SQL II

R & G - Chapter 5

SELECT DISTINCT

- SELECT DISTINCT (col list) will remove duplicates of tuples corresponding to the col list
- You can only apply DISTINCT at the start of a list of columns
- So:
  - SELECT A, DISTINCT B … is not permitted
  - But SELECT COUNT(DISTINCT A) … is OK
    - Count of number of distinct values of A

SQL

- **So far: Basic Single-Table DML queries**
  - **SELECT (with DISTINCT)/FROM/WHERE**
  - **Aggregation: GROUP BY, HAVING**
  - **Presentation: ORDER BY, LIMIT**
- Extending basic SELECT/FROM/WHERE
  - Multi-table queries: JOINs
  - Aliasing in FROM and SELECT
  - Expressions in SELECT
  - Expressions, string comparisons, connectives in WHERE
  - Extended JOINs
  - The use of NULLS
- Query Composition
  - Set-oriented operations
  - Nested queries
  - Views
  - Common table expressions

Lots to cover!
Use vitamins and sections to dig deeper.

SQL DML 1:
Basic Single-Table Queries

- **SELECT** [**DISTINCT**] *<column expression list>*
  FROM *<single table>*
  [**WHERE** *<predicate>*]
  [**GROUP BY** *<column list>*
  [**HAVING** *<predicate>*]]
  [**ORDER BY** *<column list>*]
  [**LIMIT** <integer>];

Conceptual Order of Evaluation

(5) **SELECT** [**DISTINCT**] *<col exp. list>*

(1) **FROM** *<single table>*

(2) [**WHERE** *<predicate>*]

(3) [**GROUP BY** *<column list>*

(4) [**HAVING** *<predicate>*]]

(6) [**ORDER BY** *<column list>*]

(7) [**LIMIT** <integer>];

Will omit ORDER BY and LIMIT for now since they are primarily for presentation

SQL DML 1: Basic Single-Table Queries
Conceptual Order of Evaluation

(5) **SELECT** [**DISTINCT**] *<col exp. list>* ☛ *remove (project) cols not found in list, then remove dupl. rows*
(1) **FROM** *<single table>* ☛ *for each tuple in table*
(2) [**WHERE** *<predicate>*] ☛ *remove tuples that don't satisfy predicate (selection condition)*
(3) [**GROUP BY** *<column list>* ☛ *form groups and perform all* ***necessary*** *aggregates per group*
(4) [**HAVING** *<predicate>*]] ☛ *remove groups that don't satisfy predicate*

Q: Which aggregates are necessary?
A: All the aggregates that will be referred to in the HAVING or SELECT clause

Remember: this is all **conceptual** — actual approach for execution may be very different. But will provide the same result as this conceptual approach.

Putting it all together

- **SELECT** S.dept, **AVG**(S.gpa), **COUNT**(*)
  FROM Students **AS** S
  WHERE S.state = 'MA'
  GROUP BY S.dept
  HAVING MAX(S.gpa) >= 2
  ORDER BY S.dept;

- Students (name, dept, gpa, state)
  - Start with all tuples in Students
  - Throw away those that aren't from MA
  - Group by S.dept, compute aggregates MAX(S.gpa), AVG(S.gpa), COUNT(*)
  - Throw away groups that don't have MAX(S.gpa)>=2
  - Retain only S.dept, AVG(S.GPA), COUNT(*)
  - Order by S.dept

Multi-Table Queries: Joins

- SELECT [DISTINCT] *<column expression list>*
  FROM *<table1 [AS t1], ... , tableN [AS tn]>*
  [WHERE *<predicate>*]
  [GROUP BY *<column list>*[HAVING *<predicate>*]]
  [ORDER BY *<column list>*];

SQL DML 1: Basic Single-Table Queries
Conceptual Order of Evaluation

Let’s not worry about GROUP BY and HAVING for now, back to good old SELECT-FROM-WHERE
Extending it to GROUP BY and HAVING is straightforward (as is ORDER BY and LIMIT)

(5) **SELECT** [**DISTINCT**] *<col exp. list>* ☛ *remove (project out) cols not found in list, then remove duplicate rows*
(1) **FROM *<table1><table2>…*** ☛ ***for each combinations of tuples in cross product of tables***
(2) [**WHERE** *<predicate>*] ☛ *remove **tuple combinations** that don’t satisfy predicate (selection condition)*
~~(3) [**GROUP BY** *<column list>* ☛ *form groups and perform all necessary aggregates per group*~~
~~(4) [**HAVING** *<predicate>*]] ☛ *remove groups that don’t satisfy predicate*~~

Another way to think about a multi-table query is a query on a new relation that is the cross-product of tables in the FROM clause.

This is likely a really bad way to evaluate this query! We will discuss better ways subsequently.

Cross (Cartesian) Product

- FROM clause: all pairs of tuples, concatenated

Sailors

sid	sname	rating	age
1	Popeye	10	22
2	OliveOyl	11	39
3	Garfield	1	27
4	Bob	5	19

Reserves

sid	bid	day
1	102	9/12
2	102	9/13
1	101	10/01

sid	sname	rating	age	sid	bid	day
1	Popeye	10	22	1	102	9/12
1	Popeye	10	22	2	102	9/13
1	Popeye	10	22	1	101	10/01
2	OliveOyl	11	39	1	102	9/12
…	…	…	…	…	…	…

Find sailors who've reserved a boat

```
SELECT S.sid, S.sname, R.bid
FROM Sailors AS S, Reserves AS R
WHERE S.sid=R.sid
```

sid	sname	rating	age
1	Popeye	10	22
2	OliveOyl	11	39
3	Garfield	1	27
4	Bob	5	19

sid	bid	day
1	102	9/12
2	102	9/13
1	101	10/01

sid	sname	rating	age	sid	bid	day
1	Popeye	10	22	1	102	9/12
1	Popeye	10	22	2	102	9/13
1	Popeye	10	22	1	101	10/01
2	OliveOyl	11	39	1	102	9/12
…	…	…	…	…	…	…

Find sailors who've reserved a boat cont

```
SELECT S.sid, S.sname, R.bid
FROM Sailors AS S, Reserves AS R
WHERE S.sid=R.sid
```

sid	sname	rating	age
1	Popeye	10	22
2	OliveOyl	11	39
3	Garfield	1	27
4	Bob	5	19

sid	bid	day
1	102	9/12
2	102	9/13
1	101	10/01

sid	sname	bid
1	Popeye	102
1	Popeye	101
2	OliveOyl	102

Table Aliases and Column Name Aliases

```
SELECT Sailors.sid, sname, bid
FROM Sailors, Reserves
WHERE Sailors.sid = Reserves.sid
```

Relation (range) variables (Sailors, Reserves) help refer to columns that are shared across relations.

We can also rename relations and use new variables (“AS” is optional for FROM)

```
SELECT S.sid, sname, bid
FROM Sailors AS S, Reserves AS R
WHERE S.sid = R.sid
```

We can also rename attributes too!

```
SELECT S.sid AS sailorid, sname AS sailorname, bid AS boatid
FROM Sailors AS S, Reserves AS R
WHERE S.sid = R.sid
```

More Aliases: Self-Joins

Berkeley cs186

```
SELECT x.sname AS sname1,
       x.age AS age1,
       y.sname AS sname2,
       y.age AS age2
FROM Sailors AS x, Sailors AS y
WHERE x.age > y.age
```

sid	sname	rating	age
1	Popeye	10	22
2	OliveOyl	11	39
3	Garfield	1	27
4	Bob	5	19

sname1	age1	sname2	age2
Popeye	22	Bob	19
OliveOyl	39	Popeye	22
OliveOyl	39	Garfield	27
OliveOyl	39	Bob	19
Garfield	27	Popeye	22
Garfield	27	Bob	19

- Query for pairs of sailors where one is older than the other
- Table aliases in the FROM clause
  - Needed when the same table used multiple times (“self-join”)

Arithmetic Expressions

- SELECT S.age, **S.age-5 AS age1, 2*S.age AS age2**
  FROM Sailors AS S
  WHERE S.sname = 'Popeye'

- SELECT S1.sname AS name1, S2.sname AS name2
  FROM Sailors AS S1, Sailors AS S2
  WHERE **2*S1.rating = S2.rating - 1**

sid	sname	rating	age
1	Popeye	10	22
2	OliveOyl	11	39
3	Garfield	1	27
4	Bob	5	19

String Comparisons

- Old School SQL

```
SELECT S.sname
FROM   Sailors S
WHERE  S.sname LIKE 'B_%'
```

_ = any single char; % = zero or more chars
Returns Bob

- Standard Regular Expressions

```
SELECT S.sname
FROM   Sailors S
WHERE  S.sname ~ 'B.*'
```

. = any char; * = repeat (zero or more instances of previous)
Note: can match anywhere in the string
Returns Bob and McBob

sid	sname	rating	age
1	Popeye	10	22
2	OliveOyl	11	39
3	Garfield	1	27
4	**Bob**	**5**	**19**
5	**McBob**	**3**	**35**

SQLite note: ~ not supported.

Boolean Connectives

Sid's of sailors who reserved a red **OR** a green boat

```
SELECT R.sid
FROM   Boats B, Reserves R
WHERE  R.bid=B.bid AND
        (B.color='red' OR B.color='green')
```

Boats

bid	bname	color
102	Titanic	green
101	Lusitania	red
100	Mayflower	orange

Reserves

sid	bid	day
1	102	9/12
2	102	9/13
1	100	10/01

SQL

Berkeley CS186

- So far: Basic Single-Table DML queries
  - SELECT (with DISTINCT)/FROM/WHERE
  - Aggregation: GROUP BY, HAVING
  - Presentation: ORDER BY, LIMIT
- Extending basic SELECT/FROM/WHERE
  - Multi-table queries: JOINs
  - Aliasing in FROM and SELECT
  - Expressions in SELECT
  - Expressions, string comparisons, connectives in WHERE
  - **Extended JOINs**
  - The use of NULLS
- Query Composition
  - Set-oriented operations
  - Nested queries
  - Views
  - Common table expressions

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
  - Same thing as what we've done so far, offers no additional convenience
  - Just present as a contrast to NATURAL and OUTER

Reminder

- Turn on video if you can
- Turn off audio except when speaking
- Don’t do anything you wouldn’t do normally

- Vitamin 1 deadline has been pushed
- Project 1 should still be on track

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;
```

sid	sname	rating	age
1	Popeye	10	22
2	OliveOyl	11	39
3	Garfield	1	27
4	Bob	5	19

sid	bid	day
1	102	9/12
2	102	9/13
1	101	10/01

- **ALL 3 ARE EQUIVALENT!**
- “NATURAL” means “equi-join” (i.e., identical values) for pairs of attributes with the same name

Left Outer Join

- Returns all matched rows, <u>and *preserves* all unmatched rows from the table on the **left**</u> of the join clause
  - (use NULLs in fields of non-matching tuples)
  - We'll talk about NULLs in a bit, but for now, think of it as N/A

```
SELECT s.sid, s.sname, r.bid
FROM Sailors s LEFT OUTER JOIN Reserves 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!

(3, Garfield, NULL) (4, Bob, NULL) in output

sid	sname	rating	age
1	Popeye	10	22
2	OliveOyl	11	39
3	Garfield	1	27
4	Bob	5	19

sid	bid	day
1	102	9/12
2	102	9/13
1	101	10/01

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 Reserves r RIGHT OUTER JOIN Boats 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 Reserves r FULL OUTER JOIN Boats 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!

SQLite note: RIGHT/FULL OUTER JOIN not supported.

Brief Detour: NULL Values

- Values for any data type can be NULL
  - Indicates the value is present but unknown or is inapplicable
  - Also comes naturally from Outer joins
- The presence of null complicates many issues. E.g.:
  - Selection predicates (WHERE)
  - Aggregation

NULL in the WHERE clause

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

Q: Should Popeye be in the output?

Not really.

Likewise for

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

sid	sname	rating	age
1	Popeye	NULL	22
2	OliveOyl	11	39
3	Garfield	1	27
4	Bob	5	19

NULL in the WHERE clause

sid	sname	rating	age
1	Popeye	NULL	22
2	OliveOyl	11	39
3	Garfield	1	27
4	Bob	5	19

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

This is really funky — we have a tautology in the WHERE clause, but Popeye will still not be output

To force certain outputs can use IS NULL or IS NOT NULL conditions

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

This will correctly output all tuples in this setting

More generally, we need an extension to Boolean logic to support this

Correctly Reasoning about NULLs

- Several Ingredients:
  - We need a way to evaluate unit predicates, a way to combine them, and a way to decide whether to output
- Ingredient 1: Evaluating unit predicates
  - (x op NULL) evaluates to NULL (IDK!)

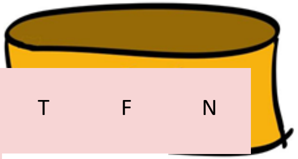
```
SELECT 100 = NULL;
SELECT 100 < NULL;
```

  - `IS NULL` evaluates to True if NULL, False otherwise
- Ingredient 3: Deciding to output
  - When the WHERE evaluates to NULL, do not output the tuple

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

- Ingredient 2: Combining predicates
  - Three-valued logic, an extension of two-valued (Boolean) logic

NULL in Boolean Logic



Three-valued logic: truth tables!

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

NOT	T	F	N
	F	T	N

Let’s build intuition by going through examples

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

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

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

General rule: NULL values are treated as “I Don’t Know” – can be either true or false

NULL and Aggregation

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

```
SELECT count(*) FROM sailors;

SELECT count(rating) FROM sailors;

SELECT sum(rating) FROM sailors;

SELECT avg(rating) FROM sailors;
```

NULL and Aggregation

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

```
SELECT count(*) FROM sailors; // count sailors

SELECT count(rating) FROM sailors; // count sailors with non-NULL ratings

SELECT sum(rating) FROM sailors; // sum of non-NULL ratings

SELECT avg(rating) FROM sailors; // avg of non-NULL ratings
```

NULLs: Summary

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

SQL

- Basic Single-Table DML queries
  - SELECT (with DISTINCT)/FROM/WHERE
  - Aggregation: GROUP BY, HAVING
  - Presentation: ORDER BY, LIMIT
- Extending basic SELECT/FROM/WHERE
  - Multi-table queries: JOINs
  - Aliasing in FROM and SELECT
  - Expressions in SELECT
  - Expressions, string comparisons, connectives in WHERE
  - Extended JOINs
  - The use of NULLS
- **Query Composition**
  - **Set-oriented operations**
  - **Nested queries**
  - **Views**
  - **Common table expressions**

Let's talk about Sets and Bags

- Set = no duplicates {🍎,🍏,🍊,🍋,🍐}
- Bag / Multi-set = duplicates allowed {🍎,🍎, 🍏,🍊,🍊,🍊}
- As we saw earlier SQL uses bag semantics
  - That is, there can be multiple copies of each tuple in a relation
- How do we "add/subtract" tuples across relations?
  - We can do so operators that enforce either bag or set-based semantics

Operators with Set Semantics

- Set: a collection of distinct elements
  - In the relational parlance: each tuple/row is unique
- Ways of manipulating/combining sets
  - A UNION B: distinct tuples in A or B
  - A INTERSECT B: distinct tuples in A and B
  - A EXCEPT B: distinct tuples in A but not in B
- Basically, we treat tuples within a relation as elements of a set

Using Set Semantics with SQL

Note: R and S are relations. They are not sets, since they have duplicates.
Assume these are all tuples: A, B, C, D, E

R = {A, A, A, A, B, B, C, D}
S = {A, A, B, B, B, C, E}

- UNION

  {A, B, C, D, E}
- INTERSECT

  {A, B, C}
- EXCEPT

  {D}

Reserves

sid	bid	day
1	102	9/12
1	102	9/12
2	101	10/01

Q: What does
(SELECT * FROM Reserves)
UNION
(SELECT * FROM Reserves)
give us?

“ALL”: Multiset Semantics

R = {A, A, A, A, B, B, C, D} = {A(4), B(2), C(1), D(1)}

S = {A, A, B, B, B, C, E} = {A(2), B(3), C(1), E(1)}

“UNION ALL”: Multiset Semantics

R = {A, A, A, A, B, B, C, D} = {A(4), B(2), C(1), D(1)}
S = {A, A, B, B, B, C, E} = {A(2), B(3), C(1), E(1)}

- UNION ALL: sum of cardinalities

  {A(4+2), B(2+3), C(1+1), D(1+0), E(0+1)}
  = {A, A, A, A, A, A, B, B, B, B, B, C, C, D, E}

Reserves

sid	bid	day
1	102	9/12
1	102	9/12
2	101	10/01

Q: What does
(SELECT * FROM Reserves)
UNION ALL
(SELECT * FROM Reserves)
give us?

"INTERSECT ALL": Multiset Semantics

R = {A, A, A, A, B, B, C, D} = {A(4), B(2), C(1), D(1)}
S = {A, A, B, B, B, C, E} = {A(2), B(3), C(1), E(1)}

- INTERSECT ALL: min of cardinalities

  {A(min(4,2)), B(min(2,3)), C(min(1,1)),
  D(min(1,0)), E(min(0,1))}
  = {A, A, B, B, C}

"EXCEPT ALL": Multiset Semantics

R = {A, A, A, A, B, B, C, D} = {A(4), B(2), C(1), D(1)}

S = {A, A, B, B, B, C, E} = {A(2), B(3), C(1), E(1)}

- EXCEPT ALL: difference of cardinalities

  {A(4-2), B(2-3), C(1-1), D(1-0), E(0-1)}

  = {A, A, D}

Set/Bag Operators

- A UNION B, A INTERSECT B, A EXCEPT B perform set-based operations treating tuples in A and B as sets
- A UNION ALL B, A INTERSECT ALL B, A EXCEPT ALL B perform bag-based operations treating tuples in A and B as bags
- **Note**: for these operations to be applied correctly, the schema for A and B must be the same!

Combining Predicates

- Subtle connections between:
  - Boolean logic in WHERE (i.e., AND, OR)
  - Set operations (i.e. INTERSECT, UNION)
- Let's see some examples…

Sid’s of sailors who reserved a red **OR** a green boat

```
SELECT R.sid
FROM Boats B, Reserves R
WHERE R.bid=B.bid AND B.color='red'

UNION

SELECT R.sid
FROM Boats B, Reserves R
WHERE R.bid=B.bid AND B.color='green'
```

VS…

```
SELECT DISTINCT R.sid
FROM Boats B,Reserves R
WHERE R.bid=B.bid AND
           (B.color='red’ OR B.color='green')
```

These two give the exact same result!

HW:

a) What if we did UNION ALL instead?
b) What if we omitted DISTINCT?

Sid’s of sailors who reserved a red **AND** a green boat

```
SELECT R.sid
FROM Boats B, Reserves R
WHERE R.bid=B.bid AND B.color='red'

INTERSECT

SELECT R.sid
FROM Boats B, Reserves R
WHERE R.bid=B.bid AND B.color='green'

VS…

SELECT DISTINCT R.sid
FROM Boats B,Reserves R
WHERE R.bid=B.bid AND
           (B.color='red’ AND B.color=‘green')
```

The first query works fine… but the second query doesn’t work. Why?

SQL

- Basic Single-Table DML queries
  - SELECT (with DISTINCT)/FROM/WHERE
  - Aggregation: GROUP BY, HAVING
  - Presentation: ORDER BY, LIMIT
- Extending basic SELECT/FROM/WHERE
  - Multi-table queries: JOINs
  - Aliasing in FROM and SELECT
  - Expressions in SELECT
  - Expressions, string comparisons, connectives in WHERE
  - Extended JOINs
  - The use of NULLS
- Query Composition
  - Set-oriented operations
  - **Nested queries**
  - Views
  - Common table expressions

Query Composition

- We’ve already seen one way of combining results across multiple queries via set and bag-based operations
- Now, we’ll talk about “nesting” queries inside other queries
    - Nesting and subqueries
    - Views to refer to frequent query expressions
    - Common Table Expressions

Nested Queries: IN

- *Names of sailors who've reserved boat #102:*

```
SELECT S.sname
FROM   Sailors S
WHERE  S.sid IN
   (SELECT  R.sid
   FROM    Reserves R
   WHERE   R.bid=102)
```

subquery

Here, the results of this subquery are treated as a (multi)set, with membership of S.sid checked in the set using the IN operator

Nested Queries: NOT IN

- *Names of sailors who've **not** reserved boat #103:*

```
SELECT  S.sname
FROM  Sailors S
WHERE  S.sid NOT IN
   (SELECT  R.sid
    FROM  Reserves R
    WHERE  R.bid=103)
```

Nested Queries with Correlation

- So far, we've studied ways to nest query results and treat it as a "set" with membership in the set checked
  - using … ***val [NOT] IN (nested query)***
- We can also check if a nested query result is empty/not
  - using … ***[NOT] EXISTS (nested query)***
- *Names of sailors who've reserved boat #102:*

```
SELECT  S.sname
FROM    Sailors S
WHERE EXISTS
   (SELECT  *
   FROM  Reserves R
   WHERE R.bid=102 AND S.sid=R.sid)
```

- Correlated subquery is **conceptually** recomputed for each Sailors tuple.

More on Set-Comparison Operators

- We've seen: `[NOT] IN, [NOT] EXISTS`
- Other forms: `op ANY, op ALL`

Find sailors whose rating is greater than that of *some* sailor called Popeye:

```
SELECT *
FROM   Sailors S
WHERE  S.rating > ANY
        (SELECT  S2.rating
        FROM  Sailors S2
        WHERE S2.sname='Popeye')
```

SQLite note: ANY/ALL not supported.

A Tough One: “Division”

- Relational Division: “Find sailors who’ve reserved all boats.”
  Said differently: “Sailors with no 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 ))
```

For S, this is the set of all boats they have not reserved

For S and B, this is the set of reservations of B for S

ARGMAX?

- 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)
```

These are exactly the same!

ARGMAX?

- 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;
```

These are not the same if there are multiple such Sailors

Views: Named Queries

CREATE VIEW *view_name* **AS** *select_statement*

- Makes development simpler, convenient
- Often used for security
- Not “materialized” [but there are materialized views as well!]

```
// Counts of reservations for red colored boats

CREATE VIEW Redcount AS

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

bid	scount
102	1

```
SELECT * from Redcount WHERE scount<10;
```

Subqueries in FROM

Like a "view on the fly"!

```
SELECT *
FROM
(SELECT B.bid, COUNT (*)
FROM Boats B, Reserves R
WHERE R.bid = B.bid AND B.color = 'red'
GROUP BY B.bid) AS Redcount(bid, scount)
WHERE scount < 10
```

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

Another “view on the fly” syntax:

```
WITH Redcount(bid, scount) AS
(SELECT B.bid, COUNT (*)
FROM Boats B, Reserves R
WHERE R.bid = B.bid AND B.color = 'red'
GROUP BY B.bid)
```

```
SELECT * FROM Reds
WHERE scount < 10
```

Can have many queries in WITH

Cascade of queries: Redcount -> UnpopularReds

```
WITH Redcount(bid, scount) AS
(SELECT B.bid, COUNT (*)
FROM Boats B, Reserves R
WHERE R.bid = B.bid AND B.color = 'red'
GROUP BY B.bid),

UnpopularReds AS
(SELECT *
FROM Redcount
WHERE scount < 10)

SELECT * FROM UnpopularReds;
```

ARGMAX GROUP BY?

- More complex variation of previous argmax
- Find the sailors 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;
```

Testing SQL Queries

- Typically not every database instance will reveal every bug in your query.
    - Eg: database instance without any rows in it!
- Best to try to reason about behavior across all instances
- Also helpful: 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!
  - Start talking about that in the next set of slides!
- We skirted questions of good database (schema) design
  - a topic we'll consider in greater depth later