**IS 441 Spring 2018 Review Class**

I. Basic Concepts and skills

1. SELECT:
   1. cannot mix row values (fields) with set values (aggregate functions)
   2. designate the table, if a field can be in two tables
   3. can use alias with AS; but this alias cannot be used later in calculation or comparison
   4. can contain a subquery (usually is used to bypass the limitation of point a above)
2. FROM:
   1. Single table – too simple to discuss
   2. Multiple tables, separated with comma, which (the tables) must later be joined using join conditions in WHERE
   3. Multiple tables, in the syntax “A JOIN B ON join condition” – watch out: that needs parentheses when there are 3+ tables
   4. Multiple tables, when one table will list all its rows no matter whether the row has a related row in another table – that is OUTER JOIN. Do NOT “abuse” outer join if there’s no logical needs as stated in sentence one in this bullet
   5. Can contain a subquery which produces data (multiple rows) that can be used just as any table
   6. Join condition:
      1. WHEERE
      2. ON
      3. i does not need parentheses, and ii MUST use parentheses if three or more tables are to be joined
      4. cannot perform outer join to join three tables: outer join is for two tables only
      5. outer join can only use ON, not WHERE
3. WHERE:
   1. Specifies conditions for rows
   2. Cannot mix with aggregate functions, because?
   3. The most “popular” place to use a subquery (when a row value is to be compared with a set value, the latter will be in a subquery)
   4. Multiple conditions MUST be connected with AND or OR, with a field be present in every part of WHERE separated by AND or OR
4. GROUP BY:
   1. GROUP BY fields that truly have group characteristics, such as Gender, Major; City, TypeOfService; Project, TypeOfEmployee; Category, etc
   2. NEVER try to group by fields that have unique values, such as DOB, Sales, Orderdate (unless this is a large data set where there’re many orders on a date that you do want to find out some common features of orders made on that date)
   3. \*\*\* With GROUP BY, we find features of GROUPS, such as AVG, SUM, COUNT, etc - \*\*no values for individual rows
5. HAVING:
   1. Specifies conditions for \*GROUPS\* - so:
   2. MUST always be after GROUP BY (can never appear before group by); and
   3. Used with aggregate functions (since it’s about GROUPS!) – this is in contrast with “b” in WHERE
   4. Repeat points “c” and “d” for WHERE
6. ORDER BY:
   1. Do NOT confuse this with GROUP BY!
   2. The \*LAST\* clause of the SELECT-statement
   3. Can have multiple fields in ORDER BY, in which … (how the multi-fields work in ORDER BY?)
   4. Can have ASC or DESC
   5. In the case of multiple fields, each can have its different order:

ORDER BY Sales DESC, OrderDate ASC

1. Don’t forget about INSERT, UPDATE

II. Scenarios of subqueries (using Restaurants as examples):

1. Those with sales higher than the average sales of ALL
   1. Field > (Subquery involving aggregate function)
2. By city: those cities whose average higher than the average sales of ALL
   1. GROUPing, HAVING group-avg > (Subquery involving aggregate function)
3. Those restaurants with sales higher than the average sales of \*ITS OWN CITY\*
   1. Passing parameter: forcing the inner query/subquery to run AVG on THE City that was passed from the outer query/main query

SELECT restaurantID, AnnualSales, City,

This part not necessary unless asked

(SELECT AVG(AnnualSales) FROM Restaurants

GROUP BY City

HAVING City = Rest\_Outside.City) AS CityAVG

FROM Restaurants Rest\_Outside

WHERE AnnualSales>

(SELECT AVG(AnnualSales) FROM Restaurants

GROUP BY City

HAVING City = Rest\_Outside.City);

1. (Using Employee DB) Those employees who were hired before their managers
   1. Self-join: employee’s manager ID = manager’s employee ID
   2. Logical condition: Employee’s dateHired < Manager’s dateHired

|  |  |
| --- | --- |
| SELECT E.Emp\_LName AS Employee, E.Emp\_hiredate AS E\_Date,  M.Emp\_LName AS Manager, M.Emp\_hiredate AS Mgr\_Date  FROM employee M, Employee E  WHERE E.Mgr\_Num = M.Emp\_Num  AND E.Emp\_hiredate < M.Emp\_hiredate;   1. All the same as the above, EXCEPT:   AND E.Emp\_Pct > M.Emp\_Pct; |  |

1. (Using Employee DB) Those employees who have higher bonus percentage than their managers
   1. Same operation as the above “4”

III. “OLD” contents:

1. ERD:
2. Functional dependency:
   1. FD exists \_\_ALWAYS\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
   2. PD only exists when \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
   3. TD is a \_\_\_\_\_Non-Key\_\_\_\_\_\_\_\_ fields determining \_\_\_other non-key fields\_\_\_\_\_\_\_\_\_\_
   4. What can we say about one field’s possible roles/positions in PD/TD? “Exclusive”
3. Normalization:
   1. Before normalization, there were N functional dependencies; after normalization, there should be \_\_\_ relations (tables)
   2. Before normalization, there were M fields (with their respected names) in one big relation (table); after normalization, there should be a total of \_\_ distinct fields in the decomposed, smaller relations (tables)
   3. Before normalization, several fields can be found logically connected in a functional dependency; after normalization, those fields should be \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
   4. Before normalization, two functional dependencies could be connected; after normalization, the relations (tables) resulted from those two functional dependencies should \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
   5. In the original functional dependency diagram, a field or a functional dependency must have a functional dependency arrow with other field(s) or other functional dependencies – otherwise that field (or functional dependency) is \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_