SQL: Queries, Constraints, Triggers

Chapter 5

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?

```
sid  sname  rating  age
22   dustin 7  45.0
31   lubber 8  55.5
58   rusty 10 35.0
```

```
sid  bid  day
22   101 10/10/96
58   103 11/12/96
```

Basic SQL Query

- `relation-list`: List of relation names (possibly with a range-variable after each name).
- `target-list`: 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`: Optional keyword indicating that the answer should not contain duplicates.
  - Default = duplicates are not eliminated

Example of Conceptual Evaluation

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

```
(sid) sname rating age (sid) bid day
22 dustin 7  45.0 22 101 10/10/96
22 dustin 7  45.0 58 103 11/12/96
31 lubber 8  55.5 22 101 10/10/96
31 lubber 8  55.5 58 103 11/12/96
58 rusty 10 35.0 22 101 10/10/96
58 rusty 10 35.0 58 103 11/12/96
```

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 bid=103
```

OR

```
SELECT sname
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

\[
\text{SELECT } S\text{.sid} \\
\text{FROM } S\text{ailors } S, \text{Reserves } R \\
\text{WHERE } S\text{.sid}=R\text{.sid}
\]

- Would adding DISTINCT to this query make a difference, i.e., could a sailor returned by the original version disappear or could a new sailor appear?
- What is the effect of replacing \( S\text{.sid} \) by \( S\text{.sname} \) in the SELECT clause? Would adding DISTINCT to this variant of the query make a difference?

Expressions and Strings

\[
\text{SELECT } S\text{.age}, age1=S\text{.age}-5, 2*S\text{.age} AS age2 \\
\text{FROM } S\text{ailors } S \\
\text{WHERE } S\text{.sname LIKE 'B_%B'}
\]

- Illustrates use of arithmetic expressions and string pattern matching
  - Find triples (age of sailor and two fields defined by expressions) for sailors whose names begin and end with B and contain at least three characters.
  - \( \text{AS} \) and \( = \) are two ways to name fields in the result.
  - \( \text{LIKE} \) is used for string matching
    - \( _ \) stands for any one character
    - \( % \) stands for 0 or more arbitrary characters.

Find sid’s of sailors who’ve reserved a red or a green boat

\[
\text{SELECT } S\text{.sid} \\
\text{FROM } S\text{ailors } S, B\text{oats } B, \text{Reserves } R \\
\text{WHERE } S\text{.sid}=R\text{.sid} \text{ AND } R\text{.bid}=B\text{.bid} \\
\text{AND (B.color='red' OR B.color='green')}\]

- UNION: Computes the union of any two union-compatible sets (which can themselves be the result of SQL queries).
- If we replace \( \text{OR} \) by \( \text{AND} \) in the first version, what do we get?
- Also available: EXCEPT (What do we get if we replace UNION by EXCEPT?)

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

\[
\text{SELECT } S\text{.sid} \\
\text{FROM } S\text{ailors } S, B\text{oats } B, \text{Reserves } R \\
\text{WHERE } S\text{.sid}=R\text{.sid} \text{ AND } R\text{.bid}=B\text{.bid} \\
\text{AND (B.color='red' AND B.color='green')}\]

- INTERSECT: Computes the intersection of any two union-compatible sets of tuples.
- Included in the SQL/92 standard, but some systems do not 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:**

\[
\text{SELECT } S\text{.sname} \\
\text{FROM } S\text{ailors } S \\
\text{WHERE } S\text{.sid }\text{IN (SELECT } R\text{.sid} \\
\text{FROM } \text{Reserves } R \\
\text{WHERE } R\text{.bid}=103)\]

- Very powerful feature of SQL: WHERE clause can itself contain an SQL query
  - And so can FROM and HAVING clauses.
  - To find sailors who have 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:**

\[
\text{SELECT } S\text{.sname} \\
\text{FROM } S\text{ailors } S \\
\text{WHERE EXISTS (SELECT } * \\
\text{FROM } \text{Reserves } R \\
\text{WHERE } R\text{.bid}=103 \text{ AND } S\text{.sid}=R\text{.sid})
\]

- EXISTS tests if the set is empty.
  - If \( \text{UNIQUE} \) is used, and \( * \) is replaced by \( R\text{.bid} \), finds sailors with at most one reservation for boat #103.
  - \( \text{UNIQUE} \) returns true if there are no duplicates in the result set.
  - Why do we have to replace \( * \) by \( R\text{.bid} \) for that query version?
  - Illustrates why, in general, subquery must be re-computed for each Sailors tuple.
More on Set-Comparison Operators

- Seen so far: IN, EXISTS, UNIQUE.
- Can also use NOT IN, NOT EXISTS, NOT UNIQUE.
- Also available: op ANY, op ALL, where op is \(<\), \(\ge\), \(\le\), or \(\ne\).
  - Note: IN same as \(\approx\) ANY, NOT IN same as \(\ne\) ALL
- 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.sname='Horatio')
  ```

Rewriting INTERSECT Queries Using IN

- 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.sname in SELECT clause. (What about INTERSECT query?)

```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')
```

Division in SQL

- Find sailors who've reserved all boats.
  - The hard way, without EXCEPT:
  
  ```sql
  (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)
  ```

  ```sql
  (2) SELECT S.sname
      FROM Sailors S
      WHERE NOT EXISTS (SELECT R.bid
      FROM Reserves R
      WHERE R.bid=B.bid
      AND R.sid=S.sid)
  ```

  ```sql
  (3) SELECT S.sname
      FROM Sailors S
      WHERE (SELECT MAX(S2.age)
      FROM Sailors S2)
      = S.age
  ```

Aggregate Operators

- Significant extension of relational algebra.

  ```sql
  SELECT COUNT (*)
  FROM Sailors S
  SELECT AVG (S.age)
  FROM Sailors S
  WHERE S.rating=10
  ```

  ```sql
  SELECT COUNT (DISTINCT S.rating)
  FROM Sailors S
  WHERE S.sname='Bob'
  ```

  ```sql
  SELECT S.sname, MAX (S.age)
  FROM Sailors S
  ```

  ```sql
  SELECT S.sname, Sage
  FROM Sailors S
  WHERE Sage = (SELECT MAX (S2.age)
  FROM Sailors S2)
  ```

Find name and age of the oldest sailor(s)

- First query is illegal. (Discussed in more depth later for GROUP BY.)
- Second query has implicit type cast (Which?)
- Third query is equivalent to second query
  - Allowed in the SQL/92 standard
  - But not supported in some systems
Motivation for Grouping

- So far: Have applied aggregate operators to all (qualifying) tuples.
- May want to apply them to each of several groups of tuples.
- E.g., 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:

\[
\text{For } i = 1, 2, \ldots, 10: \quad \text{SELECT } \text{MIN} (S.\text{age}) \quad \text{FROM} \quad \text{Sailors} \quad S \quad \text{WHERE} \quad S.\text{rating} = i
\]

Queries With GROUP BY and HAVING

\[
\text{SELECT} \quad [\text{DISTINCT}] \quad \text{target-list} \\
\text{FROM} \quad \text{relation-list} \\
\text{WHERE} \quad \text{qualification} \\
\text{GROUP BY} \quad \text{grouping-qualification} \\
\text{HAVING} \quad \text{group-qualification}
\]

- target-list contains (i) attribute names (ii) terms with aggregate operations (e.g., MIN (S.age)).
  - 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.)

Conceptual Evaluation

1. Compute cross-product of relation-list.
2. Discard tuples that fail qualification.
3. Delete ‘unnecessary’ fields.
4. Partition remaining tuples into groups by the value of attributes in grouping-list.
5. Apply group-qualification to eliminate some groups.
   - Expressions in group-qualification must have a single value per group.
   - 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 youngest sailor with age $\geq 18$ for each rating with at least 2 such sailors

\[
\text{SELECT} \quad S.\text{rating}, \quad \text{MIN} (S.\text{age}) \quad \text{AS} \quad \text{minage} \\
\text{FROM} \quad \text{Sailors} \quad S \\
\text{WHERE} \quad S.\text{age} \geq 18 \\
\text{GROUP BY} \quad S.\text{rating} \\
\text{HAVING} \quad \text{COUNT} (*) \geq 2
\]

Find age of youngest sailor with age $\geq 18$ for each rating with at least 2 such sailors

- Sailors instance:
  - Note: irrelevant attributes omitted on this and following slides.

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

\[
\text{HAVING} \quad \text{COUNT} (*) \geq 2 \text{ AND EVERY } (S.\text{age} \leq 60)
\]

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>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>35</td>
<td>rusty</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>zorba</td>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>74</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>85</td>
<td>art</td>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>95</td>
<td>bob</td>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>96</td>
<td>frodo</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)

- Shows HAVING clause can also contain a subquery.
- Compare this with the query where we considered only ratings with 2 sailors of age ≥ 18!
  - What if HAVING clause is replaced by HAVING COUNT(*) >= 2?

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

- Aggregate operations cannot be nested! WRONG:
  SELECT S.rating
  FROM Sailors S
  WHERE AVG(S.age) = (SELECT MIN (AVG (S2.age))
                      FROM Sailors S2
                      GROUP BY S2.rating)
  GROUP BY S.rating
- Correct solution (in SQL/92):
  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.
- Presence of NULL complicates many issues:
  - Special operators needed to check if value is (not) NULL.
  - Is rating>8 true or false for rating=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 do not evaluate to true.)
    - New operators (in particular, outer joins) possible and needed.
- Arithmetic operators (+, -, *, /) return NULL if any input is NULL
- Aggregate operators affected differently
  - COUNT(*) not affected
  - All others, including COUNT(column), discard NULL values before computing the aggregate
- What if all values in the column are NULL?
### Working with NULL

- **NULL op constant** evaluates to **unknown**
  - `op` is one of `<`, `>`, `<=`, `>=`, `<>`
  - What about `NULL = NULL`? **unknown**
- **NOT unknown** evaluates to **unknown**
- **true OR unknown** evaluates to **true**
- **false AND unknown** evaluates to **false**
- Definition of a duplicate: corresponding columns are either equal or both have value NULL
  - Implicitly evaluates `(NULL = NULL)` as true
- **Arithmetic operators** (`+`, `-`, `*`, `/`) return NULL if any input is NULL
- **Aggregate operators** affected differently
  - `COUNT(*)` not affected
  - All others, including `COUNT(column)`, discard NULL values before computing the aggregate
    - Compare result of `SUM(column)` to using `+` on the same set of values
    - If all values in the column are NULL, result is NULL.

### 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.

### Constraints Over Multiple Relations

- **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 Bname FROM Boats B WHERE B.bid=bid)))`
- **CREATE ASSERTION** `smallClub`
  - `(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
- **Three parts**:
  - **Event**
    - Change to the database that activates the trigger
  - **Condition**
    - Query or test that is run when the trigger is activated
  - **Action**
    - Procedure that is executed when the trigger is activated and its condition is true

### Trigger Options

- **Event** can be insert, delete, or update on DB table
- **Condition** can be a true/false statement
  - All employee salaries are less than $100K
- **Condition** can be a query
  - Interpreted as true if and only if answer set is not empty
- **Action** can perform DB queries and updates that depend on
  - Answers to query in condition part
  - Old and new values of tuples modified by the statement that activated the trigger
  - Action can also contain data-definition commands, e.g., create new tables
Trigger Timing

- Should trigger action be executed before or after the statement that activated the trigger?
  - Consider triggers on insertions
  - Trigger that initializes a variable for counting how many new tuples are inserted: execute trigger before insertion
  - Trigger that updates this count variable for each inserted tuple: execute after each tuple is inserted (might need to examine values of tuple to determine action)
- Challenge: Trigger action can fire other triggers
  - Very difficult to reason about what exactly will happen
    - Trigger can fire “itself” again
    - Unintended effects possible

Trigger Example (Oracle Syntax)

```sql
CREATE TRIGGER init_count BEFORE INSERT ON Students /* Event */
DECLARE
  count INTEGER;
BEGIN /* Action */
  count := 0;
END
CREATE TRIGGER incr_count AFTER INSERT ON Students /* Event */
WHEN (new.age < 18) /* Condition, where new refers to inserted tuple */
FOR EACH ROW
BEGIN /* Action */
  count = count + 1;
END
```

Trigger Example (SQL:1999)

```sql
CREATE TRIGGER set_count AFTER INSERT ON Students
REFERENCING NEW TABLE AS InsertedTuples /* Name for the set of newly inserted tuples */
FOR EACH STATEMENT /* Statement-level trigger */
INSERT
  INTO StatisticsTable(ModifiedTable, ModificationType, Count)
  SELECT 'Students', 'Insert', COUNT(*)
  FROM InsertedTuples
WHERE I.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 than relational algebra.
- Even queries that can be expressed in relational algebra can often be expressed more naturally in SQL.
- Many alternative ways to write a query
  - Optimizer should find 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
- SQL allows specification of rich integrity constraints
- Triggers respond to changes in the database