Lecture 6
SQL (Basic)

CS348 Spring 2025:
Introduction to Database Management

Instructor: **Xiao Hu**

Sections: 001, 002, 003

Announcements

- Project Milestone 0 Due today
  - Not be graded
- Deadline extended for Assignment 1
  - From Thursday May 29 to Sunday Jun 1
- Online office hours by IAs for Assignment 1
  - See Piazza for Zoom information
  - Friday May 23 4pm - 5pm
  - Wednesday May 28 4:30 pm – 5:30pm

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)

Take home exercises

- Using EXISTS, write a query to list the distinct IDs of groups that have at least two users

```
SELECT DISTINCT gid FROM Member m
WHERE EXISTS (SELECT m1.gid
                FROM Member m1
                WHERE m.gid = m1.gid and m.uid != m1.uid);
```

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

Take home exercises

- Using WITH-AS and (NOT) IN, write a query to list the IDs of groups that Lisa belongs to but Ralph does not

```
WITH temp AS (SELECT gid
                    FROM User u, Member m
                    WHERE u.name = 'Ralph' and u.uid = m.uid)

SELECT DISTINCT gid
FROM User u, Member m
WHERE name = 'Lisa'  AND u.uid = m.uid AND
                                gid NOT IN (SELECT gid FROM temp);
```

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

Take home exercises

- Write the same query but using EXCEPT (you may or may not use any other keywords)

```
SELECT gid FROM User u, Member m
WHERE u.name = 'Lisa' AND u.uid = m.uid
EXCEPT
SELECT gid FROM User u, Member m
WHERE u.name = 'Ralph' AND u.uid = m.uid;
```

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

What is next?

- Basic topics
  - Data-definition language (DDL): define/modify schemas, delete 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)

☞Next: NULL and JOIN

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 their *pop*?

Possible 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*

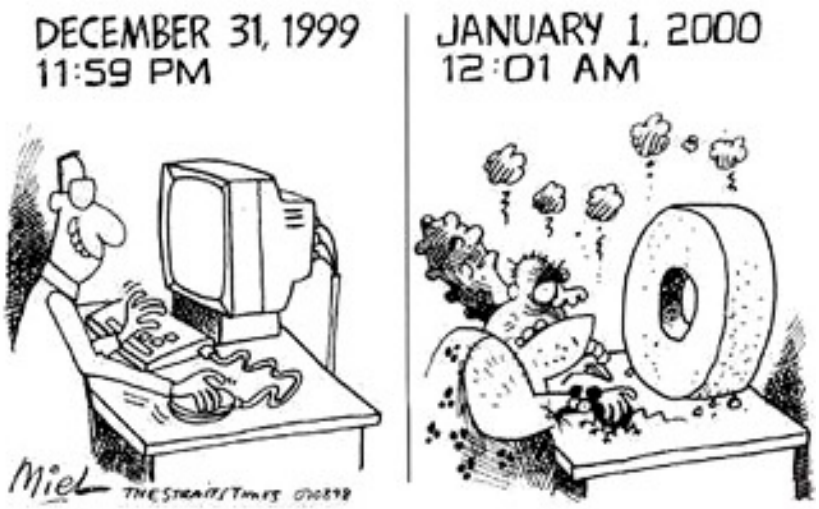
```
SELECT AVG(pop) FROM User;
```

```
SELECT AVG(pop) FROM User WHERE pop != -1;
```

Incorrect answers

Complicated

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



http://www.90s411.com/images/y2k-cartoon.jpg

Possible 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

Possible Solution 3

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

- 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

Other Rules

- 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
  - FALSE and UNKNOWN are not sufficient

- Aggregate functions ignore NULL, except COUNT(*)
  - SUM, AVG, MIN, MAX all ignore NULLs
  - COUNT(age) also ignores NULL
  - If all inputs are NULL, SUM, AVG, MIN, MAX all return NULL

Will 789 be in the output?

⟨789, “Nelson”, NULL, NULL⟩

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

```
SELECT uid FROM User WHERE name=‘Nelson’ OR pop>0.5;
```

```
SELECT uid FROM User WHERE NOT 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;
```

GROUP BY with NULL

- In a GROUP BY clause, any NULL values in the grouping column(s) are treated all the same
  - they end up in one "NULL" group

```
SELECT age, AVG(pop), COUNT(*)
FROM User
GROUP BY age;
```

age	AVG(pop)	COUNT(*)
8	0.7	3
NULL	0.6	2

User

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

- If a group has only NULLs, then aggregate function (except COUNT(*)) returns NULL.

In class exercises

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

- 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
WHERE age IS NOT NULL
GROUP BY age;
```

User

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

In class exercises

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

- What is the output of these queries?

```
SELECT name
FROM User
WHERE age IN (SELECT age FROM
User);
```

```
SELECT name
FROM User
WHERE age <= ANY(SELECT age
FROM User) AND pop IS NOT NULL;
```

```
SELECT name
FROM User
WHERE age <= ALL(SELECT age
FROM User);
```

User

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

In class exercises

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

- Write a query to find the ID of all users with non-null popularity who belong to at least one group.

```
SELECT uid FROM User
WHERE pop IS NOT NULL
AND uid IN (SELECT DISTINCT uid FROM Member);
```

JOIN

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

- Example: construct a group membership list with all groups and its members info (if non-empty)

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

```
SELECT g.gid, gname, u.uid, uname
FROM Group g JOIN Member m ON g.gid =
m.gid JOIN User u ON m.uid = u.uid;
```

```
SELECT g.gid, gname, u.uid, uname
FROM Group g, Member m, User u
WHERE g.gid = m.gid AND m.uid = u.uid;
```

Need a new JOIN operator

- Example: construct a master group membership list with all groups and its members info ~~(if non-empty)~~

```
SELECT g.gid, gname, u.uid, uname
FROM Group g JOIN Member m ON g.gid = m.gid JOIN User u ON
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, user id and user name would be NULL

Outerjoin

- An extended relational algebra operator
- A full outerjoin ($R ⟗ S$) includes:
  - All rows in the natural join 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

Example of Outerjoin

Group

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

Left/Right Outerjoin

Group

gid	*gname*
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	*gname*	*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	*gname*	*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 in SQL

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

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

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

$\approx Group \bowtie Member$

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

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

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

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

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

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

In class exercises

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

- 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

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

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

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

SQL features covered so far

- Basic topics
  - Data-definition language (DDL): define/modify schemas, delete 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

☞ Next: Modify data (INSERT/DELETE/UPDATE)

INSERT

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

- 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

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

- Delete everything from a table

```
DELETE FROM Member;
```

- Delete according to a WHERE condition
  - User 789 leaves Dead Putting Society

```
DELETE FROM Member WHERE uid=789 AND gid='dps';
```

  - Users over age 18 must be removed from Book Club

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

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

UPDATE

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

- 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

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