AS TotalSales
FROM Purchase
WHERE price > 1
GROUP BY product
```
Ordering Results

SELECT product, sum(quantity) as TotalSales
FROM purchase
GROUP BY product
ORDER BY TotalSales DESC
LIMIT 20

SELECT product, sum(quantity) as TotalSales
FROM purchase
GROUP BY product
ORDER BY sum(quantity) DESC
LIMIT 20

Equivalent, but not all systems accept both syntax forms
HAVING Clause

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

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

HAVING clause contains conditions on aggregates.
WHERE vs HAVING

• **WHERE** condition: applied to individual rows
  – Determine which rows contributed to the aggregate
  – All attributes are allowed
  – No aggregates functions allowed

• **HAVING** condition: applied to the entire group
  – Entire group is returned, or not at all
  – Only group attributes allowed
  – Aggregate functions allowed
General form of Grouping and Aggregation

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

Why?

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
Empty Groups Running Example

For the next slides, run this in postgres:

```sql
create table Purchase(pid int primary key, product text, price float, quantity int, month varchar(15));
create table Product (pid int primary key, pname text, manufacturer text);

insert into Purchase values(01, 'bagel', 1.99, 20, 'september');
insert into Purchase values(02, 'bagel', 2.50, 12, 'december');
insert into Purchase values(03, 'banana', 0.99, 9, 'september');
insert into Purchase values(04, 'banana', 1.59, 9, 'february');
insert into Purchase values(05, 'gizmo', 99.99, 5, 'february');
insert into Purchase values(06, 'gizmo', 99.99, 3, 'march');
insert into Purchase values(07, 'gizmo', 49.99, 3, 'april');
insert into Purchase values(08, 'gadget', 89.99, 3, 'january');
insert into Purchase values(09, 'gadget', 89.99, 3, 'february');
insert into Purchase values(10, 'gadget', 49.99, 3, 'march');
insert into Purchase values(11, 'orange', null, 5, 'may');

insert into product values(1, 'bagel', 'Sunshine Co.');
insert into product values(2, 'banana', 'BusyHands');
insert into product values(3, 'gizmo', 'GizmoWorks');
insert into product values(4, 'gadget', 'BusyHands');
insert into product values(5, 'powerGizmo', 'PowerWorks');
```
Empty Group Problem

**Query**: for each manufacturer, compute the total number of purchases for its products

**Problem**: a group can never be empty! In particular, count(*) is never 0

```
SELECT x.manufacturer, count(*)
FROM Product x, Purchase y
WHERE x.pname = y.product
GROUP BY x.manufacturer
```
Solution 1: Outer Join

**Query**: for each manufacturer, compute the total number of purchases for its products

Use a **LEFT OUTER JOIN**.
Make sure you count an attribute that may be NULL

```sql
SELECT x.manufacturer, count(y.product)
FROM Product x LEFT OUTER JOIN Purchase y
ON x.pname = y.product
GROUP BY x.manufacturer
```
Solution 2: Nested Query

**Query**: for each manufacturer, compute the total number of purchases for its products

Use a subquery in the **SELECT** clause

```sql
SELECT DISTINCT x.manufacturer, 
    (SELECT count(*) 
     FROM Product z, Purchase y 
     WHERE x.manufacturer = z.manufacturer 
         AND z.pname = y.product) 
FROM Product x
```

Notice second use of Product. Why?
Finding Witnesses

**Query**: for each manufacturer, find its most expensive product

Finding the maximum price is easy:
Query: for each manufacturer, find its most expensive product

Finding the maximum price is easy:

```
SELECT x.manufacturer, max(y.price)
FROM Product x, Purchase y
WHERE x.pname = y.product
GROUP BY x.manufacturer
```

…but we need to find the product that sold at that price!
Finding Witnesses

**Query**: for each manufacturer, find its most expensive product

Use a subquery in the **FROM** clause:

```
SELECT DISTINCT u.manufacturer, u.pname
FROM Product u, Purchase v,
    (SELECT x.manufacturer, max(y.price) as mprice
     FROM Product x, Purchase y
     WHERE x.pname = y.product
     GROUP BY x.manufacturer) z
WHERE u.pname = v.product
    AND u.manufacturer = z.manufacturer
    AND v.price = z.mprice
```
Finding Witnesses

**Query**: for each manufacturer, find its most expensive product

Using **WITH**:

```
WITH Temp as (SELECT x.manufacturer, max(y.price) as mprice
  FROM Product x, Purchase y
  WHERE x.pname = y.product
  GROUP BY x.manufacturer)
SELECT DISTINCT u.manufacturer, u.pname
FROM Product u, Purchase v, Temp z
WHERE u.pname = v.product
  and u.manufacturer = z.manufacturer
  and v.price = z.mprice
```