SQL: Queries, Constraints, Triggers
Part 2

Chapter 5.5-5.10
Screw-up from last lecture

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

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

- If `UNIQUE` is used to qualify the inner `SELECT` (I assume in place of `EXISTS`), and `*` is replaced by `R.bid`, then the query finds sailors with exactly one reservation for boat #103.

- The book says (page 148):
  “Closely related to EXISTS is the UNIQUE predicate. When we apply UNIQUE to a subquery, the resulting condition returns true if no row appears twice in the answer to the subquery, that is, there are no duplicates; in particular, it returns true if the answer is empty.”
Screw-up from last lecture

Here’s what I get...

```python
>>> import sqlite3
>>> cursor = db.cursor()
>>> import sqlite3
>>> db = sqlite3.connect("sailors.db")
>>> cursor.execute("""SELECT S.sname
... FROM Sailors S
... WHERE UNIQUE (SELECT R.bid
... FROM Reserves R
... WHERE R.bid=103 AND S.sid=R.sid)""")
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
sqlite3.OperationalError: near "UNIQUE": syntax error
```  

Conclusion #1:
Our “SQL” implementation does not have a “UNIQUE” Predicate

Note: ORACLE does claim to implement a predicate called UNIQUE
Furthermore, Recall Our Infant Database

- Need to add an interesting reservation

**Sailors:**

<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>
</tbody>
</table>

**Reserves:**

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/98</td>
</tr>
<tr>
<td>22</td>
<td>102</td>
<td>10/10/98</td>
</tr>
<tr>
<td>22</td>
<td>103</td>
<td>10/8/98</td>
</tr>
<tr>
<td>22</td>
<td>104</td>
<td>10/7/98</td>
</tr>
<tr>
<td>31</td>
<td>102</td>
<td>11/10/98</td>
</tr>
<tr>
<td>31</td>
<td>103</td>
<td>11/6/98</td>
</tr>
<tr>
<td>31</td>
<td>104</td>
<td>11/12/98</td>
</tr>
<tr>
<td>64</td>
<td>101</td>
<td>9/5/98</td>
</tr>
<tr>
<td>64</td>
<td>102</td>
<td>9/8/98</td>
</tr>
<tr>
<td>74</td>
<td>103</td>
<td>9/8/98</td>
</tr>
<tr>
<td>74</td>
<td>104</td>
<td></td>
</tr>
</tbody>
</table>

```python
>>> cursor.execute("""SELECT *
FROM Reserves R
WHERE bid=103"")
<sqlite3.Cursor object at 0x1004309d0>
```

```python
>>> for row in cursor:
    ... print row
...
(22, 103, u'1998-10-08')
...
```

```sql
SELECT * FROM Reserves WHERE bid=103;
```
Screw-up from last lecture

- Perhaps they meant “DISTINCT”

```python
>>> cursor.execute('"""SELECT S.sname
... FROM Sailors S
... WHERE EXISTS (SELECT DISTINCT R.bid
... FROM Reserves R
... WHERE R.bid=103 AND S.sid=R.sid)"""

<sqlite3.Cursor object at 0x1004309d0>

>>> for row in cursor:
...     print row
...
(u'Dustin',)
(u'Lubber',)
(u'Horatio',)
```

- Doesn’t work… DISTINCT gathers copies of the same tuple, whereas UNIQUE (as suggested) would test that a tuple appears exactly once
- Conclusion #2: They really meant “UNIQUE”
Screw-up from last lecture

- Can you do it using pure relational algebra without UNIQUE?

```python
>>> cursor.execute('''
SELECT S.sname
FROM Sailors S
WHERE S.sid NOT IN (SELECT R1.sid
                      FROM Reserves R1, Reserves R2
                      WHERE R1.bid=103 AND R2.bid=103
                      AND R1.sid=R2.sid AND R1.day<>R2.day)
AND S.sid IN (SELECT R3.sid
               FROM Reserves R3
               WHERE R3.bid=103 AND S.sid=R3.sid)''')
<sqlite3.Cursor object at 0x1004309d0>
>>> for row in cursor:
    ...    print row
...
(u'Lubber',)
(u'Horatio',)
```
Aggregate Operators

- Significant extension of relational algebra.
- Computation and summarization operations
- Result *aggregates* rather than each individually
- E.x. How many Sailor instances in the sailor relation?

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

- Average age of Sailors with a rating of 10?

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

- Names of all Sailors who have achieved the maximum rating

  ```sql
  SELECT S.sname
  FROM Sailors S
  WHERE S.rating = (SELECT MAX(S2.rating)
                      FROM Sailors S2)
  ```
More examples (cont)

- How many distinct ratings for Sailors less than 40 years of age?

```
SELECT COUNT (DISTINCT S.rating)
FROM Sailors S
WHERE S.age < 40.0
```

- Names of all Sailors who have achieved the maximum rating

```
SELECT AVG (DISTINCT S.age)
FROM Sailors S
WHERE S.rating=10
```
Find name and age of the oldest sailor(s)

- First query is illegal! (Aggregation operators can only appear in target-list)
- Second is illegal! (A SELECT clause that uses an aggregate operation must only use aggregate operations)
- The third query is correct.

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

```sql
SELECT S.sname, MAX(S.age)
FROM Sailors S
```

```sql
SELECT S.sname, S.age
FROM Sailors S
WHERE S.age =
(SELECT MAX (S2.age)
FROM Sailors S2)
```
Return of the Screw-up

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

SELECT S.sname
FROM Sailors S
WHERE (SELECT COUNT(*)
  FROM Reserves R
  WHERE R.bid=103 AND S.sid=R.sid) = 1

This is how I would have approached the query that I choked on last lecture.
Motivation for Grouping

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

- 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 (!):

```sql
SELECT MIN(S.age) FROM Sailors S WHERE S.rating = i
```
For $i = 1, 2, \ldots, 10$: 
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. (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, and 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. (SQL does not exploit primary key semantics here!)

- One answer tuple is generated per qualifying group.
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 >= 18
GROUP BY S.rating
HAVING COUNT(*) > 1
```

**Sailors instance:**

<table>
<thead>
<tr>
<th>sid</th>
<th>surname</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:**

<table>
<thead>
<tr>
<th>rating</th>
<th>minage</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
</tbody>
</table>
Find age of the youngest sailor with age $\geq 18$, for each rating with at least 2 such sailors.
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.

HAVING \texttt{COUNT (*)} > 1 AND EVERY (S.age $\leq$ 60)

What is the result of changing \texttt{EVERY} to \texttt{ANY}?
**Find age of the youngest sailor with age ≥ 18, for each rating with at least 2 sailors between 18 and 60.**

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

**Answer relation:**

<table>
<thead>
<tr>
<th>rating</th>
<th>minage</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
</tbody>
</table>

**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>
For each red boat, find the number of reservations for this boat

```
SELECT  B.bid, COUNT (*) AS scount
FROM    Sailors S, Boats B, Reserves R
WHERE   S.sid=R.sid AND R.bid=B.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 > 18, for each rating with at least 2 sailors (of any age)

SELECT S.rating, MIN (S.age)
FROM Sailors S
WHERE S.age > 18
GROUP BY S.rating
HAVING 1 < (SELECT COUNT (*)
            FROM Sailors S2
            WHERE S.rating=S2.rating)

❖ Shows HAVING clause can also contain a subquery.
❖ Compare this with the query where we considered only ratings with 2 sailors over 18!
❖ What if HAVING clause is replaced by:
  ▪ HAVING COUNT(*) >1
Find those ratings for which the average age is the minimum over all ratings

- Aggregate operations cannot be nested!  **WRONG:**

  ```sql
  SELECT S.rating
  FROM Sailors S
  WHERE S.age = (SELECT MIN (AVG (S2.age)) FROM Sailors S2)
  ```

- Correct solution (in SQL/92):

  ```sql
  SELECT Temp.rating, Temp.avgage
  FROM (SELECT S.rating, AVG (S.age) AS avgage
          FROM Sailors S
          GROUP BY S.rating) AS Temp
  WHERE Temp.avgage = (SELECT MIN (Temp.avgage) FROM Temp)
  ```
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.

- The presence of `null` complicates many issues. E.g.:
  - Special operators needed to check if value is/is not `null`.
  - Is `rating>8` true or false when `rating` is equal to `null`? What about AND, OR and NOT connectives?
  - We need a 3-valued logic (true, false and unknown).
  - Meaning of constructs must be defined carefully. (e.g., WHERE clause eliminates rows that don’t evaluate to true.)
  - New operators (in particular, outer joins) possible/needed.
**Integrity Constraints (Review)**

- An IC describes conditions that every *legal instance* of a relation must satisfy.
  - Inserts/deletes/updates that violate IC’s are disallowed.
  - Can be used to ensure application semantics (e.g., *sid* is a key), or prevent inconsistencies (e.g., *sname* has to be a string, *age* must be < 200)

- **Types of IC’s**: Domain constraints, primary key constraints, foreign key constraints, general constraints.
  - *Domain constraints*: Field values must be of right type. Always enforced.
General Constraints

- Useful when more general ICs than keys are involved.
- Can use queries to express constraint.
- Constraints can be named.

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

- Useful when more general ICs than keys are involved.
- Can use queries to express constraint.
- Constraints can be named.

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)))
Constraints Over Multiple Relations

- Awkward and wrong!
- If Sailors is empty, the number of Boats tuples can be anything!
- ASSERTION is the right solution; not associated with either table.

```
CREATE TABLE Sailors(
    sid INTEGER,
    sname CHAR(10),
    rating INTEGER,
    age REAL,
    PRIMARY KEY (sid),
    CHECK
    ( (SELECT COUNT (S.sid) FROM Sailors S) 
    + (SELECT COUNT (B.bid) FROM Boats B) < 100 )

CREATE ASSERTION smallClub
CHECK
( (SELECT COUNT (S.sid) FROM Sailors S) 
+ (SELECT COUNT (B.bid) FROM Boats B) < 100 )
```

Number of boats plus number of sailors is < 100
Triggers

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

- Triggers have 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(sid, name, age, rating)
    SELECT sid, name, age, rating
    FROM NewSailors N
    WHERE N.age <= 18
Summary

- SQL was an important factor in the early acceptance of the relational model; more natural than earlier, procedural query languages.
- Relationally complete; in fact, significantly more expressive power than relational algebra.
- Even queries that can be expressed in RA can often be expressed more naturally in SQL.
- Many alternative ways to write a query; optimizer should look for most efficient evaluation plan.
  - In practice, users need to be aware of how queries are optimized and evaluated for best results.
Summary (Contd.)

- NULL for unknown field values brings many complications
- SQL allows specification of rich integrity constraints
- Triggers respond to changes in the database