Homework Problem Set 9: Indexing / Performance

# Overview

In this lab, we will explore how to profile queries to determine how performance can be improved and how to write a subsequent index to improve performance.

## Learning Objectives

Upon completion of the lab, you should be able to:

* Read query plans.
* Write indexes to convert scans to seeks.
* Create indexed views and column store indexes.
* Learn to force a query to use an index.

## What You Will Need

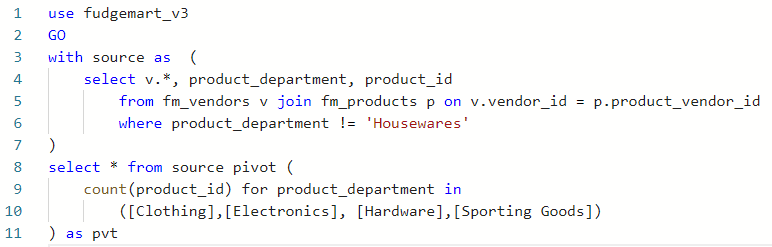
To complete this lab, you will need the learn-databases environment up and running, specifically:

* Microsoft SQL Server DBMS.
* Provision the **payroll**, **vbay**, **and fudgemart\_v3** databases using the database provisioner application <https://localhost:5000>.
* Azure Data Studio connected to SQL Server with an open query window.
* Please review the first lab if you require assistance with these tools.

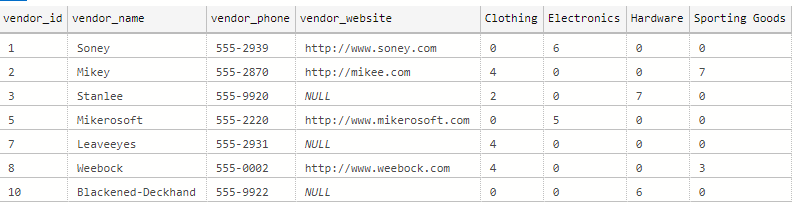
# Walkthrough

Let’s walk through the process of using the query plan to discover index candidates and then create an index to leverage for the query.

As a data analyst for **fudgemart\_v3**, imagine you need to write a query to get counts of products supplied by each vendor. This should include department (except “Housewares,” which is being discontinued), so you settle on the following PIVOT query:

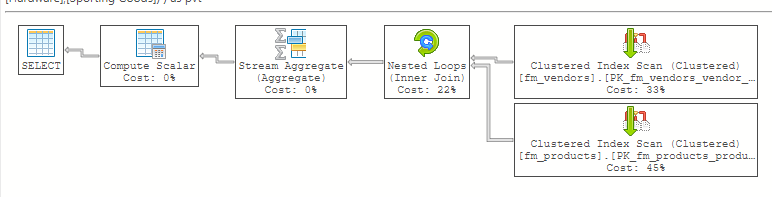


Producing the following output:

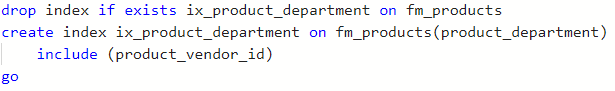


This query probably doesn’t need to be improved, but let’s assume it does and consider ways we can do it. The general guidelines are:

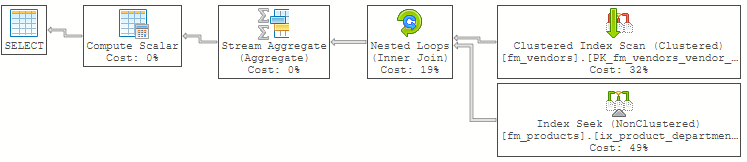
1. Seeks over scans. Jump right to the page rather than loop over all the pages until you find the one you need.
2. Nonclustered over clustered (assuming the index keys have far fewer rows than the actual table).

Upon hitting the **Explain** tool in the toolbar, you see the following plan:  


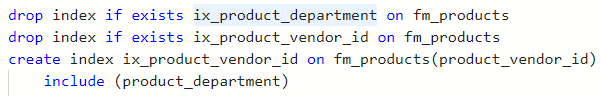
We read these plans from right to left. At the far right, we are collecting data from the tables. The top does a clustered index scan over the **fm\_vendors** table (this is expected because the table has a PK). The bottom does a clustered index scan over the **fm\_products** table (again, expected).

One way we can improve query performance is to index **product\_department** from **fm\_products**. Why? It is in the WHERE clause, and if we can seek the values versus scan them, it would be better. What should we include from this table in the index? The foreign key **product\_vendor\_id** because it’s included in the JOIN. Generally, if it’s included in the SELECT or JOIN, it should be added to the index INCLUDE clause. Why don’t we add **product\_id**? Well, as the PK, it’s included in all indexes. You decide to try this index:  


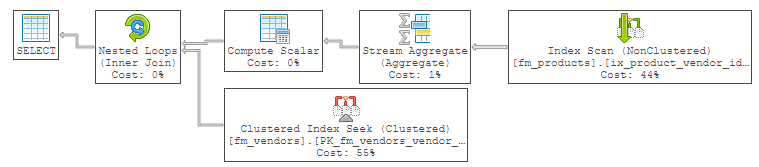
which produces this query plan:



Looks like we’ve swapped out a clustered index scan on **fm\_products** for an index seek on **fm\_products**. This should be a considerable improvement.

Let’s try something else. What if we try the reverse? We use the **product\_vendor\_id** as the index key and the **product\_department** as the included value. It makes sense because we are joining the tables on **product\_vendor\_id** so indexing this foreign key should improve the join process. You do this:  


which yields this query plan:



This plan builds the aggregates and then joins to the vendors table last. It scans an index of **fm\_products**, then seeks the clustered index of **fm\_vendors**.

* The first plan scans/loops over each vendor from the **fm\_vendor** table, seeking each product from the index.
* The second plan scans/loops over the **product\_vendor\_id** in the index, seeking each vendor from **fm\_vendors**.

Which is better? I’m inclined to believe the second is better because we are not scanning a clustered index, but the reality is, it will depend on a variety of factors, and the data here are too small to index anyway.

# Questions

Answer these questions using the problem set submission template. For any screen shots provided, please follow the guidelines for submitting a screen shot.

Write the following as SQL programs. For each, include the SQL as a screen shot with the output of the SQL code.

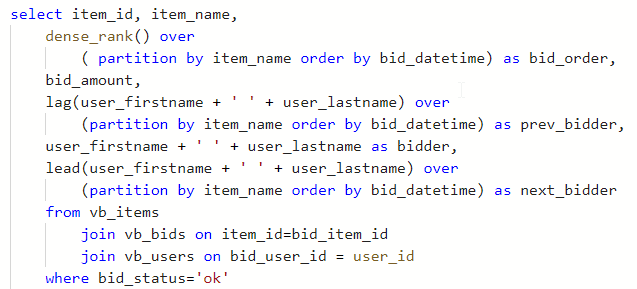
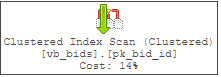
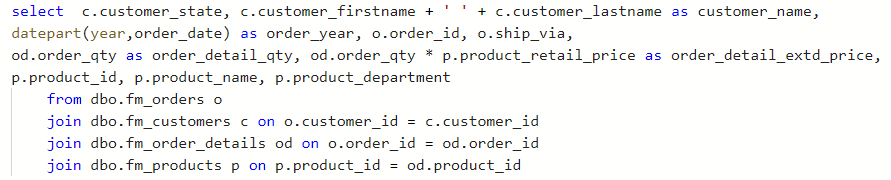
1. Using the **payroll** database, write an index to improve the performance of the following query:

select employee\_id, employee\_firstname, employee\_lastname, employee\_jobtitle

    from employees

    where employee\_jobtitle = 'Store Manager'

        or employee\_jobtitle = 'Owner'

1. Your screen shot should include the created index SQL code and the query plan demonstrating the index is being used.
2. Write another query using GROUP BY, which also uses the index you created in the first question.
3. For the following query from a previous assignment, which provides a rank of each bid on an item:  
     
   Implement the query, and run it. Provide a screen shot of the query plan, and include the portion where the **vb\_bids**, **vb\_items**,and **vb\_users** tables are selected and joined together.
4. Write an index to improve performance of the query by replacing the clustered index scan on **vb\_bids**   
     
   with an index seek on the same table. Provide a screen shot of your index code and a screen shot of the query plan demonstrating the index is being used to draw data into the query.
5. Using **fudgemart\_v3**, create a schemabound view from the following query:   
     
   Name the view **v\_orders** . Provide a screen shot of the code and sample output that conveys the query ran and created the view.
6. Write code to add a unique clustered index to the view **v\_orders**. Execute your view ( **select \* from v\_orders)**,and then observe the query plan to see if the index is being used. If the index is not being used, that’s an indication there is not enough data to warrant the index. You can force the index to be used by using the **noexpand** option on the query: **select \* from v\_orders with (noexpand)**.Provide a screen shot of code to create the index and execute the view along with the query plan showing the index is used.
7. Write code to add a columnstore index to **v\_orders**. Include all the columns from the view in the column store index. Provide screen shots of code to demonstrate you created the columnstore index and that these queries use it:  
   