CECS 323 LAB EXECUTION PLAN

OBJECTIVE:

Gain some insight into how the optimizer does its work, and the place of indexes in satisfying queries.

INTRODUCTION:

SQL is a **non**-procedural language. That means that the developer can stipulate **what** they want from the database, but not **how** to get it. This begs the question: if the developer does not determine the "how" of a query, who does? The answer is: the database optimizer.

The objective of the optimizer is to review **likely** approaches to satisfying the requirements stipulated in each SQL statement **very** quickly. Once the optimizer has surveyed its options, it must select one. Clearly, if the optimizer takes too long to reach a decision, the overall performance of the query will suffer. The optimizer's decision how to execute a SQL statement is termed the explain plan. A query of any complexity could quickly have hundreds if not thousands of possible plans, so the optimizer must quickly discard plans that have little chance of performing well. To evaluate the merits of a potential plan, the optimizer has to take into account statistics on the data, not just the overall volume, but the distribution of values across the rows of the table, the results that the SQL statement calls for, any indexes present in the pertinent tables, and other factors that have a bearing on the overall performance.

Various database management systems have ways to influence the optimizer's decision. Some of those mechanisms are more direct than others. However, as a rule I try to avoid strong-arming the optimizer simply because any approach that I impose on the optimizer is likely to eventually become obsolete. For instance, a large amount of new data might get inserted into one or more of the tables in a query. Using the database's analyze table utility to refresh the database statistics on the relevant tables will give the optimizer better input data towards a better decision. Many database administrators have batch jobs that run on a schedule to run through all the tables in each database and gather fresh statistics on a routine basis.

Just about every relational database system has some tool(s) that allow the developer to see what decision the optimizer made. Lately, these tools are often visual in nature so that a large complex query can be more easily understood. In this lab, we will use MySQL workbench to access the same data that we have been interacting with this entire semester, but in MySQL instead of Derby.

To setup the data in MySQL:

1. Get into MySQL Workbench.
    a. Preferably, do this on your local installation of the MySQL database engine. If you have not installed MySQL on your computer, do not worry, I will give you instructions along the way for how to perform this lab using the campus MySQL instance.
    b. **If you are using your own MySQL instance**, then create a new schema for this lab:
        i. Right-click any schema visible in the Schemas pane on the left.
        ii. Select "Create Schema".
        iii. Give the new schema the name: "classicmodels".
2. Run the .sql script to build the tables and load the sample data:
    a. Go to here to get the create table statements and sample data insert statements.

b. **If you are running this on the campus MySQL instance,** be sure to **update** the statement "USE `classicmodels`;". Instead, tell MySQL to use whichever the schema name is for your personal schema. That will allow you to direct the create table and insert statements to the schema of your choice.
c. Run the build script.

At this point, you should be ready to start running SQL statements and see what the optimizer does with them.

For this exercise, we want to see what the optimizer did with your SQL statements. To get that:

1. Run the query (I will tell you more about just what query in a minute), just as you always have done. Validate that you received the output that you expected (you should receive 15 rows total, for two different customers).
2. Now that the optimizer has performed your query, you can find out how it chose to execute your query. Put your cursor in the statement, go to the query menu (up at the top) and select "Explain current statement".
3. The output window will show the plan. You may have to toggle MySQL workbench to show the visual version of the plan:

just select the indicated down arrow and select "Visual Explain".

4. At this point, you will want to save that image to disk (I will tell you what to call each image in the instructions). To save the image to disk, push the disk button that is a little to the right of the "Visual Explain" pull down. This will prompt you for the location and name of the visual explain plan file. Please follow the instructions (below) regarding what name to give each of these pictures.

Within the graphical execution plan, the text in each of the rectangles summarizes what that operation entails. If it says "Filter", then MySQL is essentially weeding out rows that are not of interest in your WHERE clause. The rest of the operations will either be a sort (for the order by), "Select" which is the culmination of your select statement, or "Unknown" which usually means that it is operating on data in memory. To read about the various types of join, go here, and search for "EXPLAIN Join Types". The floating point number above the upper left hand corner of the box represents the relative cost of the operation. I frankly do not know what the algorithm is that MySQL uses to come up with that number,

but it gives you a good idea of where the query will spend its time. Then you can see the number of rows returned from each operation.

Note that if you use aliases for your tables, the execution plan will use those alias names, rather than the table names.

PROCEDURE:

- For each scenario listed below, I want you to provide:
  - The SQL that you ran
  - The image of the **execution** plan that you received from MySQL
  - A discussion of any differences that you see in the explain plan relative to the very first explain plan that you will run.
  - The Relational Algebra that best represents your query. **Some** of these, I will give you the Relational Algebra to get you started, others of them, you will need to come up with the Relational Algebra yourself. Please consult the instructions for setting up the RelaX tool here.
  - The graphical image of the Relational Algebra query that RelaX gives you. This will help you to see the similarities and the differences between the sequence of operations that your Relational Algebra calls for, versus the way that the optimizer decided to tackle the query.

  Note that each scenario starts with a word or phrase in **bold** to identify the scenario. Use that name for your .jpg file graphical execution plan that you turn in.

1. **Basic** – Build a SQL statement and run it in MySQL Workbench connected to MySQL that will provide the customer name, order number, order date, and product name for all orders placed later than March 15th, 2015 and the customer's name **starts** with 'Mini'. Returns 15 rows.
   a. The Relational Algebra for this is:

   $\tau_{customername,\ orders.orderNumber,\ orderDate,\ productName}$
   $\pi_{customername,\ orders.orderNumber,\ orderDate,\ productName}$
   $\sigma_{customername\ like\ 'Mini\%'\ \wedge\ orderDate\ >\ date('2015\text{-}03\text{-}15')}$
   (customers $\bowtie_{customers.customerNumber\ =\ orders.customerNumber}$
   orders $\bowtie_{OrderDetails.orderNumber\ =\ orders.orderNumber}$
   OrderDetails $\bowtie_{OrderDetails.productCode\ =\ products.productCode}$
   products)

2. **Inline Query** – Modify **Basic** to use two **inline** queries: one that produces only the Customers that start with 'Mini%', and another that only produces the orders placed after 3/15/2015. Join these inline query results in just as though they were tables.
3. **Parenthesis** – Modify the **Basic** statement to use parenthesis to change the join order. Join Products and OrderDetails together first, then join the rest.
4. **Date Digging** – Modify the **Basic** statement again, this time instead of using orderdate itself, use the following clause in your where:

```
(year(orderDate) > 2015 OR
(year(orderDate) = 2015 AND month(orderDate) > 3) or
```

```
(year(orderDate) = 2015 AND month(orderDate) = 3) and day(orderDate) > 15) AND …
```

   a. This gives you > '2015-03-15', just in a very roundabout way.

5. **Index** – Add an index to the customer name, then run **Basic** again.
   a. To add an index to a table in MySQL:

      create index <name> on <table name> (<list of columns>);

   b. The Relational Algebra for this is:

      $\tau_{customername,\ orders.orderNumber,\ orderDate,\ productName}$
      $\pi_{customername,\ orders.orderNumber,\ orderDate,\ productName}$
      $\sigma_{customername\ like\ 'Mini\%'\ \wedge\ orderDate > date('2015\text{-}03\text{-}15')}$
      (customers $\bowtie_{customers.customerNumber = orders.customerNumber}$
      orders $\bowtie_{OrderDetails.orderNumber = orders.orderNumber}$
      OrderDetails $\bowtie_{OrderDetails.productCode = products.productCode}$
      products)

WHAT TO TURN IN:

- See the Procedure section.
- Your team's filled out collaboration document. You can find a template for that here.