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

Product (PName, price, category, manufacturer)

Selections in SQL

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

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

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

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

Projections (and Selections) in SQL

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

```
SELECT   category
FROM     Product
```

Need DISTINCT (why?)

```
SELECT DISTINCT category
FROM     Product
```

“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

Key

<u>CName</u>	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

Product

<u>PName</u>	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

Foreign key

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 a1, a2, …, ak
FROM   R1 AS x1, R2 AS x2, …, Rn AS xn
WHERE  Conditions
```

$\text{SELECT } a_1, a_2, \ldots, a_k$
$\text{FROM } R_1 \text{ AS } x_1, R_2 \text{ AS } x_2, \ldots, R_n \text{ AS } x_n$
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 $\cup$ $\{(a_1,\ldots,a_k)\}$
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
Examples at the end of the lecture

WHERE
Examples on following slides

Avoid writing nested queries when possible;
keep in mind that sometimes it's impossible

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

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');
```

Product (<u>pname</u>, price, company)
Company(<u>cname</u>, city)

Existential Quantifiers

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

Product (<u>pname</u>, price, company)
Company(<u>cname</u>, city)

Existential Quantifiers

Find cities that have a company
that manufacture *<u>some</u>* 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! ☺

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

Universal Quantifiers

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

Universal quantifiers are hard ! ☹

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

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

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

Product (<u>pname</u>, price, company)
Company(<u>cname</u>, city)

Universal Quantifiers

De Morgan's Laws:

$\neg(A \wedge B) = \neg A \vee \neg B$
$\neg(A \vee B) = \neg A \wedge \neg B$
$\neg \forall x.\ P(x) = \exists x.\ \neg P(x)$
$\neg \exists x.\ P(x) = \forall x.\ \neg P(x)$

$\neg(A \rightarrow B) = A \wedge \neg B$

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

Universal Quantifiers

De Morgan's Laws:

$\neg(A \wedge B) = \neg A \vee \neg B$

$\neg(A \vee B) = \neg A \wedge \neg B$

$\neg \forall x.\ P(x) = \exists x.\ \neg P(x)$

$\neg \exists x.\ P(x) = \forall x.\ \neg P(x)$

$\neg(A \rightarrow B) = A \wedge \neg B$

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

=

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

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

Universal Quantifiers

De Morgan's Laws:

$\neg(A \land B) = \neg A \lor \neg B$

$\neg(A \lor B) = \neg A \land \neg B$

$\neg \forall x.\ P(x) = \exists x.\ \neg P(x)$

$\neg \exists x.\ P(x) = \forall x.\ \neg P(x)$

$\neg(A \rightarrow B) = A \land \neg B$

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

=

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

=

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

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

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

Universal Quantifiers: NOT IN

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

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

```
SELECT DISTINCT  c.city
FROM      Company c
WHERE   c.cname NOT IN (SELECT p.company
                        FROM Product p
                        WHERE p.price >= 100)
```

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

Universal Quantifiers: NOT EXISTS

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

```
SELECT DISTINCT  c.city
FROM     Company c
WHERE  NOT EXISTS (SELECT p.company
                   FROM Product p
                   WHERE c.cname = p.company AND p.price >= 100)
```

Correlated subquery!

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

Universal Quantifiers: ALL

```
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

Product (pname, price, cid)

Company(cid, cname, city)

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

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

pname	price	cid
Gizmo	19.99	c001
Gadget	999.99	c003
Camera	149.99	c001

Company

cid	cname	city
c001	Sunworks	Bonn
c002	DB Inc.	Lyon
c003	Builder	Lodtz

Q

A	B
149.99	Lodtz
19.99	Lyon

Is the mystery query monotone?

Product

pname	price	cid
Gizmo	19.99	c001
Gadget	999.99	c003
Camera	149.99	c001
iPad	499.99	c001

Company

cid	cname	city
c001	Sunworks	Bonn
c002	DB Inc.	Lyon
c003	Builder	Lodtz

Q

A	B
149.99	Lyon
19.99	Lyon
19.99	Bonn
149.99	Bonn

Monotone Queries

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

```
SELECT a1, a2, …, ak
FROM   R1 as x1, R2 as x2, …, Rn as xn
WHERE  Conditions
```

SELECT $a_1, a_2, \ldots, a_k$
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 $(a_1,\ldots,a_k)$

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

Monotone Queries

This query is not monotone:

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

pname	price	cid
Gizmo	19.99	c001

cid	cname	city
c001	Sunworks	Bonn

⇒

cname
Sunworks

pname	price	cid
Gizmo	19.99	c001
Gadget	999.99	c001

cid	cname	city
c001	Sunworks	Bonn

⇒

cname

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)

height unknown

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

FALSE	=	0
UNKNOWN	=	0.5
TRUE	=	1

Null Values

- C1 AND C2 = min(C1, C2)
- C1 OR C2 = max(C1, C2)
- NOT C1 = 1 – C1

```
SELECT *
FROM Person
WHERE  (age < 25) AND
          (height > 6 OR weight > 190)
```

E.g.
age=20
height=NULL
weight=200

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:

x IS NULL

x IS NOT NULL

```
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?)

Product(name, category)
Purchase(prodName, store)

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

Product(name, category)
Purchase(prodName, store)

Outerjoins

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(<u>name</u>, category)
Purchase(prodName, store)

Product

<u>name</u>	category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

prodName	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

name	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz
OneClick	NULL

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

Purchase(product, price, quantity)

Counting Duplicates

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
```

Purchase(product, price, quantity)

Grouping and Aggregation

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

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

product	price	quantity
Bagel	3	20
Bagel	1.50	20
Banana	0.5	50
Banana	2	10
Banana	4	10

What is the answer?

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

Product	Price	Quantity
Bagel	3	20
Bagel	1.50	20
Banana	~~0.5~~	~~50~~
Banana	2	10
Banana	4	10

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

3. SELECT: Each Group → One Answer

Product	Price	Quantity
Bagel	3	20
Bagel	1.50	20
Banana	~~0.5~~	~~50~~
Banana	2	10
Banana	4	10

Product	TotalSales
Bagel	40
Banana	20

```
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 al all
  - Only group attributes allowed
  - Aggregate functions allowed

General form of Grouping and Aggregation

```
SELECT     S
FROM       R1,…,Rn
WHERE      C1
GROUP BY   a1,…,ak
HAVING     C2
```

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

```
SELECT     S
FROM       R1,…,Rn
WHERE      C1
GROUP BY   a1,…,ak
HAVING     C2
```

Evaluation steps:

1. Evaluate FROM-WHERE using Nested Loop Semantics
2. Group by the attributes $a_1,\ldots,a_k$
3. Apply condition C2 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:

```
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');
```

Purchase(product, price, quantity)

Product(pname, manufacturer)

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
```

Purchase(product, price, quantity)

Product(pname, 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

```
SELECT x.manufacturer, count(y.product)
FROM Product x LEFT OUTER JOIN Purchase y
ON x.pname = y.product
GROUP BY x.manufacturer
```

Purchase(product, price, quantity)
Product(pname, 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

Notice second use of Product. Why?

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

Purchase(product, price, quantity)

Product(pname, manufacturer)

Finding Witnesses

Query: for each manufacturer, find its most expensive product

Finding the maximum price is easy:

Purchase(product, price, quantity)

Product(<u>pname</u>, manufacturer)

Finding Witnesses

<u>Query</u>: 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!

Purchase(product, price, quantity)

Product(<u>pname</u>, manufacturer)

Finding Witnesses

<u>Query</u>: 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
```

Purchase(product, price, quantity)

Product(pname, manufacturer)

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
```