SQL: Part II

Introduction to Databases

CompSci 317 Spring 2017

DUKE COMPUTER SCIENCE

Announcements (Wed., Feb. 08)

- Homework #1 sample solution to be posted on Sakai tonight
- Homework #2 due in 1½ weeks
- Office hours this week
  - Mon, Tues, Wed, Thurs: LSRC D105
  - Fri: LSRC A247 (as before)
  - Hopefully permanent!
- I am going to assume anyone who has not contacted me about groups, has found a group!

Review of Lecture 7

User (uid, name, age, pop)
Group (gid, name)
Member (uid, gid)

- SQL queries and semantics
- e.g. Find average popularity of members in groups of size > 100

```
5  SELECT G.name, AVG(pop)
1  FROM User U, Member M, Group G
2  WHERE U.uid = M.uid AND M.gid = G.gid
3  GROUP BY G.name
4  HAVING COUNT(*) > 100
```

- Set (no duplicates) and bag operations (duplicates allowed)
- Subqueries – FROM, WHERE, WITH
- EXISTS, IN, UNION, ALL, ANY
- Ordering – ORDER BY

Today

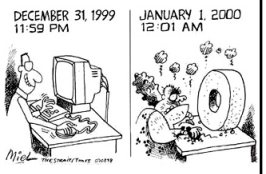
- Incomplete information and NULL
- Outerjoins
- Data modification
- Constraints

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
    - SELECT AVG(pop) FROM User;
  - Complicates applications
    - SELECT AVG(pop) 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



http://www.90s411.com/images/y2k-cartoon.jpg

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! Why?

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

in class:
x = 5, y = NULL
((x > 6) OR (y > 6)) AND (x < 10)
= ?

- TRUE = 1, FALSE = 0, UNKNOWN = 0.5!
- x AND y = $\min(x, y)$
- x 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

Are the queries equivalent?

- SELECT AVG(pop) FROM User;
  SELECT SUM(pop)/COUNT(*) FROM User;
- SELECT * FROM User;
  SELECT * FROM User WHERE pop = pop;
  SELECT * FROM User WHERE pop > 1 OR pop <= 1

☞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
  - SQL 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? Will it appear in the result?
  - It may be reasonable for the master list to include empty groups as well
    - For these groups, *uid* and *uname* columns would be NULL

Outerjoin flavors and definitions

- A full outerjoin between R and S (denoted $R \bowtie\!\!\!\!\!\!\bowtie S$) 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 ($R \bowtie\!\!\!\!\!\!\bowtie S$) includes rows in $R \bowtie S$ plus dangling R rows padded with NULL's
- A right outerjoin ($R \bowtie\!\!\!\!\!\!\bowtie S$) includes rows in $R \bowtie S$ plus dangling S rows padded with NULL's

Outerjoin examples

Group ⟗ *Member*

Group

gid	*name*
abc	Book Club
gov	Student Government
dps	Dead Putting Society
nuk	United Nuclear Workers

Group ⟕ *Member*

Member

uid	*gid*
142	dps
123	gov
857	abc
857	gov
789	foo

Group ⟖ *Member*

Outerjoin syntax

- SELECT * FROM Group LEFT OUTER JOIN Member
  ON Group.gid = Member.gid;
  ≈ $Group \underset{Group.gid=Member.gid}{\bowtie} Member$
- SELECT * FROM Group RIGHT OUTER JOIN Member
  ON Group.gid = Member.gid;
  ≈ $Group \underset{Group.gid=Member.gid}{\bowtie} Member$
- SELECT * FROM Group FULL OUTER JOIN Member
  ON Group.gid = Member.gid;
  ≈ $Group \underset{Group.gid=Member.gid}{\bowtie} Member$

☞A similar construct exists for regular ("inner") joins:
- SELECT * FROM Group JOIN Member
  ON Group.gid = Member.gid;

☞These are theta joins rather than natural joins
- Return all columns in *Group* and *Member*

☞For natural joins, add keyword NATURAL; don't use ON

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

19

INSERT

User (uid, name, age, pop)
Group (gid, name)
Member (uid, gid)

- 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!

20

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';

21

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?
  - ☞Subquery is always computed over the old table

22

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?

23

Types of SQL constraints

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

24

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);

age or pop can be NULL

- 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

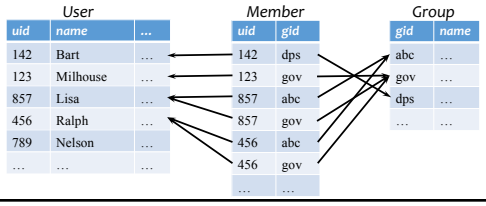
- CREATE TABLE User
  (uid INTEGER NOT NULL PRIMARY KEY,
  name VARCHAR(30) NOT NULL,
  twitterid VARCHAR(15) NOT NULL UNIQUE,
  age INTEGER,
  pop FLOAT);
- CREATE TABLE Group
  (gid CHAR(10) NOT NULL PRIMARY KEY,
  name VARCHAR(100) NOT NULL);
- CREATE TABLE Member
  (uid INTEGER NOT NULL,
  gid CHAR(10) 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));

Both work!

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

31

- 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

32

- Associated with a single table
- Only checked when a tuple/attribute is inserted/updated
  - Reject if condition evaluates to FALSE
  - TRUE and UNKNOWN are fine
- 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

33

- 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