SQL - 2

Week 7
### 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>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>58</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>Horatio</td>
<td>9</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>
</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>10/10/04</td>
</tr>
<tr>
<td>22</td>
<td>102</td>
<td>10/10/04</td>
</tr>
<tr>
<td>22</td>
<td>103</td>
<td>10/08/04</td>
</tr>
<tr>
<td>22</td>
<td>104</td>
<td>10/07/04</td>
</tr>
<tr>
<td>31</td>
<td>102</td>
<td>11/10/04</td>
</tr>
<tr>
<td>31</td>
<td>103</td>
<td>11/06/04</td>
</tr>
<tr>
<td>31</td>
<td>104</td>
<td>11/12/04</td>
</tr>
<tr>
<td>64</td>
<td>101</td>
<td>09/05/04</td>
</tr>
<tr>
<td>64</td>
<td>102</td>
<td>09/08/04</td>
</tr>
<tr>
<td>74</td>
<td>103</td>
<td>09/08/04</td>
</tr>
</tbody>
</table>

### Boats

<table>
<thead>
<tr>
<th>bid</th>
<th>bname</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Interlake</td>
<td>blue</td>
</tr>
<tr>
<td>102</td>
<td>Interlake</td>
<td>red</td>
</tr>
<tr>
<td>103</td>
<td>Clipper</td>
<td>green</td>
</tr>
<tr>
<td>104</td>
<td>Marine</td>
<td>red</td>
</tr>
</tbody>
</table>
Correlated Nested Queries (Revisit)

Find names of sailors who have reserved boat 103

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

(For finding sailors who have not reserved boat 103, we would use NOT EXISTS)
Correlated Nested Query - Division

*Find the names of sailors who have reserved ALL boats (DIVISION)*

```sql
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));
```
Correlated Nested Query 2

Alternatively,

*Find the names of sailors who have reserved ALL boats*

```sql
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 ));
```
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');
```
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');

What if there were no sailor called Horatio?

Note that IN is equivalent to = ANY
NOT IN is equivalent to <> ALL
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;
```
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 the the SELECT clause)
  – Duplicate removal
  – Example:

```
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
Last time we saw…
Example of MAX operator

*Find the name and age of the oldest sailor*

```sql
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;
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)
Example 1

“Find the account numbers opened at branches of the bank in Fairfax”

```
SELECT A.account-number
FROM account A
WHERE A.branch-id IN (SELECT B.branch-id
                        FROM branch B
                        WHERE B.branch-city= 'Fairfax')
```

IN
IN...Example 2

“Find the account numbers opened at branches 101 and 102 of the bank”

SELECT A.account-number
FROM A.account
WHERE A.branch-id IN (‘101’, ‘102’)

The `EXISTS` predicate is TRUE if and only if the Subquery returns a non-empty set.

The `NOT EXISTS` predicate is TRUE if and only if the Subquery returns an empty set.

The `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”

SELECT A.account-balance
FROM account A
WHERE EXISTS (SELECT *
FROM branch B
WHERE B.branch-city='Fairfax'
AND B.branch-id=A.branch-id)
EXISTS...Example 2

“Select all the account balances where the account has not been opened in a Fairfax branch”

SELECT A.account-balance
FROM account A
WHERE NOT EXISTS (SELECT *
    FROM branch B
    WHERE B.branch-city= ‘Fairfax’
    AND B.branch-id=A.branch-id)
EXISTS…Example 3

“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

Example 1

“Select account numbers of the accounts with the minimum balance”

SELECT A.account-number
FROM account A
WHERE A.balance <= ALL (SELECT A2.balance
FROM account A2)
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”

```
SELECT SUM(A.balance) AS total_amount
FROM account A, branch B
WHERE B.branch-city='Fairfax' AND
      B.branch-id= A.branch-id
```
 Aggregate Functions in SQL

Example 2

“Select the total number of opened accounts”

SELECT COUNT(A.account-number)
FROM account A

OR

SELECT COUNT(*)
FROM account
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;
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 (!):

\[
\text{SELECT MIN (S.age) FROM Sailors S WHERE S.rating = i}
\]

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

- 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.)

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

• The cross-product of \textit{relation-list} is computed, tuples that fail \textit{qualification} are discarded, \textquote{unnecessary} fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in \textit{grouping-list}.

• The \textit{group-qualification} is then applied to eliminate some groups. Expressions in \textit{group-qualification} must have a single value per group!  
  - In effect, an attribute in \textit{group-qualification} that is not an argument of an aggregate op also appears in \textit{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

```
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.)

<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
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 \texttt{B.color= 'red'} from the \texttt{WHERE} clause and add a \texttt{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!

```
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')
```

Can be rewritten using \texttt{EVERY} in \texttt{HAVING}:
Find the age of the youngest sailor with age $\geq 18$, for each rating with at least 2 sailors (of any age)

```
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:**

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):**

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 \texttt{null}.
  – “is null” always true or false (never unknown)
  – “is not null”

• Is \texttt{rating}>8 true or false when \texttt{rating} is equal to \texttt{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>
</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>

\[
\begin{align*}
\text{sid} & \quad \text{sname} & \quad \text{rating} & \quad \text{age} & \quad \text{bid} & \quad \text{day} \\
22 & \quad \text{dustin} & \quad 7 & \quad 45.0 & \quad 101 & \quad 10/10/96 \\
31 & \quad \text{lubber} & \quad 8 & \quad 55.5 & \quad \text{Null} & \quad \text{Null} \\
58 & \quad \text{rusty}  & \quad 10 & \quad 35.0 & \quad 103 & \quad 11/12/96 \\
\end{align*}
\]
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 \textit{FROM} clause are evaluated.
- From this, rows not satisfying the \textit{WHERE} clause are eliminated.
- The remaining rows are grouped in accordance with the \textit{GROUP BY} clause.
- Groups not satisfying the \textit{HAVING} clause are then eliminated.
- The expressions in the \textit{SELECT} list are evaluated.
- If the keyword \textit{DISTINCT} is present, duplicate rows are now eliminated.
- Evaluate \textit{UNION}, \textit{INTERSECT} and \textit{EXCEPT} for Subqueries up to this point.
- Finally, the set of all selected rows is sorted if the \textit{ORDER BY} is present.
Conclusion

• Nested queries are a very powerful feature in SQL; they help us write shorter and more efficient queries.
• Post processing on the result of queries is supported.
• Aggregation is the most complex “post processing”
  – “Group by” clause partition the results into groups
  – “Having” clause puts condition on groups (just like Where clause on tuples).