Announcements

- Homework 1
  - Everybody should have accounts by now
- Project on the Website
  - Please check website, there is work for you!
- Course outline updated on the Website
  - Reading assignments (more to come)
- SQL
  - Lots of materials last lecture and this one!
  - Make sure you understand it

Two Tough Examples

Store(sid, sname)
Product(pid, pname, price, sid)

Find all stores that sell only products with price > 100
(Equivalent formulation: find all stores s.t. all their products have price > 100)

Two Tough Examples

Two Tough Examples

For each store, find the Product ID of its most expensive product

Better:

SELECT Store.name, max(Product.price)
FROM Store, Product
WHERE Store.sid = Product.sid
GROUP BY Store.sid

But may return multiple pids:

SELECT Store.name, x.pid
FROM Store, Product x
WHERE Store.sid = x.sid and
  x.price >=
ALL (SELECT y.price
     FROM Product y
     WHERE Store.sid = y.sid)

Your pick...

SELECT Store.name
FROM Store
WHERE 100 < ALL (SELECT Product.price
                   FROM product
                   WHERE Store.sid = Product.sid)
Two Tough Examples

Finally, choose some pid arbitrarily, if there are many with highest price:

```sql
SELECT Store.name, max(x.pid)
FROM Store, Product x
WHERE Store.sid = x.sid and x.price >=
    ALL (SELECT y.price
         FROM Product y
         WHERE Store.sid = y.sid)
GROUP BY Store.name
```

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?

Nulls

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

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:
  - 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

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

E.g.: age=20
weight=NULL

Some Persons are 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 it includes all Persons

Outerjoins

Explicit joins in SQL:
Product(name, category)
Purchase(prodName, store)

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

Same as:

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

But Product.name = Purchase.name

Outerjoins

Left outer joins in SQL:
Product(name, category)
Purchase(prodName, store)

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

Product
<table>
<thead>
<tr>
<th>Name</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gismo</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>Gismo</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 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:

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

Example: Insert a new purchase to the database:

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

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

Insertion: an Example

Product(name, listPrice, category)
Purchase(prodName, buyerName, price)

prodName is foreign key in Product.name

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>200</td>
</tr>
<tr>
<td>phone</td>
<td>100</td>
</tr>
</tbody>
</table>

Task: insert in Product all prodNames from Purchase

Deletions

Example:

\[
\text{DELETE FROM PURCHASE}
\text{WHERE} \ \text{seller} = \text{'Joe'} \ \text{AND product} = \text{'Brooklyn Bridge'}
\]

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');
```

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

Data Types in SQL

- Characters:
  - CHAR(20)  -- fixed length
  - VARCHAR(40)  -- variable length
- Numbers:
  - INT, REAL plus variations
- Times and dates:
  - DATE, DATETIME (SQL Server only)
- To reuse domains:
  CREATE DOMAIN address AS VARCHAR(55)

Creating Tables

Example:

```
CREATE TABLE Person(
    name VARCHAR(30),
    social-security-number INT,
    age SHORTINT,
    city VARCHAR(30),
    gender CHAR(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 'F',
    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

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

- We are given a **workload** = a set of SQL queries plus how often they run
- What indexes should we build to speed up the workload?
- FROM/WHERE clauses favor an index
- INSERT/UPDATE clauses discourage an index
- Index selection is normally done by people, recently done automatically (SQL Server)
Defining Views

Views are relations, except that they are not physically stored.

For presenting different information to different users

Employee(ssn, name, department, project, salary)

CREATE VIEW Developers AS
SELECT name, project
FROM Employee
WHERE department = "Development"

Payroll has access to Employee, others only to Developers

A Different View

Person(name, city)
Purchase(buyer, seller, product, store)
Product(name, maker, category)

CREATE VIEW Seattle-view AS
SELECT buyer, seller, product, store
FROM Person, Purchase
WHERE Person.city = "Seattle" AND
Person.name = Purchase.buyer

We have a new virtual table:
Seattle-view(buyer, seller, product, store)

A Different View

We can later use the view:

SELECT name, store
FROM Seattle-view, Product
WHERE Seattle-view.product = Product.name AND
Product.category = "shoes"

What Happens When We Query a View?

SELECT name, Seattle-view.store
FROM Seattle-view, Product
WHERE Seattle-view.product = Product.name AND
Product.category = "shoes"

SELECT name, Purchase.store
FROM Person, Purchase, Product
WHERE Person.city = "Seattle" AND
Person.name = Purchase.buyer AND
Purchase.product = Product.name AND
Product.category = "shoes"

Types of Views

- Virtual views:
  - Used in databases
  - Computed only on-demand – slow at runtime
  - Always up to date
- Materialized views
  - Used in data warehouses
  - Pre-computed offline – fast at runtime
  - May have stale data

Updating Views

How can I insert a tuple into a table that doesn’t exist?

Employee(ssn, name, department, project, salary)

CREATE VIEW Developers AS
SELECT name, project
FROM Employee
WHERE department = "Development"

If we make the following insertion:
INSERT INTO Developers
VALUES(“Joe”, “Optimizer”)

It becomes:
INSERT INTO Employee(ssn, name, department, project, salary)
Non-Updateable Views

Person(name, city)
Purchase(buyer, seller, product, store)

CREATE VIEW City-Store AS
SELECT Person.city, Purchase.store
FROM Person, Purchase
WHERE Person.name = Purchase.buyer

How can we add the following tuple to the view?
(“Seattle”, “Nine West”)

We don’t know the name of the person who made the purchase; cannot set to NULL (why?)

Constraints in SQL

- A constraint = a property that we’d like our database to hold
- The system will enforce the constraint by taking some actions:
  - forbid an update
  - or perform compensating updates

Constraints in SQL

- Keys, foreign keys
- Attribute-level constraints
- Tuple-level constraints
- Global constraints: assertions

The more complex the constraint, the harder it is to check and to enforce

Keys

CREATE TABLE Product (name CHAR(30) PRIMARY KEY, category VARCHAR(20))

OR:

CREATE TABLE Product (name CHAR(30), category VARCHAR(20) PRIMARY KEY (name))

Other Keys

CREATE TABLE Product (productID CHAR(10), name CHAR(30), category VARCHAR(20), price INT, PRIMARY KEY (productID), UNIQUE (name, category))

There is at most one PRIMARY KEY; there can be many UNIQUE

Keys with Multiple Attributes

CREATE TABLE Product (name CHAR(30), category VARCHAR(20), price INT, PRIMARY KEY (name, category))
Foreign Key Constraints

CREATE TABLE Purchase (
    prodName CHAR(30),
    REFERENCES Product(name),
    date DATETIME)

prodName is a foreign key to Product(name)
name must be a key in Product

Foreign Key Constraints

- OR

CREATE TABLE Purchase (
    prodName CHAR(30),
    category VARCHAR(20),
    date DATETIME,
    FOREIGN KEY (prodName, category)
    REFERENCES Product(name, category)
    )

- (name, category) must be a PRIMARY KEY

What happens during updates?

Types of updates:
- In Purchase: insert/update
- In Product: delete/update

Constraints on Attributes and Tuples

- Constraints on attributes:
  - NOT NULL -- obvious meaning...
  - CHECK condition -- any condition!
- Constraints on tuples
  - CHECK condition

READING ASSIGNMENT: 7.1.5, 7.1.6
What is the difference from Foreign-Key?

CREATE TABLE Purchase (  
    prodName CHAR(30)  
    CHECK (prodName IN  
    SELECT Product.name  
    FROM Product),  
    date DATETIME NOT NULL)

Final Comments on Constraints
- Can give them names, and alter later
  - Read in the book !!!!
- We need to understand exactly when they are checked
- We need to understand exactly what actions are taken if they fail

General Assertions
CREATE ASSERTION myAssert CHECK NOT EXISTS(  
    SELECT Product.name  
    FROM Product, Purchase  
    WHERE Product.name = Purchase.prodName  
    GROUP BY Product.name  
    HAVING count(*) > 200)

Embedded SQL
- direct SQL (= ad-hoc SQL) is rarely used
- in practice: SQL is embedded in some application code
- SQL code is identified by special syntax

Impedance Mismatch
- Example: SQL in C:  
  - C uses int, char[...], pointers, etc
  - SQL uses tables
- Impedance mismatch = incompatible types

The Impedance Mismatch Problem
Why not use only one language?
- Forgetting SQL: “we can quickly dispense with this idea” [textbook, pg. 351].
- SQL cannot do everything that the host language can do.

Solution: use cursors
Transactions

Address two issues:

- Access by multiple users
  - Remember the “client-server” architecture: one server with many clients
- Protection against crashes

Multiple users: single statements

Client 1:
UPDATE Product
SET Price = Price – 1.99
WHERE pname = ‘Gizmo’

Client 2:
UPDATE Product
SET Price = Price*0.5
WHERE pname=‘Gizmo’

Two managers attempt to do a discount. Will it work?

Multiple users: multiple statements

Client 1:
INSERT INTO SmallProduct(name, price)
SELECT pname, price
FROM Product
WHERE price <= 0.99
DELETE Product
WHERE price <=0.99

Client 2:
SELECT count(*)
FROM Product
SELECT count(*)
FROM SmallProduct

What’s wrong?

Protection against crashes

Client 1:
INSERT INTO SmallProduct(name, price)
SELECT pname, price
FROM Product
WHERE price <= 0.99
DELETE Product
WHERE price <=0.99

Crash!

What’s wrong?

Transactions

- Transaction = group of statements that must be executed atomically
- Transaction properties: ACID
  - ATOMICITY = all or nothing
  - CONSISTENCY = leave database in consistent state
  - ISOLATION = as if it were the only transaction in the system
  - DURABILITY = store on disk!

Transactions in SQL

- In “ad-hoc” SQL:
  - Default: each statement = one transaction
- In “embedded” SQL:
  BEGIN TRANSACTION
  [SQL statements]
  COMMIT or ROLLBACK (=ABORT)
Transactions: Serializability

Serializability = the technical term for isolation
- An execution is *serial* if it is completely before or completely after any other function’s execution
- An execution is *serializable* if it equivalent to one that is serial
- DBMS can offer serializability guarantees

Serializability

- Enforced with locks, like in Operating Systems!
- But this is not enough:

Serializability

- Solution: two-phase locking
  - Lock everything at the beginning
  - Unlock everything at the end
- Read locks: many simultaneous read locks allowed
- Write locks: only one write lock allowed
- Insert locks: one per table

Isolation Levels in SQL

1. “Dirty reads”
   SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
2. “Committed reads”
   SET TRANSACTION ISOLATION LEVEL READ COMMITTED
3. “Repeatable reads”
   SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
4. Serializable transactions (default)
   SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

Reading assignment: chapter 8.6