SQL: Part II

CS348 Spring 2024

Instructor: Sujaya Maiyya

Sections: **002 and 003 only**

Announcements

- Assignment 1 is released: Due June 4th
- Project description is released
  - Milestone 0: not graded but due on May 23rd
- **No class next Tuesday**, May 21st (Monday schedule)

SQL features covered so far

- Query
  - SELECT-FROM-WHERE statements
  - Set/bag (DISTINCT, UNION/EXCEPT/INTERSECT (ALL))

☞Next: practice questions

☞ Nested queries

☞Aggregation and grouping

☞Ordering and limiting

In class exercises

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

- List user names whose popularity is b/w 0.5 and 0.9

```
SELECT name FROM User where pop > 0.5 or pop < 0.9;
```

- List the group ids that a user with id 134 belongs to

```
SELECT gid FROM Member where uid=134;
```

- List the group ids that Lisa belongs to

```
SELECT gid FROM Member m, User u where u.name='Lisa' and m.uid=u.uid;
```

In class exercises

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

- List the group names that Lisa belongs to

```
SELECT g.name
FROM Member m, User u, Group g
WHERE u.name='Lisa' and m.uid=u.uid and m.gid = g.gid;
```

- List user ids belonging to at least 2 groups

```
SELECT m1.uid
FROM Member m1, Member m2
WHERE m1.uid=m2.uid and m1.gid != m2.gid;
```

In class exercises

Consider this db instance:

User

uid	name	age	pop
142	Bart	10	0.9
123	Milhouse	10	0.2
857	Lisa	8	0.7
456	Ralph	8	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 gid FROM Member m, User u where u.name='Lisa' and u.uid=m.uid
```

```
SELECT gid FROM Member m, User u where u.name='Lisa' and u.uid=m.uid
UNION
SELECT gid FROM Member m, User u where u.name='Ralph' and u.uid=m.uid
```

```
SELECT gid FROM Member m, User u where u.name='Lisa' and u.uid=m.uid
UNION ALL
SELECT gid FROM Member m, User u where u.name='Ralph' and u.uid=m.uid
```

SQL features covered so far

- Query
  - SELECT-FROM-WHERE statements
  - Set/bag (DISTINCT, UNION/EXCEPT/INTERSECT (ALL))

☞Next: how to nest SQL queries

Table subqueries

- Use query result as a table
  - In set and bag operations, FROM clauses, etc.

- Example: names of users belonging to at least two groups

```
SELECT DISTINCT name
FROM User,
     (SELECT m1.uid
     FROM Member m1, Member m2
     WHERE m1.uid=m2.uid and m1.gid != m2.gid)
     AS T
WHERE User.uid = T.uid;
```

Scalar subqueries

- A query that returns a single row can be used as a value in WHERE, SELECT, etc.
- Example: users at the same age as Bart (uid=142)

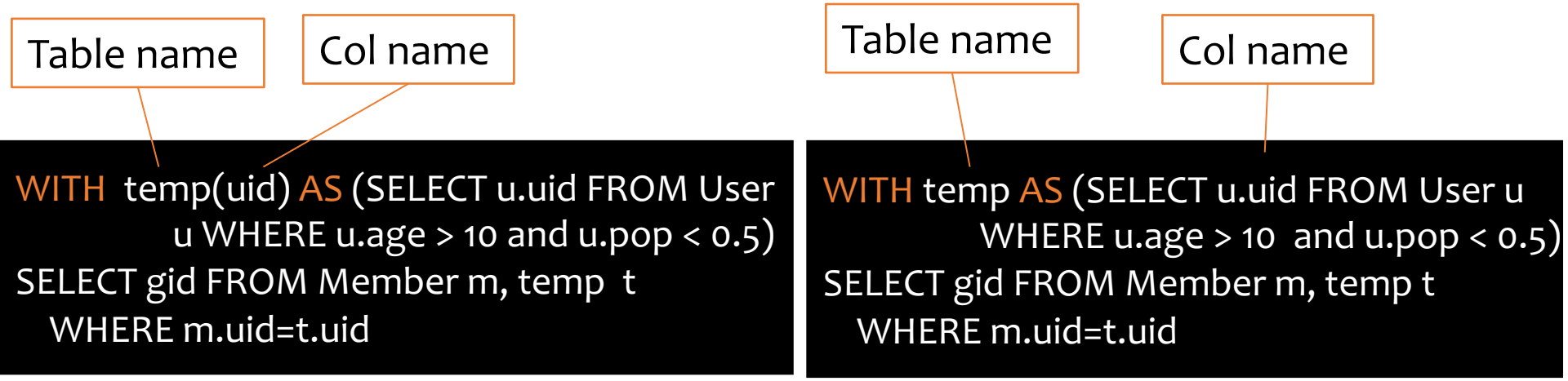
```
SELECT *
FROM User,
WHERE age = (SELECT age
             FROM User
             WHERE uid = 142);
```

- When can this query go wrong?
  - Return more than 1 row (WHERE name = 'Bart')
  - Return no rows

WITH clause

- WITH clause provides a way of defining a temporary relation whose definition is available only to the query in which the with clause occurs

- Ex: List group ids of users with age > 10 and pop < 0.5



- Supported by many but not all DBMSs
- Can be written using subqueries

IN subqueries

- x IN $(subquery)$ checks if x is in the result of $subquery$

- Example: users that have the same age as (some) Bart

```
SELECT *
FROM User,
WHERE age IN (SELECT age
              FROM User
              WHERE name = 'Bart');
```

EXISTS subqueries

- EXISTS ($subquery$) checks if the result of $subquery$ is non-empty

- Example: users that have the same age as (some) Bart

```
SELECT *
FROM User AS u,
WHERE EXISTS (SELECT * FROM User
                WHERE name = 'Bart'
                  AND age = u.age);
```

  - This happens to be a correlated subquery—a subquery that references tuple variables in surrounding queries

Quantified subqueries

- Universal quantification (for all):
  - ... WHERE $x\ op$ ALL($subquery$) ...
  - True iff for all t in the result of $subquery$, $x\ op\ t$

```
SELECT *
FROM User
WHERE pop >= ALL(SELECT pop FROM User);
```

- Existential quantification (exists):
  - ... WHERE $x\ op$ ANY($subquery$) ...
  - True iff there exists some t in $subquery$ result s.t. $x\ op\ t$

```
SELECT *
FROM User
WHERE NOT
   (pop < ANY(SELECT pop FROM User));
```

More ways to get the most popular

- Which users are the most popular?

```
Q1. SELECT *
FROM User
WHERE pop >= ALL(SELECT pop FROM User);
```

```
Q2. SELECT *
FROM User
WHERE NOT
   (pop < ANY(SELECT pop FROM User);
```

EXISTS or IN?

```
Q3. SELECT *
FROM User AS u
WHERE NOT [EXISTS or IN?]
   (SELECT * FROM User
    WHERE pop > u.pop);
```

```
Q4. SELECT * FROM User
WHERE uid NOT [EXISTS or IN?]
   (SELECT u1.uid
    FROM User AS u1, User AS u2
    WHERE u1.pop < u2.pop);
```

In class exercises

Consider this db instance:

User

uid	name	age	pop
142	Bart	10	0.9
123	Milhouse	10	0.2
857	Lisa	8	0.7
456	Ralph	8	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 name FROM User WHERE age <=ALL(SELECT age FROM User)
```

```
SELECT name FROM User WHERE pop < ANY (SELECT pop FROM User)
```

```
WITH temp AS (SELECT uid FROM User WHERE pop < ANY (
                                        SELECT pop FROM User))
SELECT name FROM User WHERE uid NOT IN (SELECT uid FROM temp)
```

```
SELECT uid FROM User u WHERE EXISTS (SELECT gid FROM Member m
                                     WHERE m.uid = u.uid)
```

Take home exercises

- Using EXISTS, write a query to list user ids belonging to at least 2 groups
- Using WITH-AS and (NOT) IN, write a query to list group ids that Lisa belongs to but Ralph does not
- Write the same query but using EXCEPT (you may or may not use any other keywords)

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

- 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;
```

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;
```

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;
```

In class exercises

Consider this db instance:

User

uid	name	age	pop
142	Bart	10	0.9
123	Milhouse	10	0.2
857	Lisa	8	0.7
456	Ralph	7	0.6

Member

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

- What is the output of these queries?

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

```
SELECT AVG(pop)  AS apop FROM User GROUP BY age
        HAVING age>5  ORDER BY  apop LIMIT 2;
```

```
SELECT AVG(pop)  AS apop FROM User GROUP BY age
        HAVING COUNT(*) >=2  ORDER BY  apop LIMIT 2;
```

```
WITH temp AS (SELECT uid, COUNT(*) AS cnt FROM Member GROUP BY uid )
SELECT name FROM User u, temp t WHERE t.uid = u.uid and
        t.cnt = (SELECT MAX(cnt) FROM temp)
```

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)
  - Outerjoins (and Nulls)
- Modification
  - INSERT/DELETE/UPDATE
- Constraints

Lecture 5

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