SQL: Part II

Introduction to Database Management

CS348 Fall 2022

Basic SQL features

- Query
  - SELECT-FROM-WHERE statements
  - Set/bag (DISTINCT, UNION/EXCEPT/INTERSECT (ALL))
  - Subqueries (table, scalar, IN, EXISTS, ALL, ANY)
  - Aggregation and grouping (GROUP BY, HAVING)
  - Ordering (ORDER)
  - Outerjoins (and Nulls)
- Modification
  - INSERT/DELETE/UPDATE
- Constraints

Lecture 4

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*

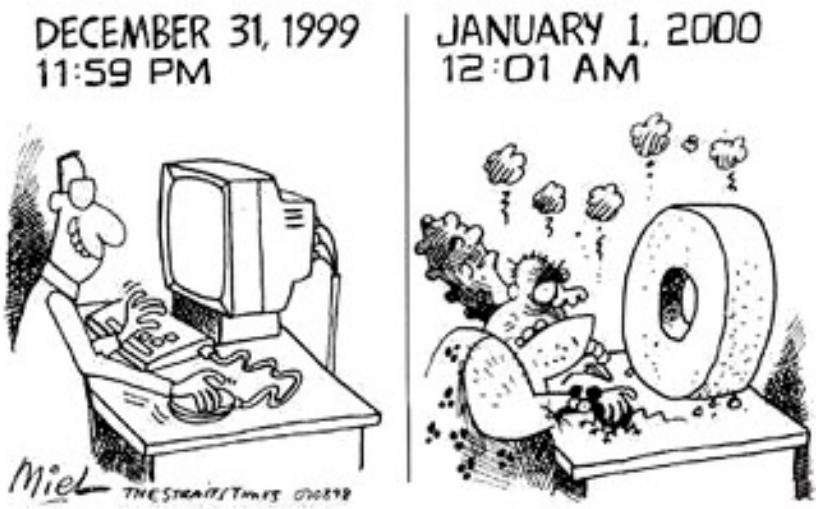
```
SELECT AVG(pop) FROM User;
```

Incorrect answers

```
SELECT AVG(pop) FROM User WHERE pop <> -1;
```

Complicated

  - Perhaps the value is not as special as you think!
    - the Y2K bug



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*)

```
SELECT AVG(pop) FROM User WHERE pop_is_valid;
```

  - Complicates schema and queries

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⟩

Three-valued logic

TRUE = 1, FALSE = 0, UNKNOWN = 0.5

x AND $y = \min(x, y)$

x OR $y = \max(x, y)$

NOT $x = 1 - x$

- Comparing a NULL with another value (including another NULL) using $=$, $>$, etc., the result is NULL
- WHERE and HAVING clauses only select rows for output if the condition evaluates to TRUE
  - NULL is not enough
- Aggregate functions ignore NULL, except COUNT(*)

Unfortunate consequences

- Q1a = Q1b?

```
Q1a. SELECT AVG(pop) FROM User;
```

```
Q1b. SELECT SUM(pop)/COUNT(*) FROM User;
```

- Q2a = Q2b?

```
Q2a. SELECT * FROM User;
```

```
Q2b SELECT * FROM User WHERE pop=pop;
```

- Be careful: NULL breaks many equivalences

Another problem

- Example: Who has NULL *pop* values?

```
SELECT * FROM User WHERE pop = NULL;
```

Does not work!

```
(SELEC * 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 examples

Group

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

Member

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

Group ⟗ *Member*

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

A full outerjoin between R and S:

- 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

Outerjoin examples

Group

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

Member

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

Group $\ltimes$ *Member*

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

- A left outerjoin ($R \ltimes S$) includes rows in $R \bowtie S$ plus dangling R rows padded with NULL's

Group $\rtimes$ *Member*

gid	*name*	*uid*
abc	Book Club	857
gov	Student Government	123
gov	Student Government	857
dps	Dead Putting Society	142
foo	NULL	789

- A right outerjoin ($R \rtimes S$) includes rows in $R \bowtie S$ plus dangling S rows padded with NULL's

Outerjoin syntax

```
SELECT * FROM Group LEFT OUTER JOIN Member
        ON Group.gid = Member.gid;
```

$\approx Group \overset{\bowtie}{}_{Group.gid=Member.gid} Member$

```
SELECT * FROM Group RIGHT OUTER JOIN Member
        ON Group.gid = Member.gid;
```

$\approx Group \overset{\bowtie}{}_{Group.gid=Member.gid} Member$

```
SELECT * FROM Group FULL OUTER JOIN Member
        ON Group.gid = Member.gid;
```

$\approx Group \overset{\bowtie}{}_{Group.gid=Member.gid} Member$

☞A similar construct exists for regular ("inner") joins:

```
SELECT * FROM Group JOIN Member ON Group.gid = Member.gid;
```

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

```
SELECT * FROM Group NATURAL JOIN Member;
```

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
  - User 789 joins Dead Putting Society

```
INSERT INTO Member VALUES (789, 'dps');
```

- Insert the result of a query
  - Everybody joins Dead Putting Society!

```
INSERT INTO Member
    (SELECT uid, 'dps' FROM User
    WHERE uid NOT IN (SELECT uid
        FROM Member
        WHERE gid = 'dps'));
```

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
- Why use constraints?
  - Protect data integrity (catch errors)
  - Tell the DBMS about the data (so it can optimize better)
- Declared as part of the schema and enforced by the DBMS

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 DECIMAL(3,0) NOT NULL,
 name VARCHAR(30) NOT NULL,
 twitterid VARCHAR(15) NOT NULL,
 age DECIMAL (2,0),
 pop DECIMAL(3,2));
```

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

```
CREATE TABLE Member
(uid DECIMAL(3,0) NOT NULL,
 gid CHAR(10) NOT NULL);
```

Key declaration examples

```
CREATE TABLE User
(uid DECIMAL(3,0) NOT NULL PRIMARY KEY,
 name VARCHAR(30) NOT NULL,
 twitterid VARCHAR(15) NOT NULL UNIQUE,
 age DECIMAL (2,0),
 pop DECIMAL(3,2));
```

At most one primary key per table

Any number of UNIQUE keys per table

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

```
CREATE TABLE Member
(uid DECIMAL(3,0) NOT NULL,
 gid CHAR(10) NOT NULL,
PRIMARY KEY(uid,gid));
```

This form is required for multi-attribute keys

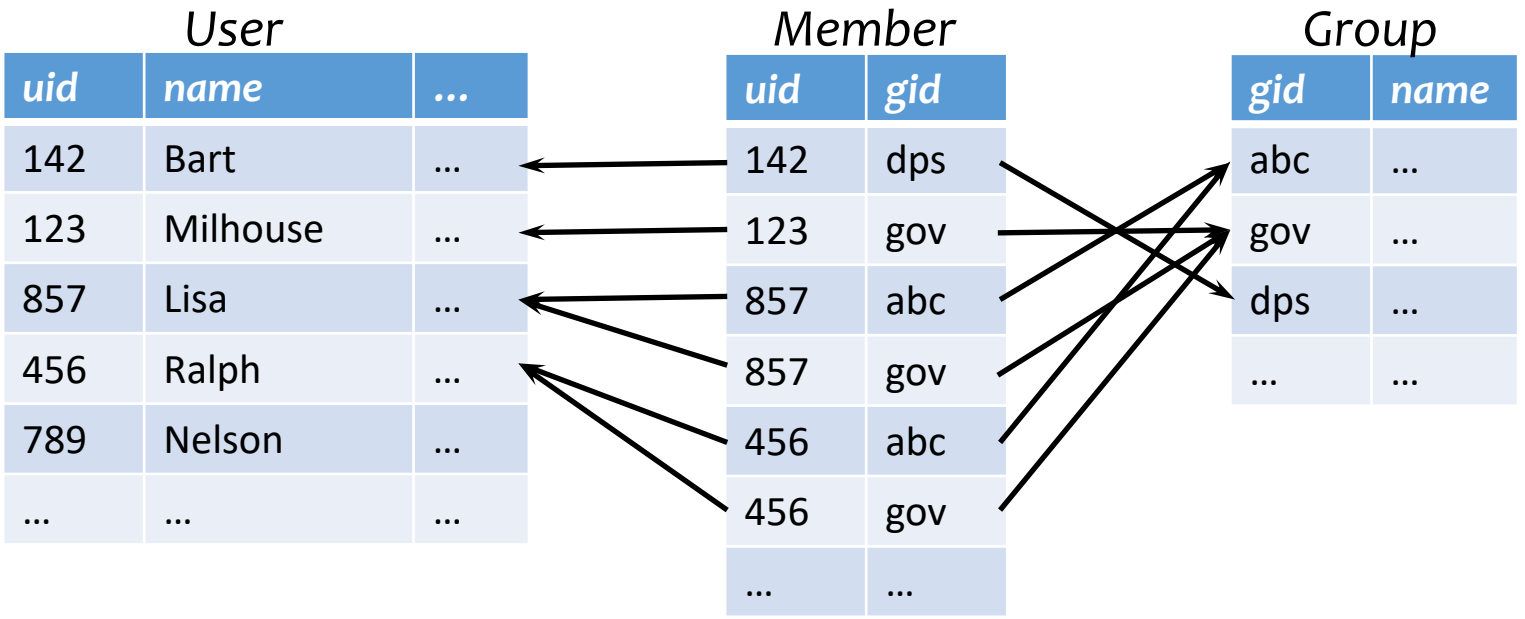
Incorrect!

```
CREATE TABLE Member
(uid DECIMAL(3,0) NOT NULL PRIMARY KEY,
 gid CHAR(10) NOT NULL PRIMARY KEY,
```

Referential integrity example

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

☞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 DECIMAL(3,0) NOT NULL REFERENCES User(uid),
 gid CHAR(10) NOT NULL,
PRIMARY KEY(uid,gid),
FOREIGN KEY (gid) REFERENCES Group(gid));
```

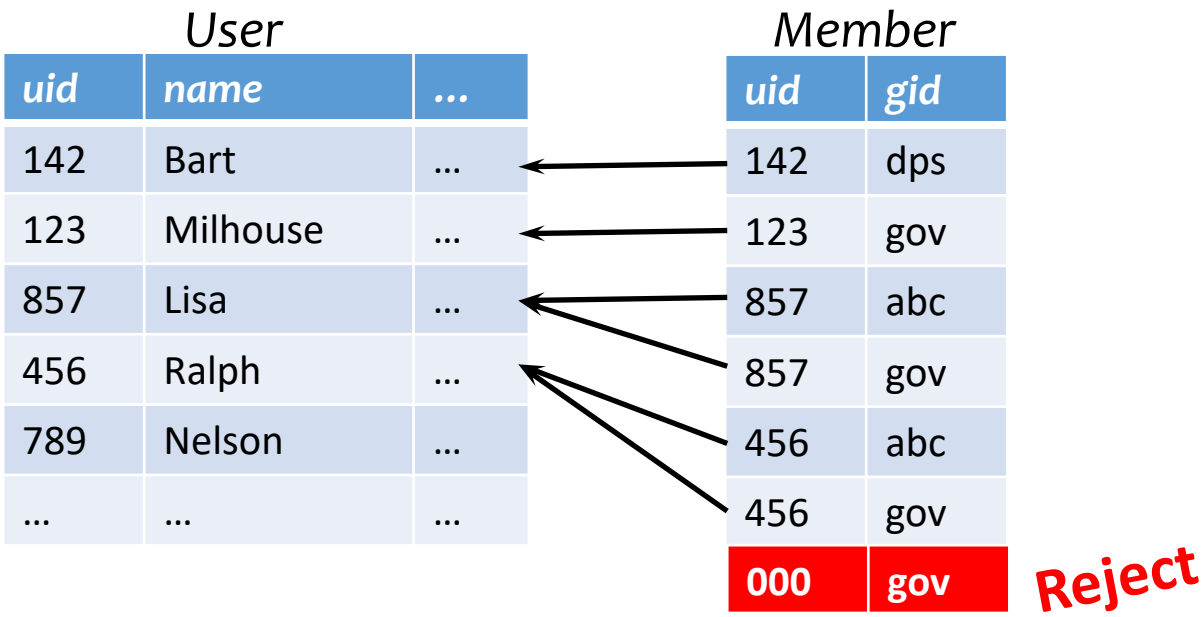
This form is required for multi-attribute foreign keys

```
CREATE TABLE MemberBenefits
(…..
FOREIGN KEY (uid,gid) REFERENCES Member(uid,gid));
```

Enforcing referential integrity

Example: *Member.uid* references *User.uid*

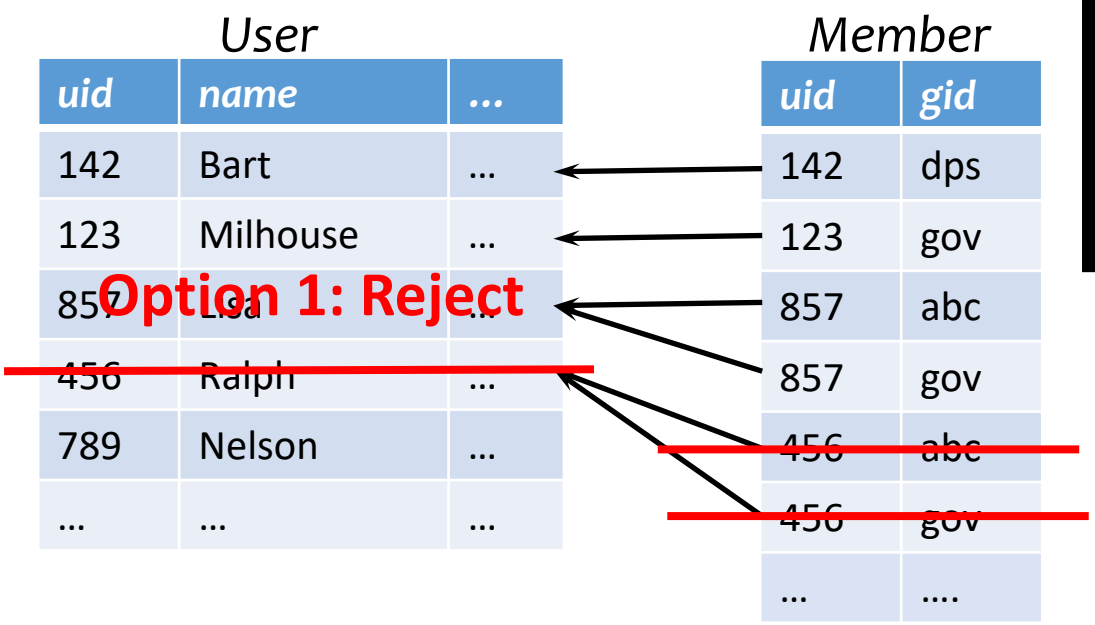
- Insert or update a *Member* row so it refers to a non-existent *uid*
  - Reject



Enforcing referential integrity

Example: *Member.uid* references *User.uid*

- Delete or update a *User* row whose *uid* is referenced by some *Member* row
  - Multiple Options (in SQL)




```
CREATE TABLE Member
(uid DECIMAL(3,0) NOT NULL
REFERENCES User(uid)
ON DELETE CASCADE,
……);
```

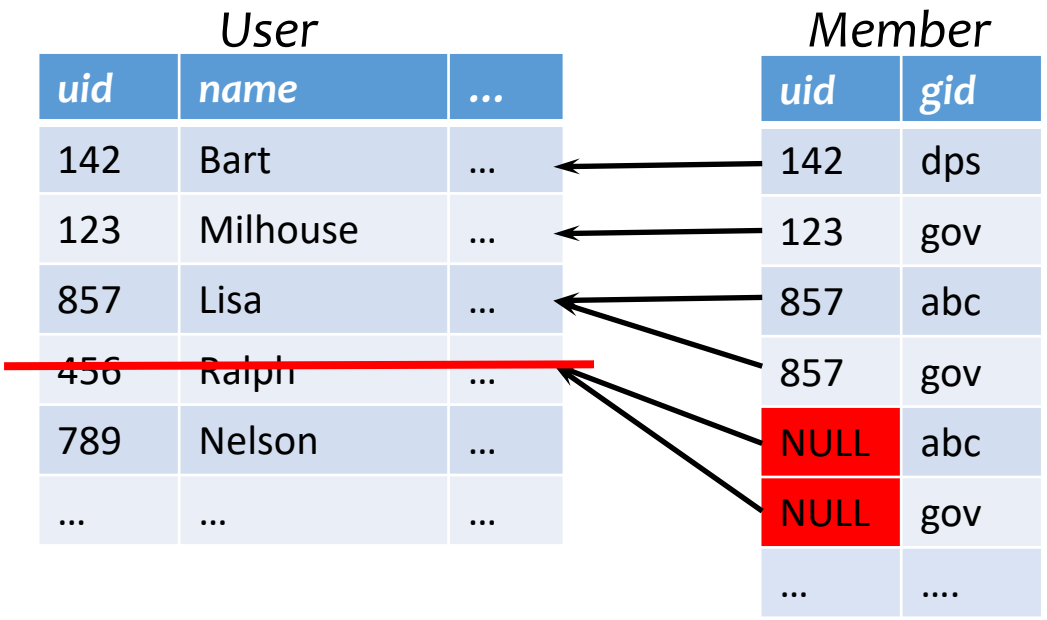
Option 2: Cascade
(ripple changes to all referring rows)

Enforcing referential integrity

Example: *Member.uid* references *User.uid*

- Delete or update a *User* row whose *uid* is referenced by some *Member* row
  - Multiple Options (in SQL)

```
CREATE TABLE Member
(uid DECIMAL(3,0) NOT NULL
REFERENCES User(uid)
ON DELETE SET NULL,
…..);
```



Option 3: Set NULL
(set all references to NULL)

Deferred constraint checking

- Example:

```
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)));
```

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)),
 ...);
```

Exercise Question: How does it differ from a referential integrity constraint (slides 26-27)?

SQL features covered so far

- Query
  - SELECT-FROM-WHERE statements
  - Set and bag operations
  - Table expressions, subqueries
  - Aggregation and grouping
  - Ordering
  - Outerjoins (and NULL)
- Modification
  - INSERT/DELETE/UPDATE
- Constraints

☞Next: triggers, views, indexes