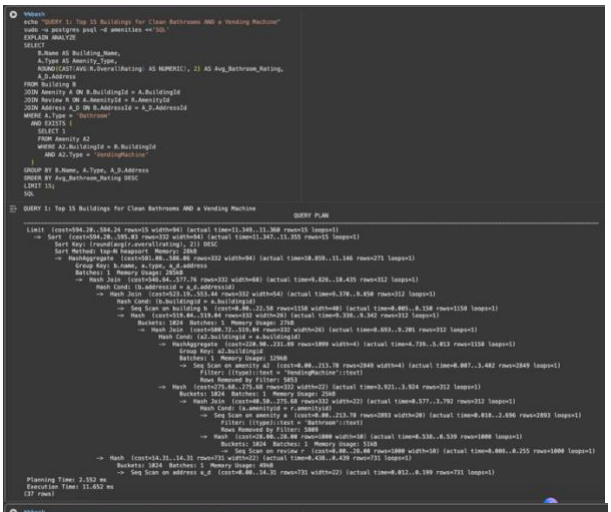
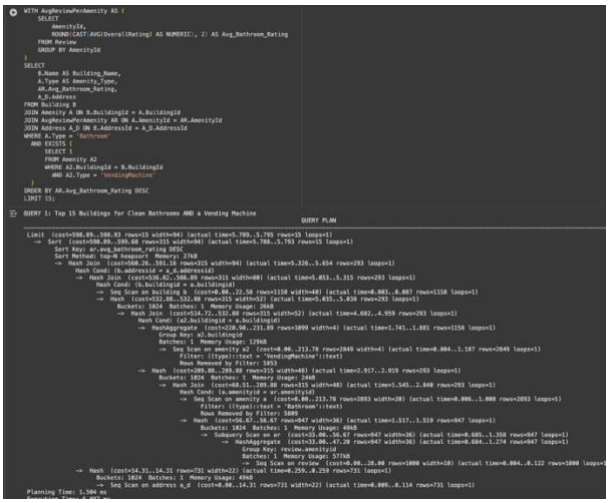
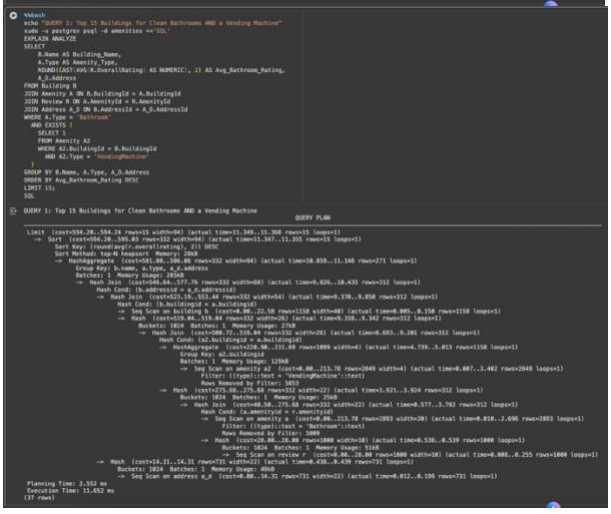
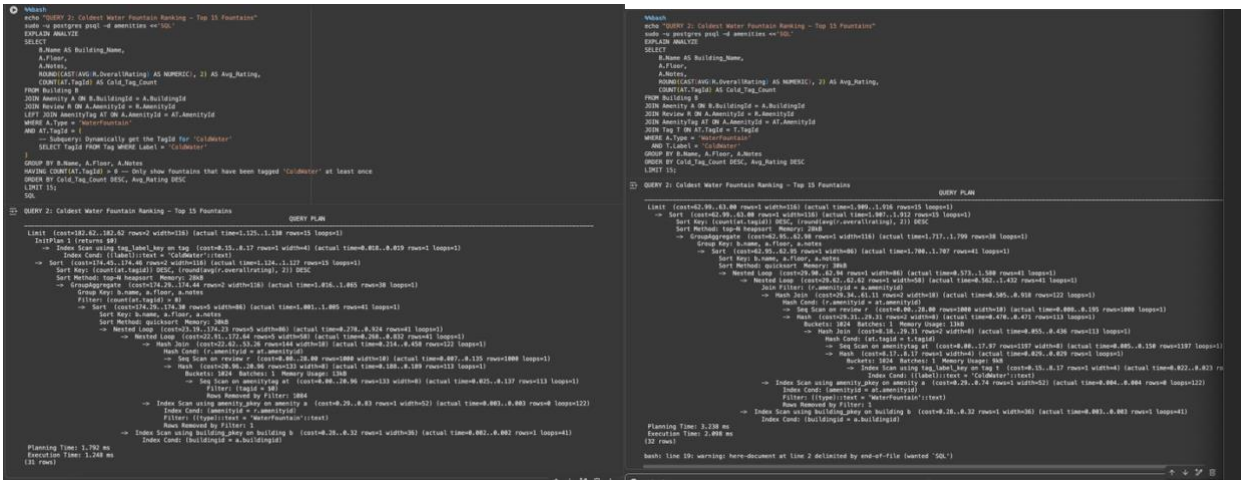
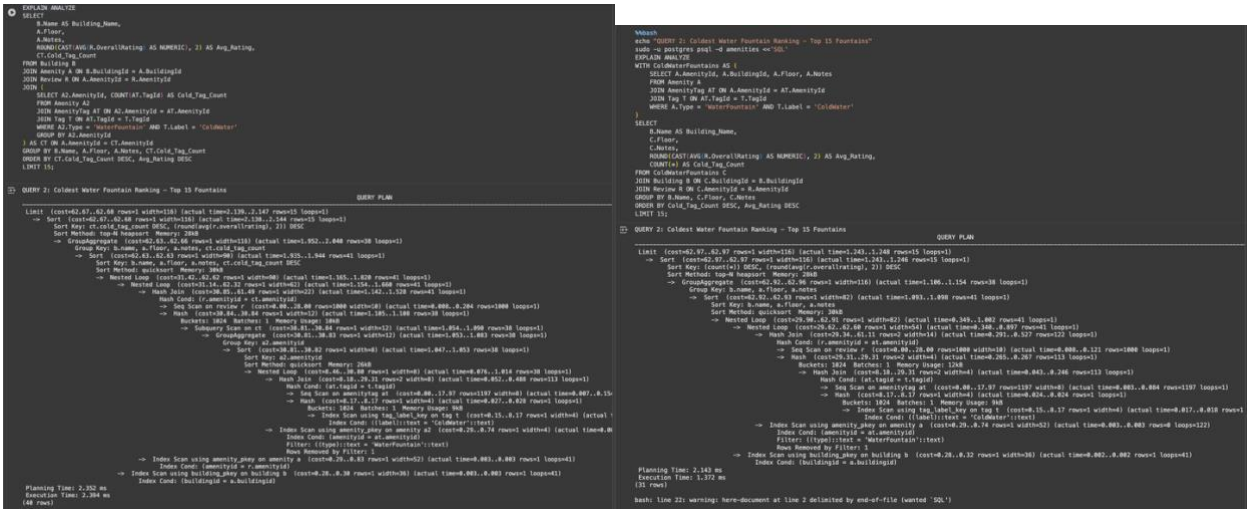




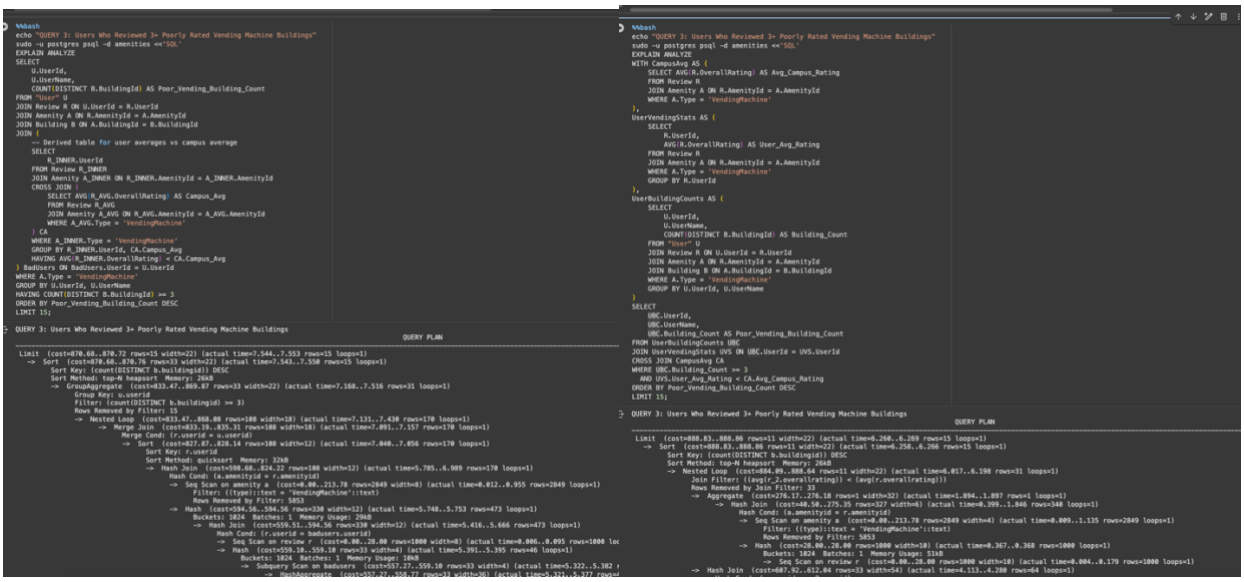


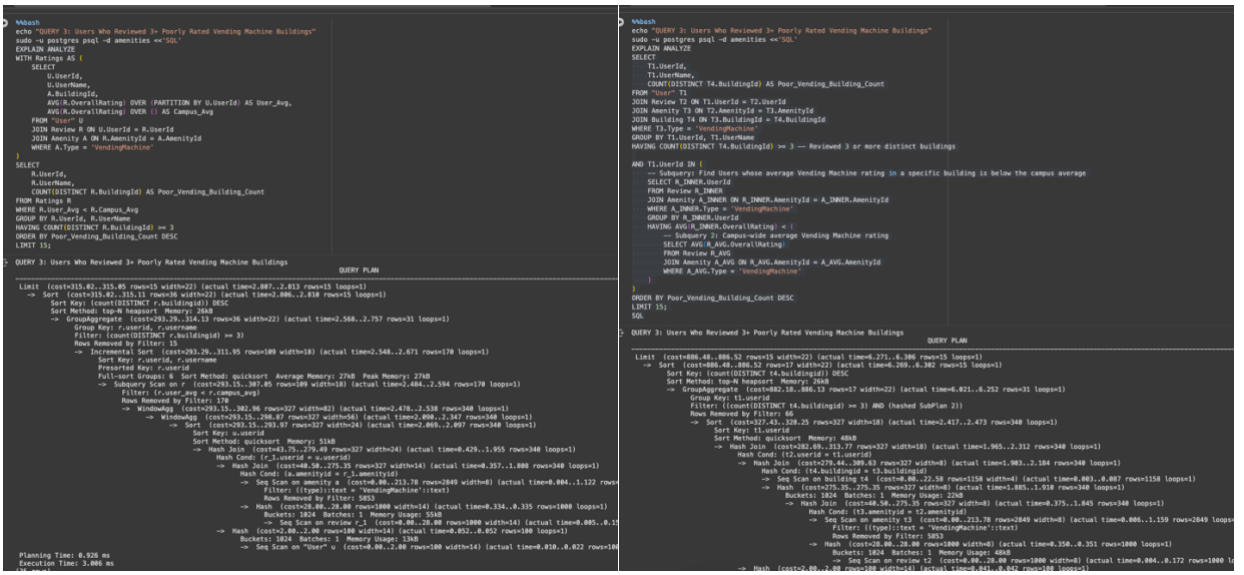
Query 1 returns the buildings with the top-rated buildings and vending machines. The original query uses hash joins and aggregation making it the fastest query, costing 594. The first alternate query relies on direct joins and filters by amenity type, it costs 611 and is sligltly slower than the original. The second alternate query uses a GROUP BY … HAVING, it costs 893 which is far too high as it uses nested loops and group aggregations. The third query uses pre-aggregated bathroom ratings then joins with vending machine data, this only costs 599 because it reduces intermediate join sizes. We will be using the original query as it is the fastest and most resource efficient.



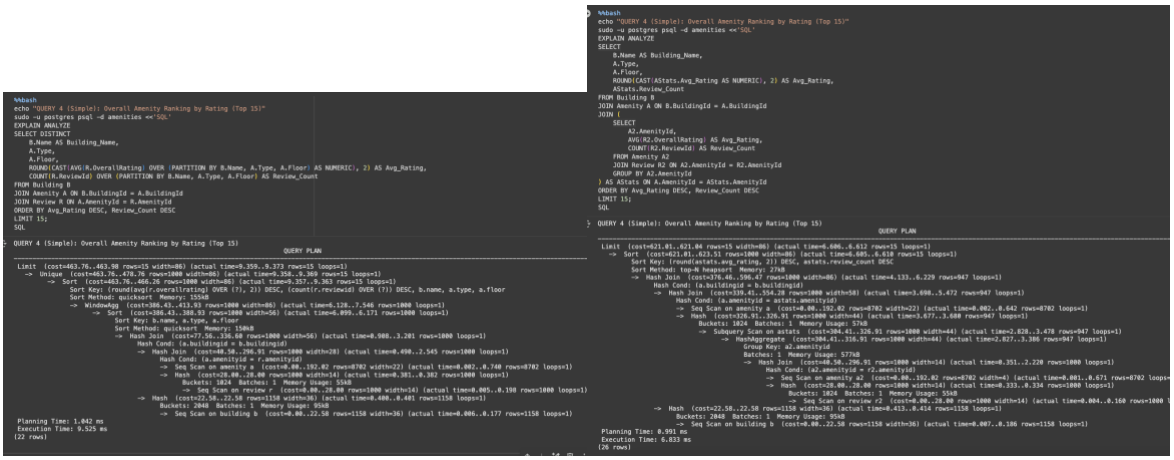


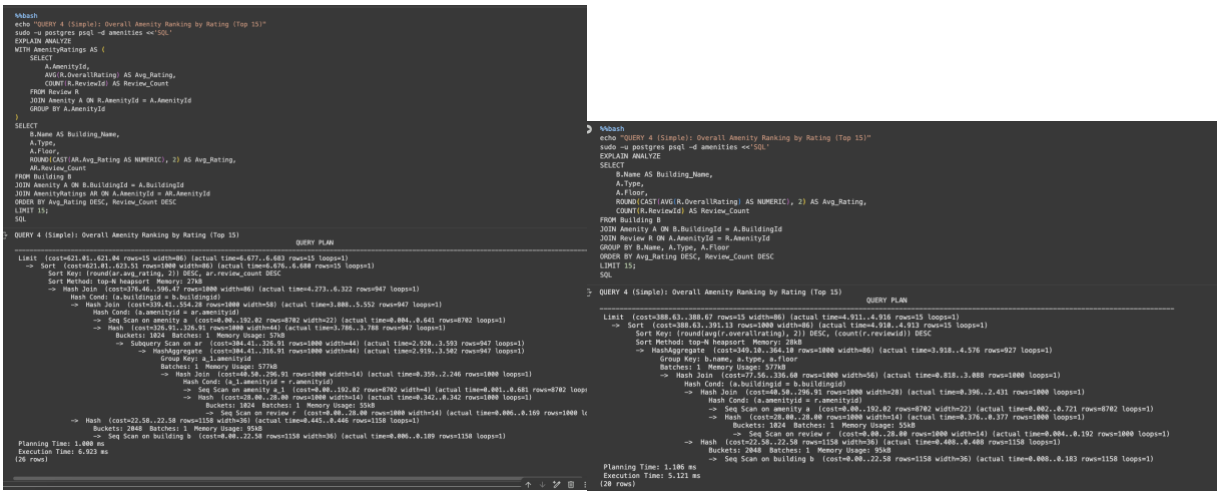
Query 2 returns the coldest water fountains by the building they are located in. The original query uses a direct join and filters by the “ColdWater” tag it is the most efficient as it costs 182. The first alternate pre joins the tag table and uses hash joins, it is much faster, as it only costs 62 and uses fewer loops and resources. The second alternative subquery uses a common table expression for reusability, it also costs 62, however it is more logically complex than the first alternate. The third alternate subquery uses aggregate tags to precompute fountains with the Cold-Water tag and reduces join volume significantly. It is the most efficient query as it has a cost of 62, so we decided this will be the query going forward.





Query 3 returns all users who reviewed at least 3 poorly rated vending machine buildings. The original code uses a subquery to find these users by comparing their vending machine ratings to the university average and has a cost of 886. The first alternate subquery uses a common table expression to join the items; it costs 888 and is more modular but slower and less resource efficient. The second alternate subquery uses a derived table instead of the subquery, it also replaces IN with a join to optimize. It costs 870, which is slightly faster than 886 but the code is much more complex. The third alternate function uses a window function instead of a subquery. It only costs 315 as it can find the campus average and summation of a user's bad reviews in a single pass, this is by far the most efficient version and will be used going forward





Query 4 returns the overall amenity ranking. It is an important query as it is a key feature for our project. The original query uses only groups and joins and costs 388. The first alternate query uses common table expressions. Because the original query is so simple, it actually slows down the query a lot to use a CTE, as this function costs 621. The second alternate query uses a derived table and joins on AmenityID. It is also way slower and more costly, costing 621, most likely due to the extra overhead of making the derived table. The third alternate query uses a window function, this function is also much slower than the original, costing 463. Because the original function is the simplest and costs the least, it will be our implementation going forward.