CS 348 Lecture 5

SQL Part 2

Semih Salihoğlu

Jan 20th, 2025

UNIVERSITY OF WATERLOO | DSg Data Systems Group

Announcements

- Project Milestone 0: due Jan 22nd !

Recap: 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

Recap: 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;

LIMIT

- The LIMIT clause specifies the number of rows to return
- ORDER BY + LIMIT: useful template for "top-k" (or "bottom-k") queries

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

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

- OFFSET: Many systems have an OFFSET clause to skip some number of rows before outputting

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 5

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

  - Not recommended for 2 reasons:
    - Hard to find a value for each data type (e.g., what to use for booleans)
  - Not universal: Many options exist and can be confusing to other people co-developing applications!
    - For numeric columns: highest, lowest, 0, or a value < 0?
    - For string columns: “Nil”, “nil”, “none”, “n/a”?

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

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

Can think of this equivalently as follows:

Let TRUE = 1, FALSE = 0, UNKNOWN = 0.5

Then:

x AND $y = \min(x, y)$

x OR $y = \max(x, y)$

NOT $x = 1 - x$

- 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;
```

Takehome ex.

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.

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

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

A full outer (natural) join 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

Similar definition for outer theta joins => what is supported in SQL

Outerjoin examples

Group

gid	name
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	name	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	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

Outer (theta) join syntax

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

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

```
SELECT * FROM User LEFT OUTER JOIN Member
        ON User.uid = Member.uid AND pop < 0.5;
```

$$\approx User \underset{User.uid=Member.uid\ AND\ pop<0.5}{\bowtie} Member$$

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

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

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

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

Natural Outer Join syntax

☞For natural joins, add keyword NATURAL; don't use ON

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


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

Inner join syntax

- Normal or "inner" join: instead of OUTER JOIN:
  - use just JOIN or INNER JOIN keywords

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

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

- Similarly for natural (inner) joins, add keyword NATURAL; don't use ON

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

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

Two orthogonal aspects of joins

1. Join Predicate: Natural vs Theta
   - Natural: join predicate is equality of common attributes
   - Theta: arbitrary predicate
2. Dangling tuples: Inner vs Outer
   - Inner: ignore dangling tuples
   - Left/Right/Full Outer: keep dangling tuples (left, right or both)

- You need to specify bothusing the following clauses this order:
  1. NATURAL vs Theta: if NATURAL keyword is used then natural, otherwise theta
  2. INNER vs LEFT/RIGHT/FULL OUTER: if omitted, inner is assumed

  Note if theta, after INNER or LEFT/RIGHT/FULL OUTER clauses, we need an ON

  E.g: This is not correct syntax:

```
SELECT * FROM Group LEFT OUTER JOIN Member;
```

Exercises

Consider this db instance:

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	Ralph	8	NULL

Member

uid	gid
857	dps
123	gov
857	abc
123	abc

- What is the output of these queries?

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

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

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

SQL features covered so far

- SELECT-FROM-WHERE statements
- Set and bag operations
- 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 over age 18 must be removed from Sports Club

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

- Some systems allow "deletions with joins in the FROM" clause. Check your systems' documentation.

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

Exercise

Consider this db instance:

Group

gid	*name*
abc	Book Club
gov	Student Government
dps	Dead Putting Society
spr	Sports Club

User

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

Member

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

- What is the output of this queries?

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

SQL features covered so far

- Query
  - SELECT-FROM-WHERE statements
  - Set and bag operations
  - Subqueries
  - Aggregation and grouping
  - Ordering
  - Outerjoins (and NULL)
- Modification
  - INSERT/DELETE/UPDATE

☞Next lectures: Constraints, schema changes, views, indexes