Database Technologies Project – Phase 2

Overview

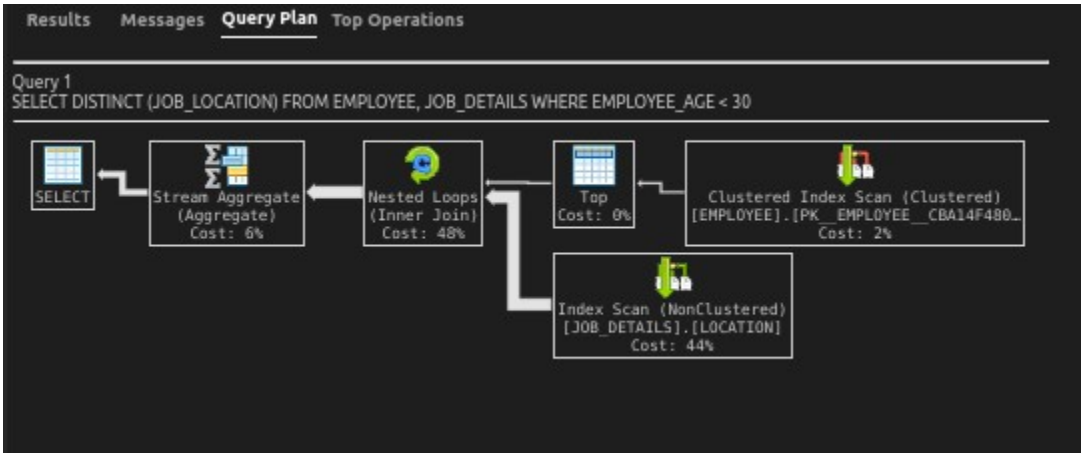
In the second phase of the project, we aim to unserstand the impact of operator choice and ordering in the selection of physical operators and overall cost of execution, as well as strategies to improve the cost of these operations. The objectives of this phase are -

1. Identify performance bottlenecks in complex queries, and suggest strategies to remedy them
2. Rewrite queries using appropriate methods to reduce the cost
3. Compare performance of multi-table joins using different order of joining the tables

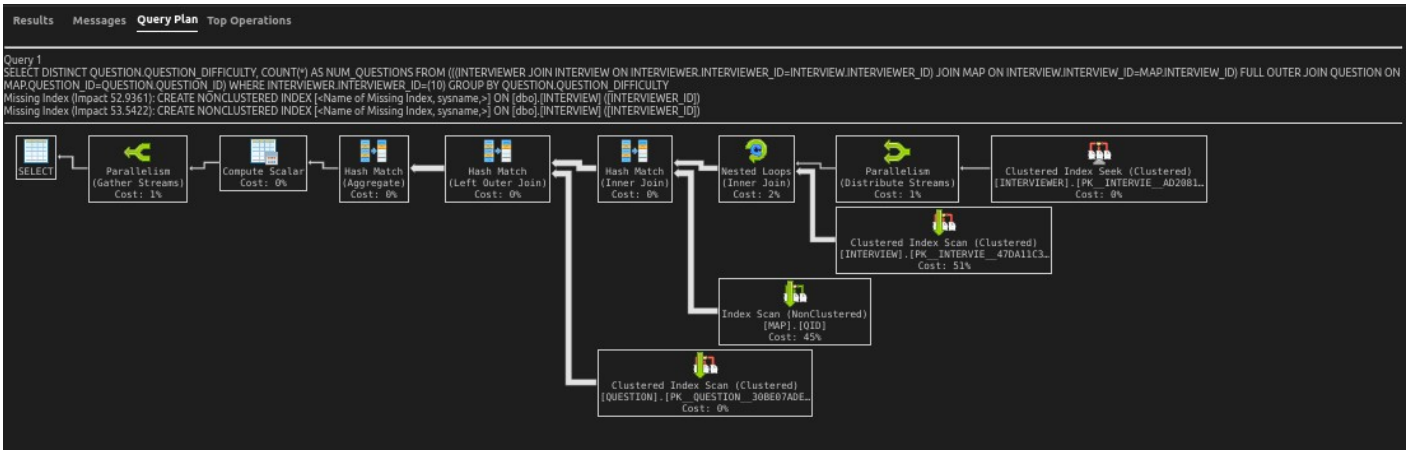
The entire project can be found at https://github.com/anihm136/DBMS_Project. Relevant SQL files are uploaded in the project submission folder as well. SQL Server 2019 on an Ubuntu 18.04 Docker container has been used for the project, mssql-cli for executing the queries and Microsoft Azure Data Studio for visualizing the execution plans

A. Queries used and their costs

We first look at some queries and their costs, and try to analyse the major contributors to the cost and how they can be reduced



Here, we can see that the two major contributors to the cost are the non-clustered index scan initially, and the nested loop join (converted from the cartesian product in the query). The cost of the index scan can be alleviated by adding an index to the LOCATION attribute. The nested loop join can potentially be optimised to a hash-based join if the join condition is stated, explicitly, such that the optimiser does not need to guess about the attributes required further in the computation



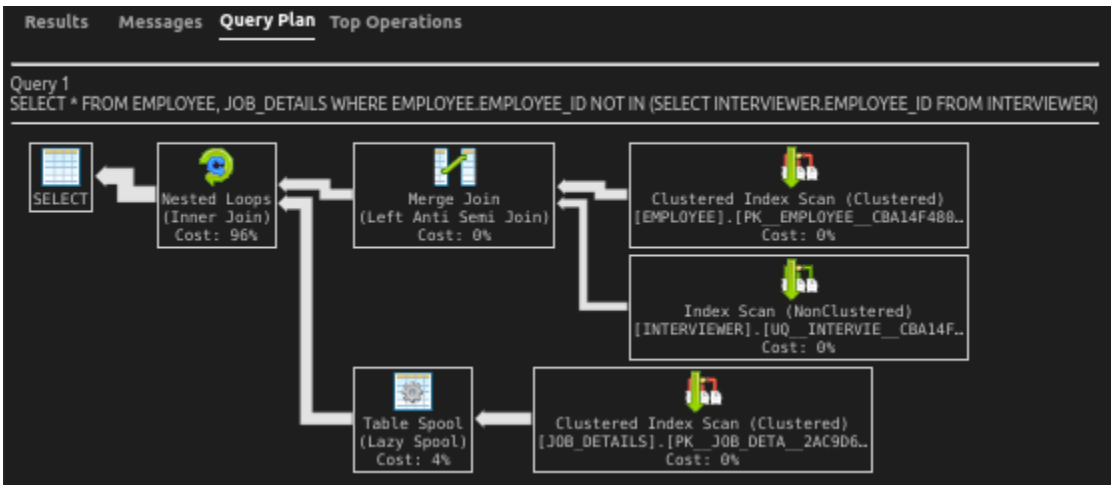
In this complex query, we can see the operation of several kinds of operators, primarily a multi-table join as well as aggregation. Once again, the major contributors to the cost appear to be the index scans, which can potentially be remedied by indexing the appropriate attributes. In addition, we see that the joins are generally optimised when explicit logical joins are performed as opposed to cartesian products being converted to joins by the optimiser.
Note that in both cases, the optimiser has correctly moved the select as far down the tree as possible

B. Comparing equivalent queries

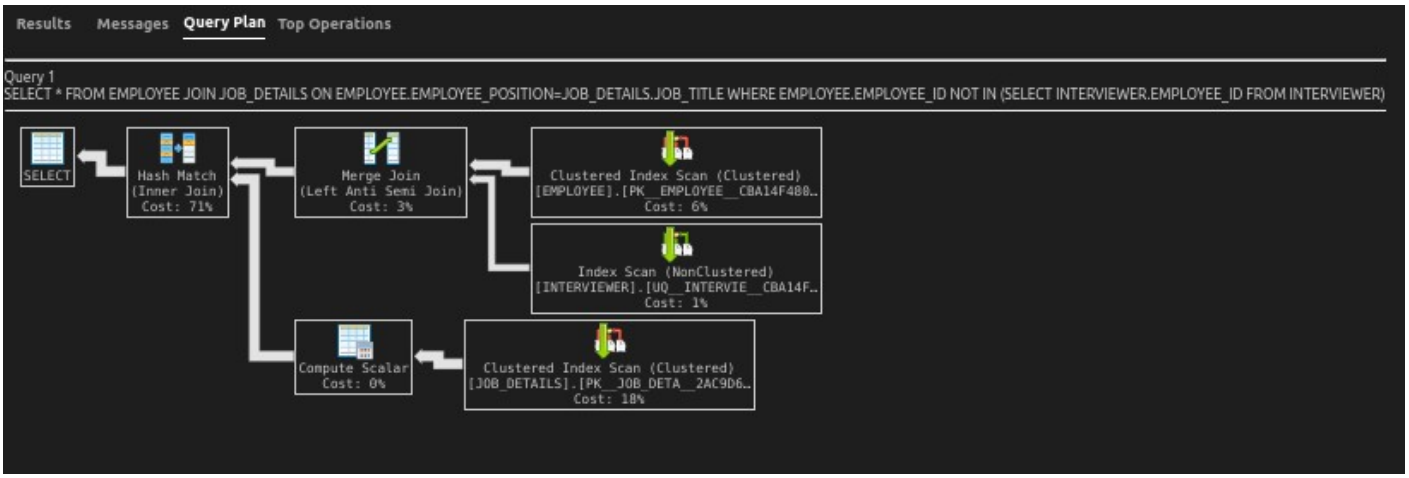
We consider two equivalent queries to select the employees and their job details, who are not interviewers -

1. SELECT * FROM EMPLOYEE, JOB_DETAILS WHERE EMPLOYEE.EMPLOYEE_ID NOT IN (SELECT INTERVIEWER.EMPLOYEE_ID FROM INTERVIEWER)
2. SELECT * FROM EMPLOYEE JOIN JOB_DETAILS ON EMPLOYEE.EMPLOYEE_POSITION=JOB_DETAILS.JOB_TITLE WHERE EMPLOYEE.EMPLOYEE_ID NOT IN (SELECT INTERVIEWER.EMPLOYEE_ID FROM INTERVIEWER)

The only difference between the two queries is that the first one uses a cartesian product, while the second one uses an explicit join on the correct attribute



Query using cartesian product

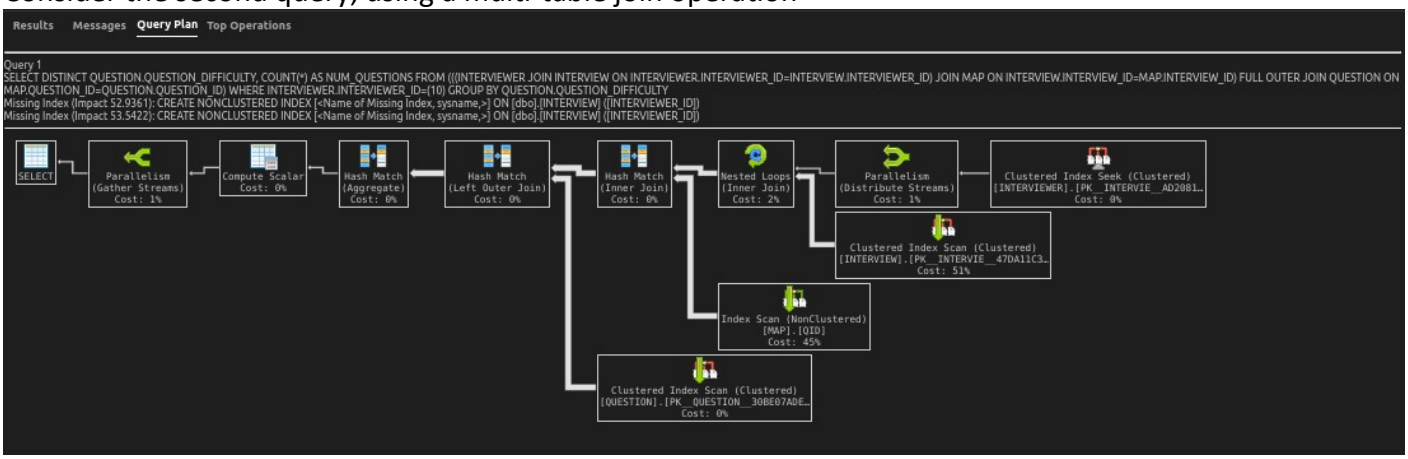


Query using explicit join

From the two visualizations we can see that the query with the explicit join has better performance than the one using cartesian product, even though the two queries perform the same task and are being optimised. The major difference we see is in the cost of the physical join when it is changed from a nested loop join to a hash join.
From this, we can deduce that, although optimisers exist to improve the performance of the queries, writing good logical queries can make a marked improvement to the performance in real time.

C. Performance of multi-table joins

Consider the second query, using a multi-table join operation



Changing the order of joins does not affect the order of physical operations, since the optimizer chooses the best order of tables for join. From this choice, we can see that the joins are distributed such that a majority of the join is handled by hash-based joins, handling the smaller relations first, before performing a nested loop join on the largest relation