**SQL: Queries, Programming, Triggers**

Chapter 5

---

**Example Instances**

<table>
<thead>
<tr>
<th>RI</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>s1</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

- 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?

---

**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 qualification.
  - 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.name, S.rating, S.age
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>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>45.0</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>45.0</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>55.5</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>55.5</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>35.0</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>35.0</td>
<td>58</td>
<td>103</td>
<td>11/12/96</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
SELECT name
FROM Sailors, Reserves
WHERE Sailors.sid = Reserves.sid
AND bid = 103
```

It is good style, however, to use range variables always.
Find sailors who’ve reserved at least one boat

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.name in the SELECT clause? Would adding DISTINCT to this variant of the query make a difference?

Expressions and Strings

SELECT age = Sage - 5, 2 * Sage AS age2
FROM Sailors S
WHERE S.name LIKE 'B_%.f'

- 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 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.

Find sides 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 (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?)

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

- 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.

Nested Queries

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

SELECT S.name
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 loop 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.name
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.
- 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; * denotes all attributes. Why do we have to replace * by R.bid?)
- Illustrates why, in general, subquery must be re-computed for each Sailors tuple.
More on Set-Comparison Operators

- We've already seen 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:
  ```sql
  SELECT *
  FROM Sailors S
  WHERE S.rating > ANY (SELECT S2.rating
  FROM Sailors S2
  WHERE S2.name = 'Horatio')
  ```

Rewriting INTERSECT Queries Using IN

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

```sql
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 rewritten 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.name 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:
  - SELECT S.name
    FROM Sailors S
    WHERE NOT EXISTS (SELECT R.bid
    FROM Boats B
    WHERE B.sid = S.sid)
  - SELECT S.name
    FROM Sailors S
    WHERE NOT EXISTS (SELECT R.bid
    FROM Boats B
    WHERE B.sid = S2.sid)

Aggregate Operators

- Significant extension of relational algebra.
  ```sql
  SELECT COUNT(*)
  FROM Sailors S
  WHERE S.rating = (SELECT MAX(S2.rating)
  FROM Sailors S2
  WHERE S2.name = 'Bob'
  AND S2.rating = 10)
  ```

GROUP BY and HAVING

- 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:
  ```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 (ii) must be a subset of grouping-list.

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

- Only S.rating and S.age are mentioned in the SELECT, GROUP BY or HAVING clauses; other attributes 'unnecessary'.
- 2nd column of result is unnamed. (Use AS to name it.)

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

- Grouping over a join of three relations
- What if we drop sailors and the condition involving S.sid?

Find the age of the youngest sailor with age > 18, for each rating with at least 2 sailors (of any age)

- Aggregate operations cannot be nested: WRONG.
- What if HAVING clause is replaced by:
  - HAVING COUNT(*) > 1

Find those ratings for which the average age is the minimum over all ratings

- 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.
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 NOT connectors?
  - We need a 3-way 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.

Embedded SQL

- SQL commands can be called from within a host language (e.g., C or COBOL) program.
  - SQL statements can refer to host variables (including special variables used to return status).
  - Must include a statement to connect to the right database.
- SQL relations are (multi-)sets of records, with no a priori bound on the number of records.
  - No such data structure in C
  - SQL supports a mechanism called a cursor to handle this.

Cursor that gets names of sailors who’ve reserved a red boat, in alphabetical order

```sql
EXEC SQL DECLARE sailor CURSOR FOR
SELECT S.sname FROM Sailors S WHERE S.sid=S.aid AND S.aid=S.bid AND S.color=’red’
ORDER BY S.sname

- Note that it is illegal to replace S.sname by, say, S.aid in the ORDER BY clause! (Why?)
- Can we add S.aid to the SELECT clause and replace S.sname by S.aid in the ORDER BY clause?
```

Embedding SQL in C: An Example

```c
char SQLSTATE[6];
EXEC SQL BEGIN DECLARE SECTION
  char c_sname[20];
  short c_minrating;
  float c_age;
EXEC SQL END DECLARE SECTION
  c_minrating = random();
EXEC SQL DECLARE sailor CURSOR FOR
SELECT S.sname, S.age FROM Sailors S WHERE S.rating > c_minrating
ORDER BY S.sname;
do {
  EXEC SQL FETCH sailor INTO :c_sname, :c_age;
  printf("%s is %d years old", c_sname, c_age);
} while (SQLSTATE != ’02000’);
EXEC SQL CLOSE sailor;
```

Database APIs: Alternative to embedding

Rather than modify compiler, add library with database calls (API)
- special standardized interface procedures/objects
- passes SQL strings from language to external system.
- Microsoft’s ODBC becoming C/C++ standard on Windows
- Sun’s JDBC a Java equivalent
- Supposedly DBMS-neutral
  - A “driver” traps the calls and translates them into DBMS-specific code
  - database can be across a network

Database Management Systems, 5th edition by Silberschatz, Korth and Sudarshan 2010
**SQL API in Java (JDBC)**

```java
Connection con = // connect
DriverManager.getConnection(url, "login", "pass");
Statement stmt = con.createStatement(); // set up stmt
String query = "SELECT name, rating FROM Sailors";
ResultSet rs = stmt.executeQuery(query);
try { // handle exceptions
    while (rs.next()) {
        String s = rs.getString("name");
        int i = rs.getInt("rating");
        System.out.println(s + ", " + i);
    }
} catch (SQLException ex) { // System.out.println(ex.getMessage());
    System.out.println(ex.getSQLState () + ex.getErrorCode () );
}
```

**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 express constraints in application semantics (e.g., sid is a key) or prevent inconsistencies (e.g., name 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.

**Constraints Over Multiple Relations**

```java
CREATE TABLE Sailors
  (sid INTEGER,
   name CHAR(10),
   rating INTEGER,
   age REAL,
   PRIMARY KEY (sid))
CHECK (rating > 1 AND rating <= 100)
```

```java
CREATE TABLE Reserves
  (name CHAR(10),
   bid INTEGER,
   day DATE,
   PRIMARY KEY (bid,day),
   CONSTRAINT multireserve
     CHECK (Interalse < 0)
   (SELECT 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 youngSailors
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.
- Embedded SQL allows execution within a host language; cursor mechanism allows retrieval of one record at a time.
- APIs such as ODBC and ODBC introduce a layer of abstraction between application and DBMS.
- SQL allows specification of rich integrity constraints.
- Triggers respond to changes in the database.