Question 1ev: Explain your answer to the previous part based on your knowledge from lectures, and details from the query plans (your explanation should include why you didn't choose certain options).

Views are different from Materialized Views in that they are calculated on demand, making their use in subsequent queries longer if not the same, they do not reduce execution time or cost because the views are calculated on demand every time the view is called. However, materialized views will reduce the cost and execution of a query because materialized views are only calculated once and are pretty much pre computed when calling these views in subsequent queries which reduces the cost and execution of these. However, since Materialized Views actually stores the data, these take more time to create compared to typical views.

Question 2d: Explain your answer based on your knowledge from lectures, and details from the query plans (your explanation should include why you didn't choose certain options).

Filtering data means that there is less data for the system to go through. Since there was only 8 rows that fit the criteria of > 2010, there was fewer rows to be dealt with when calling the view since the filter got applied when the view was created on demand because of predicate pushdown. Since filtering reduced the data so much, we also didn't need to do a hash join and nested loop join was both faster and less costly.

0.0.1 Question 3d

Please discuss your findings in the previous parts. In particular, we are interested in hearing why you think the query optimizer chooses the ultimate join approach in each of the above three scenarios. Feel free to discuss the pros and cons of each join approach as well.

If you feel stuck, here are some things to consider: Does a non-equijoin constrain us to certain join approaches? What's an added benefit in regards to the output of merge join?

In 3a, when performing an inner join, the query optimizer chose hash join because the joined tables are not sorted (making merge sort not an option) and given that there are a lot of rows to go through, the nested loop would be too CPU intensive.

In 3b, when performing an inner join and then sorting, the query optimer used merge join because merge join returns data in sorted order which has been specified by the order by statement, but also requires all input to be sorted by the join columns. We have done this on each table by sorting on playerid. In this case merge sort would reduce the time it would take to join using another method and then sort through each row by playerid because it would only have to go through each table once.

In 3d when performing a cross join to get every single combination, the optimizer chose a nested loop join since we start with one value of our outer table and iterate over every value of our inner table comparing values each time. In nested loop join, we compare every row in the first table with every row in the second table which is necessary for every single possible combinations of two players as we are essentially doing a cross product. We are also limiting the rows to 1000 which reduces the strain that this nested loop join would have on CPU making it a good option for this problem.

Question 4dii: Explain your answer based on your knowledge from lectures, and details from inspecting the query plans (your explanation should include why you didn't choose certain options).

Adding indexes helps the system look for data matching certain characteristics. Adding an index to the g_batting column decreased cost and time because we were looking for tuples that had g_batting = 10, so it sped that process up using an index scan method. Indexes can also speed up joins but in this case, since we were indexing on the salary column, there was no change to cost or execution because we were not joining on the salary column as it was exclusive to the salaries table the query plan has no use for that index.

Question 4evii Explain your answer to the previous part based on your knowledge from lectures, and details of the query plans (your explanation should include why you didn't choose certain options).

Using indexes with an AND predicate like we did 4eii reduced both cost and time because in an AND predicate we can essentially only search through one column and remove tuples that don't hold (because by AND they shouldn't be returned), which reduces the amount of filtering the system has to do. Then the filtering on g_batting is also faster due to the Bitmap Heap Scan searching which limits the which is faster than sequential search over all the tuples.

Using indexes on one column used in OR statements did not reduce time or cost because even though we can find tuples where g_batting = 10 using the index that we created for that specific column, we would still need to go through all the tuples again for the second column using sequential search for each column because even if g_batting was false, the g_all could be true. So then using index methods here would not be beneficial since we can't actually use any of the filtered true rows and the query plan just sticks to sequential search.

Using indexes on both columns reduced both the query time and cost when used in OR statements because, despite this being an OR predicate, having two indexes means that we can look at only the indexes . We still have to search every single tuple but if we have a multicolumn index we can still narrow down the search range that corressponds to only those two indexes rather than searching through all the data pages. The query plan shows that there is a reduction in cost and execution time, but it is not as significant as using an AND predicate.

0.0.2 Question 5d:

Explain your answer to the previous part based on your knowledge from lectures, and your inspection of the query plans.

Looking at the query plans, we can see that using AVG or COUNT doesn't improve performance, and even with indexing these use the same query plan. However, looking at the MIN (or MAX) aggregation, this reduces both cost and execution. Count and Average are slower aggregation functions because they both must use a sequential scan as they must read every single tuple before their output is ready. This means that indexing COUNT or AVG columns does not improve any performance because we would still need to go through every tuple which defeats the purpose of having an index, and therefore we would still use the sequential scan.

On the other hand, MIN or MAX are good functions to use with an indexed column because these queries can make use of index specific scans as they are only looking for single values. In these cases using the index would allow SQL to simply navigate to the first OR last leaf block in the index structure.

0.0.3 Question 6e:

Explain your answer to the previous part based on your knowledge from lectures, and your inspection of the query plans (your explanation should include why you didn't choose certain options).

Clustering based on the batting_pkey didn't have much of an effect on the execution time or cost of the query because since it the batting_pkey is a primary key, and therefore all unique, there was really not any added benefit (or detriment) to doing this. The number of clusters was equivalent to the number of batting_pkey's, additionally, the where clause is not using the primary key and therefore adding that specific clustering did not change anything about the query plan.

Clustering and indexing on ab from the batting table makes it such that instead of having to go through all 10k rows of the batting table, we only have to go through each value that ab can be (which is probably much smaller than 0-10k). Then we can use the index structure to optimize the where statement for ab > 500, which decreased both cost and time.

0.0.4 Question 7c:

What difference did you notice when you added an index into the salaries table and re-timed the update? Why do you think it happened?

After adding an index into the salaries table, the total time actually increased. It likely happened because updating the table by inserting many rows means that we needed to update the indexes as well. The database has to add each new entry to each index. So because 7b had an index on salary, this query took a little longer to run.

0.1 Question 8: Takeaway

In this project, we explored how the database system optimizes query execution and how users can futher tune the performance of their queries.

Familiarizing yourself with these optimization and tuning methods will make you a better data engineer. In this question, we'll ask you to recall and summarize these concepts. Who knows? Maybe one day it will help you during an interview or on a project.

In the following answer cell, 1. Name 3 methods you learned in this project. The method can be either the optimization done by the database system, or the fine tuning done by the user. 2. For each method, summarize how and why it can optimize query performance. Feel free to discuss any drawbacks, if applicable.

1. Method 1: Adding an index to typically non mutable where claused attributes. This can optimize query performance because indexing makes attributes faster to be filtered and queried by creating pointers that point to where the data is stored. Indexing also allows for efficient search methods such as B+ trees which are usually more cost effective and much faster than the base sequential search, especially for large databases. However, indexing should probably be done on attributes that are not usually updated because of the time it takes to update the indexes.
2. Method 2: Pre filtering data can help significantly optimize query performance. This comes from the general idea that the less data the query has to work with, the less time and cost it will take to run. Pre filtering the data can lower the amount of data the system needs to run in a more intensive clause such as a join or group by.
3. Method 3: Merge join method (performed by the database system) optimizes query performance by first sorting the join attributes, the creating sorted runs of blocks, and then running a single pass through both sorted runs and merges. Additionally, merge join merges in sorted order which can be helpful. Since merge join only has to do a single pass through each table it is generally cheaper and more time efficient to do than nested loop or hash joins unless sorting hasn't already happened and the sorting would be expensive.