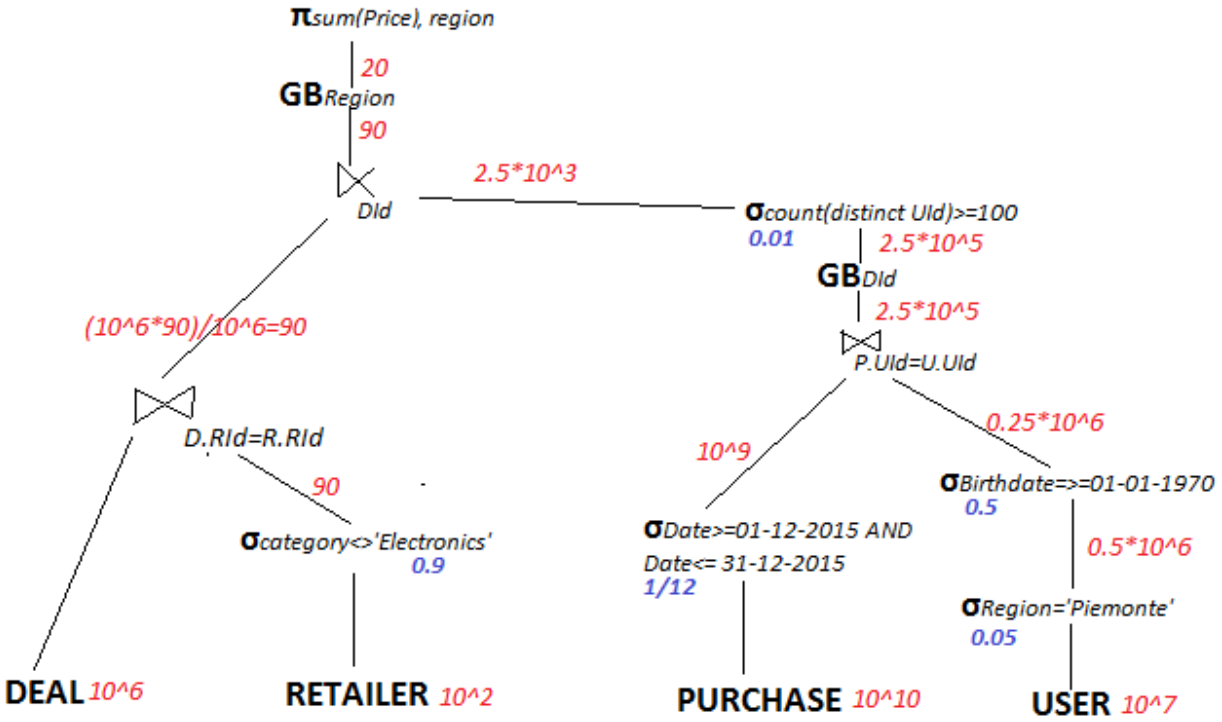
Homework #1

SECTION 1: ALGEBRAIC EXPRESSION

Query tree and cardinality analysis

Representation:
in blue, reduction factors of the select queries
in red, cardinality on the branch.



GROUPBY Anticipation

Let's start by analyzing two levels of query: the inner query and the outer query.

INNER QUERY: It is possible to anticipate the Group By on the Purchase branch, by adding a new Group By clause, based on two attributes P.UId and Did, right after the selection statement. This way we reduce the cardinality on the left branch of the Inner Join on UId, and thus the computational time required by it.

OUTER QUERY: Anticipating the Group By clause based on the Region attribute, would be neither possible nor useful. If we were to anticipate it on the left branch of the semi-join, no real benefit would be provided as the cardinality of that branch is

already low (~90). The anticipation on the other branch would not be useful either, as a Group By based on DId is already being applied.

Indices

1. Primary B+-tree index on Rid (primary key of Retailer table). This is particularly useful because Rid is a number that is used to insert tuples in their natural order and as the attribute for the join. This way, the performance of a join executed with the Nested Loop approach with Retailer as inner table would be improved by the index. In the NL, the Retailer table would be accessed by an index full scan, while the fast full scan approach is not a suitable one as all attributes are needed.
2. Secondary B+-tree on Data (of the table Purchase), as there is a selection with a good reduction factor (1/12)
3. Secondary hash index on Region (of the table User), as there is a selection with a good reduction factor but the attribute is not numerical

Group By (after anticipation)

1. $GB_{Did,UId}$: HASH, as the final result set is quite large
2. GB_{Did}: HASH, as the final result set is large and a previously executed join is done on another attribute
3. GB_{Region}: SORT, as the final result set is small and no operations previously executed applied a sorting order on the Region attribute.

Joins

1. $\bowtie_{UId}$: HASH JOIN, as both tables are large
2. $\bowtie_{RId}$: NESTED LOOP, with Retailer as Inner table (previously explained in the *Indices* section)
3. $\ltimes_{DId}$: NESTED LOOP, with the left branch as the inner table, because of its smaller size

Access Methods

1. RETAILER: Index full scan
2. DEAL: Full Table Scan
3. PURCHASE: Index Range Scan
4. USER: Index Full Scan

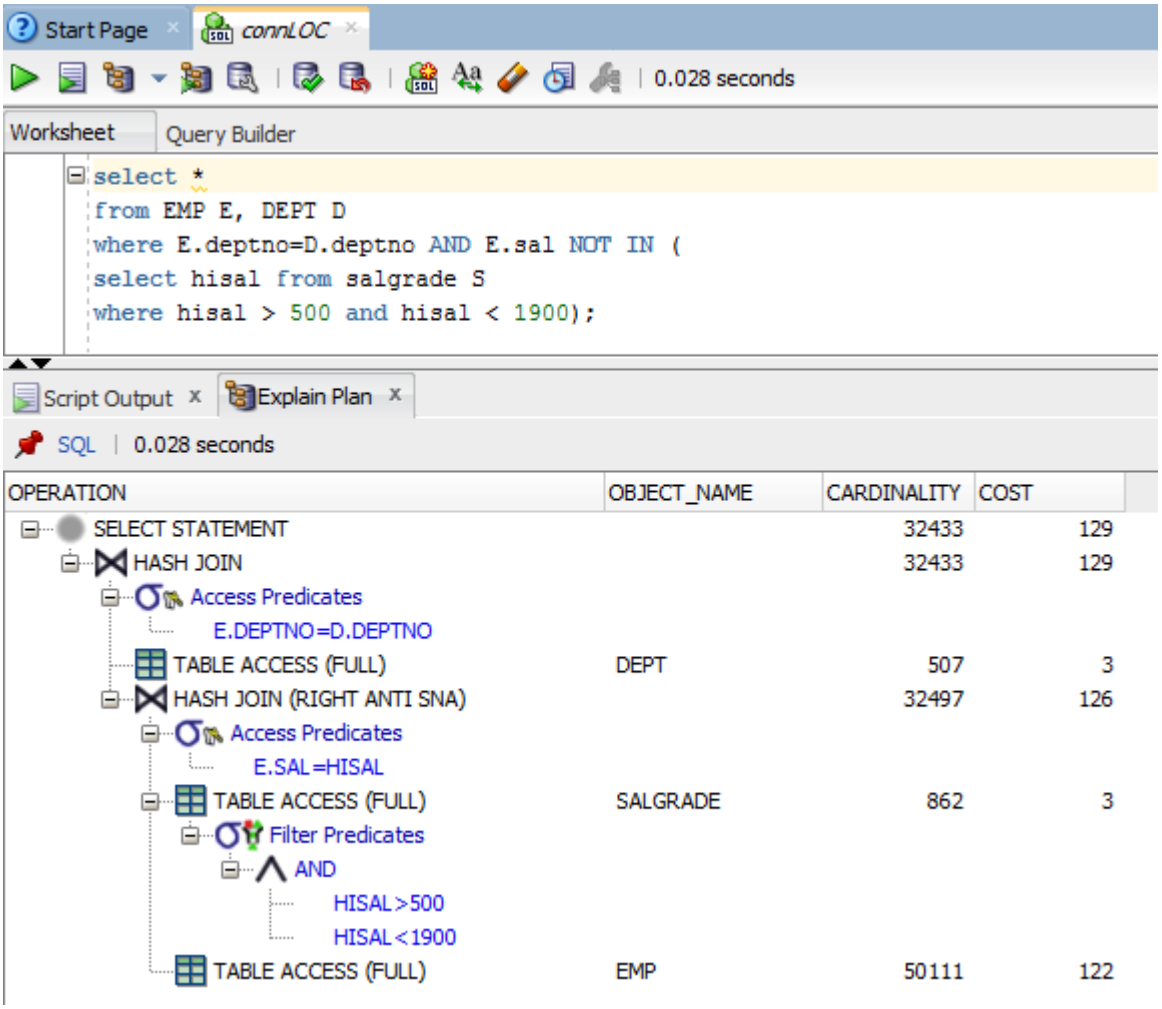
SECTION 2: SQL INDICES

The tables provided in the Oracle DB for the lab experience are three:

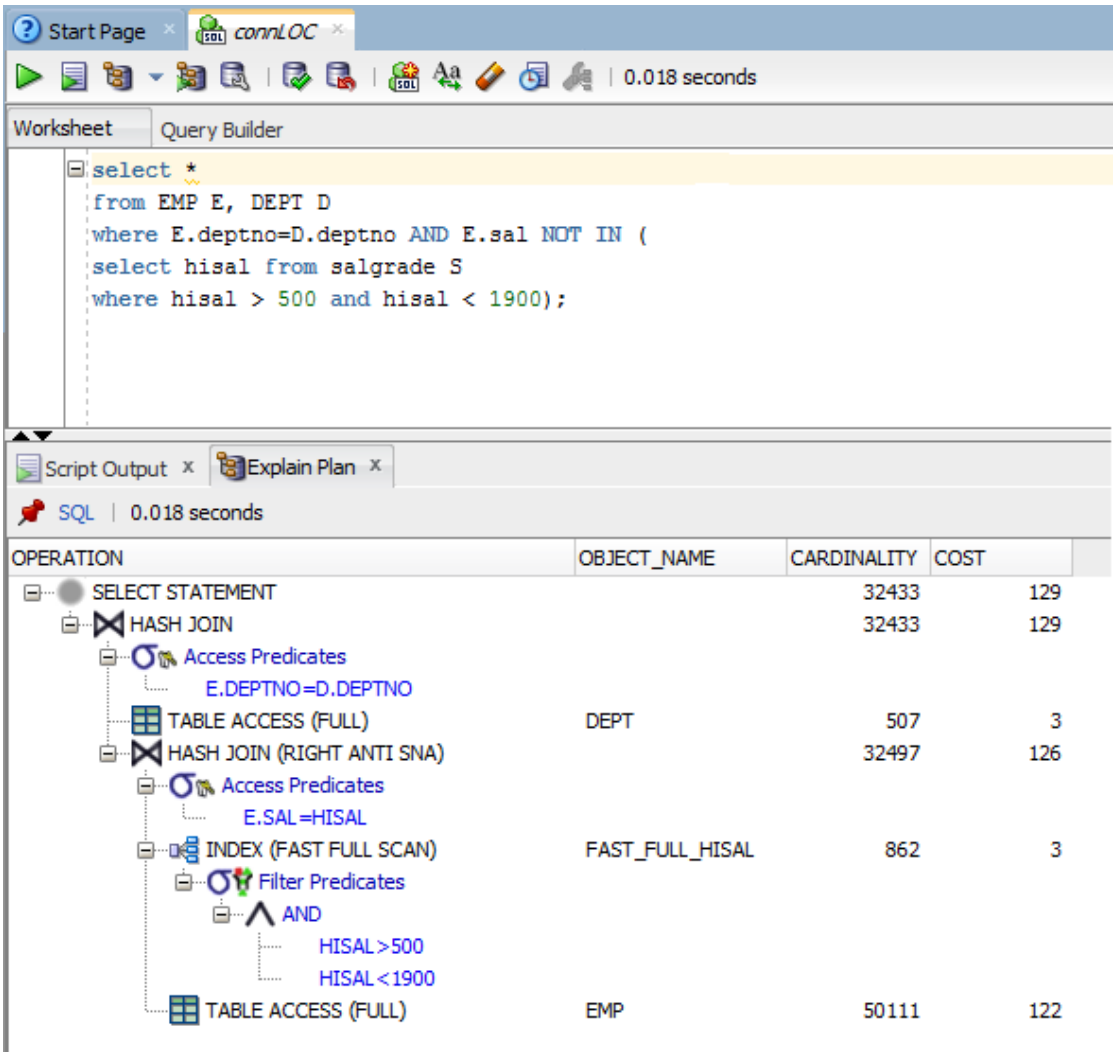
1. SALGRADE, with a cardinality of 999 (rounded to ~1000)
2. EMP, with a cardinality of 50111 (rounded to ~50000)
3. DEPT, with a cardinality of 507 (rounded to ~500)

Analyzing both queries, we can see that the WHERE condition is a RANGE. This means that HASH indexing and BITMAP indexing on SALGRADE would not provide any actual benefit. In the following considerations, B+-TREE indexing will be applied on the *hisal* attribute of the SALGRADE table.

The query #1 has a lower selectivity, due to having a larger range in the WHERE condition. Applying the previously mentioned index, would prove no real benefit, as the following images show:

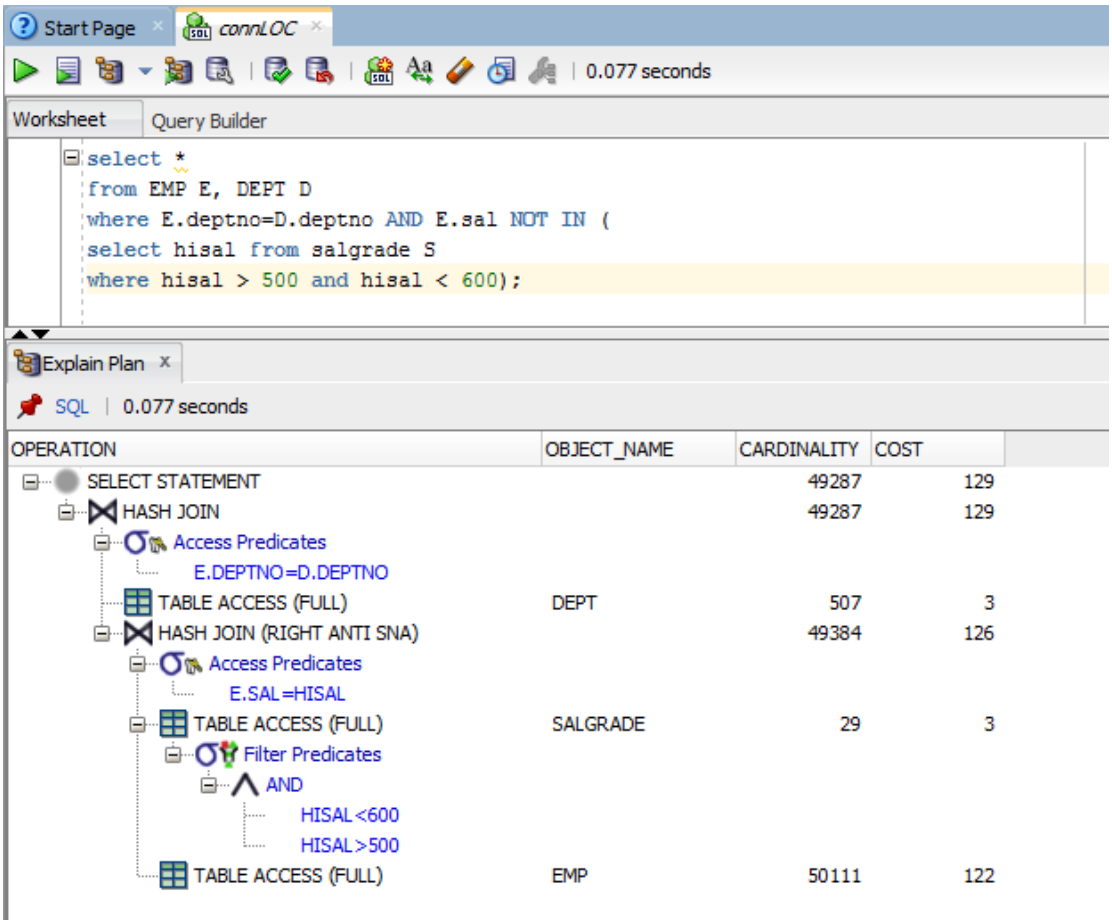


0.1 - Query #1 with no indices on SALGRADE table

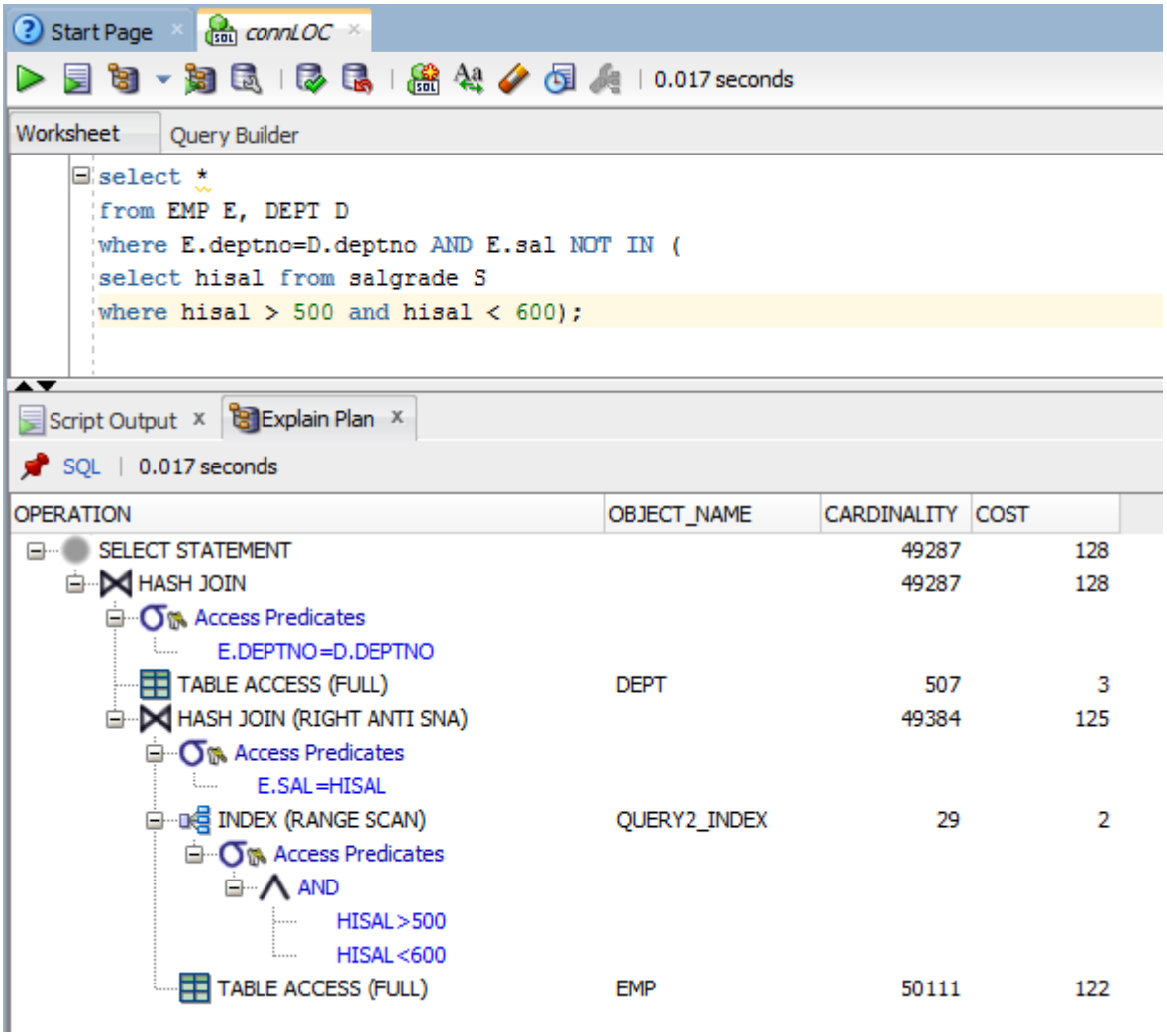


0.2 - Query #1 with index on SALGRADE table

From the image, no reduction in computational cost is shown, and just a slight improvement in the query execution time.



0.3 - Query #2 with no indices on SALGRADE table



0.4 - Query #2 with index on SALGRADE table

The index on the *hisal* attribute proves to be very useful for the second query: due to the higher selectivity of the *where* clause in the query, the index improves the computational cost from 129 to 128 and therefore the execution time of the query from 0.077 seconds to 0.017 seconds.