SQL III

R & G - Chapter 5

A Tough One: “Division”

- Relational Division: “Find sailors who’ve reserved all boats.” Said differently: “sailors with no counterexample missing boats”

```
SELECT S.sname
FROM Sailors S
WHERE NOT EXISTS
  (SELECT B.bid
  FROM Boats B
  WHERE NOT EXISTS (SELECT R.bid
                  FROM Reserves R
                  WHERE R.bid=B.bid
                  AND R.sid=S.sid ))
```

ARGMAX? Pt 1

- The sailor with the highest rating
- Correct or Incorrect?

```
SELECT MAX(S.rating)
FROM Sailors S;
```

VS

```
SELECT S.*, MAX(S.rating)
FROM Sailors S;
```

ARGMAX? Pt 2

- The sailor with the highest rating
- Correct or Incorrect? Same or different?

```
SELECT *
FROM   Sailors S
WHERE  S.rating >= ALL
 (SELECT  S2.rating
 FROM  Sailors S2)
```

VS

```
SELECT *
FROM   Sailors S
WHERE  S.rating =
  (SELECT  MAX(S2.rating)
  FROM  Sailors S2)
```

ARGMAX? Pt 3

- The sailor with the highest rating
- Correct or Incorrect? Same or different?

```
SELECT *
FROM   Sailors S
WHERE  S.rating >= ALL
  (SELECT  S2.rating
  FROM  Sailors S2)
```

VS

```
SELECT *
FROM   Sailors S
ORDER BY rating DESC
LIMIT 1;
```

“Inner” Joins: Another Syntax

```
SELECT s.*, r.bid
FROM Sailors s, Reserves r
WHERE s.sid = r.sid
AND ...
```

```
SELECT s.*, r.bid
FROM Sailors s INNER JOIN Reserves r
ON s.sid = r.sid
WHERE ...
```

Join Variants

```
SELECT <column expression list>
FROM table_name
 [INNER | NATURAL
  | {LEFT |RIGHT | FULL } {OUTER}] JOIN
 table_name
 ON <qualification_list>
WHERE …
```

- INNER is default
- Inner join what we’ve learned so far
  - Same thing, just with different syntax.

Inner/Natural Joins

```
SELECT s.sid, s.sname, r.bid
FROM Sailors s, Reserves r
WHERE s.sid = r.sid
 AND s.age > 20;
```

```
SELECT s.sid, s.sname, r.bid
FROM Sailors s INNER JOIN Reserves r
ON s.sid = r.sid
WHERE s.age > 20;
```

```
SELECT s.sid, s.sname, r.bid
FROM Sailors s NATURAL JOIN Reserves r
WHERE s.age > 20;
```

- **ALL 3 ARE EQUIVALENT!**
- “NATURAL” means equi-join for pairs of attributes with the same name

Left Outer Join

- Returns all matched rows, and *preserves* all unmatched rows from the table on the left of the join clause
  - (use nulls in fields of non-matching tuples)

```
SELECT s.sid, s.sname, r.bid
FROM Sailors2 s LEFT OUTER JOIN Reserves2 r
ON s.sid = r.sid;
```

Returns all sailors & bid for boat in any of their reservations

Note: no match for s.sid? r.bid IS NULL!

Right Outer Join

- Returns all matched rows, and *preserves* all unmatched rows from the table on the right of the join clause
  - (use nulls in fields of non-matching tuples)

```
SELECT r.sid, b.bid, b.bname
FROM Reserves2 r RIGHT OUTER JOIN Boats2 b
ON r.bid = b.bid
```

Returns all boats and sid for any sailor associated with the reservation.

Note: no match for b.bid? r.sid IS NULL!

Full Outer Join

- Returns all (matched or unmatched) rows from the tables on both sides of the join clause

```
SELECT r.sid, b.bid, b.bname
FROM Reserves2 r FULL OUTER JOIN Boats2 b
ON r.bid = b.bid
```

- Returns all boats & all information on reservations
- No match for r.bid?
  - b.bid IS NULL AND b.bname IS NULL!
- No match for b.bid?
  - r.sid IS NULL!

Views: Named Queries

CREATE VIEW *view_name*
AS *select_statement*

- Makes development simpler
- Often used for security
- Not “materialized”

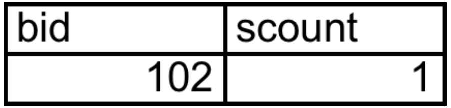
```
CREATE VIEW Redcount

AS SELECT B.bid, COUNT(*) AS scount
     FROM Boats2 B, Reserves2 R
     WHERE R.bid=B.bid AND B.color='red'
     GROUP BY B.bid
```

Views Instead of Relations in Queries

```
CREATE VIEW Redcount
AS SELECT B.bid, COUNT(*) AS scount
     FROM Boats2 B, Reserves2 R
     WHERE R.bid=B.bid AND B.color='red'
     GROUP BY B.bid;
```

```
SELECT * from redcount;
```



bid	scount
102	1

```
SELECT bname, scount
FROM Redcount R, Boats2 B
WHERE R.bid=B.bid
AND scount < 10;
```

Subqueries in FROM

Like a “view on the fly”!

```
SELECT  bname, scount
FROM Boats2 B,
(SELECT B.bid, COUNT (*)
     FROM Boats2 B, Reserves2 R
     WHERE R.bid = B.bid AND B.color = 'red'
     GROUP BY B.bid) AS Reds(bid, scount)
  WHERE  Reds.bid=B.bid
       AND scount < 10
```

WITH a.k.a. common table expression (CTE)

Another “view on the fly” syntax:

```
       WITH Reds(bid, scount) AS
       (SELECT B.bid, COUNT (*)
       FROM Boats2 B, Reserves2 R
       WHERE R.bid = B.bid AND B.color = 'red'
       GROUP BY B.bid)

SELECT bname, scount
FROM Boats2 B, Reds
WHERE Reds.bid=B.bid
AND scount < 10
```

Can have many queries in WITH

Another “view on the fly” syntax:

```
     WITH Reds(bid, scount) AS
     (SELECT B.bid, COUNT (*)
     FROM Boats2 B, Reserves2 R
     WHERE R.bid = B.bid AND B.color = 'red'
     GROUP BY B.bid),

     UnpopularReds AS
     (SELECT bname, scount
     FROM Boats2 B, Reds
     WHERE Reds.bid=B.bid
     AND scount < 10)

SELECT * FROM UnpopularReds;
```

ARGMAX GROUP BY?

- The sailor with the highest rating per age

```
     WITH maxratings(age, maxrating) AS
     (SELECT age, max(rating)
     FROM Sailors
     GROUP BY age)

SELECT S.*
  FROM Sailors S, maxratings m
 WHERE S.age = m.age
   AND S.rating = m.maxrating;
```

Brief Detour: Null Values

- Field values are sometimes unknown
  - SQL provides a special value NULL for such situations.
  - Every data type can be NULL
- The presence of null complicates many issues. E.g.:
  - Selection predicates (WHERE)
  - Aggregation
- But NULLs comes naturally from Outer joins

NULL in the WHERE clause

- Consider a tuple where `rating` IS NULL.

```
 INSERT INTO sailors VALUES
  (11, 'Jack Sparrow', NULL, 35);
```

```
SELECT * FROM sailors
WHERE rating > 8;
```

Is Jack Sparrow in the output?

NULL in comparators

- Rule: (x op NULL) evaluates to … NULL!

```
SELECT 100 = NULL;

SELECT 100 < NULL;

SELECT 100 >= NULL;
```

Explicit NULL Checks

```
SELECT * FROM sailors WHERE rating IS NULL;

SELECT * FROM sailors WHERE rating IS NOT NULL;
```

NULL at top of WHERE

- Rule: Do not output a tuple WHERE NULL

```
SELECT * FROM sailors;
SELECT * FROM sailors WHERE rating > 8;
SELECT * FROM sailors WHERE rating <= 8;
```

NULL in Boolean Logic

Three-valued logic:

NOT	T	F	N
	F	T	

AND	T	F	N
T	T	F	
F	F	F	
N			

OR	T	F	N
T	T	T	
F	T	F	
N			

```
SELECT * FROM sailors WHERE rating > 8 AND TRUE;
```

```
SELECT * FROM sailors WHERE rating > 8 OR TRUE;
```

```
SELECT * FROM sailors WHERE NOT (rating > 8);
```

General rule: NULL can take on either T or F, so answers need to accommodate either value.

NULL in Boolean Logic

Three-valued logic:

NOT	T	F	N
	F	T	N

AND	T	F	N
T	T	F	N
F	F	F	F
N	N	F	N

OR	T	F	N
T	T	T	T
F	T	F	N
N	T	N	N

```
SELECT * FROM sailors WHERE rating > 8 AND TRUE;
```

```
SELECT * FROM sailors WHERE rating > 8 OR TRUE;
```

```
SELECT * FROM sailors WHERE NOT (rating > 8);
```

General rule: NULL can take on either T or F, so answers need to accommodate either value.

NULL and Aggregation

```
SELECT count(*) FROM sailors;

SELECT count(rating) FROM sailors;

SELECT sum(rating) FROM sailors;

SELECT avg(rating) FROM sailors;
```

General rule: NULL **column values** are ignored by aggregate functions

NULLs: Summary

- x op NULL is NULL
- WHERE NULL: do not send to output
- Boolean connectives: 3-valued logic
- Aggregates ignore NULL-valued inputs

Testing SQL Queries

- SQL Fiddle pages we provide in this class will typically help you answer the questions in the worksheets and vitamins.
- But in real life:
    - not every database instance will reveal every bug in your query.
        - Eg: database instance without any rows in it!
    - Need to debug your queries
    - reasoning about them carefully
    - constructing test data.

Tips for Generating Test Data

- Generate **random data**
  - e.g. using a service like mockaroo.com
- Try to construct data that could check for the following potential errors:
  - Incorrect output schema
  - Output may be missing rows from the correct answer (false negatives)
  - Output may contain incorrect rows (false positives)
  - Output may have the wrong number of duplicates.
  - Output may not be ordered properly.

Summary

- You’ve now seen SQL—you are armed.
- A declarative language
  - Somebody has to translate to algorithms though…
  - The RDBMS implementor ... i.e. you!

Summary Cont

- The data structures and algorithms that make SQL possible also power:
  - NoSQL, data mining, scalable ML, network routing…
  - A toolbox for scalable computing!
  - That fun begins next week
- We skirted questions of good database (schema) design
  - a topic we'll consider in greater depth later