SQL: Queries, Programming, Triggers

Chapter 5

Basic SQL Query

- relation-list: A list of relation names (possibly with a range-variable after each name).
- target-list: A list of attributes 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.

Example of Conceptual Evaluation

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

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>(sid)</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>45.0</td>
<td>10</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>45.5</td>
<td>10</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>10</td>
<td>31</td>
<td>101</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>55.3</td>
<td>10</td>
<td>31</td>
<td>101</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>35.0</td>
<td>10</td>
<td>58</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>35.0</td>
<td>10</td>
<td>58</td>
<td>101</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

Example Instances

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</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 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.

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:

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

Or

```
SELECT snam
FROM   Sailors, Reserves
WHERE  Sailors.sid=Reserves.sid
       AND bid=103
```
**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.surname` 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 - 2, S.age AS age2
FROM Sailors S
WHERE S.surname LIKE '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 names to use fields in result.
- LIKE is used for string matching; _ stands for any one character and % stands for 0 or more arbitrary characters.

**Find sailors who’ve reserved a red or a green boat**

```sql
SELECT S.id
FROM Sailors S, Boats B, Reserves R
WHERE S.id = R.sid AND B.id = R.bid
```

- UNION can be used to compute the union of any two disjoint, 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?)

**Nested Queries**

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

```sql
SELECT S.name
FROM Sailors S
WHERE S.id 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 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.

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

```sql
SELECT S.name
FROM Sailors S
WHERE EXISTS (SELECT R.sid
              FROM Reserves R
              WHERE R.bid = 103 AND S.id = 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 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 >, <, <=, >=.
- Find sailors whose rating is greater than that of some sailor called Horatio:

\[
\begin{align*}
\text{SELECT} & \quad * \\
\text{FROM} & \quad \text{Sailors S} \\
\text{WHERE} & \quad \text{S.rating} > \text{ANY} (\text{SELECT S2.rating} \\
& \quad \text{FROM} \quad \text{Sailors S2} \\
& \quad \text{WHERE} \quad \text{S2.name} = \text{"Horatio"})
\end{align*}
\]

Rewriting INTERSECT Queries Using IN
- Find all sailors who've reserved both a red and a green boat:

\[
\begin{align*}
\text{SELECT} & \quad \text{S.sid} \\
\text{FROM} & \quad \text{Sailors S, Boats B, Reserves R} \\
\text{WHERE} & \quad \text{S.sid} = \text{R.sid} \text{ AND B.bid = B2.bid} \text{ AND B.color = \"red\"} \\
& \quad \text{AND} \quad \text{S.sid} \in (\text{SELECT \text{S2.sid}} \\
& \quad \text{FROM} \quad \text{Sailors S2, Boats B2, Reserves R2} \\
& \quad \text{WHERE} \quad \text{S2.sid} = \text{R2.sid} \text{ AND R2.bid = B2.bid} \\
& \quad \text{AND} \quad \text{B2.color = \"green\"})
\end{align*}
\]

- Similarly, EXCEPT queries rewritten using NOT IN.
- To find names (not all) 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

1. \[\text{(1) SELECT S.name} \\
\quad \text{FROM Sailors S} \\
\quad \text{WHERE NOT EXISTS (SELECT B.bid} \\
\quad \text{FROM Boats B} \\
\quad \text{Select B such that ...} \\
\quad \text{there is no boat B without ...} \\
\quad \text{a Reserves tuple showing S reserved B}\]

Aggregate Operators
- Significant extension of relational algebra.

\[
\begin{align*}
\text{COUNT} () & \quad \text{COUNT} ( [\text{DISTINCT}] \text{ A}) \\
\text{SUM} ( [\text{DISTINCT}] \text{ A}) & \quad \text{AVG} ( [\text{DISTINCT}] \text{ A}) \\
\text{MAX} ( ) & \quad \text{MIN}( )
\end{align*}
\]

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.

\[
\begin{align*}
\text{SELECT} & \quad \text{S.name, MAX (S.age)} \\
\text{FROM} & \quad \text{Sailors S} \\
\text{SELECT} & \quad \text{S.name, S.age} \\
\text{FROM} & \quad \text{Sailors S} \\
\text{WHERE} & \quad \text{S.age} = \text{MAX}(\text{S2.age}) \\
\quad \text{FROM} \quad \text{Sailors S2} \\
\text{SELECT} & \quad \text{S.name, S.age} \\
\text{FROM} & \quad \text{Sailors S} \\
\text{WHERE} & \quad \text{SELECT MAX (S2.age)} \\
\quad \text{FROM} \quad \text{Sailors S2} \\
\quad = \quad \text{S.age}
\end{align*}
\]

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.
  - 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!:

\[
\begin{align*}
\text{For } i = 1, 2, ..., 10: \quad & \quad \text{SELECT MIN (S.age)} \\
& \quad \text{FROM Sailors S} \\
& \quad \text{WHERE S.rating} = i
\end{align*}
\]
Queries With GROUP BY and HAVING

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

- The `target-eties` contains (i) attribute names (ii) terms with aggregate operations (e.g., MIN (S. age)).
- The attribute list (i) must be a subset of `grouping-eties`.
- 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-eties`.)

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

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

- 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

```sql
SELECT B.bid, COUNT(*) AS count
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 the age of the youngest sailor with age > 18, for each rating with at least 2 sailors (of any age)

```sql
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

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

- Aggregate operations cannot be nested! WRONG:

```sql
SELECT Temp.rating, AVG (Temp.age)
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)
```

- Correct solution in SQL/92:

```sql
SELECT Temp.rating, AVG (Temp.age)
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 1 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, order pairs) possibly 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

EXEC SQL DECLARE sinfo CURSOR FOR
SELECT S.name FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND B.bid=R.bid AND B.color='red'
ORDER BY S.name

- Note that it is illegal to replace S.name by, say, S.sid in the ORDER BY clause! (Why?)

- Can we add S.sid to the SELECT clause and replace S.name by S.sid in the ORDER BY clause?

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, presents result sets in a language-friendly way
- 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

Embedding SQL in C: An Example

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 sinfo CURSOR FOR
SELECT S.sname, S.age FROM Sailors S
WHERE S.rating > :c_minrating
ORDER BY S.sname;
do {
EXEC SQL FETCH sinfo INTO :c_sname, :c_age;
printf("%s is %d years old", :c_sname, :c_age);
} while (SQLSTATE != '02000');
EXEC SQL CLOSE sinfo;

Cursor

- Can declare a cursor on a relation or query statement (which generates a relation).
- Can open a cursor, and repeatedly fetch a tuple then move the cursor, until all tuples have been retrieved.
  - Can use a special clause, called ORDER BY, in queries that are accessed through a cursor, to control the order in which tuples are returned.
  - Fields in ORDER BY clause must also appear in SELECT clause.
  - The ORDER BY clause, which orders answer tuples, is only allowed in the context of a cursor.
- Can also modify/delete tuple pointed to by a cursor.

Database Management Systems, R. Ramakrishnan and J. Gehrke
SQL API in Java (JDBC)

```
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 r = rs.getInt("rating");
        System.out.print("s = ", s);
        System.out.print("r = ", r);
    }
} catch (SQLException e) {
    System.out.println(ex.getMessage());
    ex.printStackTrace();
}
```

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.

```
CREATE TABLE Sailors
  ( sid INTEGER,
    sname CHAR(60),
    rating INTEGER,
    age REAL,
    PRIMARY KEY (sid),
    CHECK (rating >= 1)
    AND rating <= 10)
```

```
CREATE TABLE Reserves
  ( sname CHAR(60),
    bid INTEGER,
    day DATE,
    PRIMARY KEY (bid, day),
    CONSTRAINT Interlake CHECK ('Interlake' =
    (SELECT R.name FROM Boats B
    WHERE R.bid = bid)));
```

Constraints Over Multiple Relations

```
CREATE TABLE Sailors
  ( sid INTEGER,
    sname CHAR(60),
    rating INTEGER,
    age REAL,
    PRIMARY KEY (sid),
    CHECK (SELECT COUNT(S.sid) FROM Sailors S
    + (SELECT COUNT(B.bid) FROM Boats B) < 100)
    AND sid is empty, the number of Boats
    tuples can be anything)
```

```
CREATE TABLE Reserves
  ( sname CHAR(60),
    bid INTEGER,
    day DATE,
    PRIMARY KEY (bid, day),
    CONSTRAINT Interlake CHECK
    (SELECT R.name FROM Boats B
    WHERE R.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 trigger 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 YoungSailor(sid, name, age, rating)
    SELECT sname, 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.