Introduction to Database Systems
CSE 444

Lecture 04: SQL
April 7, 2008

Outline

• The Project
• Nulls (6.1.6)
• Outer joins (6.3.8)
• Database Modifications (6.5)

The Project

• Application:
  – Boutique online music and book store
• Project:
  – Create database, access through a Web interface
  – Import real data and develop inventory logic
  – Customer checkout
  – Advanced functionality (TBD)

The Project

• Team:
  – Two people
  – Find partner now!
• Tools:
  – SQL Server 2005
  – Visual Studio 2005
  – C# 2.0
  – ASP.NET 2.0
### The Project

Phase 1: posted now, due April 18
- Create a schema
- Populate the database: fake data for now
- Access through a simple Web interface

### 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= NULL then 4*(3-x)/7 is still NULL
- If x= NULL then x="Joe" is UNKNOWN
- In SQL there are three boolean values:
  
<table>
<thead>
<tr>
<th>Boolean Value</th>
<th>Value</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>

### Rule in SQL: include only tuples that yield TRUE

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

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

Rule in SQL: include only tuples that yield TRUE
Null Values

Unexpected behavior:

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

Some Persons are not included!

Null Values

Can test for NULL explicitly:

- x IS NULL
- x IS NOT NULL

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

Now it includes all Persons

Outerjoins

Explicit joins in SQL = “inner joins”:

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

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

Same as:

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

But Products that never sold will be lost!

Outerjoins

Left outer joins in SQL:

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

```sql
SELECT Product.name, Purchase.store
FROM Product LEFT OUTER JOIN Purchase ON
Product.name = Purchase.prodName
```

```sql
SELECT Product.name, Purchase.store
FROM Product, Purchase
WHERE Product.name = Purchase.prodName
```
Application

Compute, for each product, the total number of sales in ‘September’
Product(name, category)
Purchase(prodName, month, store)

SELECT Product.name, count(*)
FROM Product, Purchase
WHERE Product.name = Purchase.prodName
and Purchase.month = ‘September’
GROUP BY Product.name

What’s wrong?

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

Now we also get the products who sold in 0 quantity
Modifying the Database

Three kinds of modifications
• Insertions
• Deletions
• Updates

Sometimes they are all called “updates”

Insertions

General form:

\[
\text{INSERT INTO } R(A_1, \ldots, A_n) \ \text{VALUES} \ (v_1, \ldots, v_n)
\]

Example: Insert a new purchase to the database:

\[
\text{INSERT INTO } \text{Purchase(buyer, seller, product, store)} \\
\text{VALUES } (\text{‘Joe’, ‘Fred’, ‘wakeup-clock-espresso-machine’, ‘The Sharper Image’})
\]

Missing attribute → NULL.
May drop attribute names if give them in order.

Insertion: an Example

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

The query replaces the VALUES keyword.
Here we insert \textit{many} tuples into \textit{PRODUCT}

Product got corrupted and we need to fix it:

<table>
<thead>
<tr>
<th>prodName</th>
<th>buyerName</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>camera</td>
<td>John</td>
<td>200</td>
</tr>
<tr>
<td>gizmo</td>
<td>Smith</td>
<td>80</td>
</tr>
<tr>
<td>camera</td>
<td>Smith</td>
<td>225</td>
</tr>
</tbody>
</table>

Task: Insert in \textit{Product} all \textit{prodNames} from \textit{Purchase}
**Insertion: an Example**

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

<table>
<thead>
<tr>
<th>name</th>
<th>listPrice</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>gizmo</td>
<td>100</td>
<td>Gadgets</td>
</tr>
<tr>
<td>camera</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Insertion: an Example**

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

<table>
<thead>
<tr>
<th>name</th>
<th>listPrice</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>gizmo</td>
<td>100</td>
<td>Gadgets</td>
</tr>
<tr>
<td>camera</td>
<td>200</td>
<td>-</td>
</tr>
<tr>
<td>camera ??</td>
<td>225 ??</td>
<td>-</td>
</tr>
</tbody>
</table>

---

**Deletions**

Example:

```sql
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:

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

So far we have seen the Data Manipulation Language, DML.
Next: Data Definition Language (DDL)

Data types:
- Defines the types.

Data definition: defining the schema.
- Create tables
- Delete tables
- Modify table schema

Indexes: to improve performance

---

Creating Tables

**CREATE TABLE** Person(

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>VARCHAR(30),</td>
</tr>
<tr>
<td>social-security-number</td>
<td>INT,</td>
</tr>
<tr>
<td>age</td>
<td>SHORTINT,</td>
</tr>
<tr>
<td>city</td>
<td>VARCHAR(30),</td>
</tr>
<tr>
<td>gender</td>
<td>VARCHAR(30),</td>
</tr>
<tr>
<td>Birthdate</td>
<td>DATE</td>
</tr>
</tbody>
</table>

);

---

Deleting or Modifying a Table

**Deleting:**
- Example: `DROP Person;` Exercise with care !!

**Altering:** (adding or removing an attribute).
- Example: `ALTER TABLE Person
  ADD phone CHAR(16);`
  - `ALTER TABLE Person
  DROP age;`

What happens when you make changes to the schema?

---

Default Values

Specifying default values:

**CREATE TABLE** Person(

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>VARCHAR(30),</td>
</tr>
<tr>
<td>social-security-number</td>
<td>INT,</td>
</tr>
<tr>
<td>age</td>
<td>SHORTINT DEFAULT 100,</td>
</tr>
<tr>
<td>city</td>
<td>VARCHAR(30) DEFAULT ‘Seattle’,</td>
</tr>
<tr>
<td>gender</td>
<td>CHAR(1) DEFAULT ‘?’,</td>
</tr>
<tr>
<td>Birthdate</td>
<td>DATE</td>
</tr>
</tbody>
</table>

The default of defaults: NULL
Indexes

Really important to speed up query processing time.

Suppose we have a relation

\[ \text{Person (name, age, city)} \]

\[
\text{SELECT * FROM Person WHERE name = "Smith"}
\]

Sequential scan of the file Person may take long

Creating Indexes

Syntax:

\[
\text{CREATE INDEX nameIndex ON Person(name)}
\]

Indexes can be useful in range queries too:

\[
\text{CREATE INDEX ageIndex ON Person (age)}
\]

B+ trees help in:

\[
\text{SELECT * FROM Person WHERE age > 25 AND age < 28}
\]

Why not create indexes on everything?
Creating Indexes

Indexes can be created on more than one attribute:

**Example:**
```
CREATE INDEX doubleindex ON Person (age, city)
```

**Helps in:**
```
SELECT * FROM Person WHERE age = 55 AND city = "Seattle"
```

**and even in:**
```
SELECT * FROM Person WHERE age = 55
```

**But not in:**
```
SELECT * FROM Person WHERE city = "Seattle"
```

The Index Selection Problem

- Why not build an index on every attribute? On every pair of attributes? Etc.?

- The index selection problem is hard: balance the query cost v.s. the update cost, in a large application workload