Introduction to Database Systems  
CSE 444

Lecture 04: SQL

October 3, 2007

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

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>Code</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

Can test for NULL explicitly:

- x IS NULL
- x IS NOT NULL

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

Rule in SQL: include only tuples that yield TRUE

Some Persons are not included!

Null Values

Explicit joins in SQL = “inner joins”:
Product(name, category)
Purchase(prodName, store)

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

Left outer joins in SQL:

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

But Products that never sold will be lost!
Application

Compute, for each product, the total number of sales in ‘September’

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

```sql
SELECT Product.name, count(*)
FROM Product, Purchase
WHERE Product.name = Purchase.prodName
  and Purchase.month = 'September'
GROUP BY Product.name
```

Now we also get the products who sold in 0 quantity

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

Modifying the Database

Three kinds of modifications
- **Insertions**
- **Deletions**
- **Updates**

Sometimes they are all called “updates”

Insertions

General form:

```sql
INSERT INTO R(A1, ..., An) VALUES (v1, ..., vn)
```

Example: Insert a new purchase to the database:

```sql
INSERT INTO Purchase(buyer, seller, product, store)
VALUES ('Joe', 'Fred', 'wakeup-clock-espresso-machine', 'The Sharper Image')
```

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

```sql
INSERT INTO PRODUCT(name)
SELECT DISTINCT Purchase.product
FROM Purchase
WHERE Purchase.date > "10/26/01"
```

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

Insertion: an Example

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

Task: insert in Product all prodNames from Purchase

Insertion: an Example

<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>
</tbody>
</table>
| camera| ?? 225    | ?? -     | Depends on the implementation

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(
    name VARCHAR(30),
    social-security-number INT,
    age SHORTINT,
    city VARCHAR(30),
    gender BIT(1),
    Birthdate DATE
);
```

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(
    name VARCHAR(30),
    social-security-number INT,
    age SHORTINT DEFAULT 100,
    city VARCHAR(30) DEFAULT 'Seattle',
    gender CHAR(1) DEFAULT '?',
    Birthdate DATE
);
```

The default of defaults: NULL

Indexes

REALLY important to speed up query processing time.

Suppose we have a relation

```
Person (name, age, city)
```

```
SELECT * FROM Person WHERE name = "Smith"
```

Sequential scan of the file Person may take long

Indexes

- Create an index on name:

```
INDEX ON Person (name)
```

B+ trees have fan-out of 100s: max 4 levels!
Will discuss in the second half of this course
Creating Indexes

Syntax:

```
CREATE INDEX nameIndex ON Person(name)
```

Indexes can be useful in range queries too:

```
CREATE INDEX ageIndex ON Person (age)
```

B+ trees help in:

```
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