SFU

Database Systems I

CMPT 354 Summer 2024

Zhengjie Miao

5 June 2024

Announcements (Wed. June 5)

- Assignment 2 (Due June 7)
- Midterm I
  - Sample released
    - Just an example of the format and style
  - Exam will be 75 min (not 80 min!)
  - More instructions will be out

SQL features covered so far

- `SELECT-FROM-WHERE` statements
- Set and bag operations
- Table expressions, subqueries
- Aggregation and grouping
- Ordering
- `WITH`
- `NULL`'s and outerjoins
- Data modification statements, Constraints

Common table expressions (WITH)

Optional column names

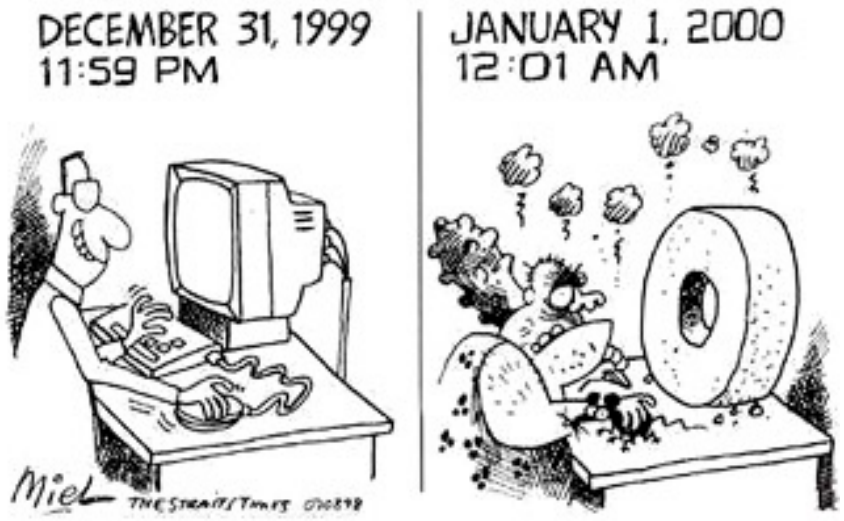
```
WITH table1(column11, column12, …)
  AS (query_definition_1),
 table2(column21, column22, …)
  AS (query_definition_2), …
actual_query;
```

Some DBMS will optimize the query before assigning the result to another name

Optional additional definitions

- Defines temporary tables to be used by
  - Other tables defined in the same WITH
    - Even recursively (more on it later in this course)
  - *actual_query*
- The whole statement returns the result of *actual_query* only

WITH example

- Again: names of users who poked others more than others poked them
  - ```
    WITH T(uid) AS
       ((SELECT uid1 FROM Poke)
        EXCEPT ALL
        (SELECT uid2 FROM Poke))
    SELECT DISTINCT name
    FROM User, T
    WHERE User.uid = T.uid;
    ```
    ```

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;
      ```
    - 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)
  - 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)
- Evaluating aggregation functions (except COUNT) on an empty collection returns NULL; converting an empty collection to a scalar also gives NULL

Three-valued logic

- TRUE = 1, FALSE = 0, UNKNOWN = 0.5
- x AND $y = \min(x, y)$
- x OR $y = \max(x, y)$
- NOT $x = 1 - x$
- 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

x	¬x
F	T
U	U
T	F

x∧y		y		
		F	U	T
x	F	F	F	F
	U	F	U	U
	T	F	U	T

x∨y		y		
		F	U	T
x	F	F	U	T
	U	U	U	T
	T	T	T	T

Unfortunate consequences

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

Why will the second query return you a smaller number?

- ```
  SELECT * FROM User;
  SELECT * FROM User WHERE pop = pop;
  ```
  - Not equivalent

if pop is null, comparing null with itself will return unknown.

- 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;`
    ```

COALESCE(*arg1, arg2, …*)

- Another special built-in function for handling NULLs
- Accepts any number of arguments, and returns the first one that is not NULL
  - E.g., COALESCE(NULL,'n/a') returns 'n/a'
- Example: find the lowest popularity among the Simpsons, but if there are no Simpsons, return zero instead of NULL
  - ```
    SELECT COALESCE(MIN(pop), 0.0)
    FROM User
    WHERE name LIKE '%Simpson';
    ```

☞COALESCE has a counterpart called NULLIF, which returns NULL if two values are equal

- E.g., NULLIF(*v*,'n/a') returns NULL if *v* equals 'n/a'
    ```

Outerjoin motivation

- Example: a master group membership list with all groups ant 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 definitions & examples

Ask professor if we can bring any written material or if we can have our labtops

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* (denoted $R \text{⟗} S$) 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

Outerjoin definitions & 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

A left outerjoin ($R \mathbin{⟕} S$) 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 ($R \mathbin{⟖} S$) 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;`
- 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
- `WITH`
- `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;`

To remove items from a table, use the DELETE FROM keywords

- 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 examples

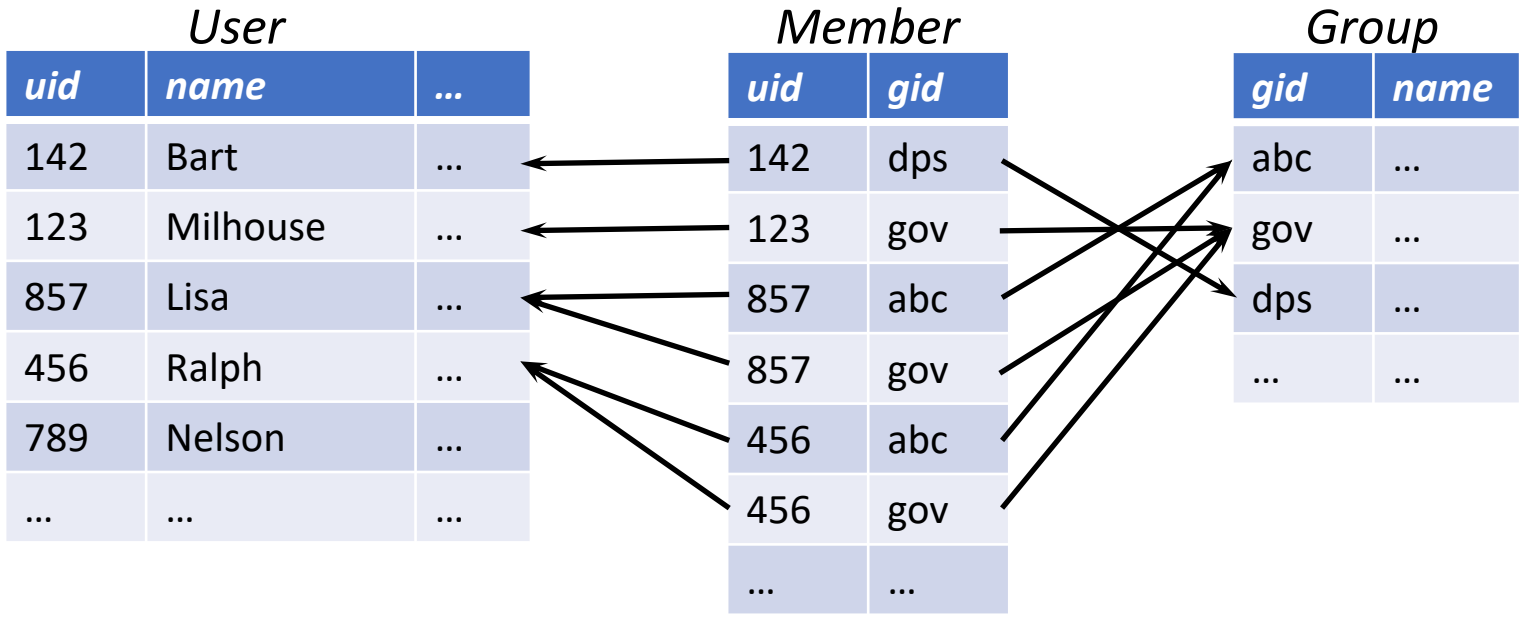
- ```
  CREATE TABLE User
  (uid INTEGER NOT NULL PRIMARY KEY,
   name VARCHAR(30) NOT NULL,
   twitterid VARCHAR(15) NOT NULL UNIQUE,
   age INTEGER,
   pop FLOAT);
  ```
- ```
  CREATE TABLE Group
  (gid CHAR(10) NOT NULL PRIMARY KEY,
   name VARCHAR(100) NOT NULL);
  ```
- ```
  CREATE TABLE Member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

At most one PRIMARY KEY per table

Any number of UNIQUE keys per table

This form is required for multi-attribute keys

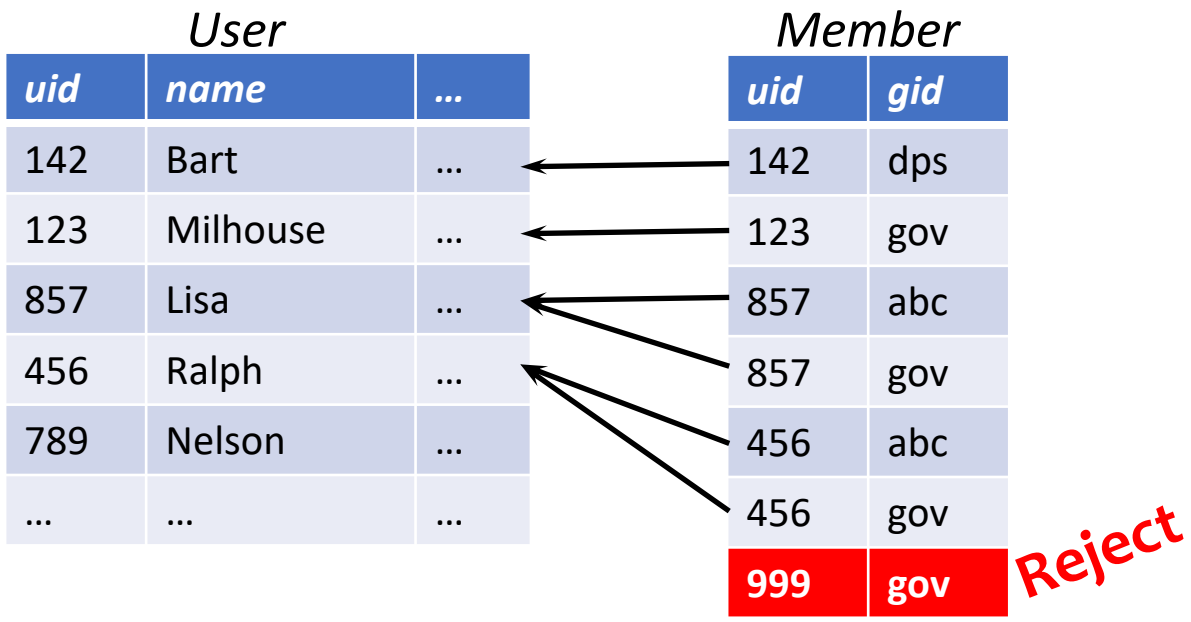
```
CREATE TABLE Member PRIMARY KEY,
(uid INTEGER NOT NULL PRIMARY KEY,
 gid CHAR(10) NOT NULL);
```

Wrong!!
```

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));
    ```

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

This form is useful for multi-attribute foreign keys
```

Enforcing referential integrity

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

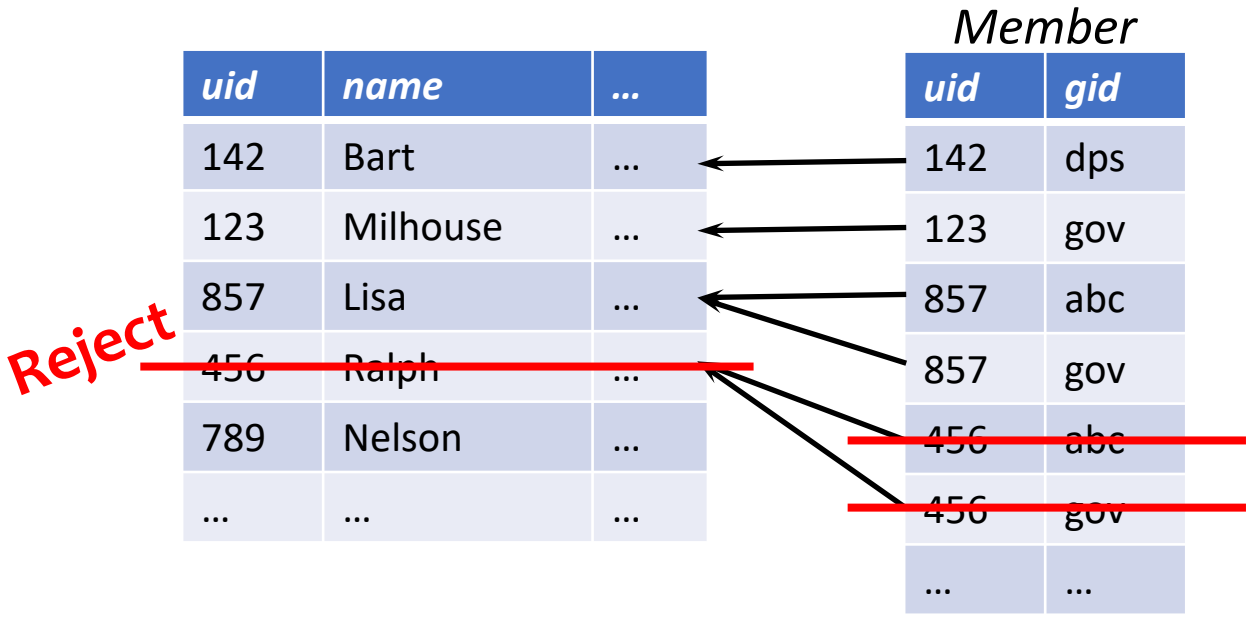
- Insert or update a *Member* row so it refers to a non-existent *uid*
  - Reject



Enforcing referential integrity

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

- Delete or update a *User* row whose *uid* is referenced by some *Member* row
  - Option 1: Reject
  - Option 2: Cascade --- ripple changes to all referring rows

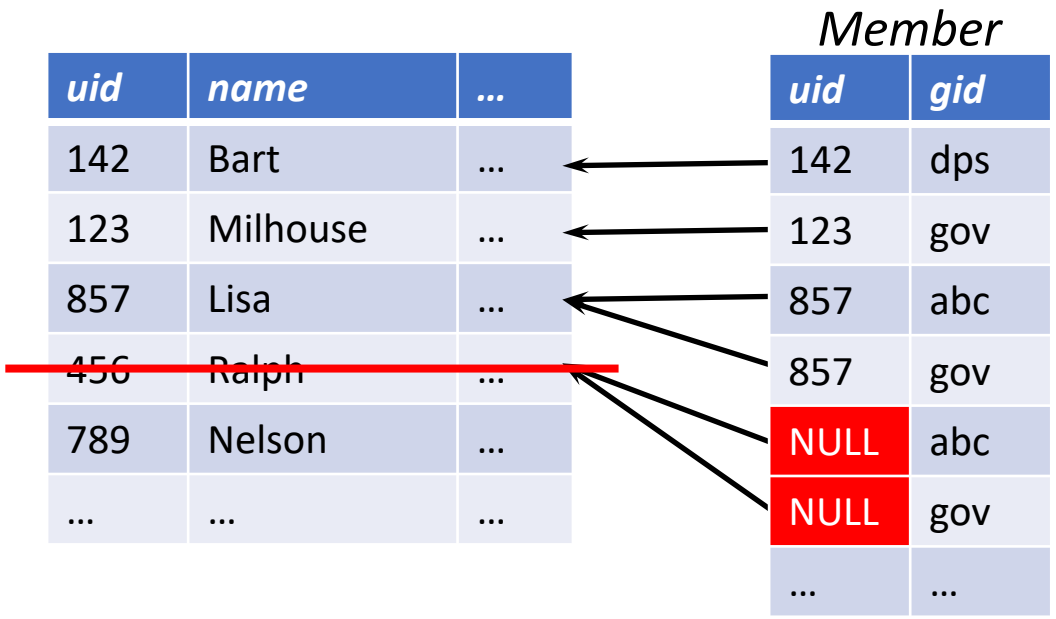



```
CREATE TABLE Member
(uid INT NOT NULL REFERENCES
User(uid) ON DELETE CASCADE,
…..);
```

Enforcing referential integrity

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

- Delete or update a *User* row whose *uid* is referenced by some *Member* row
  - Option 3: Set NULL --- set all references to NULL




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

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
  - Use keyword `deferred`
    ```

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)));
    ```

Can include multiple tables

☞ 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

Schema modification

- How to add constraints once the schema is defined??
  - Add or Modify attributes/domains
    - `ALTER TABLE table_name ADD COLUMN column_name`
    - `ALTER TABLE table_name RENAME COLUMN old_name TO new_name`
    - `ALTER TABLE table_name DROP COLUMN column_name`
    - `ALTER TABLE table_name ALTER COLUMN column_name datatype`
  - Add or Remove constraints
    - `ALTER TABLE Member ADD CONSTRAINT fk_user FOREIGN KEY(uid) REFERENCES User(uid)`
    - `ALTER TABLE Member DROP CONSTRAINT fk_user`

SQL features covered so far

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

☞Next: triggers and views