SQL: Queries, Constraints, Triggers

Chapter 5

DML versus DDL

- Data Manipulation Language (DML)
  - posing queries and operating on tuples
- Data Definition Language (DDL)
  - operating on tables/views

Example Instances

- We will use these instances of the Sailors and Reserves relations in our examples.
- If the key for the Reserves relation contained only the attributes sid and bid, how would the semantics differ?
Basic SQL Query

```sql
SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification
```

- **relation-list** A list of relation names (possibly with a range-variable after each name).
- **target-list** A list of attributes of relations in relation-list
- **qualification** Comparisons (Attr op const or Attr1 op Attr2, where op is one of <, >, =, ≤, ≥, ≠) combined using AND, OR and NOT.
- **DISTINCT** is an optional keyword indicating that the answer should not contain duplicates. Default is that duplicates are not eliminated!

Conceptual Evaluation Strategy

- **Semantics of an SQL query** defined in terms of the following conceptual evaluation strategy:
  - Compute the cross-product of relation-list.
  - Discard resulting tuples if they fail qualifications.
  - Delete attributes that are not in target-list.
  - If DISTINCT is specified, eliminate duplicate rows.
- This strategy is probably the least efficient way to compute a query! An optimizer will find more efficient strategies to compute the same answers.

Example of Conceptual Evaluation

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

<table>
<thead>
<tr>
<th>(sid)</th>
<th>name</th>
<th>rating</th>
<th>age</th>
<th>(sid)</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Justin</td>
<td>45.0</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Justin</td>
<td>45.0</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>55.5</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>55.5</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>Rusty</td>
<td>35.0</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>Rusty</td>
<td>35.0</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
<td></td>
</tr>
</tbody>
</table>
A Note on Range Variables

- Really needed only if the same relation appears twice in the FROM clause. The previous query can also be written as:

```sql
SELECT S.name
FROM Sailors S, Reserves R
WHERE S.sid=R.sid AND bid=103
```

OR

```sql
SELECT S.name
FROM Sailors S, Reserves R
WHERE S.sid=R.sid AND bid=103
```

It is good style, however, to use range variables always!

Find sailors who’ve reserved at least one boat

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

- Would adding DISTINCT to this query make a difference?
- What is the effect of replacing S.sid by S.sname in the SELECT clause? Would adding DISTINCT to this variant of the query make a difference?

Expressions and Strings

```sql
SELECT S.age, age1=S.age-5, 2*S.age AS age2
FROM Sailors S
WHERE S.sname LIKE 'B_%B'
```

- 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.
- AS and = are two ways to name fields in result.
- LIKE is used for string matching. _ stands for any one character and % stands for 0 or more arbitrary characters.
UNION: Can be used to compute the union of any two union-compatible sets of tuples (which are themselves 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')
```

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.
- Contrast symmetry of the UNION and INTERSECT queries with how much the other versions differ.

```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')
```

**Nested Queries**

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

```sql
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 clause can itself contain an SQL query! (Actually, so can FROM and HAVING clauses.)
- 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:

```
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 re-computed 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 >,<,<=,<>`
- Find sailors whose rating is greater than that of some sailor called Horatio:

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

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')
```

- Similarly, EXCEPT queries re-written using NOT IN.
- To find names (not sid’s) of Sailors who’ve reserved both red and green boats, just replace `S.sid` by `S.sname` in SELECT clause. (What about INTERSECT query?)
Division in SQL

Find sailors who’ve reserved all boats.

Let’s do it the hard way, without EXCEPT:

(1) SELECT S.sname
    FROM Sailors S
    WHERE NOT EXISTS (SELECT B.bid
                        FROM Boats B
                        EXCEPT
                        (SELECT R.bid
                         FROM Reserves R
                         WHERE R.bid=B.bid
                         AND R.sid=S.sid)))

Sailors S such that ...
there is no boat B without ...
a Reserves tuple showing S reserved B

Aggregate Operators

Significant extension of relational algebra.

<table>
<thead>
<tr>
<th>COUNT (*)</th>
<th>COUNT ([DISTINCT] A)</th>
<th>SUM ([DISTINCT] A)</th>
<th>AVG ([DISTINCT] A)</th>
<th>MAX (A)</th>
<th>MIN (A)</th>
</tr>
</thead>
</table>
| SELECT COUNT (*)
  FROM Sailors S |
| SELECT S.sname
  FROM Sailors S
  WHERE S.rating=10 |
| 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 name and age of the oldest sailor(s)

The first query is illegal! (We’ll look into the reason a bit later, when we discuss 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.
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 (!):

```sql
For i = 1, 2, ..., 10:
  SELECT MIN (S.age)
  FROM Sailors S
  WHERE S.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. (A group is a set of tuples that have the same value for all attributes in grouping-list.)

```sql
SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification
GROUP BY grouping-list
HAVING group-qualification
```

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 ≥ 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
```

Answer relation:

<table>
<thead>
<tr>
<th>rating</th>
<th>minage</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>9</td>
<td>25.5</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>11</td>
<td>35.0</td>
</tr>
<tr>
<td>12</td>
<td>55.5</td>
</tr>
<tr>
<td>13</td>
<td>25.5</td>
</tr>
<tr>
<td>14</td>
<td>35.0</td>
</tr>
</tbody>
</table>

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

```
rating age  
7  45.0 
8  55.5 
9  35.0 
10 35.0 
```

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

```
HAVING COUNT(*) > 1 AND EVERY (S.age <= 60)
```

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

**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>nauty</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>zebra</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>94</td>
<td>bob</td>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>96</td>
<td>Fido</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>

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) AS minage
FROM Sailors S
WHERE S.age > 18
GROUP BY S.rating
HAVING COUNT(*) > 1
```

- 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.)

**Adding and Deleting Tuples**

- Can insert a single tuple using:

  ```sql
  INSERT INTO Students (sid, name, login, age, gpa) VALUES (53688, 'Smith', 'smith@ee', 18, 3.2)
  ```

- Can delete all tuples satisfying some condition (e.g., name = Smith):

  ```sql
  DELETE FROM Students S
  WHERE S.name = 'Smith'
  ```

  **Powerful variants of these commands are available; more later!**
Updating Tuples

Creating Relations in SQL

- Creates the Students relation. Observe that the type (domain) of each field is specified, and enforced by the DBMS whenever tuples are added or modified.
- As another example, the Enrolled table holds information about courses that students take.

```
CREATE TABLE Students
(sid: CHAR(20),
name: CHAR(20),
login: CHAR(10),
age: INTEGER,
gpa: REAL)
```

```
CREATE TABLE Enrolled
(sid: CHAR(20),
cid: CHAR(20),
grade: CHAR(2))
```

Destroying and Altering Relations

- Destroys the relation Students. The schema information and the tuples are deleted.
- The schema of Students is altered by adding a new field; every tuple in the current instance is extended with a null value in the new field.

```
DROP TABLE Students
```

```
ALTER TABLE Students
ADD COLUMN firstYear: integer
```

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.
  - How to handle **DROP TABLE** if there’s a view on the table?
  - **DROP TABLE** command has options to let the user specify this.

Questions