

SQL: Part II

Introduction to Databases
CompSci 316 Spring 2019



Announcements (Thu., Jan. 31)

- Homework #1 due next Tuesday (Feb 5) 11:59pm
 - Extra credit problems due on Feb 8 (Friday) 11:59 pm
- Project mixer next Tuesday in class (first half, 2nd half regular lecture)
 - Presentation by Elliott Bolzan (your UTA) about their project in the last semester – do not miss it!
 - You will get an idea how much work and what output is expected
 - Please let me know by next Monday if you want to make a pitch in front of the class (to recruit teammates)!
- Sudeepa's office hours Wednesdays 1:30-2:30 pm, LSRC D325.

Project resources

- Working web dev examples in PHP, Flask, and Play/Java for course VM
 - See "Help" on course website for more details
- Duke Co-Lab offerings
 - Many interesting "Roots" courses
 - Build Your First iPhone or iPad App, Making Your Website Interactive, Intro to React.js, Introduction to Linux, etc.
 - Advance registration required
 - Office hours on full-stack web/app development



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



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
- $x \text{ AND } y = \min(x, y)$
- $x \text{ OR } y = \max(x, y)$
- $\text{NOT } x = 1 - x$
- 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
 - 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?
 - 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 $R \bowtie S$) 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** ($R \ltimes S$) includes rows in $R \bowtie S$ plus dangling R rows padded with NULL's
- A **right outerjoin** ($R \rtimes S$) includes rows in $R \bowtie S$ plus dangling S rows padded with NULL's

Outerjoin examples

Group		Group \bowtie Member		
gid	name	gid	name	uid
abc	Book Club	abc	Book Club	857
gov	Student Government	gov	Student Government	123
dps	Dead Putting Society	gov	Student Government	857
nuk	United Nuclear Workers	dps	Dead Putting Society	142
		nuk	United Nuclear Workers	NULL

Member		Group \bowtie Member		
uid	gid	gid	name	uid
142	dps	abc	Book Club	857
123	gov	gov	Student Government	123
857	abc	gov	Student Government	857
857	gov	dps	Dead Putting Society	142
789	foo	nuk	United Nuclear Workers	NULL
		foo	NULL	789

Outerjoin syntax

- SELECT * FROM Group **LEFT OUTER JOIN** Member ON Group.gid = Member.gid;
 \approx Group \ltimes Member
 - SELECT * FROM Group **RIGHT OUTER JOIN** Member ON Group.gid = Member.gid;
 \approx Group \rtimes Member
 - SELECT * FROM Group **FULL OUTER JOIN** Member ON Group.gid = Member.gid;
 \approx Group \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

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?
 - 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 \Rightarrow best speedup for queries
- Any number of **UNIQUE** keys per table
 - Typically implies a **secondary index**
 - Pointers to rows are stored inside the index \Rightarrow 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”

User			Member			Group	
uid	name	...	uid	gid		gid	name
142	Bart	...	142	dps	→	abc	...
123	Milhouse	...	123	gov	→	gov	...
857	Lisa	...	857	abc	→	dps	...
456	Ralph	...	857	gov	→
789	Nelson	...	456	abc	→
...	456	gov	→

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));
- ← This form is useful for multi-attribute foreign keys

Enforcing referential integrity

Example: Member.uid references User.uid

- Insert or update a Member row so it refers to a non-existent uid
 - Reject
- _____
- _____
- 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));
- _____
- 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/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

- 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