CSE544
SQL
Wednesday, March 31, 2004

Administrivia
- Sign up for the 544 mailing list!
- Assignment 1 is released. The deadline for first part is 7th April.

SQL Introduction
Standard language for querying and manipulating data
Structured Query Language

Many standards out there:
- ANSI SQL
- SQL92 (a.k.a. SQL2)
- SQL99 (a.k.a. SQL3)
- Vendors support various subsets of these
- What we discuss is common to all of them

SQL
- Data Definition Language (DDL)
  - Create/alter/delete tables and their attributes
  - Following lectures...
- Data Manipulation Language (DML)
  - Query one or more tables - discussed next!
  - Insert/delete/modify tuples in tables
- Transact-SQL
  - Idea: package a sequence of SQL statements
  - Won’t discuss in class

Data in SQL
1. Atomic types, a.k.a. data types
2. Tables built from atomic types

Data Types in SQL
- Characters:
  - CHAR(20) -- fixed length
  - VARCHAR(40) -- variable length
- Numbers:
  - BIGINT, INT, SMALLINT, TINYINT
  - REAL, FLOAT -- differ in precision
  - MONEY
- Times and dates:
  - DATE
  - DATETIME -- SQL Server
- Others... All are simple
Tables in SQL

<table>
<thead>
<tr>
<th>Product</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>

Tables Explained

- A tuple = a record
  - Restriction: all attributes are of atomic type
- A table = a set of tuples
  - Like a list
  - ...but it is unordered: no first(), no next(), no last().
- No nested tables, only flat tables are allowed!
  - We will see later how to decompose complex structures into multiple flat tables.

Attributes names

- Product
- Price
- Category
- Manufacturer

Tables Explained

- The schema of a table is the table name and its attributes:
  - Product(PName, Price, Category, Manufacturer)
- A key is an attribute whose values are unique; we underline a key
  - Product(PName, Price, Category, Manufacturer)

SQL Query

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

```
SELECT attributes
FROM relations (possibly multiple, joined)
WHERE conditions (selections)
```

Simple SQL Query

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

```
SELECT PName, Price, Manufacturer
FROM Product
WHERE Price > 100
```
A Notation for SQL Queries

SELECT Name, Price, Manufacturer
FROM Product
WHERE Price > 100

Output Schema

Name, Price, Manufacturer

Product(Price, Category, Manufacturer)

Select*

FROM Product
WHERE PName LIKE '%gizmo%'

The LIKE operator

- s LIKE p: pattern matching on strings
- p may contain two special symbols:
  - % = any sequence of characters
  - _ = any single character

Find all products whose name mentions 'gizmo':

Eliminating Duplicates

SELECT DISTINCT category
FROM Product

Output Schema

Category
Gadgets
Photography
Household

Compare to:

SELECT category
FROM Product

Output Schema

Category
Gadgets
Photography
Household

Ordering the Results

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

Output Schema

Name, Price, Category, Manufacturer

Ordering is ascending, unless you specify the DESC keyword.

Ties are broken by the second attribute on the ORDER BY list, etc.
**Ordering the Results**

```
SELECT DISTINCT category
FROM Product
ORDER BY category
```

**Joins in SQL**

- Connect two or more tables:

```
Product (PName, Price, Category, Manufacturer)
Company (CName, StockPrice, Country)
```

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

```
SELECT PName, Price
FROM Product
JOIN Company
ON Manufacturer=CName AND Country='Japan'
AND Price <= 200
```

**Joins**

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

Find all countries that manufacture some product in the ‘Gadgets’ category.

```
SELECT Country
FROM Product
JOIN Company
ON Manufacturer=CName AND Category='Gadgets'
WHERE Product.CName
```

**Joins in SQL**

```
SELECT Country, StockPrice
FROM Product
JOIN Company
ON Manufacturer=CName AND Category='Gadgets'
WHERE Product.CName
```
**Joins**

```sql
SELECT DISTINCT person.name, company.address
FROM person, company
WHERE person.works for = company.name
```

Find names of people living in Seattle that bought some product in the 'Gadgets' category, and the names of the stores they bought such product from.

**Disambiguating Attributes**

- Sometimes two relations have the same attr:
- Person(pname, address, worksfor)
- Company(cname, address)

```sql
SELECT DISTINCT person.pname, company.address
FROM person, company
WHERE worksfor = company.name
```

**Tuple Variables in SQL**

```sql
SELECT DISTINCT x.name
FROM purchase AS x, purchase AS y
WHERE x.product = y.product AND y.store = 'BestBuy'
```

Find all stores that sold at least one product that was sold at 'BestBuy':

**Tuple Variables**

General rule:
tuple variables introduced automatically by the system:

```sql
SELECT name
FROM product
WHERE price > 100
```

Becomes:

```sql
SELECT product.name
FROM product AS product
WHERE product.price > 100
```

Doesn’t work when Product occurs more than once:
In that case the user needs to define variables explicitly.

**Meaning (Semantics) of SQL Queries**

1. Nested loops:

```java
Answer = {}
for x1 in R1 do
    for x2 in R2 do
        ...
        for xn in Rn do
            if Conditions then Answer = Answer \cup \{(a1, ..., ak)\}
    return Answer
```

2. Parallel assignment

```java
Answer = {}
for all assignments x1 in R1, ..., xn in Rn do
    if Conditions then Answer = Answer \cup \{(a1, ..., ak)\}
return Answer
```
First Unintuitive SQLism

SELECT DISTINCT R.A
FROM R, S, T
WHERE R.A=S.A OR R.A=T.A

Looking for $R \cap (S \cup T)$

But what happens if $T$ is empty?

Renaming Columns

<table>
<thead>
<tr>
<th>Product</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Game</td>
<td>$19.99</td>
<td>Gadgets</td>
<td>GameWorks</td>
</tr>
<tr>
<td>Powergamer</td>
<td>$29.99</td>
<td>Gadgets</td>
<td>GameWorks</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$249.99</td>
<td>Photography</td>
<td>Canon</td>
</tr>
</tbody>
</table>

SELECT P.name AS prodName, Price AS askPrice
FROM Product
WHERE Price > 100

Query with renaming

Conserving Duplicates

(SELECT name
FROM Person
WHERE City="Seattle")
UNION
(SELECT name
FROM Person, Purchase
WHERE buyer=name AND store="The Bon")

Similarly, you can use INTERSECT and EXCEPT.
You must have the same attribute names (otherwise: rename).

Subqueries

A subquery producing a single value:

```
SELECT Purchase.product
FROM Purchase, Person
WHERE buyer =
  (SELECT name
   FROM Person
   WHERE ssn = '123456789');
```

In this case, the subquery returns one value.
If it returns more, it's a run-time error.

Can say the same thing without a subquery:

```
SELECT Purchase.product
FROM Purchase, Person
WHERE buyer = name AND ssn = '123456789'
```

This is equivalent to the previous one when the ssn is a key and '123456789' exists in the database;
otherwise they are different.
Subqueries Returning Relations

Find companies that manufacture products bought by Joe Blow.

```
SELECT Company.name
FROM Company, Product
WHERE Company.name = Product.maker
AND Product.name IN
(SELECT Purchase.product
FROM Purchase
WHERE Purchase.buyer = 'Joe Blow');
```

Here the subquery returns a set of values: no more runtime errors.

---

Subqueries Returning Relations

Equivalent to:

```
SELECT Company.name
FROM Company, Product, Purchase
WHERE Company.name = Product.maker
AND Product.name = Purchase.product
AND Purchase.buyer = 'Joe Blow';
```

Is this query equivalent to the previous one?

- Beware of duplicates!

---

Removing Duplicates

```
SELECT DISTINCT Company.name
FROM Company, Product
WHERE Company.name = Product.maker
AND Product.name IN
(SELECT Purchase.product
FROM Purchase
WHERE Purchase.buyer = 'Joe Blow');
```

```
SELECT DISTINCT Company.name
FROM Company, Product
WHERE Company.name = Product.maker
AND Product.name = Purchase.product
AND Purchase.buyer = 'Joe Blow';
```

Now they are equivalent.

---

Subqueries Returning Relations

You can also use: \( s > \text{ALL} \; R \)
\( s > \text{ANY} \; R \)
\( \text{EXISTS} \; R \)

Find products that are more expensive than all those produced
By "Gizmo-Works"

```
SELECT name
FROM Product
WHERE price > \text{ALL} (SELECT price
FROM Purchase
WHERE maker = 'Gizmo-Works');
```

---

Question for Database Fans and their Friends

- Can we express this query as a single \text{SELECT-FROM-WHERE} query, without subqueries?
- Hint: show that all SFW queries are \text{monotone} (figure out what this means). A query with \text{ALL} is not monotone

---

Correlated Queries

```
SELECT DISTINCT title
FROM Movie AS s
WHERE year <> \text{ANY}
(SELECT year
FROM Movie
WHERE title = s.title);
```

Note (1) scope of variables (2) this can still be expressed as single SFW
Complex Correlated Query

Product (pname, price, category, maker, year)
- Find products (and their manufacturers) that are more expensive than all products made by the same manufacturer before 1972

```
SELECT DISTINCT pname, maker
FROM Product AS x
WHERE price > ALL (SELECT price
FROM Product AS y
WHERE x.maker = y.maker AND y.year < 1972);
```

Powerful, but much harder to optimize!

Existential/Universal Conditions

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

Find all companies s.t. some of their products have price < 100

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

Existential: easy! J

Existential/Universal Conditions

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

Find all companies s.t. all of their products have price < 100

```
SELECT DISTINCT Company.cname
FROM Company
WHERE Company.cname IN (SELECT Product.company
FROM Product
WHERE Product.price >= 100)
```

Universal: hard! L

Existential/Universal Conditions

1. Find the other companies: i.e. s.t. some product ≥ 100

```
SELECT DISTINCT Company.cname
FROM Company
WHERE Company.cname IN (SELECT Product.company
FROM Product
WHERE Product.price >= 100)
```

2. Find all companies s.t. all their products have price < 100

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

Aggregation

```
SELECT Avg(price)
FROM Product
WHERE maker="Toyota"
```

SQL supports several aggregation operations:
SUM, MIN, MAX, AVG, COUNT
**Aggregation: Count**

```
SELECT Count(*)
FROM Product
WHERE year > 1995
```

Except COUNT, all aggregations apply to a single attribute.

**Aggregation: Count**

COUNT applies to duplicates, unless otherwise stated:

```
SELECT Count(category)
FROM Product
WHERE year > 1995
```

Better:

```
SELECT Count(DISTINCT category)
FROM Product
WHERE year > 1995
```

---

**Simple Aggregation**

Purchase(product, date, price, quantity)

Example 1: find total sales for the entire database

```
SELECT Sum(price * quantity)
FROM Purchase
```

Example 1': find total sales of bagels

```
SELECT Sum(price * quantity)
FROM Purchase
WHERE product = 'bagel'
```

---

**Simple Aggregations**

Purchase

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>0.85</td>
<td>15</td>
</tr>
<tr>
<td>Banana</td>
<td>10/22</td>
<td>0.52</td>
<td>7</td>
</tr>
<tr>
<td>Banana</td>
<td>10/19</td>
<td>0.52</td>
<td>17</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/20</td>
<td>0.85</td>
<td>20</td>
</tr>
</tbody>
</table>

---

**Grouping and Aggregation**

Usually, we want aggregations on certain parts of the relation.

Purchase(product, date, price, quantity)

Example 2: find total sales after 9/1 per product.

```
SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '9/1'
GROUPBY product
```

Let’s see what this means...

**Grouping and Aggregation**

1. Compute the FROM and WHERE clauses.
2. Group by the attributes in the GROUPBY clause.
3. Select one tuple for every group (and apply aggregation)

**SELECT** can have (1) grouped attributes or (2) aggregates.
First compute the **FROM-WHERE** clauses (date > "9/1") then **GROUP BY** product:

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banana</td>
<td>10/19</td>
<td>0.52</td>
<td>17</td>
</tr>
<tr>
<td>Banana</td>
<td>10/22</td>
<td>0.52</td>
<td>7</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/20</td>
<td>0.85</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>0.85</td>
<td>15</td>
</tr>
</tbody>
</table>

Then, aggregate

<table>
<thead>
<tr>
<th>Product</th>
<th>TotalSales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>$29.75</td>
</tr>
<tr>
<td>Banana</td>
<td>$12.48</td>
</tr>
</tbody>
</table>

**GROUP BY v.s. Nested Queries**

```
SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > "9/1"
GROUP BY product
```

```
SELECT DISTINCT x.product, (SELECT Sum(y.price*y.quantity)
FROM Purchase y
WHERE x.product = y.product
AND y.date > '9/1')
AS TotalSales
FROM Purchase x
WHERE x.date > "9/1"
```

**Another Example**

<table>
<thead>
<tr>
<th>Product</th>
<th>SumSales</th>
<th>MaxQuantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banana</td>
<td>$12.48</td>
<td>17</td>
</tr>
<tr>
<td>Bagel</td>
<td>$29.75</td>
<td>20</td>
</tr>
</tbody>
</table>

For every product, what is the total sales and max quantity sold?

```
SELECT product, Sum(price*quantity) AS SumSales
FROM Purchase
WHERE date > "9/1"
GROUP BY product
HAVING Sum(quantity) > 30
```

**GENERAL FORM OF GROUPING AND AGGREGATION**

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

S = may contain some of group-by attributes a1,..,ak
and/or any aggregates but NO OTHER ATTRIBUTES
C1 = is any condition on the attributes in R1,..,Rn
C2 = is any condition on aggregate expressions

**HAVING Clause**

Same query, except that we consider only products that had at least 100 buyers.

```
SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > "9/1"
GROUP BY product
HAVING Sum(quantity) > 30
```

HAVING clause contains conditions on aggregates.
General form of Grouping and Aggregation

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

Evaluation steps:
1. Compute the FROM-WHERE part, obtain a table with all attributes in \(R_1, \ldots, R_n\)
2. Group by the attributes \(a_1, \ldots, a_k\)
3. Compute the aggregates in \(C_2\) and keep only groups satisfying \(C_2\)
4. Compute aggregates in \(S\) and return the result

Examples of Queries with Aggregation

Web pages, and their authors:

\[
\begin{align*}
\text{Author(login, name)} \\
\text{Document(url, title)} \\
\text{Wrote(login, url)} \\
\text{Mentions(url, word)}
\end{align*}
\]

- Find all authors who wrote at least 10 documents
  \(\text{Author(login, name)}\)
  \(\text{Wrote(login, url)}\)
- Attempt 1: with nested queries
  \[
  \text{SELECT DISTINCT Author.name} \\
  \text{FROM Author} \\
  \text{WHERE count(SELECT Wrote.url} \\
  \text{FROM Wrote} \\
  \text{WHERE Author.login=Wrote.login) } \\
  > 10
  \]

- Find all authors who wrote at least 10 documents:
  - Attempt 2: SQL style (with GROUP BY)
  \[
  \text{SELECT Author.name} \\
  \text{FROM Author, Wrote} \\
  \text{WHERE Author.login=Wrote.login} \\
  \text{GROUP BY Author.login, Author.name} \\
  \text{HAVING count(wrote.url) > 10}
  \]
  No need for DISTINCT: automatically from GROUP BY

- Find all authors who have a vocabulary over 10000 words:
  \[
  \text{SELECT Author.name} \\
  \text{FROM Author, Wrote, Mentions} \\
  \text{WHERE Author.login=Wrote.login AND Wrote.url=Mentions.url} \\
  \text{GROUP BY Author.name} \\
  \text{HAVING count(distinct Mentions.word) > 10000}
  \]
  Look carefully at the last two queries: you may be tempted to write them as a nested queries, but in SQL we write them best with GROUP BY

NULLS in SQL

- Whenever we don't have a value, we can put a NULL
- Can mean many things:
  - Value does not exist
  - Value exists but is unknown
  - Value not applicable
  - Etc.
- The schema specifies for each attribute if can be null (nullable attribute) or not
- How does SQL cope with tables that have NULLs?
Null Values

- If \( x = \text{NULL} \) then \( 4\cdot(3-x)/7 \) is NULL
- If \( x = \text{NULL} \) then \( x = "\text{Joe}" \) is UNKNOWN
- In SQL there are three boolean values:
  - FALSE = 0
  - UNKNOWN = 0.5
  - TRUE = 1

Null Values

Unexpected behavior:

\[
\text{SELECT *}
\text{FROM Person}
\text{WHERE age < 25 OR age >= 25}
\]

Some Persons are not included!

Null Values

Can test for NULL explicitly:

\[
\text{SELECT *}
\text{FROM Person}
\text{WHERE age < 25 OR age >= 25 OR age IS NULL}
\]

Now it includes all Persons

Outerjoins

\[
\text{SELECT product.name, purchase.store}
\text{FROM product, purchase}
\text{WHERE product.name = purchase.prodName}
\]

But products that never sold will be lost!

Null Values

- \( C_1 \text{ AND } C_2 = \min(C_1, C_2) \)
- \( C_1 \text{ OR } C_2 = \max(C_1, C_2) \)
- \( \text{NOT } C_1 = 1 - C_1 \)

E.g.

\[
\text{SELECT *}
\text{FROM Person}
\text{WHERE (age < 25) AND (height > 6 OR weight > 190)}
\text{AND (age=20 AND height=NULL AND weight=200)}
\]

E.g.

\[
\text{SELECT *}
\text{FROM Person}
\text{WHERE age < 25 OR age >= 25 OR age IS NULL}
\]

Left outer joins in SQL:

\[
\text{SELECT product.name, purchase.store}
\text{FROM product LEFT OUTER JOIN purchase ON}
\text{product.name = purchase.prodName}
\]
Modifying the Database
Three kinds of modifications
- Insertions
- Deletions
- Updates

Sometimes they are all called “updates”

Insertions

\[
\text{INSERT INTO PRODUCT(name)} \\
\text{SELECT DISTINCT Purchase.product} \\
\text{FROM Purchase} \\
\text{WHERE Purchase.date > “10/26/01”}
\]

The query replaces the VALUES keyword. Here we insert many tuples into PRODUCT

Insertions

\[
\text{INSERT INTO R(A1, ..., An) VALUES (v1, ..., vn)}
\]

Example: Insert a new purchase to the database:

\[
\text{INSERT INTO Purchase(buyer, seller, product, store)} \\
\]

Missing attribute \(\rightarrow\) NULL.
May drop attribute names if give them in order.

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 the both left and right tuples even if there’s no match

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>

Insertions

Suppose database got corrupted and we need to fix it:

<table>
<thead>
<tr>
<th>Product</th>
<th>Purchase</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>listPrice</td>
</tr>
<tr>
<td>camera</td>
<td>John</td>
</tr>
<tr>
<td>camera</td>
<td>Smith</td>
</tr>
<tr>
<td>camera</td>
<td>Smith</td>
</tr>
</tbody>
</table>

Task: insert in Product all prodNames from Purchase
**Insertion: an Example**

- **Insert into Product**

```sql
INSERT INTO Product(name)
SELECT DISTINCT prodName
FROM Purchase
WHERE prodName NOT IN (SELECT name FROM Product)
```

- **Example**

```
<table>
<thead>
<tr>
<th>name</th>
<th>listPrice</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>gismo</td>
<td>100</td>
<td>Gadgets</td>
</tr>
<tr>
<td>camera</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>camera ??</td>
<td>225</td>
<td>-</td>
</tr>
</tbody>
</table>
```

---

**Deletions**

- **Example**

```
DELETE FROM Purchase
WHERE seller = 'Joe' AND product = 'Brooklyn Bridge'
```

Factoid about SQL: there is no way to delete only a single occurrence of a tuple that appears twice in a relation.

---

**Updates**

- **Example**

```
UPDATE PRODUCT
SET price = price/2
WHERE Product.name IN
(SELECT product
FROM Purchase
WHERE Date = 'Oct, 25, 1999');
```