SQL: Part II

CS348

Instructor: Sujaya Maiyya

SQL features covered so far

- SELECT-FROM-WHERE statements
- Set and bag operations
- Subqueries
  - Subqueries allow queries to be written in more declarative ways (recall the "most popular" query)
  - But in many cases, they don't add expressive power

☞ Next: aggregation and grouping

Aggregates

- Standard SQL aggregate functions: COUNT, SUM, AVG, MIN, MAX

- Example: number of users under 18, and their average popularity
  - COUNT(*) counts the number of rows

```
SELECT COUNT(*), AVG(pop)
FROM User
WHERE age <18;
```

COUNT (*)	AVG (pop)
6	0.625

Aggregates with DISTINCT

- Example: How many users belong to groups?

```
SELECT COUNT(*)
FROM (SELECT DISTINCT uid FROM Member);
```

Is equivalent to

```
SELECT COUNT(DISTINCT uid)
FROM Member;
```

Grouping

- SELECT … FROM … WHERE …
  GROUP BY *list_of_columns*;

- Example: compute average popularity for each age group

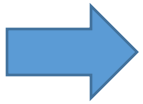
```
SELECT age, AVG(pop)
FROM User
GROUP BY age;
```

Example of computing GROUP BY

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

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

Compute GROUP BY: group rows according to the values of GROUP BY columns



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

Compute SELECT for each group

age	*avg_pop*
10	0.55
8	0.50

Semantics of GROUP BY

SELECT … FROM … WHERE … GROUP BY …;

1. Compute FROM ($\times$)
2. Compute WHERE (σ)
3. Compute GROUP BY: group rows according to the values of GROUP BY columns
4. Compute SELECT for each group (π)
   - For aggregation functions with DISTINCT inputs, first eliminate duplicates within the group

☞ Number of groups =
number of rows in the final output

Aggregates with no GROUP BY

- An aggregate query with no GROUP BY clause = all rows go into one group

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

Group all rows into one group

Aggregate over the whole group

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

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

avg_pop
0.525

Restriction on SELECT

- If a query uses aggregation/group by, then every column referenced in SELECT must be either
  - Aggregated, or
  - A GROUP BY column

Why?

☞This restriction ensures that any SELECT expression produces only one value for each group

```
SELECT uid, age FROM User GROUP BY age;
```

WRONG!

```
SELECT uid, MAX(pop) FROM User;
```

WRONG!

HAVING

- Used to filter groups based on the group properties (e.g., aggregate values, GROUP BY column values)

- SELECT … FROM … WHERE … GROUP BY … HAVING $condition$;
  1. Compute FROM ($\times$)
  2. Compute WHERE (σ)
  3. Compute GROUP BY: group rows according to the values of GROUP BY columns
  4. Compute HAVING (another σ over the groups)
  5. Compute SELECT (π) for each group that passes HAVING

HAVING examples

- Find average popularity for each age group over 10

```
SELECT age, AVG(pop)
FROM User
GROUP BY age
HAVING age >10;
```

  - Can be written using WHERE without table subqueries

```
SELECT age, AVG(pop)
FROM User
WHERE age >10
GROUP BY age;
```

HAVING examples

- List the average popularity for each age group with more than a hundred users

```
SELECT age, AVG(pop)
FROM User
GROUP BY age
HAVING COUNT(*)>100;
```

  - Can be written using WHERE and table subqueries

```
SELECT T.age, T.apop
FROM (SELECT age, AVG(pop) AS apop, COUNT(*) AS gsize
        FROM User GROUP BY age) AS T
WHERE T.gsize>100;
```

SQL features covered so far

- SELECT-FROM-WHERE statements
- Set and bag operations
- Subqueries
- Aggregation and grouping
  - More expressive power than relational algebra

☞Next: ordering output rows

ORDER BY

- SELECT [DISTINCT] …
  FROM … WHERE … GROUP BY … HAVING …
  ORDER BY *output_column* [ASC|DESC], …;

- ASC = ascending, DESC = descending

- Semantics: After SELECT list has been computed and optional duplicate elimination has been carried out, sort the output according to ORDER BY specification

ORDER BY example

- List all users, sort them by popularity (descending) and name (ascending)

```
SELECT uid, name, age, pop
FROM User
ORDER BY pop DESC, name;
```

  - ASC is the default option
  - Strictly speaking, only output columns can appear in ORDER BY clause (although some DBMS support more)
  - Can use sequence numbers instead of names to refer to output columns: ORDER BY 4 DESC, 2;

Discouraged: hard to read!

LIMIT

- The LIMIT clause specifies the number of rows to return

- E.g., Return top 3 users with highest popularities

```
SELECT uid, name, age, pop
FROM User
ORDER BY pop DESC
LIMIT 3;
```

SQL features so far

- 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)
  - Missing values

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*

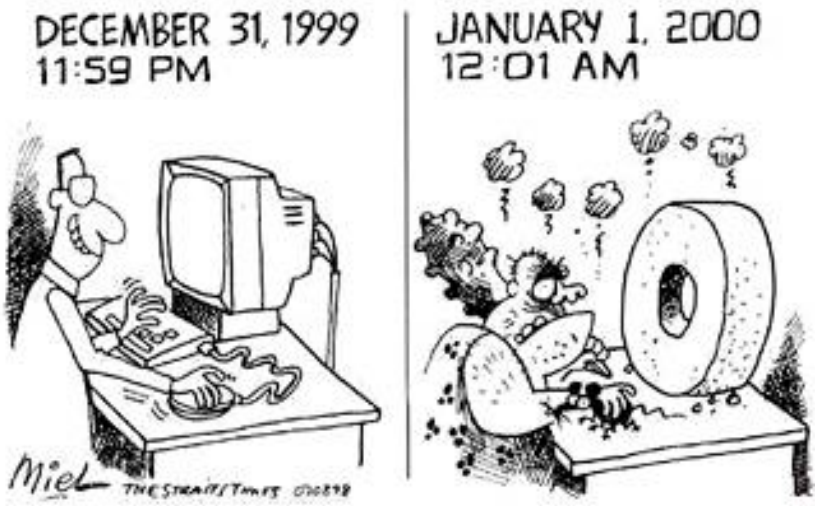
```
SELECT AVG(pop) FROM User;
```

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.

SQL features so far

- 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)
  - Missing values
- Outerjoins
- Modification
  - INSERT/DELETE/UPDATE
- Constraints