CompSci 516
Database Systems

Lecture 4-5
More SQL
( Joins, Subqueries, Group-by,
Nulls, Views, Constraints )

Instructor: Sudeepa Roy
Announcements

• In-person classes starting Thursday
  – Also live streaming and recording
Next: different types of joins

• Theta-join
• Equi-join
• Natural join
• Outer Join
Condition/Theta Join

**SELECT** * * 
**FROM** Sailors S, Reserves R 
**WHERE** S.sid=R.sid and age >= 40

Form cross product, discard rows that do not satisfy the condition

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>
**Equi Join**

A special case of theta join
Join condition only has equality predicate =

```
SELECT * 
FROM Sailors S, Reserves R 
WHERE S.sid=R.sid and age = 45
```

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>
# Natural Join

```sql
SELECT * 
FROM Sailors S NATURAL JOIN Reserves R
```

A special case of equi join
Equality condition on ALL common predicates (sid)
Duplicate columns are eliminated

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

**Right Table:***

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>
Outer Join

\[
\text{SELECT } S.\text{sid}, R. \text{bid} \\
\text{FROM Sailors S LEFT OUTER JOIN Reserves R} \\
\text{ON S.sid=R.sid}
\]

Preserves all tuples from the left table whether or not there is a match if no match, fill attributes from right with null
Similarly RIGHT/FULL outer join

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>
Expressions and Strings

- Illustrates use of arithmetic expressions and string pattern matching
- **Find triples (of ages of sailors and two fields defined by expressions) for sailors**
  - whose names begin and end with B and contain at least three characters
- LIKE is used for string matching. `_` stands for any one character and `%` stands for 0 or more arbitrary characters
  - You will need these often

```sql
SELECT S.age, age1=S.age-5, 2*S.age AS age2
FROM Sailors S
WHERE S.sname LIKE 'B_%B'
```
Find sid’s of sailors who’ve reserved a red or a green boat

- **UNION**: Can be used to compute the union of any two *union-compatible* sets of tuples
  - can themselves be the result of SQL queries
- If we replace **OR** by **AND** in the first version, what do we get?
- Also available: **EXCEPT** (What do we get if we replace **UNION** by **EXCEPT**?)

```sql
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
AND (B.color=’red’ OR B.color=’green’)
```

```sql
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
AND B.color=’red’
UNION
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
AND B.color=’green’
```
Find sid’s of sailors who’ve reserved a red and a green boat
Find sid’s of sailors who’ve reserved a red and a green boat

- Does not work ->
- **INTERSECT**: Can be used to compute the intersection of any two union-compatible sets of tuples.
  - Included in the SQL/92 standard, but some systems don’t support it

```sql
SELECT S.sid
FROM Sailors S, Boats B1, Reserves R1,
     Boats B2, Reserves R2
WHERE S.sid=R1.sid AND R1.bid=B1.bid
     AND S.sid=R2.sid AND R2.bid=B2.bid
     AND (B1.color='red' AND B2.color='green')
```

```sql
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
     AND B.color='red'
INTERSECT
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
     AND B.color='green'
```
Nested Queries

Find names of sailors who’ve reserved boat #103:

```
SELECT  S.sname
FROM    Sailors S
WHERE   S.sid IN (SELECT  R.sid
                      FROM    Reserves R
                      WHERE   R.bid=103)
```

- A very powerful feature of SQL:
  - a WHERE/FROM/HAVING clause can itself contain an SQL query
- To find sailors who’ve not reserved #103, use NOT IN.
- To understand semantics of nested queries, think of a nested loops evaluation
  - For each Sailors tuple, check the qualification by computing the subquery
Nested Queries with Correlation

Find names of sailors who’ve reserved boat #103:

\[
\text{SELECT } S.\text{sname} \\
\text{FROM } \text{Sailors } S \\
\text{WHERE } \text{EXISTS} \left( \text{SELECT } * \right) \\
\quad \text{FROM } \text{Reserves } R \\
\quad \text{WHERE } \text{R.bid}=103 \text{ AND } S.\text{sid}=R.\text{sid} \right)
\]

- \text{EXISTS} is another set comparison operator, like \text{IN}
- Illustrates why, in general, subquery must be re-computed for each Sailors tuple
Nested Queries with Correlation

Find names of sailors who’ve reserved boat #103 at most once:

```sql
SELECT S.sname
FROM Sailors S
WHERE UNIQUE (SELECT R.bid
                FROM Reserves R
                WHERE R.bid=103 AND S.sid=R.sid)
```

- If `UNIQUE` is used, and `*` is replaced by `R.bid`, finds sailors with at most one reservation for boat #103
  - `UNIQUE` checks for duplicate tuples
More on Set-Comparison Operators

• We’ve already seen IN, EXISTS and UNIQUE

• Can also use NOT IN, NOT EXISTS and NOT UNIQUE.

• Also available: $op$ ANY, $op$ ALL, $op$ IN
  – where $op$ : $>$, $<$, $=$, $<=$, $>=$

• Find sailors whose rating is greater than that of some sailor called Horatio
  – similarly ALL

```sql
SELECT *
FROM Sailors S
WHERE S.rating > ANY (SELECT S2.rating
    FROM Sailors S2
    WHERE S2.sname='Horatio')
```
Recall: Aggregate Operators

Check yourself:
What do these queries compute?

```
SELECT COUNT (*)
FROM Sailors S
```

```
SELECT AVG (S.age)
FROM Sailors S
WHERE S.rating=10
```

```
SELECT COUNT (DISTINCT S.rating)
FROM Sailors S
WHERE S.sname='Bob'
```

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

```
SELECT AVG (DISTINCT S.age)
FROM Sailors S
WHERE S.rating=10
```

```
SELECT AVG ( [DISTINCT] A)
SUM ( [DISTINCT] A)
AVG ( [DISTINCT] A)
MAX (A)
MIN (A)
```

single column
Motivation for Grouping

• So far, we’ve applied aggregate operators to all (qualifying) tuples
  – Sometimes, we want to apply them to each of several groups of tuples

• Consider: Find the age of the youngest sailor for each rating level
  – In general, we don’t know how many rating levels exist, and what the rating values for these levels are!
  – Suppose we know that rating values go from 1 to 10; we can write 10 queries that look like this (need to replace $i$ by num):

\[
\text{SELECT MIN (S.age)} \\
\text{FROM Sailors S} \\
\text{WHERE S.rating = i}
\]

For $i = 1, 2, \ldots, 10$: 
Group-by evaluation semantics

Always start from “FROM” -- form cross product
Apply “WHERE” -- filter out some tuples (rows)
Apply “GROUP BY” -- partition tuples into groups by the value of the grouping attributes
Apply “HAVING” -- eliminate groups that do not satisfy the condition
Apply “SELECT” -- compute aggregates for each group

- Expressions in HAVING must have a single value per group
  In effect, an attribute in HAVING that is not an argument of an aggregate op also appears in GROUP-BY attributes list like, “…GROUP BY bid, sid HAVING bid = 3”
- One answer tuple is generated per qualifying group
- A subset of GROUP BY attributes can appear in SELECT: SELECT A, SUM(C) FROM R GROUP BY A, B

```
5  SELECT  S.rating,  MIN (S.age) AS minage
1  FROM  Sailors S
2  WHERE  S.age >= 18
3  GROUP BY  S.rating
4  HAVING  COUNT (*) > 1
```
Find age of the youngest sailor with age $\geq 18$, for each rating with at least 2 such sailors.

SELECT S.rating, MIN (S.age) AS minage
FROM Sailors S
WHERE S.age $\geq 18$
GROUP BY S.rating
HAVING COUNT (*) > 1

Sailors instance:

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>29</td>
<td>brutus</td>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>32</td>
<td>andy</td>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>64</td>
<td>horatio</td>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>71</td>
<td>zorba</td>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>74</td>
<td>horatio</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>85</td>
<td>art</td>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>95</td>
<td>bob</td>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>96</td>
<td>frodo</td>
<td>3</td>
<td>25.5</td>
</tr>
</tbody>
</table>

Answer relation:
Find age of the youngest sailor with age $\geq 18$, for each rating with at least 2 such sailors.

Step 1: Form the cross product: FROM clause
(some attributes are omitted for simplicity)

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
</tbody>
</table>

SELECT S.rating, MIN (S.age) AS minage
FROM Sailors S
WHERE S.age $\geq 18$
GROUP BY S.rating
HAVING COUNT (*) $> 1$
Find age of the youngest sailor with age >= 18, for each rating with at least 2 such sailors.

**Step 2: Apply WHERE clause**

```
SELECT S.rating, MIN(S.age) AS minage
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT(*) > 1
```
Find age of the youngest sailor with age $\geq 18$, for each rating with at least 2 such sailors.

Step 3: Apply GROUP BY according to the listed attributes

```
SELECT S.rating, MIN(S.age) AS minage
FROM Sailors S
WHERE S.age $\geq$ 18
GROUP BY S.rating
HAVING COUNT(*) > 1
```
Find age of the youngest sailor with age $\geq 18$, for each rating with at least 2 such sailors.

**Step 4: Apply HAVING clause**

The *group-qualification* is applied to eliminate some groups

```sql
SELECT S.rating, MIN(S.age) AS minage
FROM Sailors S
WHERE S.age $\geq$ 18
GROUP BY S.rating
HAVING COUNT (*) > 1
```

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
</tbody>
</table>

Duke CS, Spring 2022
Find age of the youngest sailor with age >= 18, for each rating with at least 2 such sailors.

**Step 5: Apply SELECT clause**
Apply the aggregate operator
At the end, one tuple per group

```sql
SELECT S.rating, MIN(S.age) AS minage
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT(*) > 1
```
Nulls and Views in SQL
Null Values

- Field values in a tuple are sometimes
  - unknown, e.g., a rating has not been assigned, or
  - inapplicable, e.g., no spouse’s name
- SQL provides a special value null for such situations.
Standard Boolean 2-valued logic

• True = 1, False = 0
• Suppose X = 5
  – \((X < 100) \text{ AND } (X \geq 1)\) is \(T \land T = T\)
  – \((X > 100) \text{ OR } (X \geq 1)\) is \(F \lor T = T\)
  – \((X > 100) \text{ AND } (X \geq 1)\) is \(F \land T = F\)
  – \(\text{NOT}(X = 5)\) is \(\neg T = F\)

• Intuitively,
  – \(T = 1, F = 0\)
  – For \(V_1, V_2 \in \{1, 0\}\)
  – \(V_1 \land V_2 = \text{MIN}(V_1, V_2)\)
  – \(V_1 \lor V_2 = \text{MAX}(V_1, V_2)\)
  – \(\neg(V_1) = 1 - V_1\)
2-valued logic does not work for nulls

• Suppose rating = null, X = 5
• Is rating > 8 true or false?
• What about AND, OR and NOT connectives?
  – (rating > 8) AND (X = 5)?
• What if we have such a condition in the WHERE clause?
3-Valued Logic For Null

- **TRUE** (= 1), **FALSE** (= 0), **UNKNOWN** (= 0.5)
  - unknown is treated as 0.5

- Now you can apply rules from 2-valued logic!
  - For \( V_1, V_2 \in \{1, 0, 0.5\} \)
    - \( V_1 \land V_2 = \text{MIN}(V_1, V_2) \)
    - \( V_1 \lor V_2 = \text{MAX}(V_1, V_2) \)
    - \( \neg V_1 = 1 - V_1 \)

- Therefore,
  - NOT UNKNOWN = UNKNOWN
  - UNKNOWN OR TRUE = TRUE
  - UNKNOWN AND TRUE = UNKNOWN
  - UNKNOWN AND FALSE = FALSE
  - UNKNOWN OR FALSE = UNKNOWN
New issues for Null

• The presence of null complicates many issues. E.g.:
  – Special operators needed to check if value IS/IS NOT NULL
  – Be careful!
  – “WHERE X = NULL” does not work!
  – Need to write “WHERE X IS NULL”

• Meaning of constructs must be defined carefully
  – e.g., WHERE clause eliminates rows that don’t evaluate to true
  – So not only FALSE, but UNKNOWNs are eliminated too
  – very important to remember!

• Arithmetic with NULL
  – all of +, -, *, / return null if any argument is null

• Can force ”no nulls” while creating a table
  – sname char(20) NOT NULL
  – primary key is always not null
Aggregates with NULL

- What do you get for
- SELECT count(*) from R1?
- SELECT count(rating) from R1?

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

R1
Aggregates with NULL

What do you get for

SELECT count(*) from R1?

SELECT count(rating) from R1?

Ans: 3 for both

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

R1
Aggregates with NULL

### R1

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

- What do you get for
- SELECT count(*) from R1?
- SELECT count(rating) from R1?
- Ans: 3 for both

### R2

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>null</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

- What do you get for
- SELECT count(*) from R2?
- SELECT count(rating) from R2?
Aggregates with NULL

- What do you get for
- SELECT count(*) from R1?
- SELECT count(rating) from R1?
- Ans: 3 for both

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

R1

- What do you get for
- SELECT count(*) from R2?
- SELECT count(rating) from R2?
- Ans: First 3, then 2

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>null</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

R2
Aggregates with NULL

- **COUNT, SUM, AVG, MIN, MAX** (with or without DISTINCT)
  - Discards null values first
  - Then applies the aggregate
  - Except count(*)
- If only applied to null values, the result is null

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>null</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

**R2**
- SELECT sum(rating) from R2?
- Ans: 17

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>null</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>null</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>null</td>
<td>35</td>
</tr>
</tbody>
</table>

**R3**
- SELECT sum(rating) from R3?
- Ans: null
Views

• A view is just a relation, but we store a definition, rather than a set of tuples

    CREATE VIEW YoungActiveStudents (name, grade)
    AS SELECT S.name, E.grade
    FROM Students S, Enrolled E
    WHERE S.sid = E.sid and S.age<21

• Views can be dropped using the DROP VIEW command

• Views and Security: Views can be used to present necessary information (or a summary), while hiding details in underlying relation(s)
  • the above view hides courses “cid” from E

• More on views later in the course
Can create a new table from a query on other tables too

```
SELECT S.name, E.grade
INTO YoungActiveStudents
FROM Students S, Enrolled E
WHERE S.sid = E.sid and S.age<21
```
“WITH” clause – very useful!

• You will find “WITH” clause very useful!

```sql
WITH Temp1 AS
  (SELECT .... ..),
Temp2 AS
  (SELECT .... ..)
SELECT X, Y
FROM TEMP1, TEMP2
WHERE....
```

• Can simplify complex nested queries
Overview: General Constraints

- Useful when more general ICs than keys are involved

- There are also **ASSERTIONS** to specify constraints that span across multiple tables

- There are **TRIGGERS** too: procedure that starts automatically if specified changes occur to the DBMS

```sql
CREATE TABLE Sailors
    ( sid INTEGER,
      sname CHAR(10),
      rating INTEGER,
      age REAL,
      PRIMARY KEY (sid),
      CHECK ( rating >= 1 AND rating <= 10 )
)
```

```sql
CREATE TABLE Reserves
    ( sname CHAR(10),
      bid INTEGER,
      day DATE,
      PRIMARY KEY (bid,day),
      CONSTRAINT noInterlakeRes
        CHECK (‘Interlake’ <>
            ( SELECT B.bname
              FROM Boats B
              WHERE B.bid=bid)))
```
Triggers

• **Trigger**: procedure that starts automatically if specified changes occur to the DBMS

• **Three parts:**
  – Event (activates the trigger)
  – Condition (tests whether the triggers should run)
  – Action (what happens if the trigger runs)

```sql
CREATE TRIGGER youngSailorUpdate
  AFTER INSERT ON SAILORS
  REFERENCING NEW TABLE NewSailors
  FOR EACH STATEMENT
  INSERT
    INTO YoungSailors(sid, name, age, rating)
  SELECT sid, name, age, rating
  FROM NewSailors N
  WHERE N.age <= 18
```
Summary: SQL

- **SQL** has a huge number of constructs and possibilities
  - You need to learn and practice it on your own
  - Given a problem, you should be able to write a SQL query and verify whether a given one is correct

- **Pay attention to NULLs**

- **Can limit answers using “LIMIT” or “TOP” clauses**
  - e.g. to output TOP 20 results according to an aggregate
  - also can sort using ASC or DESC keywords
Summary

• Relational Data
• SQL
  – Semantic
  – Join
  – Simple Aggregates
  – Nested Queries

• You will learn these further and run yourself on PostGres on Thursday in the in-class lab on SQL!