

Operations ... SQL Specifics



pm @ daiict



Certain SQL specific features

- ORDER BY, LIMIT, OFFSET
- Functions for computed attributes in projection
- Certain issues related to NULLs,
 - UNKNOWN is third TRUTH value while evaluating SQL expressions
 - IS NULL as additional predicate
- **UNION/INTERSECT/EXCEPT ALL** in SQL
- Correlated Sub-Queries
- EXISTS
- Some (relation as BAG) comparison operators



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
```



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>**



Issues with Null Values



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-
SELECT **e.salary*1.1** from employee as e;
Select * from employee as e where **e.salary > 50000**;
- 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?



Issues with Null Values

- 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 UNKNOWN.
- When we compare a NULL value with another value including NULL, result is UNKNOWN.
- UNKNOWN is treated as third truth (in addition to TRUE and FALSE) value in SQL where clause evaluation



Truth values for UNKNOWN

- NOT
 - NOT UNKNOWN -> UNKNOWN
- AND
 - TRUE AND UNKNOWN -> UNKNOWN
 - FALSE AND UNKNOWN -> FALSE
 - UNKNOWN AND UNKNOWN -> UNKNOWN
- OR
 - TRUE OR UNKNOWN -> TRUE
 - FALSE OR UNKNOWN -> UNKNOWN
 - UNKNOWN OR UNKNOWN -> UNKNOWN



Null Values and Comparisons

- While evaluating WHERE clause tuples with UNKNOWN or FALSE truth values are not included in result
- Following query will not include any tuple where either of value is NULL irrespective of 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 UNKNOWN. 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 ;
```



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, INTERSECT, 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 / INTERSECT / EXCEPT ALL in SQL

- UNION ALL
 - count of an element e in result is sum of count in R and S
- INTERSECT ALL
 - $\min(\text{count-}r, \text{count-}s)$ 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.



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
- 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

- FROM clause of SQL SELECT can have a sub-query, as following-

```
SELECT e.ssn, fname, dno, dname
FROM employee AS e NATURAL JOIN (SELECT
mgrssn AS ssn, dno, dname FROM
department) AS dept;
```

- In queries like above naming to relation returned by subquery is required, even if the relation name is not used; and that is the only relation as in query below-

```
SELECT * FROM (SELECT mgrssn AS ssn,
dno, dname FROM department) AS dept;
```



Subquery in WHERE clause

- We have seen sub-query in **IN**, as
WHERE . . . IN (SELECT . . .)
- For example,
**SELECT * FROM employee WHERE ssn IN
(SELECT essn FROM works_on);**



Subquery in WHERE clause

- When used in where clause and a sub-query returns a single column, single tuple relation, it can be interpreted as single value as following
- **SELECT pname FROM project
WHERE dno = (SELECT dno FROM department
WHERE dname = 'Research') ;**
- Note: resultant relation of sub-query is getting compared with a attribute value – appropriate conversion takes place;
 - Has a underlying assumption that sub-query return a single tuple



Execution of Subquery

- Consider the query below-

```
SELECT pname FROM project
WHERE dno = (SELECT dno FROM department
WHERE dname = 'Research' );
```

- Can you guess logical execution of the query?
 - For each tuple of p from project
 - Select the tuple if p.dno= (select dno from department ...)
 - Should the inner query (in blue) be executed for every tuple p?



Execution of Subquery

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

```
SELECT pname FROM project  
WHERE dno = (SELECT dno FROM department  
WHERE dname = 'Research' );
```

- It is typically executed as following: Execute inner query, and let us say returns **5**, and then place it in outer query, and the query to be executed becomes following -

```
SELECT pname FROM project  
WHERE dno = 5;
```
- This is what dbms query optimizer will be able to do!



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



Execution of Correlated Sub-Queries

- Consider same query

```
SELECT ssn, fname 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 ssn, fname FROM employee as e
WHERE salary > (SELECT AVG(salary)
FROM employee WHERE dno = e.dno)`
- can be re-written as-
`SELECT ssn, fname, 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.ssn = e1.superssn);`
- Select employees having dependents older than 18 years:
`SELECT * FROM employee AS e WHERE ssn IN (SELECT essn FROM dependent AS d WHERE essn = e.ssn AND age(d.bdate) > interval '18 years');`
- Attempt re-writting them without correlated query.



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.essn
= e.ssn 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.



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 ssn, fname FROM employee WHERE salary

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

SELECT ssn, fname 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 ssn, fname FROM employee WHERE salary

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

SELECT ssn, fname FROM employee WHERE salary

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

SELECT ssn, fname FROM employee WHERE salary

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

SELECT ssn, fname 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 = (...);`