**IS 441 Week 15 Class Summary and Highlights (VERY Important; Could Be A Bit Complex)**

Dr. Yue Zhang

Class outline:

|  |  |  |
| --- | --- | --- |
| 1. Multi-table queries;
2. OUTER JOIN;
 | 1. Self Join
2. Suqueries;
 | 1. Exam 2 comments
 |

1. Multi-Table Queries
2. The “Table1 JOIN Table2 ON (join condition)” option:

Challenge 1: proper use of parentheses to enclose the correct pairs of tables to be joined;

Challenge 2: correct statement of the join condition;

Challenge 3: proper sequence of tables to be joined –

|  |  |
| --- | --- |
| FROM **((**Table1 INNER JOIN Table2 ON T1.PK=T2.FK**)**  INNER JOIN Table3 ON T2.PK=T3.FK) **INNER JOIN Table4 ON T3.PK=T4.FK**Not easy to write the statement correctly in syntax and logic | 1432 |

1. \*\* Recommendation: Use the format: FROM T1, T2, T3 WHERE (*state the join conditions in any order*) – as long as the join conditions correctly join the right pairs, which pair appears first and which second and which third bears no significance.
2. OUTER JOIN
	* + 1. ONLY WHEN a table’s ALL records will be displayed even some of the records do NOT have a match with records in another table in the join, should OUTER JOIN be used. In other words, if the target is to list the records that have a match, outer join is not only unnecessary but WRONG.
			2. In terms of number of rows in the final display of the query:
				1. INNER JOIN output has <= (less than or equal to) the number of the rows in the BIGGER table, since the JOIN may “filter out” some records which did not have a match.
				2. OUTER JOIN output has as many rows as the “dominating table” (explained immediately 🡺), since one of the two tables has ALL its records listed even some of them do not have a match with records in another table. In this case I call this table (the one with ALL records listed) as the “dominating table” (quotation marks mean “not a standard term; my ‘invention’ just to make it easy to refer to this kind of objects”).
				3. Graphical illustration:

INNER JOIN:

|  |  |  |
| --- | --- | --- |
| **Blue Table** | **Green Table** | **Inner Join** |
|  |  |  |

OUTER JOIN: BlueTable LEFT JOIN GreenTable 🡺 🡺 🡺 ALL from Blue, + matched rows from Green

|  |  |  |
| --- | --- | --- |
| **Blue Table** | **Green Table** | **OUTer Join** |
|  |  |  |

* + - 1. Syntax:
				1. FROM “Dominating-table” LEFT JOIN another-table
				2. FROM another-table RIGHT JOIN “Dominating-table”
				3. a and b have the same outcome.
			2. Recap on syntax and logical outcomes (in the context of the graph above):
				1. Blue LEFT JOIN Green
				2. Blue RIGHT JOIN Green
				3. Green LEFT JOIN Blue
				4. Green RIGHT JOIN Blue

“Verdict”:

a is what is depicted in the graph;

b is wrong (opposite to the graph);

c is wrong (opposite to the graph);

d is the same as a (“Blue LEFT JOIN Green” equivalent to “Green RIGHT JOIN Blue”);

1. Self Join
2. Concept and logic: Self Join corresponds to Unary Relationship in ERD. In implementation, self join relates some rows of a table to some OTHER rows of THE SAME table.

\*\*\*Note: Self join does NOT mean a row joins with itself! ! !

1. Implementation situation: one physical table is seen as two logical tables or “views”.
2. Understanding with example: relate employees with THEIR managers who are also rows in the Employee table:
	1. Name the same physical table as TWO DIFFERENT LOGICAL VIEWS – E (for employee) and M (for manager): SEALECT… FROM Employee E, Employee M, WHERE (join condition)
	2. Join condition: WHERE E.ManagerID = M.EmployeeID

Meaning: non-manager employee’s manager ID, equals (matches) manager’s employee ID – makes sense right?

* 1. I would call the above “**Crossed**”:

E. ManagerID

 =

M. EmployeeID

The above is just a “memory scheme”; please do not “over-interpret” it; I just tried to provide a memory tool to assist you to remember it BEFORE you fully understand it.

1. Subqueries (this part is LONG)
2. Overall/General: subqueries can be used to replace most o the components in a SELECT statement:
	1. In SELECT-clause to create a column to be displayed;
	2. In FROM-clause, to create a table/view as the data source;
	3. In WHERE-clause, to provide a value/values for comparison (>, =, <) or “fitting in among” (IN) for a row;
	4. In HAVING-clause, to perform similar roles as the above, yet for a group;
	5. In ORDER BY-clause, to order by a field that does not originally exist in the table but created by the subquery.
	6. c and d (WHERE and HAVING) are the most common, while e could be least common (haven’t seen an example, only logically possible).
3. Examples of **non-correlated** suqueries – the **subquery is executed ONLY ONCE**
4. Subquery in SELECT (and also in WHERE):

|  |  |
| --- | --- |
| SELECT RestaurantID, AnnualSales, (SELECT AVG(AnnualSales) FROM Restaurants) AS TableAvgFROM Restaurants WHERE AnnualSales > (SELECT AVG(AnnualSales) FROM Restaurants);  |  (Only capture half ) |

Roles of subqueries:

The one in WHERE-clause allows the row value AnnualSales to be compared with the average sales of the whole table;

The one in SELECT-clause allows the display of the whole-table average which would otherwise prohibited by the SQL rule – row values and set values cannot be in the same SELECT-clause.

1. Subquery in FROM:

Scenario: Find the average of the top sales performers in each city.

Analysis: the “top sales performers in each city” is a MAX with GROUP BY City – handled by the subquery; then the average of such values (top values) would be another aggregate function – handled by the outer query -- in the scope of the whole table.

Implementation: a query with the function AVG, with the argument being the outputs of a subquery producing top-performer sales.

|  |  |
| --- | --- |
| Code: | Output: |
| SELECT AVG(MaxSales) As AvgTop FROM**(SELECT MAX(AnnualSales) AS MaxSales FROM Restaurants GROUP BY City)**; |  |

\*\* Note: In this question, the inner query passed its outcome not only in value but also with a variable name MaxSale. The passing of the value was made possible by the reference of that variable in the aggregate function AVG(MaxSale) in the outer query.

1. Subquery in WHERE – the most common:

Example 1:

|  |  |
| --- | --- |
| SELECT RestaurantID, annualsalesFROM RestaurantsWHERE annualsales > **(SELECT AVG(annualsales) from restaurants)**;(Only capture half of the output screen) 🡺🡺 |  |

Example 2: Use IN

|  |  |
| --- | --- |
| SELECT RestaurantID, Annualsales, cityFROM RestaurantsWHERE Annualsales IN(Select Max(Annualsales) from Restaurants GROUP BY City)order by annualsales desc; | (Only captured half of the output screen) |

\*\* Note: “=” and “IN”:

“=” is used to compare a field with ONE value; “=” cannot be followed by multiple values;

“IN” allows a field to be compared with multiple values, and returns a TRUE if the field’s value equals ONE OF the several values in the parentheses after IN: “your value is among one of us.”

\*\*\* In this example, please understand that the subquery could return multiple values – a MAX is to be found for each city. So, the output of the subquery equivalent to the parentheses containing multiple values after IN in a simple case, such as IN (600000, 750000, 350000). The outer query then executes the “comparison of the field AnnualSales with the valueS – multiple values – until a match is found or until the whole list of values have been exhausted.”

1. Subquery in HAVING: Condition for a group

Example: Find the cities and their average sales for those cities whose average sales are higher than the overall average in the whole table.

|  |  |
| --- | --- |
| SELECT City, AVG(AnnualSales) AS CityAvgFROM RestaurantsGROUP BY CityHAVING AVG(AnnualSales) >(SELECT AVG(AnnualSales) FROM Restaurants)ORDER BY AVG(AnnualSales); |  |

1. **Correlated** subquery: Often times, a parameter is passed from outer query into the subquery, that when each row in the outer query is evaluated, the subquery will be executed once and return corresponding value(s) to the outer query for its processing, and then move forward to the next row in the outer query when the above process repeats. – So the **subquery would be executed N times**.

Example: Find restaurants whose sales are higher than the avg of ITS OWN CITY. “AVG of its own city” needs the city for AVG calculation being THE CITY of the restaurant 🡪 passing parameter.

|  |  |
| --- | --- |
| SELECT restaurantid, annualsales, cityfrom restaurants **R**where annualsales >(select avg(annualsales) from restaurantsgroup by city having city = **R**.city); |  |