1. During the lecture, we’ve mainly discussed the inner join algorithms. However, in practice, we have (left, right, or full) outer joins as well. Consider two tables L and R. L left outer join R states that the result table must contain all rows from L, and if a row in L does not have a matching row in R, we will output that row with the corresponding part in R padded with NULL. Discuss how to extend block nested loop join and GRACE hash join to support left outer joins efficiently.

**HINT:** be careful about the asymmetricity of join processing.

2. Suppose you have a table R with |R| pages and a table S with |S| pages, and 4 < |R| < |S|. We want to join the relations, and neither R nor S is ordered by the join key. We have the following assumptions:

- The GRACE hash join will recursively partition the input branch for “join build” until the hash table of single partition can fit in memory to be used by the in memory hash join algorithm.
- Hash join can automatically use the smaller input branch for “join build”, while the nested loop join can automatically use the smaller input branch as the outer loop data.

a. Assuming there are in total four (4) memory buffer pages, how many disk I/Os are needed to join R and S by the block nested-loop join algorithm?

b. If you want join R and S by the GRACE hash join with only one partitioning pass, what is the minimum number of memory pages needed for the join algorithm? Briefly explain your answers.

c. Assuming there are four (4) memory buffer pages, we want to join R and S by the GRACE hash join algorithm using multiple recursive partitioning passes. How many I/Os are needed?
Solution
1. In general, to support $L$ left outer join $R$, we need to keep track of whether each row in $L$ finds a match in $R$ or not. For all rows in $L$ which do not have a match in $R$, we need to output these rows with the part of $R$ padded with NULLs.

For block-nested loop join, if $L$ is the outer loop table, we can simply keep track of whether each row in the current block of $L$ has found a match. After all rows in $R$ have been iterated, we output rows in the current block of $L$ which does not have a match. This won’t increase the original I/O cost.

If $L$ is the inner loop table, things are more trickier since after each pass of $L$, we have to materialize the indicators of the $L$ table (assuming $L$ does not fit into memory). At the end of the join processing, we need to scan the $L$ table again to produce additional rows.

For grace hash join, if $L$ is the join build table (i.e., building the hash table), then we attach each row in the hash table with an additional indicator, which indicates whether there is an match in $R$ or not. At the end of join processing, we simply scan the hash table to output rows with indicator being false.

If $L$ is the join probe table, then after each probe, if it does not find a match, we immediately output that row.

2.

a. We use one page for reading an $S$ page, one page for writing join results, and the remaining two pages to read $R$ pages. So the number of iterations is $|R|/2$. The total number of disk I/Os is $|R| + (|R|/2) * |S|$.

b. $|R|/(M-1) + 2 <= M$
$R <= (M-1)(M-2)$
$M > 1 + \sqrt{R}$

c. Suppose we need $k$ partitioning pass, we use one page as the input and three pages for three output partitions. So after $k$ passes, the size of each $R$ partition is $|R|/3^k$, which should be within 2 for the final build-probe phase. In the last phase, we need 2 pages for an $R$ partition, one page for reading an $S$ partition, and one page for the join output. So we have: $|R|/3^k <= 2.$ The number of iterations is: $k >= \log_3(|R|/2)$.

The total number of disk I/Os is: $(k * 2 + 1) * (|R| + |S|)$. 