

# SQL: Part II

CS348 Spring 2023

Instructor: Sujaya Maiyya

Sections: **002 and 004 only**

# Announcements

- Assignment 1 is released: Due **May 30<sup>th</sup>**
- Project description is released
  - Milestone 0: not graded but due on **May 25<sup>th</sup>**
- **No class next Tuesday**, May 23<sup>rd</sup> (Monday schedule)

# 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







# 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
  - 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*)
  - $\langle 789, \text{"Nelson"}, \text{NULL}, \text{NULL} \rangle$

Truth table?



# Three-valued logic

TRUE = 1, FALSE = 0, UNKNOWN = 0.5

$x \text{ AND } y = \min(x, y)$

$x \text{ OR } y = \max(x, y)$

$\text{NOT } x = 1 - x$

$x$	$y$	$x \text{ AND } y$	$x \text{ OR } y$	$\text{NOT } x$
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(\*)**

# 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;
```

# 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 ⋈ Member

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

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  $\bowtie$  Member

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

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** ( $R \bowtie S$ ) includes rows in  $R \bowtie S$  plus dangling  $R$  rows padded with NULL's

Group  $\bowtie$  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;
```

$$\approx \text{Group} \bowtie_{\text{Group.gid}=\text{Member.gid}} \text{Member}$$

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

$$\approx \text{Group} \bowtie_{\text{Group.gid}=\text{Member.gid}} \text{Member}$$

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

$$\approx \text{Group} \bowtie_{\text{Group.gid}=\text{Member.gid}} \text{Member}$$

☞ 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;
```

# 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 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

- 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);
```

This form is  
required for multi-  
attribute keys

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

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

**Incorrect!**





# 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**

User			Member	
<i>uid</i>	<i>name</i>	...	<i>uid</i>	<i>gid</i>
142	Bart	...	142	dps
123	Milhouse	...	123	gov
857	Lisa	...	857	abc
456	Ralph	...	857	gov
789	Nelson	...	456	abc
...	...	...	456	gov
			000	gov

**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)

User			Member	
uid	name	...	uid	gid
142	Bart	...	142	dps
123	Milhouse	...	123	gov
857	Lea	...	857	abc
<del>456</del>	<del>Ralph</del>	<del>...</del>	857	gov
789	Nelson	...	<del>456</del>	<del>abc</del>
...	...	...	<del>456</del>	<del>gov</del>
			...	....

**Option 1: Reject**

```
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)

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

```
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 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 set of operations (transactions)
  - 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*

Can include multiple tables

```
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)),  
  ...);
```

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),  
  ...);
```



# Schema modification

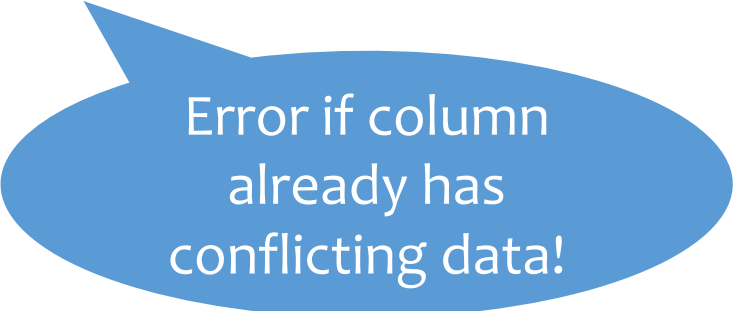
- How to add constraints once the schema is defined??
- Add or Modify attributes/domains
- Add or Remove constraints

# 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

Domain change:

- *Alter table* table\_name *Alter column* column\_name datatype



Error if column  
already has  
conflicting data!

# Add or Remove constraints

- *Alter table* `table_name` *Add constraint*  
`constraint_name constraint_condition`

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

- *Alter table* `table_name` *Drop constraint*  
`constraint_name`

```
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
    - Outerjoins (and NULL)
  - Modification
    - INSERT/DELETE/UPDATE
  - Constraints
- ☞ Next lecture: triggers, views, indexes

# Two ways to practice queries

- School servers have db2 installed
  - Instructions in db2tutorial.pdf posted along with the project description
  - The JDBC example also provides instructions for the same
- The textbook's website has an SQLite db that runs in the browser: <https://www.db-book.com/university-lab-dir/sqljs.html>