Announcements

- HW1 is published on Sakai:
  - Resources -> HW -> HW1 folder
  - Due on 09/20 (Thurs), 11:55 pm, no late days
  - Start now!
  - Submission instructions for gradescope to be updated (will be notified through piazza)
- Your piazza and sakai accounts should be active
  - if not on piazza, send me an email
- Occasional Pop up quizzes will start
  - Bring a laptop in class

Recap: Lecture 2

- XML overview
  - differences with relational model and transformation
- SQL
  - Creating/modifying relations
  - Specifying integrity constraints
  - Key/candidate key, superkey, primary key, foreign key
  - Conceptual evaluation of SQL queries

Today's topic

- More SQL
  - joins
  - group bys and aggregates
  - nested queries
  - NULLs
  - views

Acknowledgement:
The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.

Joins

- Condition/Theta-Join
- Equi-Join
- Natural-Join
- (Left/Right/Full) 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>name</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>name</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>
**Find sid’s of sailors who’ve reserved a red or a green boat**

- Assume a Boats relation

- **UNION:** Can be used to compute the union of any two union-compatible sets of tuples
  - can themselves be the result of SQL queries

- **Also available:** `EXCEPT` (What do we get if we replace `UNION` by `EXCEPT`?)

**Find sid’s of sailors who’ve reserved a red and a green boat**

- Sailors (`sid, sname, rating, age`)
- Reserve(`sid, bid, day`)
- Boats (`bid, bname, color`)

**Equi Join**

\[
\text{SELECT * FROM Sailors S, Reserves R WHERE S.sid = R.sid and age = 45}
\]

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>cid</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>31</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>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

A special case of theta join

Join condition only has equality predicate +

**Natural Join**

\[
\text{SELECT * FROM Sailors S NATURAL JOIN Reserves R}
\]

<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>22</td>
<td>101</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>58</td>
<td>103</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>31</td>
<td>101</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>58</td>
<td>103</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>58</td>
<td>103</td>
</tr>
</tbody>
</table>

A special case of equi join

Equality condition on ALL common predicates (sid)

Duplicate columns are eliminated

**Outer Join**

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

<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>31</td>
<td>null</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

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

**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, `'B_%B'` stands for any one character and `%` stands for 0 or more arbitrary characters
  - You will need these often

\[
\text{SELECT } S \text{.age, age1=S.age-5, age2=S.age AS age2 FROM Sailors S WHERE S.sname LIKE 'B_%B'}
\]
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)

- 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

Nested Queries with Correlation

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

- SELECT S.sname FROM Sailors S
  WHERE EXISTS (SELECT * FROM Reserves R WHERE R.bid=103 AND S.sid=R.sid)

- EXISTS is another set comparison operator, like IN
  - Illustrates why, in general, subquery must be recomputed for each Sailors tuple

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

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

Find sid’s of sailors who’ve reserved a red and a green boat

- SELECT sid FROM Reserves WHERE color IN ('red', 'green')
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):

  $$\text{For } i = 1, 2, \ldots, 10: \quad \text{SELECT } \text{MIN} (S.\text{age}) \text{ FROM Sailors S } \text{WHERE } S.\text{rating} = i$$

Queries With GROUP BY and HAVING

- The target-list contains
  - (i) attribute names
  - (ii) terms with aggregate operations (e.g., MIN (S.age))
- The attribute list (i) must be a subset of grouping-list
  - Intuitively, each answer tuple corresponds to a group, and these attributes must have a single value per group
  - Here a group is a set of tuples that have the same value for all attributes in grouping-list

Conceptual Evaluation

- The cross-product of relation-list is computed
- Tuples that fail qualification are discarded
- Unnecessary’ fields are deleted
- The remaining tuples are partitioned into groups by the value of attributes in grouping-list
- The group-qualification is then applied to eliminate some groups
- Expressions in group-qualification must have a single value per group
  - In effect, an attribute in group-qualification that is not an argument of an aggregate op also appears in grouping-list
  - like “…GROUP BY bid, sid HAVING bid = 3”
- One answer tuple is generated per qualifying group

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

<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>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>

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

<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>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>
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.

Nulls and Views in SQL

Standard Boolean 2-valued logic

- True = 1, False = 0
- Suppose X = 5
  - (X < 100) AND (X >= 1) is T \( \wedge \) T = T
  - (X > 100) OR (X >= 1) is F \( \vee \) T = T
  - (X > 100) AND (X <= 1) is F \( \wedge \) T = F
  - NOT(X = 5) is \( \neg \)T = F
- Intuitively,
  - \( T \ominus 1, F \ominus 0 \)
  - For V1, V2 = (1, 0)
  - V1 \( \wedge \) V2 = MIN(V1, V2)
  - V1 \( \vee \) V2 = MAX(V1, V2)
  - \( \neg \)(V1) = 1 \( \ominus \) V1
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 V1, V2 = (1, 0, 0.5)
  - V1 \ V2 = MIN(V1, V2)
  - V1 \ V2 = MAX(V1, V2)
  - ¬(V1) = 1 – V1
- Therefore,
  - NOT UNKNOWN = UNKNOWN
  - UNKNOWN OR TRUE = TRUE
  - UNKNOWN AND FALSE = 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!
- But NULL allows new operators (e.g. outer joins)
- 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?
Aggregates with NULL

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

- What do you get for
  - `SELECT count(*)` from `R2`?
  - `SELECT count(rating)` from `R2`?

Views

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

“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

Can create a new table from a query on other tables too

```sql
SELECT .. INTO .. FROM .. WHERE

SELECT S.name, E.grade INTO YoungActiveStudents
FROM Students S, Enrolled E
WHERE S.sid = E.sid and S.age < 21
```
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)

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

Summary

- 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

Rewriting INTERSECT Queries Using IN

Find sid's of sailors who've reserved both a red and a green boat:

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid = R.sid
AND R.bid = B.bid
AND B.color = 'red'
AND S.sid IN (SELECT S2.sid
FROM Sailors S2, Boats B2, Reserves R2
WHERE S2.sid = R2.sid
AND R2.bid = B2.bid
AND B2.color = 'green')
```

Find name and age of the oldest sailor(s)

- The first query is illegal!
  - Recall the semantic of GROUP BY
- The third query is equivalent to the second query
  - and is allowed in the SQL/92 standard, but is not supported in some systems

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

```
SELECT S.sname, S.age
FROM Sailors S
WHERE (SELECT MAX(S2.age)
FROM Sailors S2)
= S.age
```
For each red boat, find the number of reservations for this boat

SELECT B.bid, COUNT(*) AS count
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND B.bid=R.bid AND B.color='red'
GROUP BY B.bid

- Grouping over a join of three relations.
- What do we get if we remove \( B.color='red' \) from the \( WHERE \) clause and add a \( HAVING \) clause with this condition?
- What if we drop Sailors and the condition involving \( S.sid \)?

Find age of the youngest sailor with age \( \geq 18 \), for each rating with at least 2 such sailors and with every sailor under 60.

SELECT S.rating, MIN(S.age) AS minage
FROM Sailors S
WHERE S.age \( \geq 18 \) AND S.age \( \leq 60 \)
GROUP BY S.rating
HAVING COUNT(*) > 1

Answer relation: Sailors instance:

<table>
<thead>
<tr>
<th>sid</th>
<th>name</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: rating | minage
---|---
3 | 25.5
7 | 35.0
8 | 25.5

Select S.rating, MIN(S.age) AS minage
FROM Sailors S
WHERE S.age = 18
GROUP BY S.rating
HAVING COUNT(*) > 1 AND EVERY(S.age <= 60)