SQL: Part II

Introduction to Databases
CompSci 316 Fall 2014
Announcements (Thu., Sep. 18)

• **Homework #1 sample solution** to be posted on Sakai by tomorrow

• We are working on resolving the **websubmit** issue
  • Use Chrome and IE for now

• **Homework #2** due in two weeks
Incomplete information

• Example: User \((uid, name, age, pop)\)

• Value unknown
  • We do not know Nelson’s age

• Value not applicable
  • Suppose \(pop\) is based on interactions with others on our social networking site
  • Nelson is new to our site; what is his \(pop\)?
Solution 1

• Dedicate a value from each domain (type)
  • \( pop \) cannot be \(-1\), so use \(-1\) as a special value to indicate a missing or invalid \( pop \)
  • Leads to incorrect answers if not careful
    • \( \text{SELECT AVG}(\text{pop}) \ \text{FROM User} \);
  • Complicates applications
    • \( \text{SELECT AVG}(\text{pop}) \ \text{FROM User WHERE pop <> -1} \);
• Perhaps the value is not as special as you think!
  • Ever heard of the Y2K bug? “00” was used as a missing or invalid year value
Solution 2

• A valid-bit for every column
  • User (uid, name, name_is_valid, age, age_is_valid, pop, pop_is_valid)
• Complicates schema and queries
  • SELECT AVG(pop) FROM User WHERE pop_is_valid;
Solution 3

• Decompose the table; missing row = missing value
  • **UserName** *(uid, name)*
    **UserAge** *(uid, age)*
    **UserPop** *(uid, pop)*
  • **UserID** *(uid)*
• Conceptually the cleanest solution
• Still complicates schema and queries
  • How to get all information about users in a table?
  • Natural join doesn’t work!
SQL’s solution

• A special value **NULL**
  • For every domain
  • Special rules for dealing with NULL’s

• Example: *User (uid, name, age, pop)*
  • ⟨789, “Nelson”, NULL, NULL⟩
Computing with NULL’s

• When we operate on a NULL and another value (including another NULL) using +, −, etc., the result is NULL

• Aggregate functions ignore NULL, except COUNT(*) (since it counts rows)
Three-valued logic

• TRUE = 1, FALSE = 0, UNKNOWN = 0.5
• \( x \text{ AND } y = \min(x, y) \)
• \( x \text{ OR } y = \max(x, y) \)
• NOT \( x = 1 - x \)
• When we compare a NULL with another value (including another NULL) using =, >, etc., the result is UNKNOWN
• WHERE and HAVING clauses only select rows for output if the condition evaluates to TRUE
  • UNKNOWN is not enough
Unfortunate consequences

• SELECT AVG(pop) FROM User;
  SELECT SUM(pop)/COUNT(*) FROM User;
  • Not equivalent
  • Although AVG(pop)=SUM(pop)/COUNT(pop) still

• SELECT * FROM User;
  SELECT * FROM User WHERE pop = pop;
  • Not equivalent

☞ Be careful: NULL breaks many equivalences
Another problem

• Example: Who has NULL pop values?
  • SELECT * FROM User WHERE pop = NULL;
    • Does not work; never returns anything
  • (SELECT * FROM User)
    EXCEPT ALL 
    (SELECT * FROM User WHERE pop = pop);
    • Works, but ugly
  • Introduced special, built-in predicates
    IS NULL and IS NOT NULL
    • SELECT * FROM User WHERE pop IS NULL;
Outerjoin motivation

- Example: a master group membership list
  - SELECT g.gid, g.name AS gname,
        u.uid, u.name AS uname
  FROM Group g, Member m, User u
  WHERE g.gid = m.gid AND m.uid = u.uid;
- What if a group is empty?
- It may be reasonable for the master list to include empty groups as well
  - For these classes, uid and uname columns would be NULL
Outerjoin flavors and definitions

• A **full outerjoin** between $R$ and $S$ (denoted $R \bowtie S$) includes all rows in the result of $R \bowtie S$, plus
  • “Dangling” $R$ rows (those that do not join with any $S$ rows) padded with NULL’s for $S$’s columns
  • “Dangling” $S$ rows (those that do not join with any $R$ rows) padded with NULL’s for $R$’s columns

• A **left outerjoin** ($R \bowtie S$) includes rows in $R \bowtie S$ plus dangling $R$ rows padded with NULL’s

• A **right outerjoin** ($R \bowtie S$) includes rows in $R \bowtie S$ plus dangling $S$ rows padded with NULL’s
### Outerjoin examples

<table>
<thead>
<tr>
<th>gid</th>
<th>name</th>
<th>uid</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>Book Club</td>
<td>857</td>
</tr>
<tr>
<td>gov</td>
<td>Student Government</td>
<td>123</td>
</tr>
<tr>
<td>gov</td>
<td>Student Government</td>
<td>857</td>
</tr>
<tr>
<td>dps</td>
<td>Dead Putting Society</td>
<td>142</td>
</tr>
<tr>
<td>nuk</td>
<td>United Nuclear Workers</td>
<td>NULL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>gid</th>
<th>name</th>
<th>uid</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>Book Club</td>
<td>857</td>
</tr>
<tr>
<td>gov</td>
<td>Student Government</td>
<td>123</td>
</tr>
<tr>
<td>gov</td>
<td>Student Government</td>
<td>857</td>
</tr>
<tr>
<td>dps</td>
<td>Dead Putting Society</td>
<td>142</td>
</tr>
<tr>
<td>nuk</td>
<td>United Nuclear Workers</td>
<td>NULL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>gid</th>
<th>name</th>
<th>uid</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>Book Club</td>
<td>857</td>
</tr>
<tr>
<td>gov</td>
<td>Student Government</td>
<td>123</td>
</tr>
<tr>
<td>gov</td>
<td>Student Government</td>
<td>857</td>
</tr>
<tr>
<td>dps</td>
<td>Dead Putting Society</td>
<td>142</td>
</tr>
<tr>
<td>foo</td>
<td>NULL</td>
<td>789</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>gid</th>
<th>name</th>
<th>uid</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>Book Club</td>
<td>857</td>
</tr>
<tr>
<td>gov</td>
<td>Student Government</td>
<td>123</td>
</tr>
<tr>
<td>gov</td>
<td>Student Government</td>
<td>857</td>
</tr>
<tr>
<td>dps</td>
<td>Dead Putting Society</td>
<td>142</td>
</tr>
<tr>
<td>nuk</td>
<td>United Nuclear Workers</td>
<td>NULL</td>
</tr>
<tr>
<td>foo</td>
<td>NULL</td>
<td>789</td>
</tr>
</tbody>
</table>
Outerjoin syntax

• SELECT * FROM Group LEFT OUTER JOIN Member
  ON Group.gid = Member.gid;
  ≅ $\text{Group}_{\text{Group.gid}=\text{Member.gid}} \bowtie \text{Member}$

• SELECT * FROM Group RIGHT OUTER JOIN Member
  ON Group.gid = Member.gid;
  ≅ $\text{Group}_{\text{Group.gid}=\text{Member.gid}} \bowtie \text{Member}$

• SELECT * FROM Group FULL OUTER JOIN Member
  ON Group.gid = Member.gid;
  ≅ $\text{Group}_{\text{Group.gid}=\text{Member.gid}} \bowtie \text{Member}$

These are theta joins rather than natural joins
• Return all columns in Group and Member

A similar construct exists for regular (“inner”) joins:
• SELECT * FROM Group JOIN Member
  ON Group.gid = Member.gid;
SQL features covered so far

• SELECT–FROM–WHERE statements
• Set and bag operations
• Table expressions, subqueries
• Aggregation and grouping
• Ordering
• NULL’s and outerjoins

Next: data modification statements, constraints
**INSERT**

- Insert one row
  - `INSERT INTO Member VALUES (789, 'dps');`
  - User 789 joins Dead Putting Society

- Insert the result of a query
  - `INSERT INTO Member
      (SELECT uid, 'dps' FROM User
       WHERE uid NOT IN (SELECT uid
                         FROM Member
                         WHERE gid = 'dps'));
  - Everybody joins Dead Putting Society!
DELETE

• Delete everything from a table
  • `DELETE FROM Member;`

• Delete according to a `WHERE` condition

Example: User 789 leaves Dead Putting Society
  • `DELETE FROM Member
    WHERE uid = 789 AND gid = 'dps';`

Example: Users under age 18 must be removed from United Nuclear Workers
  • `DELETE FROM Member
    WHERE uid IN (SELECT uid FROM User
          WHERE age < 18)
    AND gid = 'nuk';`
UPDATE

• Example: User 142 changes name to “Barney”
  • UPDATE User
    SET name = 'Barney'
    WHERE uid = 142;

• Example: We are all popular!
  • UPDATE User
    SET pop = (SELECT AVG(pop) FROM User);
    • But won’t update of every row causes average pop to change?
    $\triangleright$ Subquery is always computed over the old table
Constraints

• Restrictions on allowable data in a database
  • In addition to the simple structure and type restrictions imposed by the table definitions
  • Declared as part of the schema
  • Enforced by the DBMS

• Why use constraints?
  • Protect data integrity (catch errors)
  • Tell the DBMS about the data (so it can optimize better)
Types of SQL constraints

- NOT NULL
- Key
- Referential integrity (foreign key)
- General assertion
- Tuple- and attribute-based CHECK’s
NOT NULL constraint examples

• CREATE TABLE User
  (uid INTEGER NOT NULL,
   name VARCHAR(30) NOT NULL,
   twitterid VARCHAR(15) NOT NULL,
   age INTEGER,
   pop FLOAT);

• CREATE TABLE Group
  (gid CHAR(10) NOT NULL,
   name VARCHAR(100) NOT NULL);

• CREATE TABLE Member
  (uid INTEGER NOT NULL,
   gid CHAR(10) NOT NULL);
Key declaration

• At most one PRIMARY KEY per table
  • Typically implies a primary index
  • Rows are stored inside the index, typically sorted by the primary key value ⇒ best speedup for queries

• Any number of UNIQUE keys per table
  • Typically implies a secondary index
  • Pointers to rows are stored inside the index ⇒ less speedup for queries
Key declaration examples

• CREATE TABLE User
  (uid INTEGER NOT NULL PRIMARY KEY,
   name VARCHAR(30) NOT NULL,
   twitterid VARCHAR(15) NOT NULL UNIQUE,
   age INTEGER,
   pop FLOAT);

• CREATE TABLE Group
  (gid CHAR(10) NOT NULL PRIMARY KEY,
   name VARCHAR(100) NOT NULL);

• CREATE TABLE Member
  (uid INTEGER NOT NULL,
   gid CHAR(10) NOT NULL,
   PRIMARY KEY(uid, gid));

This form is required for multi-attribute keys
Referential integrity example

- **Member.uid** references **User.uid**
  - If an *uid* appears in **Member**, it must appear in **User**
- **Member.gid** references **Group.gid**
  - If a *gid* appears in **Member**, it must appear in **Group**

That is, no “dangling pointers”
Referential integrity in SQL

• Referenced column(s) must be PRIMARY KEY
• Referencing column(s) form a FOREIGN KEY
• Example
  • CREATE TABLE Member
    (uid INTEGER NOT NULL
     REFERENCES User(uid),
    gid CHAR(10) NOT NULL,
    PRIMARY KEY(uid, gid),
    FOREIGN KEY gid REFERENCES Group(gid));
Enforcing referential integrity

Example: *Member.uid references User.uid*

- Insert or update a *Member* row so it refers to a non-existent *uid*
  - Reject
- Delete or update a *User* row whose *uid* is referenced by some *Member* row
  - Reject
  - **Cascade**: ripple changes to all referring rows
  - **Set NULL**: set all references to NULL
  - All three options can be specified in SQL
Deferred constraint checking

• No-chicken-no-egg problem
  • CREATE TABLE Dept
    (name CHAR(20) NOT NULL PRIMARY KEY,
     chair CHAR(30) NOT NULL
     REFERENCES Prof(name)));
  CREATE TABLE Prof
    (name CHAR(30) NOT NULL PRIMARY KEY,
     dept CHAR(20) NOT NULL
     REFERENCES Dept(name)));

  • The first INSERT will always violate a constraint!

• Deferred constraint checking is necessary
  • Check only at the end of a transaction
  • Allowed in SQL as an option

• Curious how the schema was created in the first place?
  • ALTER TABLE ADD CONSTRAINT (read the manual!)
General assertion

• **CREATE ASSERTION** *assertion_name* CHECK *assertion_condition*;

• *assertion_condition* is checked for each modification that could potentially violate it

• Example: *Member.uid* references *User.uid*
  
  • **CREATE ASSERTION** *MemberUserRefIntegrity* CHECK (NOT EXISTS (SELECT * FROM Member WHERE uid NOT IN (SELECT uid FROM User)));

☞ In SQL3, but not all (perhaps no) DBMS supports it
Tuple- and attribute-based CHECK’s

• Associated with a single table
• Only checked when a tuple or an attribute is inserted or updated
• Examples:
  • CREATE TABLE User(...
    age INTEGER
    CHECK(age IS NULL OR age > 0),
  ...
  ...
  • CREATE TABLE Member
    (uid INTEGER NOT NULL
    CHECK(uid IN
      (SELECT uid FROM User)),
  ...
  ...
  • Is it a referential integrity constraint?
  • Not quite; not checked when User is modified
SQL features covered so far

• Query
  • SELECT–FROM–WHERE statements
  • Set and bag operations
  • Table expressions, subqueries
  • Aggregation and grouping
  • Ordering
  • Outerjoins

• Modification
  • INSERT/DELETE/UPDATE

• Constraints

☞ Next: triggers, views, indexes