Lecture 7: SQL (Basic/Advanced)

CS348 Spring 2025:
Introduction to Database Management

Instructor: **Xiao Hu**

Sections: 001, 002, 003

SQL features covered so far

- Basic topics
  - Data-definition language (DDL): define/modify schemas, drop relations
  - Data-manipulation language (DML): query data
    - 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)
    - NULL and JOIN

    and modify data (INSERT/DELETE/UPDATE)

Take home exercises

- What is the output of these queries?

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	Alice	8	NULL

Member

uid	*gid*
857	dps
123	gov
857	abc
123	abc

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

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

```
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';
```

What is next?

- Basic topics
  - Data-definition language (DDL): define/modify schemas, drop relations
  - Data-manipulation language (DML): query data
    - 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)
    - NULL and JOIN

    and modify data (INSERT/DELETE/UPDATE)
  - Constraints

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
- Tuple- and attribute-based CHECK
- General assertion

Example of NOT NULL

```
CREATE TABLE User
(uid INT NOT NULL,
 name VARCHAR(30) NOT NULL,
 twitterid VARCHAR(15),
 age INT NOT NULL,
 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);
```

Incorrect

```
INSERT INTO User (uid, age)
VALUES (389, 18);
```

Incorrect

```
INSERT INTO User VALUES (789,
'Nelson', NULL, NULL, NULL);
```

Examples of PRIMARY KEY

option 1

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

At most one primary key per table

PRIMARY KEY should not contain NULL values

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

option 2

Option 2 is required for multi-attribute keys

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

Incorrect!

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

Examples of UNIQUE

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

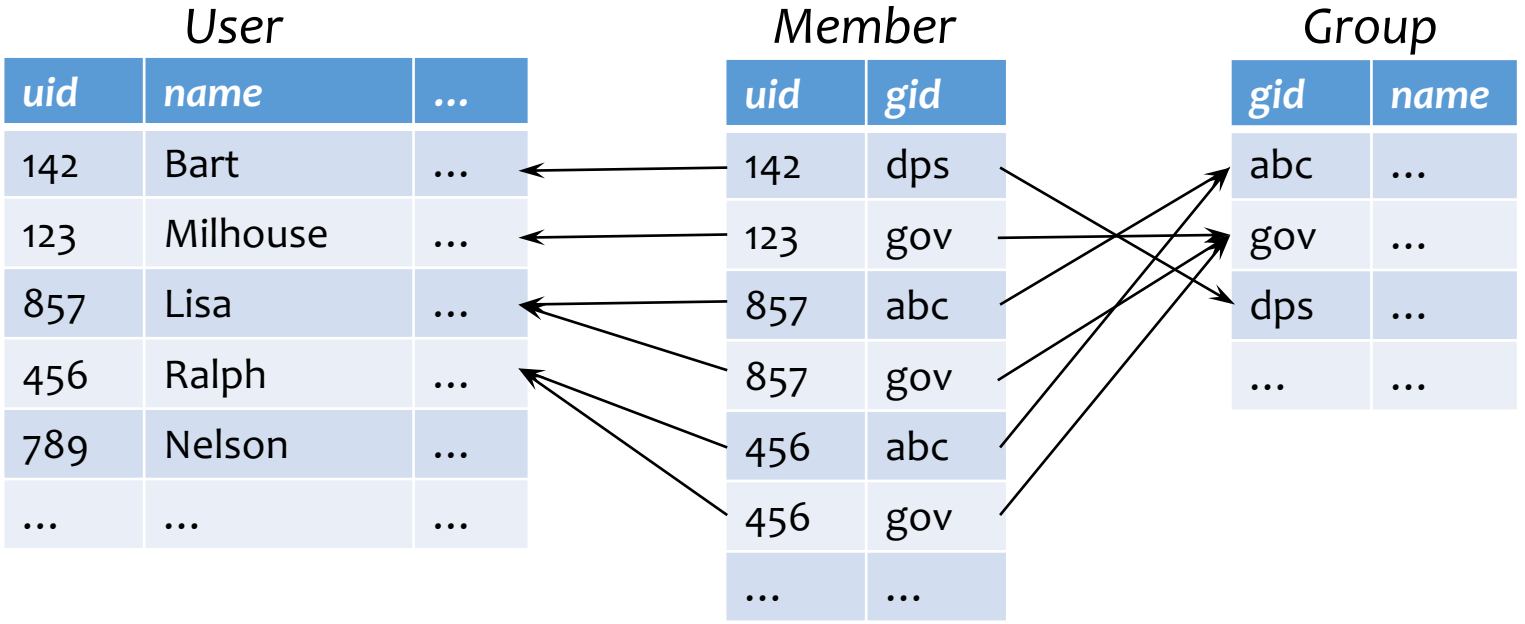
Any number of UNIQUE keys per table

When defining a UNIQUE constraint, NULL values are usually treated as distinct from each other.

(Recap) Referential integrity

- 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
  - Some systems allow both PRIMARY KEY and UNIQUE
- Referencing column(s) form a FOREIGN KEY
- Example

```
CREATE TABLE User (…
uid INT NOT NULL PRIMARY KEY);
```

```
CREATE TABLE Group (…
gid CHAR(10) NOT NULL PRIMARY KEY);
```

```
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));
```

option 1

option 2

Referential integrity in SQL

- Example

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

```
CREATE TABLE MemberBenefits (…
 uid INT NOT NULL REFERENCES Member(uid),
 gid CHAR(10) NOT NULL REFERENCES Member(gid))
```

Are they equivalent?

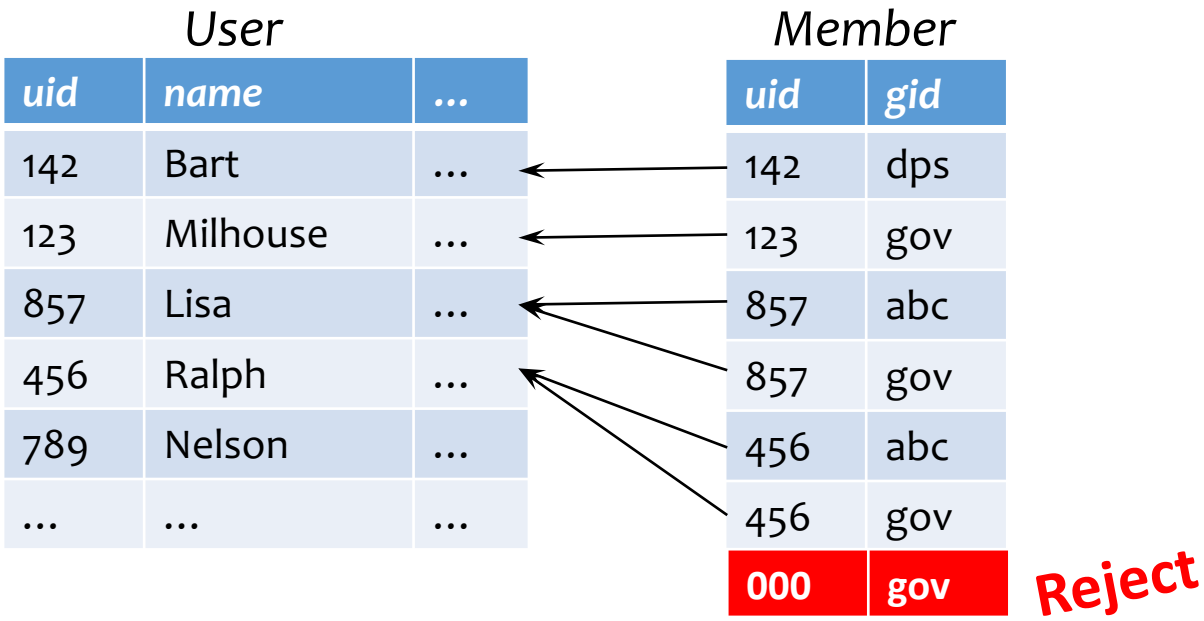
```
CREATE TABLE MemberBenefits (…
 FOREIGN KEY (uid,gid) REFERENCES Member(uid,gid));
```

Option 2 is required for **multi-attribute** foreign keys

Enforcing referential integrity

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

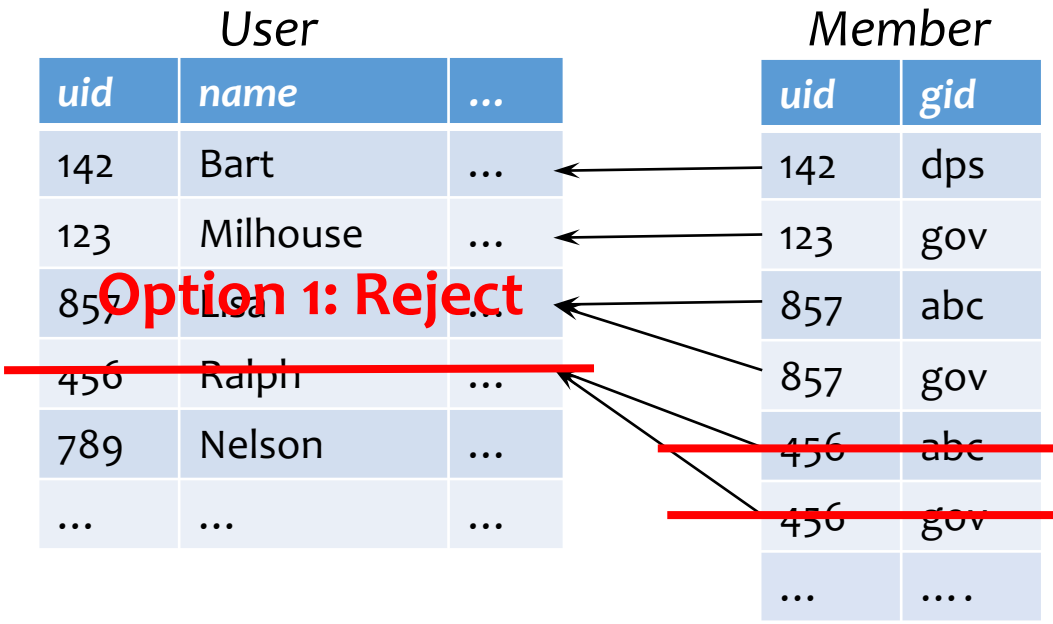
- Insert or update a *Member* row whose uid refers to a non-existent *uid in User*
  - Reject



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)



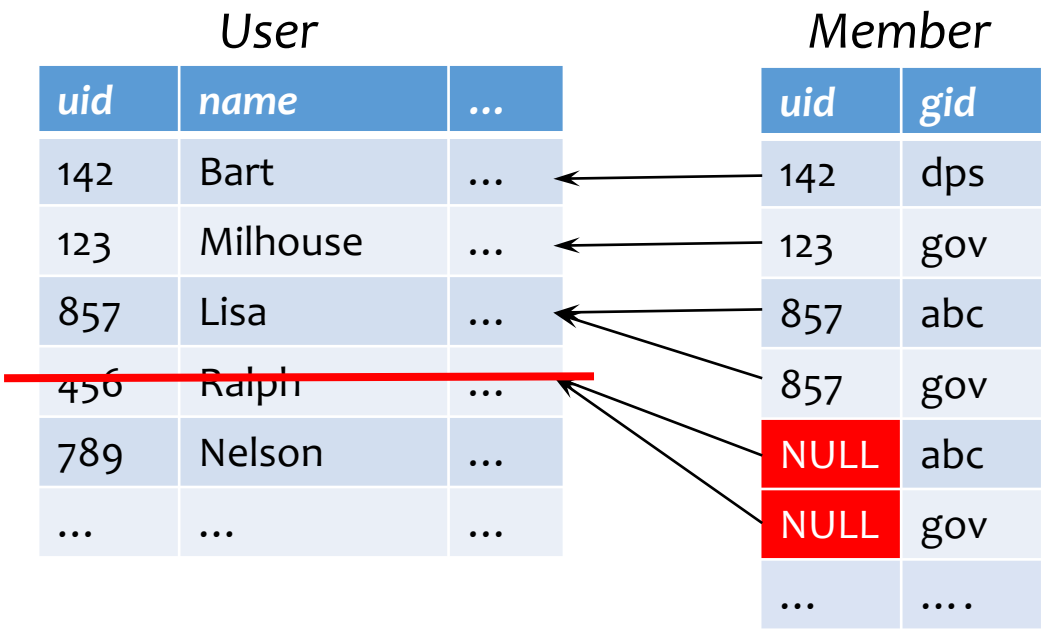

```
CREATE TABLE Member
(uid INT NOT NULL
REFERENCES User(uid)
ON DELETE CASCADE,... );
```

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)

Deferred constraint check (optional)

- 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 set of operations (transactions)
  - Allowed in SQL as an option
  - Use keyword deferred

Tuple- and attribute-based CHECK

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

(Recap) WHERE and HAVING clauses should evaluate to TRUE

- Examples: each user has age above 0 or NULL

How about dropping age IS NULL?

```
CREATE TABLE User(...
 age INTEGER CHECK(age IS NULL OR age > 0), ...);
```

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

Tuple- and attribute-based CHECK

- Examples: if a *uid* appears in *Member*, it must appear in some *User* row

Checked when Member is modified but not when User is modified

```
CREATE TABLE Member
(uid INT NOT NULL,
 CHECK(uid IN (SELECT uid FROM User)), ...);
```

Are they equivalent?

```
CREATE TABLE Member
(uid INT NOT NULL,
 FOREIGN KEY (uid) REFERENCES User(uid));
```

Checked when Member or User is modified

Post-Lecture: General assertion

- Can involve multiple tables!
- CREATE ASSERTION … CHECK *assertion_condition*
- Checked for any modification that could potentially violate it
  - Reject if condition evaluates to FALSE or UNKNOWN
  - TRUE is required
- Example: *Member.uid* references *User.uid*

```
CREATE ASSERTION MemberUserRefIntegrity
CHECK (NOT EXISTS
        (SELECT * FROM Member
         WHERE uid NOT IN
                (SELECT uid FROM User)));
```

Checked when Member or User is modified

CHECK condition must be TRUE!

Modifying constraints

Cannot directly update a constraint itself once it has been created

- Add constraint

```
ALTER TABLE Member
ADD CONSTRAINT fk_user FOREIGN KEY(uid) REFERENCES User(uid);
```

```
ALTER TABLE User
ADD CONSTRAINT chk_pop CHECK (pop >= 0);
```

```
ALTER TABLE User
ADD CONSTRAINT unique_name UNIQUE (name);
```

- Delete constraint

```
ALTER TABLE Member DROP CONSTRAINT fk_user
```

In class exercises

Write a DDL to create the *MemberBenefits* table

- *MemberBenefits* references the *Member*
- (*uid*,*gid*) forms the primary key of *MemberBenefits* table
- Assume discount is of type INT (and *uid* is INT and *gid* is string with a max of 30 characters)

Member

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

MemberBenefits

uid	*gid*	*discount*
857	dps	10
123	gov	25
857	abc	5

```
CREATE TABLE MemberBenefits
(uid INT, gid VARCHAR(30), discount INT,
  PRIMARY KEY (uid,gid),
  FOREIGN KEY (uid,gid) REFERENCES Member(uid,gid));
```

In class exercises

Assume all foreign key references are set to ON DELETE SET NULL (Assume the db allows this, just for this exercise)

- What happens when user 857 is deleted from the *User* table? (Recall *Member* table references *uid* of *User* table)

Member

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

In class exercises

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

- Assume the User table requires *pop* column values to be between 0 and 1. Complete the following DDL statement.

```
CREATE TABLE User
(uid INT PRIMARY KEY,
 name VARCHAR(30),
 age INT,
 pop DECIMAL(3,2) CHECK (0 <= pop AND pop <= 1));
```

In class exercises

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 an 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')));
```

In class exercises

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

- Assume each group contains at least 4 and at most 5 members. Create an assertion to check this constraint.

```
CREATE ASSERTION  group_memberships
CHECK (
  NOT EXISTS (
    SELECT gid FROM Member GROUP BY gid
    HAVING COUNT(*) > 5 or COUNT(*) < 4));
```

Corner case: when Member table is empty, nothing is returned in NOT EXISTS subquery and CHECK condition is true. This is still fine since nothing is stored in Member, hence constraint is preserved automatically.

In class exercises

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

- Assume each group has an average popularity among its members of at least 0.2. Create an assertion to check this constraint.

```
CREATE ASSERTION avg_popularity_threshold
CHECK (
 NOT EXISTS (
   SELECT gid
   FROM Member m JOIN User u ON m.uid = u.uid
   GROUP BY gid
   HAVING AVG(pop) < 0.2));
```

SQL features covered so far

- Basic topics:
  - Data-definition language (DDL): define/modify schemas, drop relations
  - Data-manipulation language (DML): query data
    - 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)
    - NULL and JOIN

    and modify data (INSERT/DELETE/UPDATE)
  - Constraints:
    - (NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK, ASSERTION)

What is next?

- Advanced topics:
  - View

Views

- A view is like a "virtual" table
  - Defined by a query, which describes how to compute the view contents on the fly
  - Stored as a query by DBMS instead of query contents
  - Can be used in queries just like a regular table

```
CREATE VIEW PopGroup AS
SELECT * FROM User
WHERE uid IN (SELECT uid FROM Member
                WHERE gid = 'popgroup');
```

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

```
SELECT AVG(pop) FROM
(SELECT * FROM User
WHERE uid IN (SELECT uid FROM
Member WHERE gid = 'popgroup'))
AS popGroup;
```

```
DROP VIEW popGroup;
```

Post-Lecture: Views and Table Subquery

- By Default, a view is simply a stored query and the query result is NOTphysically stored separately
  - But, systems can choose to materialize view (i..e., physically store the query results)

- A table subquery is a temporary table that physically stores data and it only exists in the duration of the whole query that uses it

Why use views?

- To hide complexity from users
- To hide data from users
- Logical data independence
- To provide a uniform interface

Modifying views

- Does it even make sense, since views are virtual?
- It does make sense if we want users to really see views as tables
- Goal: modify base tables such that the modification would appear to have been done on the view

```
CREATE VIEW UserPop AS SELECT uid, pop FROM User;
```

```
DELETE FROM UserPop WHERE uid = 123;
```

is translated to

```
DELETE FROM User WHERE uid = 123;
```

An impossible case

```
CREATE VIEW PopularUser AS
        SELECT uid, pop FROM User
        WHERE pop >= 0.8;
```

```
INSERT INTO PopularUser VALUES(987, 0.3);
```

- No matter what we do on *User*, the inserted row will not be in *PopularUser*

A case with too many possibilities

```
CREATE VIEW AveragePop(pop) AS
        SELECT AVG(pop) FROM User;
```

Renamed column

```
UPDATE AveragePop SET pop = 0.5;
```

- Set everybody's *pop* to 0.5?
- Adjust everybody's *pop* by the same amount?
- Just lower one user's *pop*?

SQL92 updateable views

- More or less just single-table selection queries
  - No join
  - No aggregation or group by
  - No subqueries
  - Attributes not listed in SELECT must be nullable
- Arguably somewhat restrictive
- Still might get it wrong in some cases
  - See the slide titled “An impossible case”
  - Adding WITH CHECK OPTION to the end of the view definition will make DBMS reject such modifications

Materialized views

- Materialized views: Some systems allow view tables to be physically stored in database
  - If the actual relations used in the view definition change, the view is kept up-to-date
- Used to enhance performance: avoid recomputing the view each time
- View maintenance: updating the materialized view upon base table changes
  - Immediately or lazily, up to the DBMS
  - (Still a very popular research topic, in particular over fully dynamic data)

In class exercises

- What is the output of these queries?

User

uid	*name*	*age*	*pop*
142	Bart	10	0.9
123	Milhouse	10	0.2
857	Lisa	8	0.7
456	Ralph	7	0.3

Member

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

```
CREATE VIEW ageGroups(age,cnt) AS
(SELECT age, COUNT(*) FROM User GROUP BY age)
```

```
SELECT age FROM ageGroups
WHERE cnt = (SELECT MAX(cnt) FROM ageGroups);
```

POST-Lecture

Given any DML that would violate the view's filter

- If WITH CHECK OPTION: reject outright
- If WITH CHECK OPTION is not specified: it is possible to "sneak" rows into the base table through the view -- these rows simply won't appear in the view
  - However, such rows can only be inserted into the base table if they still satisfy the table's constraints

POST-Lecture

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

- Assume there is a CHECK constraint on User table such that age is above 0 and below 140

```
CREATE VIEW youngUsers AS
        (SELECT * FROM User WHERE age < 25) WITH CHECK OPTION;
```

- What happens to the following statements?

```
INSERT INTO youngUsers VALUES (835, 'Alex', 30, 0.2);
```

```
INSERT INTO youngUsers VALUES (923, 'James', 150, 0.3);
```

  - (835, 'Alex', 30, 0.2) will not be inserted into User, and it also will not appear in youngUser
  - Same for (923, 'James', 150, 0.3)

POST-Lecture

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

- Assume there is a CHECK constraint on User table such that age is above 0 and below 140

```
CREATE VIEW youngUsers AS
        (SELECT * FROM User WHERE age < 25);
```

- What happens to the following statements?

```
INSERT INTO youngUsers VALUES (835, 'Alex', 30, 0.2);
```

```
INSERT INTO youngUsers VALUES (923, 'James', 150, 0.3);
```

  - (835, 'Alex', 30, 0.2) can be inserted into User table but it will not appear in youngUser
  - (923, 'James', 150, 0.3) will not be inserted into User (since there is a CHECK constraint on the User table itself) and it also will not appear in youngUser

What is next?

- Basic topics:
  - Data-definition language (DDL)
  - Data-manipulation language (DML)
  - Constraints
- Advanced topics:
  - View
  - Triggers
  - Indexes
  - Recursion
  - Programming (optional; if we have time)