SQL: Queries, Programming, Triggers

Chapter 5, Cow Book

or

See http://sqlzoo.net/
SQL Language

- DDL: Data definition language
- DML: Data manipulation language
- Embedded and Dynamic SQL
- Triggers
- Security
- Transaction Management
- Remote Database access
Basic SQL Query

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

- Semantics/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.

- Not an efficient evaluation plan! (Optimizer picks efficient plans)
### Example of Conceptual Evaluation

SELECT S.name, A.hours FROM Senators S, Attendance A
WHERE S.ssn = A.ssn and A.date = '24-Sept-2010'

<table>
<thead>
<tr>
<th>ssn</th>
<th>name</th>
<th>email</th>
<th>age</th>
<th>income</th>
<th>ssn</th>
<th>date</th>
<th>hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>11-111</td>
<td>Bob</td>
<td><a href="mailto:bob@ca.gov">bob@ca.gov</a></td>
<td>51</td>
<td>100.1</td>
<td>11-111</td>
<td>12-Aug-2010</td>
<td>1.1</td>
</tr>
<tr>
<td>22-222</td>
<td>Jane</td>
<td><a href="mailto:jane@mi.gov">jane@mi.gov</a></td>
<td>54</td>
<td>130.1</td>
<td>11-111</td>
<td>12-Aug-2010</td>
<td>1.1</td>
</tr>
<tr>
<td>33-333</td>
<td>Jane</td>
<td><a href="mailto:jane@wi.gov">jane@wi.gov</a></td>
<td>51</td>
<td>99.8</td>
<td>33-333</td>
<td>24-Sept-2010</td>
<td>4.1</td>
</tr>
</tbody>
</table>
Find senators who attended the ‘24-Sept-2010’ session

SELECT S.name
FROM Senators S, Attendance A
WHERE S.ssn = A.ssn and date = ‘24-Sept-2010’

- Add DISTINCT to this query. Effect?
- Replace S.name by S.ssn.
  Effect of adding DISTINCT to this query

RA: $\pi_{\text{name}} (\sigma_{\text{date} = '24-Sept-2010'} (\text{Senators} \bowtie \text{Attendance}))$

- Equivalent SQL?
- What is the schema of Senators $\bowtie$ Attendance?
A Note on Range Variables

• Needed only if the same relation appears twice in the FROM clause.

SELECT S.name, A.hours
FROM Senators S, Attendance A
WHERE S.ssn = A.ssn and date = '24-Sept-2010'

OR

SELECT Senators.name, Attendance.hours
FROM Senators , Attendance
WHERE Senators.ssn = Attendance.ssn and Attendance.date = '24-Sept-2010'

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

SELECT S.name, S.age, age1=S.age+2, S.income/S.age AS iar
FROM Senators S
WHERE S.sname LIKE 'Ja_%Doe'
ORDER BY S.name

- Illustrates use of arithmetic expressions and string pattern matching
- **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.
- **Collation**: sort order for character sets
Find senators who attended either the ‘24-Sept-2010’ or ‘25-Sept-2010’ session

- **UNION**: Compute the union of two *union-compatible* sets of tuples
  - Same number/types of fields.
- **Also available**: INTERSECT and EXCEPT (What do we get if we replace **UNION** by **EXCEPT**?)
- **SQL oddities**: duplicates with union, except, intersect
  - Default: eliminate duplicates!
  - Use ALL to keep duplicates

```sql
SELECT S.ssn
FROM Senators S, Attendance A
WHERE S.ssn = A.ssn
and (A.date = '24-Sept-2010' or A.date = '25-Sept-2010')

SELECT S.ssn
FROM Senators S, Attendance A
WHERE S.ssn = A.ssn
and A.date = '24-Sept-2010'
UNION
SELECT S.ssn
FROM Senators S, Attendance A
WHERE S.ssn = A.ssn
and A.date = '25-Sept-2010'
```
Find senators who attended both the ‘24-Sept-2010’ and ‘25-Sept-2010’ session

- **INTERSECT**: Compute the intersection of any two union-compatible sets of tuples.

- In the SQL/92 standard, but some systems don’t support it.

```sql
SELECT S.ssn
FROM Senators S, Attendance A1,
     Attendance A2
WHERE S.ssn = A1.ssn and S.ssn = A2.ssn
and     A1.date = '24-Sept-2010'
and     A2.date = '25-Sept-2010'

SELECT S.ssn
FROM Senators S, Attendance A
WHERE S.ssn = A.ssn
and     A.date = '24-Sept-2010'
INTERSECT
SELECT S.ssn
FROM Senators S, Attendance A
WHERE S.ssn = A.ssn
and     A.date = '25-Sept-2010'
```

**Key field!**
What happens if S.name is used
Nested Queries

Find names of sailors who’ve reserved boat #103

SELECT  S.sname
FROM    Sailors S
WHERE   S.sid IN (SELECT  R.sid
                   FROM    Reserves R
                   WHERE   R.bid=103)

Can you rewrite this to not use a nested query?

• Powerful feature of SQL
  – 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
• Conceptual Evaluation: nested loops  For each Sailors tuple, check the qualification by computing the subquery.
Nested Queries with Correlation

---Find names of sailors who’ve reserved boat #103---
Find names of sailors with exactly one reservation for boat #103

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

Why R.bid?

- `EXISTS` tests if the set is not empty
- `UNIQUE` returns true if the row appears only once
- 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:  \textit{op} ANY, \textit{op} ALL, \textit{op} is \textit{<, ≤, >, ≥, =, ≠}

• Find sailors whose rating is greater than that of some sailor called Horatio:

\[
\begin{align*}
\text{SELECT } & * \\
\text{FROM } & \text{Sailors S} \\
\text{WHERE } & \text{S.rating} > \text{ANY} \left( \text{SELECT } \text{S2.rating} \right) \\
& \text{FROM } \text{Sailors S2} \\
& \text{WHERE } \text{S2.sname} = \text{‘Horatio’}
\end{align*}
\]
Rewriting Except Queries Using NOT IN

Find sailors (sid) who’ve reserved some boat for ‘24-Sept-2010’ but have no reservations for ‘09-Oct-2010’

SELECT S.sid FROM Sailors S, Reserves R WHERE S.sid=R.sid AND R.day='24-Sept-2010' EXCEPT (SELECT S2.sid FROM Sailors S2, Reserves R2 WHERE S2.sid=R2.sid AND R2.day='09-Oct-2010')

• Similarly, INTERSECT queries re-written using IN.
Division in SQL

Find sailors who’ve reserved all boats.

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.sid=S.sid))

(2) SELECT S.sname 
    FROM Sailors S 
    WHERE NOT EXISTS (SELECT B.bid 
        FROM Boats B 
    WHERE NOT EXISTS (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

Sailors (sid, sname, rating, age) 
Reserves (sid, bid, day) 
Boats (bid, bname, color)
Aggregate Operators

SELECT COUNT (*) FROM Sailors S

SELECT COUNT (DISTINCT S.name) FROM Sailors S

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

SELECT S.sname FROM Sailors S WHERE S.rating= (SELECT MAX(S2.rating) FROM Sailors S2)

COUNT (*)
COUNT ( [DISTINCT] A)
SUM ( [DISTINCT] A)
AVG ( [DISTINCT] A)
MAX (A) Can use Distinct
MIN (A) Can use Distinct

single column

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

- The first query is illegal! (wait for GROUP BY.)
- Q3 is allowed in the SQL/92 standard, but not supported in some systems

<table>
<thead>
<tr>
<th>Query</th>
<th>SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>SELECT S.sname, MAX (S.age) FROM Sailors S</td>
</tr>
<tr>
<td>Q2</td>
<td>SELECT S.sname, S.age FROM Sailors S WHERE S.age = (SELECT MAX (S2.age) FROM Sailors S2)</td>
</tr>
<tr>
<td>Q3</td>
<td>SELECT S.sname, S.age FROM Sailors S WHERE (SELECT MAX (S2.age) FROM Sailors S2) = S.age</td>
</tr>
</tbody>
</table>

How many tuples in the result?
GROUP BY and HAVING

- Apply aggregate to each of several *groups* of tuples
- Find the age of the youngest sailor *for each rating level*
  - Don’t know: # rating levels, and rating values
  - Suppose we did know that rating values go from 1 to 10
    - we can write 10 queries that look like this (!):

For $i = 1, 2, \ldots, 10$:

SELECT MIN (S.age) FROM Sailors S
WHERE S.rating = $i$

\[
\text{SELECT MIN (S.age), S.rating FROM Sailors S GROUP BY S.rating}
\]
Queries With GROUP BY and HAVING

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

- The **target-list** contains
  - Attribute names: must be a subset of **grouping-list**.
  - Terms with aggregate operations (e.g., \( \text{MIN} (S.age) \)).
- The **group-qualification**
  - Must have a single value per group

How many tuples in the result?
Conceptual Evaluation

- Cross-product -> discard tuples -> apply projection
  -> partition into groups using the *grouping-list* attribute values
  -> eliminate groups that don’t satisfy the *group-qualification*

- Expressions in *group-qualification* 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 $\geq 18$, for each rating with at least 2 such sailors

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

- 2nd column of result is unnamed. (Use AS to name it)

<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>71</td>
<td>zorba</td>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>64</td>
<td>horatio</td>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>29</td>
<td>brutus</td>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

Answer relation

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
</tbody>
</table>

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

\[
\text{SELECT B.bid, COUNT (*) AS scount
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid AND B.color='red'
GROUP BY B.bid}
\]

\[
\text{SELECT B.bid, COUNT (*) AS scount
FROM Sailors S, Boats B, Reserves R
GROUP BY B.bid}
\]

Would this work? note: one color per bid
Find the 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  1 < (SELECT  COUNT (*) FROM  Sailors S2
              WHERE  S.rating=S2.rating)

• Subquery in the HAVING clause
• Compare this with the query where we considered only ratings with 2 sailors over 18!
Find ratings for which the average age is the minimum of the average age over all ratings

- Aggregate operations cannot be nested! WRONG:

```sql
SELECT S.rating
FROM Sailors S
WHERE AVG(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

• Represent
  – unknown (e.g., rating not assigned) or
  – inapplicable (e.g., no spouse’s name)

• Complications with nulls:
  – Operators to check if value is/is not null.
  – Is rating > 8 true or false when rating is null?
    • Answer: Evaluate to unknown
  – What about AND, OR and NOT connectives?
    • Need 3-valued logic (true, false and unknown)
      – Not unknown = unknown
  – WHERE clause eliminates rows that don’t evaluate to true
  – New operators (in particular, outer joins) possible/needed.
## Outer Join

### 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>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>10/10/99</td>
</tr>
</tbody>
</table>

Select S.sid, R.bid  
From Sailors S NATURAL LEFT OUTER JOIN Reserves R

### Result

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
</tr>
<tr>
<td>58</td>
<td>null</td>
</tr>
</tbody>
</table>

Similarly:
- RIGHT OUTER JOIN
- FULL OUTER JOIN
Embedded SQL

• Call SQL commands from a host language (e.g., C) 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.
Cursors

• 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.
  – Special clause, called ORDER BY, in cursor queries to control the order in which tuples are returned.
  – Fields in ORDER BY must also appear in SELECT clause.
• Can also modify/delete tuple pointed to by a cursor
Cursor that gets names of sailors who’ve reserved a red boat, in alphabetical order

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

• Can we replace $S.sname$ by $S.sid$ in the ORDER BY clause!
  – Every column in the ORDER BY clause must appear in the SELECT clause
Integrity Constraints

• An IC describes conditions that every *legal instance* of a relation must satisfy.
  – Inserts/deletes/updates that violate IC’s are disallowed.

• *Types of IC’s*: Domain constraints, primary key constraints, foreign key constraints, general constraints.

• Can create new domains, Domain Constraints:
  – CREATE DOMAIN LegalRatings INTEGER DEFAULT 0
    CHECK (VALUE >= 1 and VALUE <= 10)
  – Create Table Sailor (..., rating LegalRatings, ...)
  – Underlying domain is still Integers for comparison

• Can create new types: CREATE TYPE AllRatings as INTEGER
  – Underlying domain is now a new type. Can’t compare with INTEGER without a cast. None of the aggregates on INTEGER work on AllRatings
Table Constraints

- More general ICs than key constraints
- 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 )
)

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 & 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

• Three parts:
  – Event (activates the trigger)
  – Condition (tests whether the triggers should run)
  – Action (what happens if the trigger runs)
    • Before and After Triggers

• Trigger Execution
  – Row-level Triggers: Once per row
  – Statement-level Triggers: Once per SQL statement
Triggers: Example

CREATE TRIGGER init_count BEFORE INSERT ON Students /* Event */
DECLARE
  count INTEGER /* Action */
BEGIN
  count := 0
END

CREATE TRIGGER incr_count AFTER INSERT ON Student /* Event */
WHEN (new.age < 18) /* Condition */
FOR EACH ROW
BEGIN /* Action */
  count := count + 1;
END
Triggers

• First trigger executed *before* the activating statement, second executes *after* the activating statement.

• Options:
  – “BEFORE”
  – “AFTER”
  – “INSTEAD OF” (only valid on views)

• In combination with:
  – “FOR EACH ROW” - execute once per modified record
  – (default) - execute once per activating statement.
    Can also specify using “FOR EACH STATEMENT”

• In combination with:
  – “INSERT”
  – “DELETE”
  – “UPDATE”
Triggers

• Referring to values
  – Old
  – New
  – Set of changed record
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
Triggers v/s Constraints

• Often used to maintain consistency
  – Can you use a foreign key?
  – Foreign keys are not defined operationally
• Constraints are easier to understand than triggers
• Triggers are more powerful.
  – Often used to fill out fields in a form
  – Check complex actions (such as credit limit in a shopping application)
  – Check preferred customer status
  – Generate logs for auditing and security checks.
  – Internally can be used by the DBMS for replication management.