Operations … SQL Specifics

pm jat @ daiict

Things that we talk about

- Result of SQL SELECT [SET or MULTI-SET] ?
- Nulls as special values in SQL
- Sub-queries in SQL
- EXISTS and NOT EXISTS in SQL
- Functions and Operators in SQL
- ORDER BY
- LIMIT and OFFSET

Operations … SQL Specifics

Result of SQL SELECT [SET or MULTI-SET] ?

Result of SQL SELECT [SET or MULTI-SET] ?

- By Definition, relations are set; but implementations may permit duplicate tuples and such relations are called *bag (multi-set)*
- Normally stored relations (base) relations should still be sets, because most relations have Primary Key
- However SQL SELECT does not produce SET, their results are often bags
  - possibly because duplicate removal is expensive.
  - We can use DISTINCT keyword for returning sets
- On the other hand, SQL SET operation (UNION/INTERSECT/EXCEPT) produce a valid “SET”.
  - possibly because duplicate removal comes as natural outptcome of executing set operations on relations.
  - But again, SQL allows producing bags too here through UNION ALL, EXCEPT ALL, and so

UNION/INTERSECT/EXCEPT ALL in SQL

- Compare result of following queries:

```
SELECT super_eno FROM employee; --Q1
SELECT mgr_eno FROM department; --Q2

SELECT super_eno FROM employee
UNION
SELECT mgr_eno FROM department; --Q3

SELECT super_eno FROM employee
UNION ALL
SELECT mgr_eno FROM department; --Q4
```

UNION ALL in SQL

- It just combines tuples from operand relations. The figure should depict the operation.

R

super_eno
105
105
null
106
106
102
106
102
101

S

mgr_eno
102
106
104
106

R UNION S

super_eno
null
106
102
104
105

R UNION ALL S

super_eno
105
105
null
106
106
102
106
102
101
102
106
104
106

INTERSECT ALL in SQL

- Common occurrences of tuples are included in the result.
- Let us say a tuple t appears m times in R and n times in S; then min(m,n) times the tuple t would appear in the result.

R

super_eno
105
105
null
106
106
102
106
102
101

S

mgr_eno
102
106
104
106

R INTERSECT S

super_eno
102
106

R INTERSECT ALL S

super_eno
102
106
106

EXCEPT ALL in SQL

- Occurrences of tuples are subtracted from the right operand relation.
- Let us say a tuple t appears m times in R and n times in S; then **m-n** times, the tuple t would appear in the result.

R

super_eno
105
105
null
106
106
102
106
102
101

S

mgr_eno
102
106
104
106

R EXCEPT S
super_eno
null
101
105

R EXCEPT ALL S
super_eno
null
101
105
105
102
106

Operations … SQL Specifics

Nulls as special values in SQL

Meaning of Null Values for an attribute

- NULLs require special treatment in different usage contexts in SQL.
- Because an attribute having NULL could mean either of the following-
  - Value is unknown, that is not available right now
  - No value for the attribute applicable to the tuple
- We have already seen how it is treated in aggregate operations.
  - Hope that treatment makes sense to you in this context of usage!
- Let us see, how it is interpreted when it appears in
  - Arithmetic expressions
  - Logical expressions in predicate

Meaning of Null Values for an attribute

- For example, how following expressions would be evaluated for tuples where salary is NULL?
- Arithmetic expression

  SELECT e.salary*1.1 from employee as e;
- Logical expression in predicate

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

  Select * from employee as e where e.salary > 50000 and dno=4;
- Let us take one by one

“Null Value” in arithmetic expressions

- Arithmetic expressions (+,-,*,/) involving null values result null
- There for expression “e.salary*1.1” would evaluate to NULL for employees having NULL for salary!
- Make sense?

Interpretation of “Null Value” in predicate

- Now suppose we have following query

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

- How these expressions would be evaluated for tuples where salary is NULL?
- Actually Interpret e here as tuple variable that ranges over all tuples of employee relations.
- For example, consider a query: select * from r where <condition>

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

- Let us say a query is executed as following:

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

Interpretation of “Null Value” in predicate

- In predicate expressions, the occurrence of NULL is read as “UNKNOWN”
- That means, in expressions like `t.a < 10`, `t.a` is taken as “UNKNOWN” when it happens to be NULL.
- For this purpose, SQL defines “UNKNOWN” as the third truth value (in addition to TRUE and FALSE) for evaluating logical expressions; and
- While evaluating WHERE clause tuples with UNKOWN or FALSE truth values are not included in the resultset
- So, now we have the answer to the question that how expressions in the following query would be evaluated for tuples where salary is NULL.

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

Truth values for UNKOWN

- **NOT**
  - **NOT UNKOWN -> UNKWON**
- **AND**
  - **TRUE AND UNKOWN -> UNKOWN**
  - **FALSE AND UNKOWN -> FALSE**
  - **UNKWON AND UNKOWN -> UNKOWN**
- **OR**
  - **TRUE OR UNKOWN -> TRUE**
  - **FALSE OR UNKOWN -> UNKOWN**
  - **UNKWON OR UNKOWN -> UNKOWN**

Null Values and Comparisons

- Following query will not include any tuple where either of value in 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 super_eno = NULL;
```

- This is so because Null = Null is also UNKOWN. For checking an attribute for having a NULL value, SQL provides IS NULL (and IS NOT NULL)
- We write as following for such situations –

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

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 r1
                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 refers to a tuple of the outer query then it is a correlated sub-query.
- Consider the following query –

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

- Are you able to determine, what does it compute?
  "List employees, whose salary is more than the department average"

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 the following:
  For each tuple of the outer query, execute the inner query.
- Note that it can not be executed once for all tuples of the outer query, as the case be with the un-related inner query, and we have to execute **SUB-Query for every tuple of the outer query**
- This is an 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 eno FROM
dependent AS d WHERE d.eno = 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

Sub-queries in Update statements

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

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

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.dob) > 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.

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 eno,
   fname || ' ' || minit || '. ' || lname AS name,
          current_date - dob AS age FROM employee;

SELECT eno, hours*50 AS amount FROM works_on;

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

SELECT * FROM employee
       WHERE upper(ename) = 'AMIT';

SELECT eno FROM dependent WHERE age(d.dob) > 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
```