CS 450

SQL - 3

A Note on Subquery

SELECT S.sname

FROM Sailors S,

      (SELECT R.sid

       FROM Reserves R

       WHERE R.bid = 101) Temp

WHERE S.sid = Temp.sid;

Naming the
temporary relation

ANY/SOME, 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');

Alternative is to use SOME, which is equivalent to ANY operator.

What if there are several sailors named Horatio?

Definition of "Any" (or "Some") Clause

F <comp> **any** $r \Leftrightarrow \exists\, t \in r$ such that (F <comp> t), where <comp> can be:

$<, \leq, >, =, \neq$

(5 < **any**
0
5
6
) = true

(read: 5 < any tuple in the relation)

(5 < **any**
0
5
) = false

(5 = **any**
0
5
) = true

(5 ≠ **any**
0
5
) = true (since 0 ≠ 5)

(= **any**) ≡ **in**
However, (≠ **any**) ≢ **not in**

Substitute the "any" with "some", and you'll get the same result.

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');

Definition of All Clause

- F \<comp\> **all** $r \Leftrightarrow \forall t \in r$ (F \<comp\> *t)*

$$(5 < \textbf{all} \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}) = \text{false}$$

$$(5 < \textbf{all} \begin{array}{|c|} \hline 6 \\ \hline 10 \\ \hline \end{array}) = \text{true}$$

$$(5 = \textbf{all} \begin{array}{|c|} \hline 4 \\ \hline 5 \\ \hline \end{array}) = \text{false}$$

$$(5 \neq \textbf{all} \begin{array}{|c|} \hline 4 \\ \hline 6 \\ \hline \end{array}) = \text{true (since } 5 \neq 4 \text{ and } 5 \neq 6)$$

($\neq$ all) $\equiv$ not in
However, (= **all**) $\neq$ **in**

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:

  ```
  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

If using DISTINCT, the ORDER BY list must match the SELECT list

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

- **COUNT**(A): The number of values in the column A
- **SUM**(A): The sum of all values in column A
- **AVG**(A): The average of all values in column A
- **MAX**(A): The maximum value in column A
- **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

Null Values and Aggregates

- Total all salaries

    SELECT SUM (I.salary)
    FROM Instructor I

    – Above statement ignores null amounts

    – Result is *null* if there is no non-null amount

- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes

- What if collection has only null values?

    – count returns 0

    – all other aggregates return null

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 (!):

            For i = 1, 2, ... , 10:

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

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 (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 <u>*single value per group*</u>!

  - 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

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

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
71	zorba	10	16.0
64	horatio	7	35.0
29	brutus	1	33.0
58	rusty	10	35.0

- 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.)

rating	age
1	33.0
7	45.0
7	35.0
8	55.5
10	35.0

rating	
7	35.0

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 *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!

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)
FROM Sailors S
WHERE S.age >= 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)

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

Review on 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

AND	False	True	Unknown
False	False	False	False
True	False	True	Unknown
Unknown	False	Unknown	Unknown

OR	False	True	Unknown
False	False	True	Unknown
True	True	True	True
Unknown	Unknown	True	Unknown

	NOT
False	True
True	False
Unknown	Unknown

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

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

(left outer-join)

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

=

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

In Oracle (version 9 or later)

SELECT *

FROM Sailors S LEFT OUTER JOIN Reserves R

ON S.sid = R.sid;

How about: SELECT S.sid, count(R.bid)

FROM Sailors S LEFT OUTER JOIN Reserves R

ON S.sid = R.sid

GROUP BY S.sid;

Or: SELECT S.sid, count(*)

FROM Sailors S LEFT OUTER JOIN Reserves R

ON S.sid = R.sid

GROUP BY S.sid;

In Oracle (Older version)

SELECT *

FROM Sailors S, Reserves R

WHERE S.sid = R.sid(+);

How about: SELECT S.sid, count(R.bid)

FROM Sailors S, Reserves R

WHERE S.sid = R.sid(+)

GROUP BY S.sid;

Or: SELECT S.sid, count(*)

FROM Sailors S, Reserves R

WHERE S.sid = R.sid(+)

GROUP BY S.sid;

More outer joins

- Left outer join (older version)

  + sign on the right in Oracle:

  Select * from R, S where R.id=S.id(+)

- Right outer join (older version)

  + 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.