SQL: Part III

CS348 Spring 2024

Instructor: Sujaya Maiyya

Sections: **002 and 003 only**

Announcements

- Project Milestone 0: due May 25th !
- Assignment 1: Due June 4th
  - Marmoset will be open tomorrow

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 5

Incomplete information

- Example: *User* (*uid*, *name*, *age*, *pop*)
- Value unknown
  - We do not know Nelson's *pop*
- Value not applicable
  - Suppose *pop* is based on interactions with others on our social networking site
  - Nelson is new to our site; what is their *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*

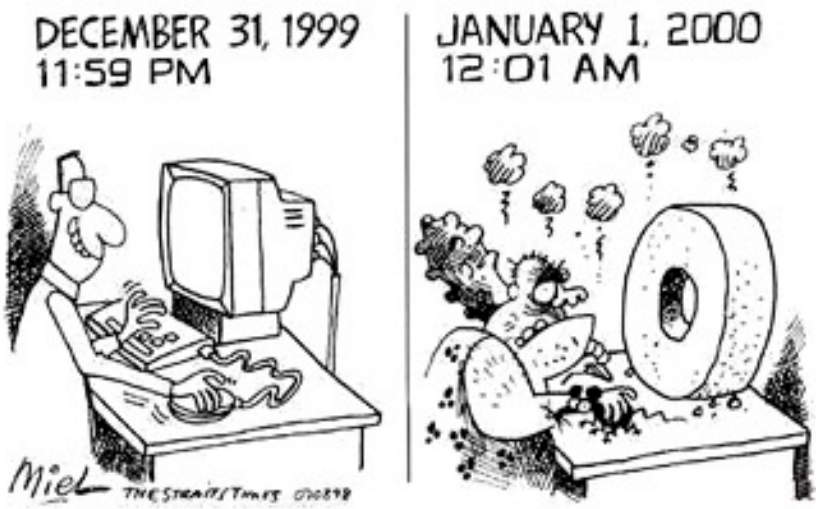
```
SELECT AVG(pop) FROM User;
```

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=1;
```

  - Complicates schema and queries
    - Need almost double the number of columns

Solution 3

- Decompose the table; missing row = missing value
  - *UserName* (*uid*, *name*) → Has a tuple for Nelson
  - *UserAge* (*uid*, *age*) → No entry for Nelson
  - *UserPop* (*uid*, *pop*) → No entry for Nelson
  - *UserID* (*uid*) → Has a tuple for Nelson

- 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 (i.e., any datatype)

- Example: *User* (*uid*, *name*, *age*, *pop*)
  - ⟨789, "Nelson", NULL, NULL⟩

- Special rules for dealing with NULL's

```
SELECT * FROM User WHERE name='Nelson' AND pop > 0.5 ??
```

Three-valued logic

TRUE = 1, FALSE = 0, UNKNOWN = 0.5

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

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

NOT x = $1 - x$

x	y	x AND y	x OR y	NOT x
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	UNKNOWN	UNKNOWN	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
UNKNOWN	TRUE	UNKNOWN	TRUE	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN
UNKNOWN	FALSE	FALSE	UNKNOWN	UNKNOWN
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	UNKNOWN	FALSE	UNKNOWN	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

- 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(*)

Will 789 be in the output?

⟨789, “Nelson”, NULL, NULL⟩

```
SELECT uid FROM User where name=‘Nelson’ AND pop>0.5;
```

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!

```
(SELECT * FROM User)
EXCEPT
(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;
```

In class exercises

Consider this db instance:

User

uid	name	age	pop
142	Bart	NULL	0.9
123	Milhouse	8	NULL
857	Lisa	8	0.7
456	Nelson	8	NULL
324	Ralph	NULL	0.3

Member

uid	gid
857	dps
123	gov
857	abc
857	gov
456	abc
456	gov

- What is the output of these queries?

```
SELECT uid FROM User where age > 5 OR pop < 0.5;
```

```
SELECT uid FROM User where age > 5 AND pop < 0.5;
```

```
SELECT avg(pop), count(*) FROM User GROUP BY age;
```

```
SELECT name FROM User WHERE age IN (SELECT age FROM User
                                    WHERE name = 'Bart');
```

Take home ex.

User (*uid* int, *name* string, *age* int, *pop* float)
Group (*gid* string, *name* string)
Member (*uid* int, *gid* string)

- For the previous db instance, what is the output for:

```
SELECT avg(pop), count(*) FROM User WHERE age IS NOT NULL
GROUP BY age;
```

```
SELECT MAX(pop), count(*) FROM User GROUP BY age;
```

- Write a query to find all users (uids) with non-null popularity who belong to at least one group.

Need for a new join query

- Example: construct a master group membership list with all groups and its members info

```
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
spr	Sports Club

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
spr	Sports Club	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
spr	Sports Club

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
spr	Sports Club	NULL

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

Group ⟖ *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 ($R \bowtie S$) 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;
```

$\approx Group \underset{Group.gid=Member.gid}{\bowtie} Member$

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

$\approx Group \underset{Group.gid=Member.gid}{\bowtie} Member$

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

$\approx Group \underset{Group.gid=Member.gid}{\bowtie} Member$

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

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

Theta join: gid is repeated

Natural join: gid appears once

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

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

In class exercises

Consider this db instance:

Group

gid	gname
abc	Book Club
gov	Student Government
dps	Dead Putting Society
spr	Sports Club

User

uid	uname	age	pop
142	Bart	10	0.9
123	Milhouse	10	NULL
857	Lisa	8	0.7
456	Ralph	8	NULL

Member

uid	gid
857	dps
123	gov
857	abc
123	abc

- What is the output of these queries?

```
SELECT u.name as uname, g.name as gname FROM User u NATURAL JOIN
Member m NATURAL JOIN Group g;
```

```
SELECT u.name as uname, m.gid FROM User u LEFT OUTER JOIN Member m
ON u.uid=m.uid;
```

```
SELECT COUNT(m.gid), COUNT(g.name) FROM Member m RIGHT OUTER JOIN
Group g ON g.gid=m.gid;
```

SQL features covered so far

- SELECT-FROM-WHERE statements
- Set and bag operations
- Table expressions, subqueries
- Aggregation and grouping
- Ordering
- NULLs and outerjoins

☞Next: data modification statements, constraints

INSERT

- Insert one row
  - User 789 joins Dead Putting Society

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

```
INSERT INTO User (uid, name) VALUES (389, ‘Marge');
```

- 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 over age 18 must be removed from Sports Club

```
DELETE FROM Member
WHERE uid IN (SELECT uid FROM User WHERE age > 18)
      AND gid = 'spr';
```

```
DELETE m FROM Member m NATURAL JOIN User u WHERE
u.age > 18 AND m.gid='spr';
```

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

In class exercises

Consider this db instance:

Group

gid	name
abc	Book Club
gov	Student Government
dps	Dead Putting Society
spr	Sports Club

User

uid	name	age	pop
142	Bart	10	0.9
123	Milhouse	10	NULL
857	Lisa	8	0.7
456	Ralph	8	NULL

Member

uid	gid
857	dps
123	gov
857	abc
123	abc

- What is the output of these queries?

```
INSERT INTO Member (SELECT u.uid, 'spr' FROM User u WHERE u.age >= 10
                    AND u.pop IS NOT NULL);
```

```
DELETE m, g FROM Member m NATURAL JOIN Group g WHERE g.gid='dps';
```

```
UPDATE User u NATURAL JOIN Member m SET u.age=11, u.pop=0.4, m.gid='spr'
WHERE u.uid=123 and m.gid='gov';
```

Constraints

- Restricts what data is allowed 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 INT NOT NULL,
 name VARCHAR(30) NOT NULL,
 twitterid VARCHAR(15) NOT NULL,
 age INT,
 pop DECIMAL(3,2));
```

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

```
CREATE TABLE Member
(uid INT NOT NULL,
 gid CHAR(10) NOT NULL);
```

Key declaration examples

```
CREATE TABLE User
(uid INT NOT NULL PRIMARY KEY,
 name VARCHAR(30) NOT NULL,
 twitterid VARCHAR(15) NOT NULL UNIQUE,
 age INT,
 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 INT NOT NULL,
 gid CHAR(10) NOT NULL,
PRIMARY KEY(uid,gid));
```

This form is required for multi-attribute keys

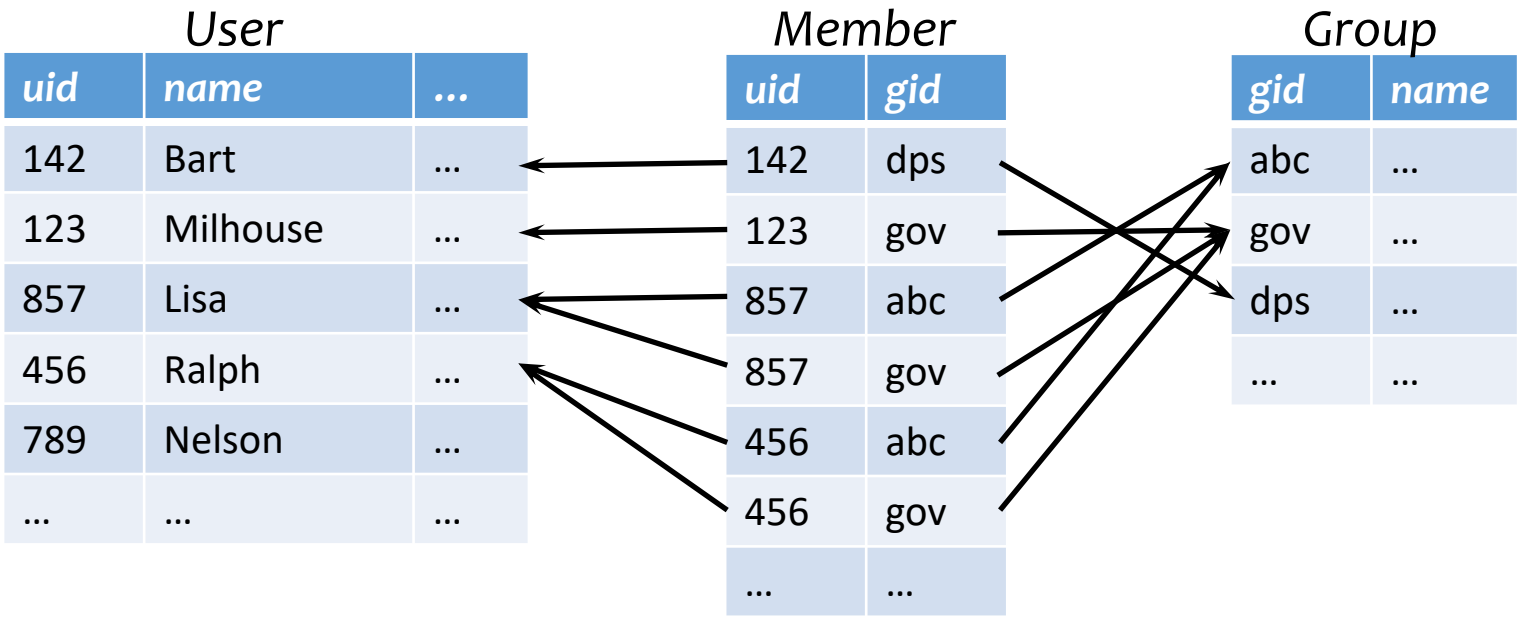
```
CREATE TABLE Member
(uid INT NOT NULL PRIMARY KEY,
 gid CHAR(10) NOT NULL PRIMARY KEY,
```

Incorrect!

Referential integrity example

- If a *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

Some system allow them to be non-PK but must be UNIQUE

```
CREATE TABLE Member
(uid INT NOT NULL REFERENCES User(uid),
 gid CHAR(10) NOT NULL,
PRIMARY KEY(uid,gid),
FOREIGN KEY (gid) REFERENCES Group(gid));
```

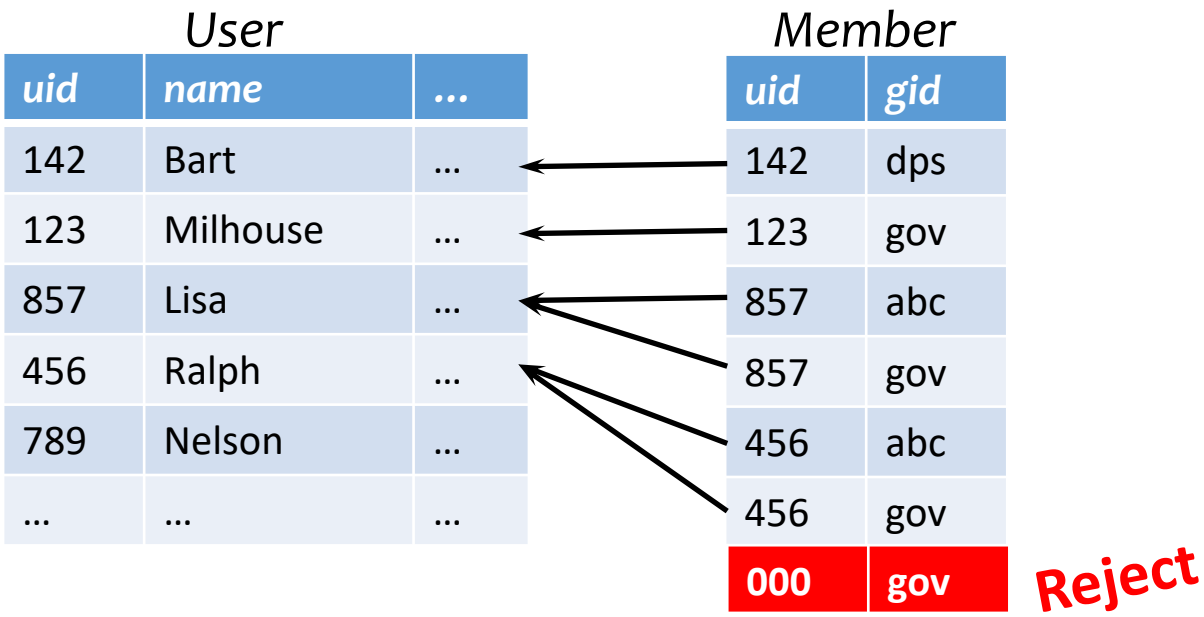
This form is required for multi-attribute foreign keys

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

Enforcing referential integrity

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

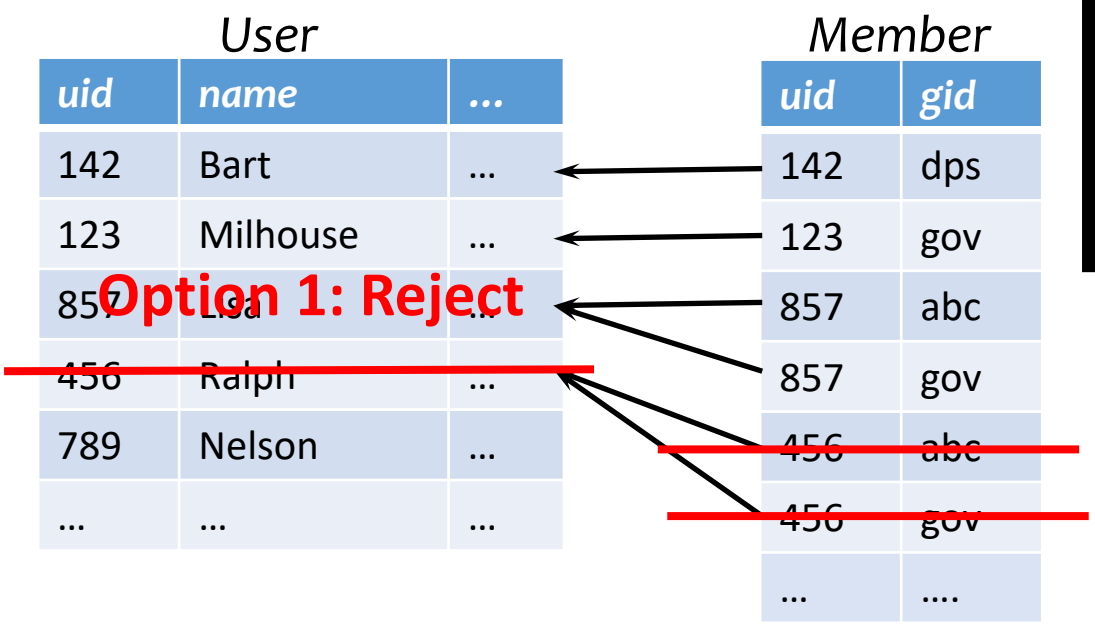
- Insert or update a *Member* row so it refers to a non-existent *uid*
  - Reject



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 INT NOT NULL
REFERENCES User(uid)
ON DELETE CASCADE,
……);
```

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

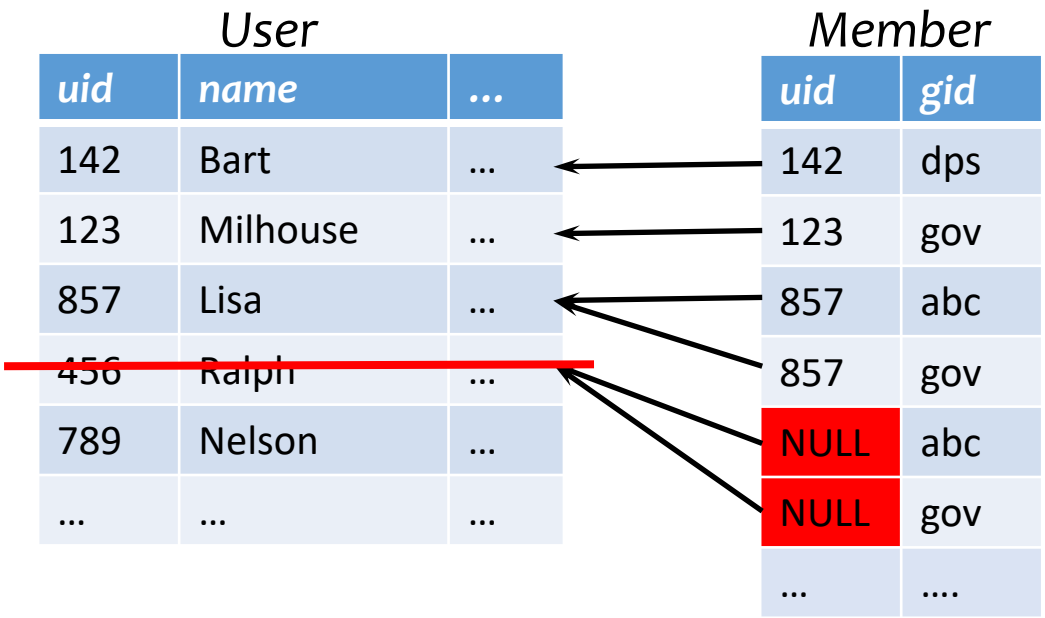
Enforcing referential integrity

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 INT NOT NULL
REFERENCES User(uid)
ON DELETE SET NULL,
…..);
```

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



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 (EXISTS
      (SELECT * FROM Member
       WHERE uid IN
       (SELECT uid FROM User)));
```

Can include multiple tables

Assertions are statements that must always be true

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)),
 ...);
```

Checked when new tuples are added to Member but not when User is modified

Naming constraints

- It is possible to name constraints (similar to assertions)

```
CREATE TABLE User(...
 age INT, constraint minAge check(age IS NULL OR age > 0),
 ...);
```

In class ex.

User (*uid* int, *name* string, *age* int, *pop* float)
Group (*gid* string, *name* string)
Member (*uid* int, *gid* string)

- Write a DDL statement to create the User table with a Primary key constraint and check that *pop* is between 0 and 1.

In class ex.

User (*uid* int, *name* string, *age* int, *pop* float)
Group (*gid* string, *name* string)
Member (*uid* int, *gid* string)

- Write a DDL statement to create the User table with a Primary key constraint and check that *pop* is between 0 and 1.

```
CREATE TABLE User
(uid INT PRIMARY KEY,
 name VARCHAR(30) NOT NULL,
 age INT,
 pop DECIMAL(3,2) CHECK(pop IS NULL OR (age >= 0 AND pop < 1));
```

In class ex.

User (*uid* int, *name* string, *age* int, *pop* float)
Group (*gid* string, *name* string)
Member (*uid* int, *gid* string)

- Say every user with pop >=0.9 must belong to the Book Club (gid='abc'). Create as assertion to check this constraint.

In class ex.

User (*uid* int, *name* string, *age* int, *pop* float)
Group (*gid* string, *name* string)
Member (*uid* int, *gid* string)

- Say every user with pop >=0.9 must belong to the Book Club (gid='abc'). Create as assertion to check this constraint.

```
CREATE ASSERTION BookClubMembership
CHECK (NOT EXISTS
        (SELECT uid FROM User WHERE pop >= 0.9 AND
         uid NOT IN (SELECT uid FROM Member WHERE gid='abc')));
```

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 lecture: schema changes, triggers, views, indexes