SQL: The Query Language

Example Instances

<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>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<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?

Basic SQL Query

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

- Set-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

<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>7</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>7</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>8</td>
<td>55.5</td>
<td>31</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</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>10</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 R.bid=103
  ```
  ```sql
  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.sname in the SELECT clause? Would adding DISTINCT to this variant of the query make a difference?

Find sid's of sailors who've reserved a red or 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' OR B.color='green')

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 sid's of sailors who've reserved 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 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.

Expressions and Strings

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

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 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)

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

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 IN, EXISTS and UNIQUE. Can also use NOT IN, NOT EXISTS and NOT UNIQUE.
- Also available: op SOME, op ALL, op IN, >=, <=, ≠.
- Find sailors whose rating is greater than that of some sailor called Horatio:
  \[
  \text{SELECT } * \\
  \text{FROM } \text{Sailors S} \\
  \text{WHERE } \text{S.rating} > \text{SOME (SELECT S2.rating}} \\
  \text{FROM } \text{Sailors S2} \\
  \text{WHERE } \text{S2.sname='Horatio')}
  \]

Aggregate Operators

- Significant extension of relational algebra.
- Also available: op SOME, op ALL, op IN.
- Find name and age of the oldest sailor(s):
  \[
  \text{SELECT } \text{S.sname}, \text{MIN (S.age)} \\
  \text{FROM } \text{Sailors S} \\
  \text{WHERE } (\text{SELECT MAX (S2.age))} \\
  \text{FROM } \text{Sailors S2} \\
  \text{= S.age}
  \]

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.

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

Queries With GROUP BY and HAVING

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

Conceptual Evaluation

- The cross-product of \text{relation-list} is computed, tuples that fail \text{qualification} are discarded, 'unnecessary' fields are deleted, as before.
- The remaining tuples are partitioned into groups by the value of attributes in \text{grouping-list}.
- The \text{group-qualification} is then applied to eliminate some groups.
- 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
```

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

```sql
```

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

Null Values

- Field values in a tuple are sometimes unknown (e.g., a rating has not been assigned) or nonexistent (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.)
  - New operators (in particular, outer joins) possible/needed.

Summary

- 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 (aggregates, arithmetic, sorting, grouping, string matching…)
- Even queries that can be expressed in RA can often be expressed more naturally in SQL.

Summary (cont’d)

- Nulls (unknown or nonexistent) force a 3-valued logic and odd behavior
- 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.
- SQL3 (SQL:1999) adds nested relational and object-oriented features to SQL (later in course)