SQL - 2

Week 7
BETWEEN and AND operators

- The **BETWEEN** and **AND** operator selects a range of data between two values.
- These values can be numbers, text, or dates.
BETWEEN and AND Example

Find the names of sailors whose age is between 25 and 35

```
SELECT S.sname
FROM Sailors S
WHERE S.age BETWEEN 25 AND 35;
```
ANY and ALL operators

Find sailors whose rating is better than some sailor named Horatio

SELECT S.sid
FROM Sailors S
WHERE S.rating > ANY (SELECT S2.rating
    FROM Sailors S2
    WHERE S2.sname= ‘Horatio’
);

(Can you find the probable bug in this SQL query??)

Hint: what if there are several sailors named Horatio?
Using ALL operator

Find sailors whose rating is better than every sailor named Horatio

SELECT S.sid
FROM Sailors S
WHERE S.rating > ALL(SELECT S2.rating
                        FROM Sailors S2
                        WHERE S2.sname=‘Horatio’);
Post Processing

• Processing on the result of an SQL query:
  – Sorting: can sort the tuples in the output by any column (even the ones not appearing in the SELECT clause)
  – Duplicate removal
  – Example:

```sql
SELECT DISTINCT S.sname
FROM Sailors S, Reserves R
WHERE S.sid=R.sid AND R.bid=103
ORDER BY S.sid ASC, S.sname DESC;
```

• Aggregation operators
Aggregate operators

• What is aggregation?
  – Computing arithmetic expressions, such as Minimum or Maximum

• The aggregate operators supported by SQL are: COUNT, SUM, AVG, MIN, MAX
Aggregate Operators

• \textbf{COUNT}(A): The number of values in the column A
• \textbf{SUM}(A): The sum of all values in column A
• \textbf{AVG}(A): The average of all values in column A
• \textbf{MAX}(A): The maximum value in column A
• \textbf{MIN}(A): The minimum value in column A

(We can use DISTINCT with COUNT, SUM and AVG to compute only over non-duplicated columns)
Using the COUNT operator

Count the number of sailors

SELECT COUNT (*)
FROM Sailors S;
Example of SUM operator

Find the sum of ages of all sailors with a rating of 10

SELECT  SUM (S.age)
FROM   Sailors S
WHERE  S.rating=10;
Example of AVG operator

Find the average age of all sailors with rating 10

SELECT AVG (S.age)  
FROM Sailors S  
WHERE S.rating=10;
Example of MAX operator

Find the name and age of the oldest sailor

SELECT S.sname, MAX(S.age)
FROM Sailors S;

But this is illegal in SQL!!
Correct SQL Query for MAX

```
SELECT S.sname, S.age
FROM Sailors S
WHERE S.age = ( SELECT MAX(S2.age) 
               FROM Sailors S2 )
```
Alternatively…

SELECT S.sname, S.age
FROM Sailors S
WHERE ROWNUM <= 1
ORDER BY S.age DESC;
Another Aggregate Query

Count the number of different sailor names

SELECT COUNT (DISTINCT S.sname) 
FROM Sailors S
Banking Examples

branch (branch-id, branch-city, assets)

customer (customer-id, customer-name, customer-city)

account (account-number, branch-id, balance)

loan (loan-number, branch-id, amount)

depositor (customer-id, account-number)

borrower (customer-id, loan-number)
IN...Example 1

“Find the account numbers opened at branches of the bank in Fairfax”
IN...Example 2

“Find the account numbers opened at branches 101 and 102 of the bank”
The \textit{EXISTS} predicate is TRUE if and only if the Subquery returns a non-empty set.

The \textit{NOT EXISTS} predicate is TRUE if and only if the Subquery returns an empty set.

The \textit{NOT EXISTS} can be used to implement the SET DIFFERENCE operator from relational algebra.
EXISTS...Example 1

“Select all the account balances where the account has been opened in a branch in Fairfax”

What about “… has not been opened..”?
EXISTS...Example 2

“Find customers who opened accounts in all branches in Fairfax”
EXISTS…Example 2

“Find customers who opened accounts in all branches in Fairfax”

SELECT C.customer-id
FROM customer C
WHERE NOT EXISTS (SELECT B.branch-id
    FROM branch B
    WHERE B.branch-city=‘Fairfax’
    EXCEPT
    SELECT A.branch-id
    FROM depositor D, account A
    WHERE D.customer-id = C.customer-id AND
        D.account-number = A.account-number)
Quantified Comparison Predicate
(ANY/ALL) Example

“Select account numbers of the accounts with the minimum balance”
Aggregate Functions in SQL… revisited

SQL provides five built-in aggregate functions that operate on sets of column values in tables:

\[ \text{COUNT()}, \text{MAX()}, \text{MIN()}, \text{SUM()}, \text{AVG}(). \]

With the exception of \text{COUNT()}, these set functions must operate on sets that consist of simple values—that is, sets of numbers or sets of character strings, rather than sets of rows with multiple values.
Aggregate Functions in SQL

Example 1

“Select the total amount of balance of the account in branches located in Fairfax”
Aggregate Functions in SQL
Example 2

“Select the total number of opened accounts”
Value functions

• Values can be transformed before aggregated:
  Select sum(S.A/2) from S;

• An interesting decode function (Oracle specific):
  decode(value, if1, then1, if2, then2, ..., else):

  Select sum(decode(major, 'CS', 1, 0)) as Num_CS_Stu,
       sum(decode(major, 'CS', 0, 1)) as Num_NonCS_Stu
  From student ;

  if (major == 'CS')
    result = 1;
  else
    result = 0;
Value functions

• Example:

Transcript (sid:integer, Dept:string, Course_no:integer, Grade:{‘A’,’B’,’C’,’F’})
Write a query to compute a given student’s GPA
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 (!):

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

For $i = 1, 2, \ldots, 10$:
Queries With GROUP BY and HAVING

$$\begin{align*}
\text{SELECT} & \quad \text{[DISTINCT]} \ target-list \\
\text{FROM} & \quad relation-list \\
\text{WHERE} & \quad qualification \\
\text{GROUP BY} & \quad grouping-list \\
\text{HAVING} & \quad group-qualification
\end{align*}$$

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

\[
\text{SELECT S.rating, MIN (S.age)} \\
\text{FROM Sailors S} \\
\text{WHERE S.age} \geq 18 \\
\text{GROUP BY S.rating} \\
\text{HAVING COUNT (*)} > 1
\]

- 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.)

<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>

Answer relation

<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>

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
</tbody>
</table>
For each red boat, find the number of reservations for this boat

```
SELECT  B.bid,  COUNT (*) AS scount
FROM    Boats B, Reserves R
WHERE   R.bid=B.bid AND B.color=‘red’
GROUP BY  B.bid
```

- Grouping over a join of two relations.
- What do we get if we remove $B.color=‘red’$ from the WHERE clause and add a HAVING clause with this condition?

```
SELECT  B.bid,  COUNT (*) AS scount
FROM    Boats B, Reserves R
WHERE   R.bid=B.bid
GROUP BY  B.bid
HAVING  B.color=‘red’
```

Illegal!

*Can be rewritten using EVERY in HAVING:*

```
SELECT  B.bid,  COUNT (*) AS scount
FROM    Boats B, Reserves R
WHERE   R.bid=B.bid
GROUP BY  B.bid
HAVING EVERY(B.color=‘red’)
```
Find the age of the youngest sailor with age $\geq 18$, for each rating with at least 2 sailors (of any age)

```sql
SELECT S.rating, MIN (S.age)
FROM Sailors S
WHERE S.age $\geq$ 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!
Find those ratings for which the average age is the minimum over all ratings

- Aggregate operations cannot be nested!

**WRONG:**

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

**Correct solution in SQL/92 (but does not work in Oracle):**

```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)
```
Continue from previous

However, this should work on Oracle 8 (or later):

```
SELECT S.rating
FROM Sailors S
GROUP BY S.rating
HAVING AVG(S.age) = (SELECT MIN (AVG (S2.age))
    FROM Sailors S2
    Group by rating);
```

Can use nested aggregates with Group By
Null Values

• We use *null* when the column value is either *unknown* or *inapplicable*.

• A comparison with at least one null value always returns *unknown*.

• SQL also provides a special comparison operator *IS NULL* to test whether a column value is *null*.

• To incorporate nulls in the definition of duplicates we define that two rows are duplicates if corresponding rows are equal or both contain *null*.
Deal with the null value

- Special operators needed to check if value is/is not null.
  - “is null” always true or false (never unknown)
  - “is not null”
- Is rating > 8 true or false when rating is equal to null?
  - Actually, it’s unknown.
  - Three-valued logic
Three valued logic

<table>
<thead>
<tr>
<th>AND</th>
<th>False</th>
<th>True</th>
<th>Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
<td>Unknown</td>
</tr>
<tr>
<td>Unknown</td>
<td>False</td>
<td>Unknown</td>
<td>Unknown</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OR</th>
<th>False</th>
<th>True</th>
<th>Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
<td>Unknown</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Unknown</td>
<td>Unknown</td>
<td>True</td>
<td>Unknown</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NOT</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>Unknown</td>
<td>Unknown</td>
</tr>
</tbody>
</table>
Other issues with the null value

• WHERE and HAVING clause eliminate rows that don’t evaluate to true (i.e., rows evaluate to false or unknown).

• Aggregate functions ignore nulls (except count(*))

• DISTINCT treats all nulls as the same
Outer Joins

• Let R and S be two tables. The outer join preserves the rows of R and S that have no matching rows according to the join condition and outputs them with nulls at the non-applicable columns.

• There exist three different variants: left outer join, right outer join and full outer join.
## Outer joins

(left outer-join)

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th></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></td>
<td>22</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></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></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

=  

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</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>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>Null</td>
<td>Null</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>
In Oracle

```
Select *
From Sailor S, Reserve R
Where S.sid = R.sid(+);
```

**How about:**

```
Select S.sid, count(R.bid)
From Sailor S, Reserve R
Where S.sid = R.sid(+)
Group by S.sid;
```

**OR**

```
Select S.sid, count(*)
From Sailor S, Reserve R
Where S.sid = R.sid(+)
Group by S.sid;
```
More outer joins

• Left outer join
  + sign on the right in Oracle:
  Select * from R, S where R.id=S.id(+)

• Right outer join
  + sign on the left in Oracle:
  Select * from R, S where R.id(+)=S.id

• Full outer join
  – not implemented in Oracle 8
  – Added for Oracle 9 (or later)
    • Use full text instead of +’s: “full outer join”, “left outer join”, “right outer join”, “inner join”
Overall:
Conceptual order in query evaluation

• First the relational products of the tables in the FROM clause are evaluated.
• From this, rows not satisfying the WHERE clause are eliminated.
• The remaining rows are grouped in accordance with the GROUP BY clause.
• Groups not satisfying the HAVING clause are then eliminated.
• The expressions in the SELECT list are evaluated.
• If the keyword DISTINCT is present, duplicate rows are now eliminated.
• Evaluate UNION, INTERSECT and EXCEPT for Subqueries up to this point.
• Finally, the set of all selected rows is sorted if the ORDER BY is present.