SQL: Joins, Constraints & Triggers

Problem Set #1 is due before midnight next Tuesday.

Problem Set #2 is posted either tonight or tomorrow morning.
Controlling Output Order

- SQL’s “ORDER BY” clause is used to sort tuples in either ascending or descending order.
- ORDER BY specifies attributes used in the sort.

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>Brutus</td>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>85</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>95</td>
<td></td>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td>15</td>
<td>25.5</td>
</tr>
<tr>
<td>74</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>74</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>58</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>85</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>95</td>
<td></td>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>29</td>
<td></td>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>95</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td>1</td>
<td>33.0</td>
</tr>
</tbody>
</table>

SELECT * FROM Sailors WHERE age > 18 ORDER BY rating

SELECT * FROM Sailors WHERE age > 18 ORDER BY rating DESC

SELECT * FROM Sailors WHERE age > 18 ORDER BY rating DESC, sname ASC
Controlling output size

- The “LIMIT” clause is used to limit the number of tuples returned by a “SELECT” statement.
- Useful for seeing a small number of examples, or “top-X” in combination with “ORDER BY”.

```
SELECT *
FROM Sailors
LIMIT 5
```

```
<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>
</tbody>
</table>
```

```
SELECT *
FROM Sailors
ORDER BY rating DESC
LIMIT 5
```

```
<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>58</td>
<td>Rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>74</td>
<td>Horatio</td>
<td>9</td>
<td>35.0</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>22</td>
<td>Dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
</tbody>
</table>
```
Changing a table value

Thus far we've inserted entire rows into tables, but SQL also provides commands for UPDATEs

```
UPDATE  relation
SET     attr-value-list
WHERE   qualification
```

An attr-value-list is a comma separated list of

\[ \text{attribute}_1 = \text{expression}_1, \text{attribute}_2 = \text{expression}_2 \]

The WHERE qualification can be any valid set of filtering terms including nest queries with IN and EXCEPT, but usually it is a $<\text{primary key}> = \text{value}$
UPDATE example

UPDATE Sailors
SET rating = rating + 1
WHERE sid = 85

UPDATE Sailors
SET rating = rating + 1
WHERE rating < 10
AND surname LIKE "%us%"
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?
  ▪ Creates the need for 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.)

❖ Joins can also generate null entries
Creating a Tiny database

Using iSQL.parser("tiny.db", mode='w'), you can execute the following:

Sailors:

CREATE TABLE Sailors(
   sid INTEGER PRIMARY KEY,
   sname TEXT,
   rating INTEGER,
   age REAL)

INSERT INTO Sailors(sid,sname,rating,age)
   VALUES (22, 'dustin', 7, 45.0),
          (31, 'lubber', 8, 55.5),
          (58, 'rusty', 10, 35.0)

SELECT * FROM Sailors

The PRIMARY KEY designation is a simple CONSTRAINT in SQL. Each PRIMARY KEY must be unique, and whether it is is checked and enforced on INSERTS.
Creating a Tiny database

Using `iSQL.parser("tiny.db", mode='w')`, you can execute the following:

**Boats:**

```
CREATE TABLE Boats(
    bid INTEGER PRIMARY KEY,
    bname TEXT,
    color TEXT)
```

```
INSERT INTO Boats
VALUES (101, 'Interlake', 'blue'),
       (102, 'Interlake', 'red'),
       (103, 'Clipper', 'green')
```

```
SELECT * FROM Boats
```
Creating a Tiny database

And now a relation between these two entities:

**Reserves:**

```sql
CREATE TABLE Reserves(
    sid INTEGER,
    bid INTEGER,
    day DATE,
    PRIMARY KEY(sid,bid),
    FOREIGN KEY(sid) REFERENCES Sailors(sid),
    FOREIGN KEY(bid) REFERENCES Boats(bid)
);

INSERT INTO Reserves
VALUES(22, 101, '1996-10-10'),
(31, 103, '1996-11-12');

SELECT * FROM Reserves;
```

A composite PRIMARY KEY (i.e. composed of more than one attribute) is defined separately at the end of the CREATE.

A FOREIGN KEY is another common constraint. It implies that this attribute is type compatible with the referenced attribute in another table. Optionally it can disable insertions unless the value inserted matches a value in a row with the referenced table.
Types of JOINS

- Tables from our “tiny” sailor database

<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>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</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>1996-10-10</td>
</tr>
<tr>
<td>31</td>
<td>103</td>
<td>1996-11-12</td>
</tr>
</tbody>
</table>

- An “implied” join (in the WHERE clause)

```sql
SELECT S.sname, R.day
FROM Sailors S, Reserves R
WHERE S.sid=R.sid
```

- An “explicit” join (in the FROM clause)

```sql
SELECT S.sname, R.day
FROM Sailors S JOIN Reserves R ON S.sid=R.sid
```

- `INNER` implies *ONLY* tuples that share the join condition appear in the result set. It is the default JOIN.

- `NATURAL` implies that rows from each table are combined if
  1. they have the same attribute name
  2. they have the same attribute value
Left JOINS

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>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</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>1996-10-10</td>
</tr>
<tr>
<td>31</td>
<td>103</td>
<td>1996-11-12</td>
</tr>
</tbody>
</table>

Boats:

<table>
<thead>
<tr>
<th>bid</th>
<th>bname</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Interlake</td>
<td>blue</td>
</tr>
<tr>
<td>102</td>
<td>Interlake</td>
<td>red</td>
</tr>
<tr>
<td>103</td>
<td>Clipper</td>
<td>green</td>
</tr>
</tbody>
</table>

- A “Left” JOIN returns a tuple for every row of the first, “left”, relation, even if it requires adding “Null” values to the output relations

```
SELECT S.sname, R.day
FROM Sailors S LEFT JOIN Reserves R ON S.sid=R.sid
```

```
SELECT S.sname, R.day
FROM Sailors S NATURAL LEFT JOIN Reserves R
```

- Notice that every row from Sailors has a corresponding row in the result
(BTW Null maps to None in Python)
Right JOINS

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>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</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>1996-10-10</td>
</tr>
<tr>
<td>31</td>
<td>103</td>
<td>1996-11-12</td>
</tr>
</tbody>
</table>

Boats:

<table>
<thead>
<tr>
<th>bid</th>
<th>bname</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Interlake</td>
<td>blue</td>
</tr>
<tr>
<td>102</td>
<td>Interlake</td>
<td>red</td>
</tr>
<tr>
<td>103</td>
<td>Clipper</td>
<td>green</td>
</tr>
</tbody>
</table>

- Likewise a “Right” join returns a tuple for every row in the second, “right”, relation.

```
SELECT R.day, B.bname
FROM Reserves R NATURAL RIGHT JOIN Boats B
```

- Here there is a corresponding row in the result for every row in "Boats".

```
SELECT R.day, B.bname
FROM Boats B NATURAL LEFT JOIN Reserves R
```

Some databases (like the one we’ll use this semester) do not support right joins. But, left and right are arbitrary.
The FULL OUTER JOIN keyword returns all rows from all tables with the specified attributes joined or null if there is no match.

```
SELECT S.sname, R.day, B.bname
FROM (Sailors S NATURAL LEFT JOIN Reserves R)
    FULL OUTER JOIN Boats B ON R.bid=B.bid
```
Emulating FULL OUTER JOIN

We can always emulate a FULL JOIN using the UNION of two oriented JOINs

```sql
SELECT S.sname, R.day, B.bname
FROM (Sailors S NATURAL LEFT JOIN Reserves R) LEFT JOIN Boats B USING(bid)
UNION
SELECT S.sname, R.day, B.bname
FROM Boats B LEFT JOIN (Sailors S NATURAL LEFT JOIN Reserves R) USING(bid)
```

<table>
<thead>
<tr>
<th>sname</th>
<th>day</th>
<th>bname</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>None</td>
<td>Interlake</td>
</tr>
<tr>
<td>dustin</td>
<td>1996-10-10</td>
<td>Interlake</td>
</tr>
<tr>
<td>lubber</td>
<td>1996-11-12</td>
<td>Clipper</td>
</tr>
<tr>
<td>rusty</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>

Same answer as before, since order doesn't matter
Integrity Constraints (IC)

- 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 nonempty 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 Constraint CHECKs

- CHECK clause
- Useful when more general ICs than keys are involved.
- Example: All ratings must be between 1 and 10

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

Comp 521 – Files and Databases

Fall 2020
More complicated CHECKs

- Constraints can be named.
- Checks can contain nested subqueries
- Example: Disallow reservations of boats named “Interlake”
  ```sql
  CREATE TABLE Reserves(
  sid INTEGER,
  bid INTEGER,
  day DATE,
  PRIMARY KEY (bid,day),
  CONSTRAINT NoInterlakeIfLessThan7 CHECK ('Interlake' <> (SELECT B.bname FROM Boats B WHERE B.bid=bid)
  OR 7 <= (SELECT S.rating FROM Sailor S WHERE S.sid=sid))
  ```
- “bid” and “sid” refer to values from the associated INSERT or UPDATE
Constraints Over Multiple Relations

- Awkward and wrong!
- Check is done only when inserting Sailors. What about Boats?
- 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 )
```
Triggers

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

❖ Triggers have three parts:
  - *Event* (that activates the trigger)
  - *Condition* (tests whether the triggers should run)
  - *Action* (what happens if the trigger runs)
Triggers: Example

- Suppose there was a rule that “no one with a rating less than 5 can reserve a green boat”. The following trigger would enforce this rule, and generate a failure message:

```sql
CREATE TRIGGER RatingRuleForGreen
BEFORE INSERT ON Reserves
BEGIN
SELECT RAISE(FAIL, 'Sailor is not qualified')
WHERE EXISTS (SELECT * FROM Sailors, Boats
WHERE sid = new.sid AND rating < 5
AND bid = new.bid AND color = 'green');
END;
```

- Note the special variable `new` is used for accessing parameters of the invoking INSERT query.
Changes in one table can cause side-effects in other tables via triggers

Example “Event Logging”

We know dates of reservations, but not when they were made. This can be remedied using a trigger as follows:

```sql
CREATE TRIGGER insertLog
AFTER INSERT ON Reserves
BEGIN
    INSERT INTO ReservesLog (sid, bid, resDate, madeDate)
    VALUES (new.sid, new.bid, new.date, DATE('NOW'));
END;
```
Summary

❖ NULLs provide a means for representing “unspecified” attribute values
❖ NULLs can be generated by special JOINs
❖ Wide range of JOIN operations-- Some retain the cardinality of specified relations
❖ SQL allows specification of rich integrity constraints
❖ Triggers respond to changes in the database