An ambitious student tries to implement an OR clause by executing two filter operations, each with the same input and output files, but with different predicates. Discuss what is the problem with this approach and how you would fix it.

This is a problem because the second filter will overwrite the results of the first one, leading to the loss of all the tuples that satisfy the first predicate but not the second. This is an incorrect implementation of an OR clause, which requires outputting all tuples that satisfy either predicate. We can fix this by writing the output of each filter operation to a separate intermediate output, and then the results should be combined into a final output while ensuring duplicate tuples are handled appropriately. This ensures that the final result correctly reflects the union of both predicate results.

The aggregate operation supports grouping by a single field. Discuss how you could extend this to support grouping by multiple fields. How can you implement the HAVING clause?

In order to support grouping by multiple fields, We can structure Group d to store a vector of field_t as the group key instead of a single field. In this the tuples with the same combination of values across all specified group fields would be grouped together via comparison between vectors. Then A hashmap can be used to retrieve these multi-field keys.

To implement the HAVING clause step we can add after aggregation an additional filtering step before the output, where the aggregated result of each group is evaluated against the HAVING condition. Only groups satisfying this condition are populated in the output.

What is the complexity of your join operation? How could you improve it?

Since we are using a nested loop join, where for each tuple in the left table scans all tuples in the right table, the time complexity is of O(N × M), where N and M are the number of tuples in the left and right tables respectively. This join is highly redundant for big datasets

To improve it we can use a hash join that can be used when the join predicate is equal. This involves building a hash table on the join field of the smaller relation and then probing it with tuples from the other relation, reducing the expected time complexity to O(N + M).

Given the cardinality of a selection predicate, can you estimate the IO cost of the query? What other factors would you need to consider to estimate the IO cost?

Yes, knowing the cardinality helps estimate the I/O cost by indicating how many pages are likely to be accessed to retrieve the matching records. However, to accurately estimate I/O cost, we also need to consider Page layout, tuple density, Index availability, Selectivity of the predicate and the Buffer pool size. These factors can help in determining whether a full scan, index scan, or random I/O is needed, significantly impacting the overall I/O cost.

In the previous assignment we introduced a join operation. How would you estimate the cardinality of a join between two tables based on the histograms of the join columns?

To estimate the cardinality of a join using histograms of the join columns, we compare the frequency distributions of the join attributes in both tables. For an equi-join, the standard estimate is:

$$\text{Estimated Cardinality} = \frac{T_R \times T_S}{\max(V_R, V_S)}$$

$$T_R = \text{Number of tuples in the left (or first) relation}$$
$$T_S = \text{Number of tuples in the right (or second) relation}$$
$$V_R = \text{Number of distinct values in the join attribute of the left relation}$$
$$V_S = \text{Number of distinct values in the join attribute of the right relation}$$

A table is stored in a file that consists of 150000 pages. Assume the cardinality of a predicate is 1000, a leaf page can fit 50 tuples, and the table is stored in a BTreeFile with 3 levels (root -> internal -> leaf). How many pages would you need to read to evaluate the predicate? What if the table was stored in a HeapFile?

In first case since we have

Cardinality = 1000 tuples
Each leaf page can hold 50 Tuple
1000/50 = 20 leaf pages need to be read

Since the depth of B+ tree = 3 To reach one leaf, we read
1 root + 1 internal + 1 leaf = 3 pages

For 20 leaves, we can reuse the root and internal pages. So, we need
1 root + 1 internal+ 20 leaf pages = 22 pages

However in case of an heapfile we have to scan all 150,000 pages to find matches unless there's a supporting index and thus the I/O cost = 150,000 page reads