Operations … SQL Specifics

pm @ daiict

Issues with Null Values

- An attribute having NULL could mean either of following-
  - Value is unknown or not available right now
  - Value is not application for the tuple: a employee not having supervisor will have null in this attribute
- Consider following two SQL statements-

Null

SELECT e.salary*1.1 from employee as e; NULL

Select * from employee as e where e.salary > 50000; true

- Interpret e here as tuple variable that ranges over all tuples of employee relations. Try finding result of expressions in blue for tuples where salary is NULL?

How Where clause is used?

- Given a SQL statement

```
select * from r
    where <condition>
```

Here r can be any “relational expression”; i.e. r1 join r2 or so

- You can think of it as following

```
for tuple t ∈ r
    if (cond is true)
        add t to resultset
```

Issues with Null Values

1.1 * e.salary Null

- Arithmetic expressions (+,-,*,/) involving null values result null value for result
- When NULL values appears for attributes used in WHERE clause then boolean expression like this `t.a < 10` then interpretation of attribute reference is UNKOWN.
- When we compare a NULL value with another value including NULL, result is UNKNOWN.
- UNKOWN is treated as third truth (in addition to TRUE and FALSE) value in SQL where clause evaluation

e.salary > 50000 and dno = 4 ;

F UNKNOWN ~~FALE~~ AND TRUE → UNKNOWN

Truth values for UNKOWN

- NOT
  - NOT UNKOWN -> UNKWON
- AND
  - TRUE AND UNKOWN -> UNKOWN
  - FALSE AND UNKOWN -> FALSE
  - UNKWON AND UNKOWN -> UNKOWN
- OR ~~eno~~ salary > 5000 AND OR dno = 4
  - TRUE OR UNKOWN -> <u>TRUE</u>
  - FALSE OR UNKOWN -> UNKOWN
  - UNKWON OR UNKOWN -> UNKOWN

Null Values and Comparisons

- While evaluating WHERE clause tuples with UNKOWN or FALSE truth values are not included in result
- Following query will not include any tuple where either of value ~~in~~ is NULL irrespective value in other attribute

```
SELECT * FROM EMPLOYEE WHERE
bdate < DATE '2001-01-01' AND salary > 30000
```

- Following query will not include a tuple only when both are NULL, if one of attribute meets the condition then it will get included in result

```
SELECT * FROM EMPLOYEE WHERE
bdate < DATE ‘2001-01-01’ OR salary > 30000
```

Null Values and Comparisons – IS NULL

- Following will not give desired result. Why? -

```
SELECT * FROM employee
             WHERE superssn = NULL;
```

- This is so because Null = Null is also UNKOWN. For checking an attribute for having NULL value, SQL provides IS NULL (and IS NOT NULL)

- We write as following for such situations –

```
SELECT * FROM employee
    WHERE superssn IS NULL ;
```

IS NOT NULL

Sub-queries in SQL

Subquery in SQL

- A Query that is part of another query is *subquery*. A subquery may also have subquery, and so forth upto any level

Select ✓ (Select ✓)

- A subquery in SQL is written as a *query expression* enclosed in parentheses, and is in following form-
  " (SELECT . . . FROM …) "
  as a part of some existing query

- **Result of sub-query is again a relation**;

Subquery in FROM clause

- Here is an example

```
select ename, dno, salary from employee natural join
    (select eno from works_on natural join project
    where pno=1) as r;

select r1.*, m.salary as manager_salary from
    (select e.eno as enum, e.ename as emp_name,
                 e.salary as emp_salary,
                 d.mgr_eno as manager
          FROM employee e join department d
                on e.dno=d.dno) as r2
                 join employee m on (manager=eno)
    where emp_salary > m.salary;
```

Subquery in FROM clause

```
select eno, ename, salary from employee

natural join

(select eno from employee

except

select mgr_eno from department) as r;
```

Subquery in WHERE clause

```
select * from employee where eno not in
(select eno from works_on);
```

- Sub query in FROM clause is never a problem. However, when subquery in where clause, there needs to be few cautions to be noted!
- Recall that where clause is evaluated for every tuple of operand relation?
  - Does it mean; subquery here needs to be executed N times?
  - Note subquery here may also be called as inner query!

Execution of Subquery

- **SUB-Query may not execute for every tuple of outer query**
- Consider another query-

```
SELECT * FROM student WHERE
progid IN (SELECT pid FROM program
                WHERE did = 'EE' );
```

- Typically, after execution of inner query, outer query may be translated to:

```
SELECT * FROM student WHERE
progid IN (BEC, BEE);
```

- However this optimization may not be possible when you have “*correlated sub-query*”

Correlated Sub-Queries

- When inner query makes a reference to tuple of outer query then it is correlated sub-query. Consider following query -

- List employees, whose salary is more than department average:

```
SELECT eno, ename FROM employee as e
WHERE salary > (SELECT AVG(salary) FROM
employee WHERE dno = e.dno)
```

Execution of Correlated Sub-Queries

- Consider same query

```
SELECT eno, ename FROM employee as e
WHERE salary > (SELECT AVG(salary)
FROM employee WHERE dno = e.dno)
```

- Logically, it is as following: For each tuple of outer query, execute inner query.
- Note that it can not be executed once for all tuples of outer query, as the case be with un-related inner query, and we have to execute **SUB-Query for every tuple of outer query**
- This is identified problem with correlated sub-queries.

Correlated Sub-Queries could be expensive to execute – therefore should be avoided

- Correlated queries are expensive to execute, and can be avoided; for example the previous example

-
  ```
  SELECT eno, ename FROM employee as e
  WHERE salary > (SELECT AVG(salary)
  FROM employee WHERE dno = e.dno)
  ```

- can be re-written as-
  ```
  SELECT eno, ename, salary FROM employee as e
  NATURAL JOIN (SELECT dno, AVG(salary) as
  avg_sal FROM employee GROUP BY dno) as av
  WHERE salary > av.avg_sal;
  ```

more Correlated Sub-queries

- List down employees having salary greater than their immediate supervisors.

  ```
  select * from employee as e1 where e1.salary >
  (select salary from employee as e2 where e2.eno =
  e1.super_eno);
  ```

- Select employees having dependents older than 18 years:

  ```
  SELECT * FROM employee AS e WHERE eno IN (SELECT
  essn FROM dependent AS d WHERE essn = e.eno AND
  age(d.bdate) > interval '18 years');
  ```

- Attempt re-writting them without correlated query.

Compare a values with a bag of values (SQL)

- For example consider following two queries [Find out employee who have salary greater some or all employees of dno = 4]

SELECT enon, ename FROM employee WHERE salary

    > SOME (SELECT salary FROM employee WHERE dno = 4);

SELECT eno, ename FROM employee WHERE salary

    > ALL (SELECT salary FROM employee WHERE dno = 4);

Compare a values with a bag of values (SQL)

- Note the equivalences:

```
SELECT enon, ename FROM employee WHERE salary
          > SOME (SELECT salary FROM employee WHERE dno = 4); and
SELECT enon, ename FROM employee WHERE salary
          > (SELECT min(salary) FROM employee WHERE dno = 4);
```

```
SELECT enon, ename FROM employee WHERE salary
          > ALL (SELECT salary FROM employee WHERE dno = 4); and
SELECT enon, ename FROM employee WHERE salary
          > (SELECT max(salary) FROM employee WHERE dno = 4);
```

Compare a values with a bag of values (SQL)

- Comparative operators could be, one of following-

  >SOME, >=SOME, <=SOME, <SOME, =SOME, <>SOME

  >ALL, >=ALL, <=ALL, <ALL, =ALL, <>ALL

- Note: it can be easily proved that

  =SOME is identical to **IN**, and
  <>SOME is not identical to **NOT IN**

  = ALL is not identical to **IN**

  <> ALL (mean = NONE) and is same as **NOT IN**, and
  Earlier versions of SQL used **ANY** for **SOME**; today both keywords are used as synonymous.

Sub-queries in Update statements

-
  ```
  UPDATE employee
      SET salary = salary * 1.1
      WHERE ssn IN ( ... );
  ```

-
  ```
  DELETE employee
      WHERE ssn = ( ... );
  ```

EXISTS and NOT EXISTS in SQL

- Checks for emptiness of a relation and returns true or false.

- **EXISTS(r)** can be interpreted as “is there some tuple exists in relation r”

- **EXISTS(r)** returning true says that argument relation r is not empty

- Similarly, **NOT EXISTS(r)** returning true says that argument relation r empty

Example EXISTS

- List employees who have dependents older than 18 years

- **`SELECT * FROM employee AS e WHERE EXISTS (SELECT * FROM dependent AS d WHERE d.eno = e.eno AND age(d.bdate) > interval '18 years');`**

SQL- EXISTS and IN

- While they might appear to be serving similar purposes, semantically are different.
- Both appear as part of predicate in WHERE clause of SELECT
- IN:
  - Syntax: **`x IN ( r )`**
  - Meaning: checks existence of tuple x in relation r, if found returns true, other wise false. Normally x is a scalar value and r is a single column relation.
- EXISTS:
  - Syntax: **`EXISTS ( r )`**
  - Meaning: checks if r is a non empty relation. Returns true if the relation has at least one tuple, otherwise false.
- In both above cases **`r`** is a *relational expression* resulting a relation.

Bags and Relational Operations

Relational Operations and multiset (or bag)

- By Definition, relations are set; but implementations may permit duplicate tuples and such relations are called *bags*
- Normally stored relations (base) relations should still be sets, because most relations have Primary Key
- However SQL SELECT results are often bags, possibly because duplicate removal is expensive.
- To get *set* you use DISTINCT keyword

SQL and Multiset (or Bag)

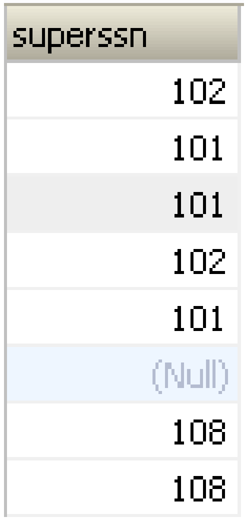
- SET operations, that are UNION, INTESECT, and EXCEPT in SQL yield their result as SET, that means duplicates are removed
- SQL however provides options by which you can have bag results by adding ALL keyword to operation name, i.e. UNION ALL, EXCEPT ALL or so.
- Let us see an example-

UNION/INTERSECT/EXCEPT ALL in SQL

- Compare result of following queries:

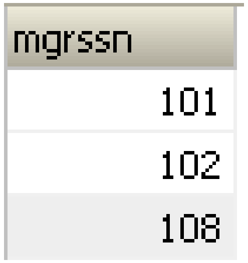
```
SELECT superssn FROM employee; --Q1
SELECT mgrssn FROM department; --Q2

SELECT superssn FROM employee
UNION
SELECT mgrssn FROM department; --Q3

SELECT superssn FROM employee
UNION ALL
SELECT mgrssn FROM department; --Q4
```

R



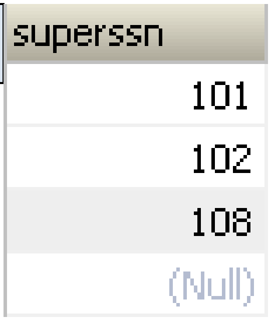
superssn
102
101
101
102
101
(Null)
108
108

S



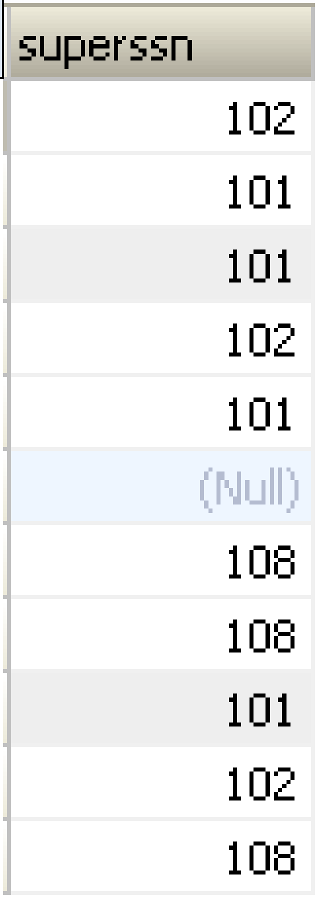
mgrssn
101
102
108

R UNION S



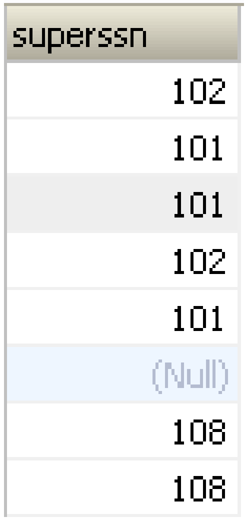
superssn
101
102
108
(Null)

R UNION ALL S



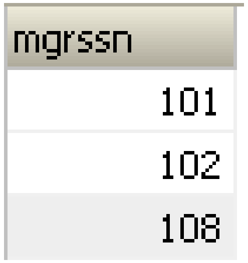
superssn
102
101
101
102
101
(Null)
108
108
101
102
108

R



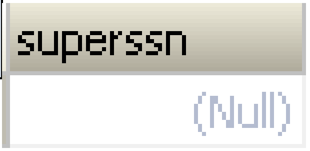
superssn
102
101
101
102
101
(Null)
108
108

S



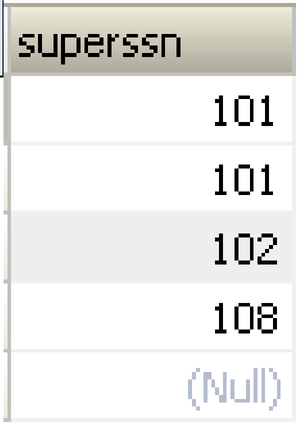
mgrssn
101
102
108

R EXCEPT S



superssn
(Null)

R EXCEPT ALL S



superssn
101
101
102
108
(Null)

UNION/INTESECT/EXCEPT ALL in SQL

- UNION ALL
  - count of an element e in result is sum of count in R and S
- INTERSECT ALL
  - min(count-r, count-s) of an element in R and S, is taken as result
- EXCEPT ALL:
  - Every occurrence of an element e in S decreases its count in R by one.

Functions and Operators in SQL

Functions and Operators

- SQL provides various functions and operators that can be used to create a new attribute in resultant relations

- There are typically, type conversion, arithmetic operators, mathematical, and string manipulation operators and functions. For example: substring, upper, lower, sqrt, ln, etc.

- Details for PostgreSQL functions can be seen at: http://intranet.daiict.ac.in/~pm_jat/postgres/html/functions.html.

Examples

```
SELECT ssn,
   fname || ' ' || minit || '. ' || lname AS name,
   current_date - bdate AS age FROM employee;

SELECT essn, hours*50 AS amount FROM works_on;

SELECT upper(fname) AS name, ln(salary) AS x FROM
   employee;

SELECT * FROM employee
       WHERE upper(fname) = 'FRANKLIN';

SELECT essn FROM dependent WHERE age(d.bdate) > interval
   '18 years');
```

BETWEEN and LIKE in SQL

- **BETWEEN, LIKE** are used in predicate:
  - **SELECT …. WHERE A BETWEEN 10 TO 20;**
  - **SELECT … WHERE A1 LIKE '%IX%‘ OR A2 LIKE ‘ABC%‘ OR A3 LIKE ‘%XYZ';**
  - **SELECT … WHERE A1 LIKE ‘_X_%‘;**

- Also: **NOT BETWEEN** and **NOT LIKE**.

Regular Expression Matching in PostgreSQL

- PostgreSQL also allows regular expression matching in string match using IS SIMILAR TO <reg-ex>

ORDER BY

- ORDER BY CLAUSE is used for ordering the resultant tuples of a SQL query.
- Following statements returns row-set from employee table, and rows are sorted based on salary. To order in descending order, we add DESC keyword after attribute name.

```
select * from employee order by salary;
select * from employee order by salary desc;
```

- Following statement returns row-set from employee table, and rows are sorted in ascending order of dno, and within dno all rows are sorted on salary in descending order-

```
select * from employee order by dno, salary desc;
```

LIMIT and OFFSET

- Examples below should be self explanatory
- Gives top three earners

```
select * from employee order by salary desc limit 3
```

- Gives next two earners after top 3

```
select * from employee order by salary desc
          offset 3 limit 2
```