Why’s it blank?

Nobody knows the answers D: LoL

😊

(͒˃̩̩⌂˂̩̩ ͒): (ᵕ̣̣̣̣̣̣﹏ᵕ̣̣̣̣̣̣)

(͒˃̩̩⌂˂̩̩ ͒):

Good luck everyobdy 😀

1.a)

This optimization pushed down the selection through the projection. Selections generally tend to reduce the cardinality of tuples flowing out, which leaves less work for the operators above to complete.

* In By-reference bulk, it would likely reduce the page faults as the size for the intermediate structure would be smaller, so more of it would fit in the cache, and we’d waste fewer resources retrieving the structure from memory.  
  However, this assumes the general case. If all attributes were larger than 9, then this would have no effect on the performance.   
  Once our tuple reaches the grouping operator, the optimization has no benefit.
* In Volcano, we send individual tuples up through the system.  
  Since we have no information about the schema, it is possible that we have, for example, 20 attributes per tuple. Therefore, it might be less advantageous to send the selection through first, because we’d be more likely to exceed the page size per each tuple, so perhaps the projection first would have been better.  
  When it comes to the grouping operator, there is no difference.

1.b.i)

Assume a volcano style processing system:

# of tuples in R = 10^8 tuples

# of tuples in S = 10^7 tuples

# of tuples in T = 10^6 tuples

Size of R = 1.25 \* 10^8 pages

Size of S = 1.25 \* 10^7 pages

Size of T = 6.25 \* 10^4 pages

Pool size = 2.62144 \* 10^6 pages

Plan 1:

* Scan T: 6.25 \* 10^4 page faults
* Hash table needs to store 10^6 tuples, so over allocate by a factor of 2 for 1.25 \* 10^5 pages. This fits in the cache so no page faults during probing or building.
* Scan S: 1.25 \* 10^7 page faults
* We have 10^6 output tuples based on the assumption. These flow into the next join which is a pipeline breaker, and we have to build our table that way. We magically know our size, and we need to store 4B + 4B = 8B for the output tuple (we’ve joined T with S so in product we’ll have (T.b, S.a). We assume this instead of the cross product as illustrated in the query.   
  Therefore, our hash table must store 10^6 tuples at 8 B per, so 8 \* 10^6 B / 64 B = 125000 pages. This fits in the buffer pool so no page faults during building or probing
* Scan R: 1.25 \* 10^8
* Volcano produces tuples with no I/O at the top.

Sum = 137,562,500 page faults

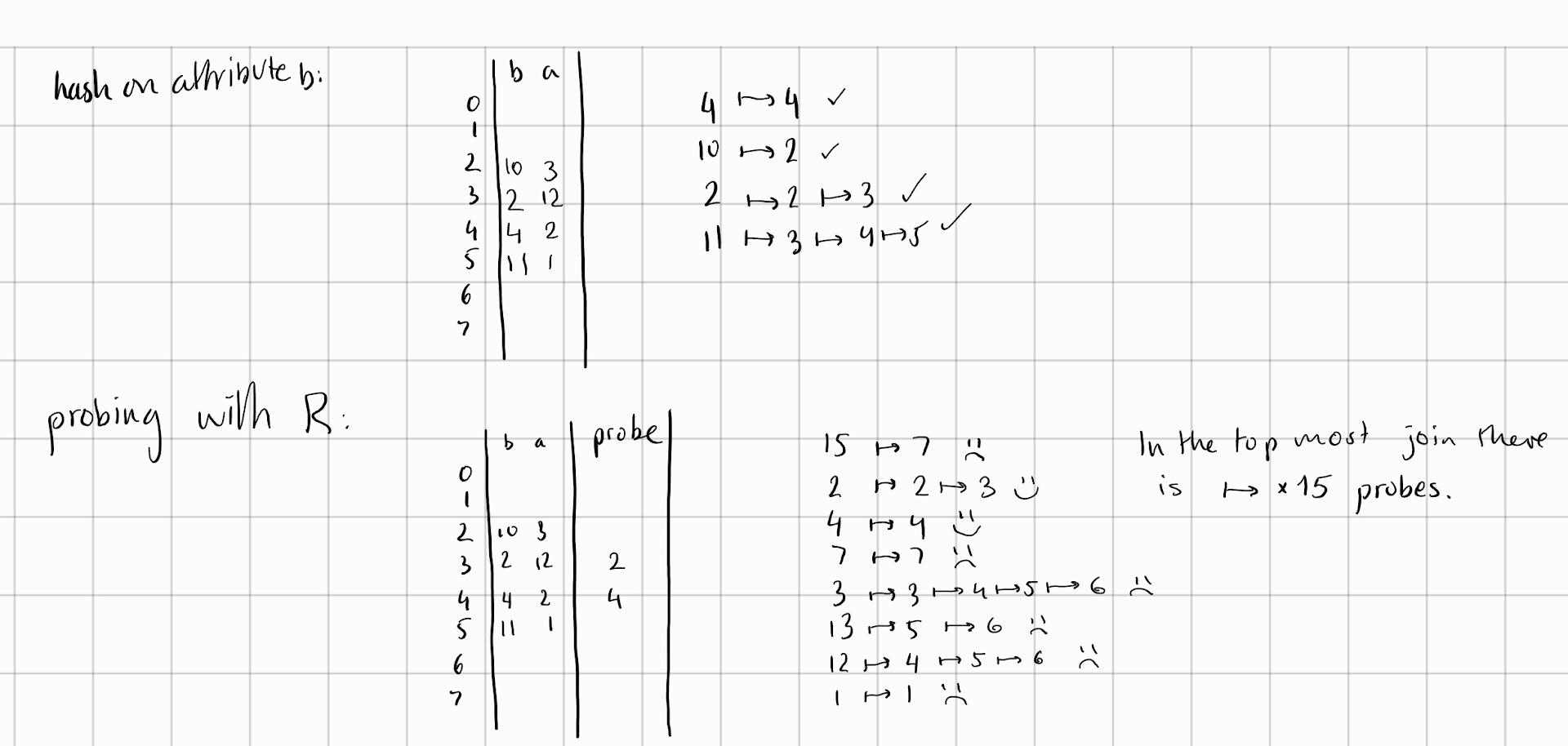
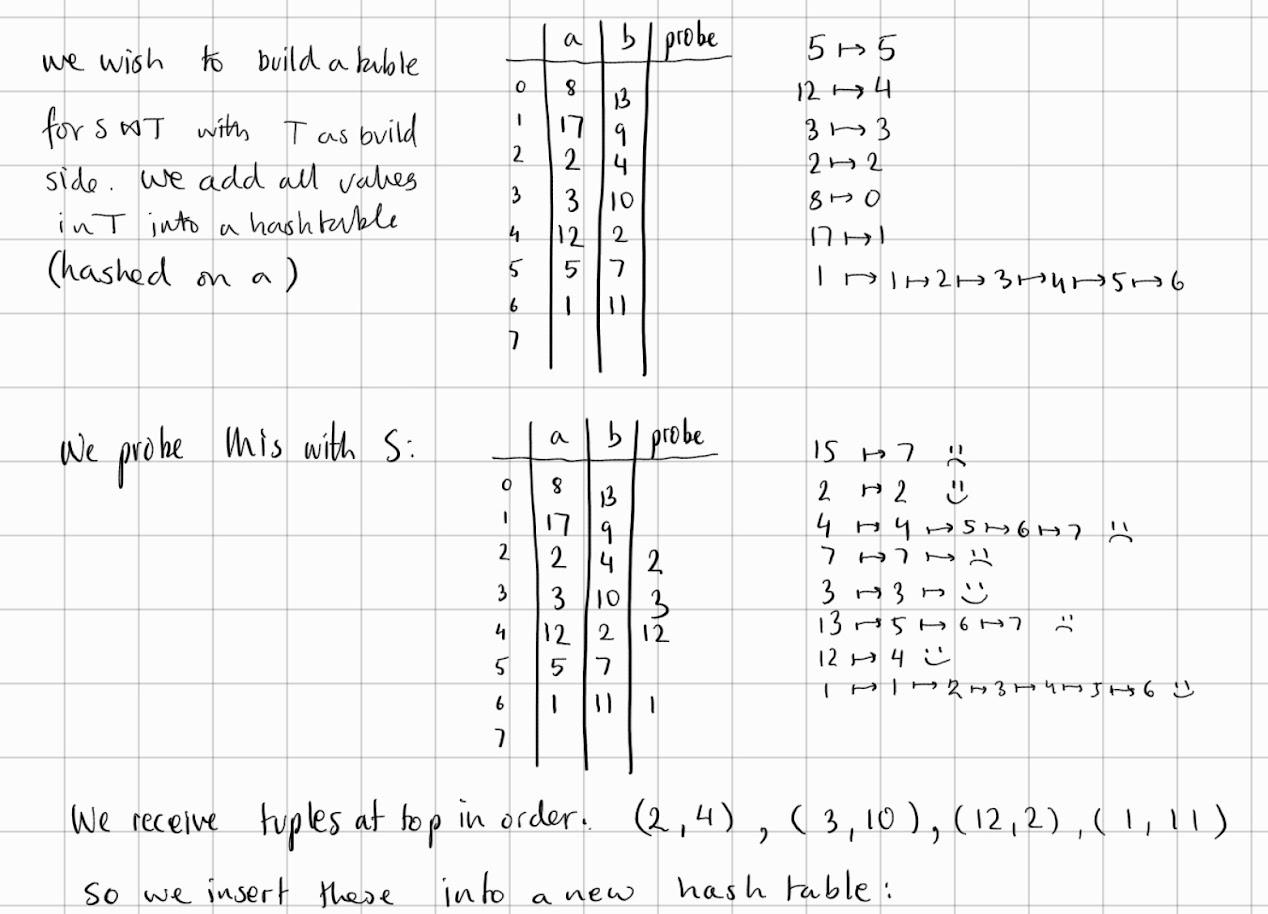
Plan 2:

* Scan R: 1.25 \* 10^8 page faults
* Since in the assumptions we bulid on the RHS, we should build a hashtable for S. Therefore, build a table that can store 10^7 tuples at 8B each. Therefore we overallocate by 2, and get 10^7 tuples \* 8B \* 2 overallocation / 64 B per page = 2500000 = 2.5 \* 10^6  
  This fits in our buffer pool, so there is no cost for each build and probe.
* Scan S: 1.25 \* 10^7 page faults
* Similarly, we build a hash table for T. At 6.25 \* 10^4 pages, we overallocate by 2 to get some number (idk maths) but it fits in the buffer, therefore there is no I/O associated with building and probing.
* Scan T: 6.25 \* 10^4 pages

Sum = 137,562,500 page faults

Both of these are as good as each other.

1.b.ii)



7 probes to build the final table, 15 probes to probe the table.

c.i)

1. Small Tables: If one of the tables being joined is very small, nested loop join can be the most efficient option. In this case, the smaller table is read into memory and then looped over for each row in the larger table. Since the smaller table fits entirely in memory, there is no need for expensive disk I/O operations. This makes nested loop join very fast for small tables.
2. Low Selectivity Joins: If the join condition is not very selective, meaning that it matches a large proportion of the rows in each table, nested loop join can be the most appropriate option. In this case, the join condition is evaluated for each row in the inner table for each row in the outer table. While this can be computationally expensive, it is still faster than other join algorithms such as sort-merge join or hash join, which require expensive sorting or hashing operations.
3. Non-Equi Joins: If the join condition is a non-equi join, meaning that it cannot be expressed as a simple equality comparison, nested loop join may be the only option. Non-equi joins are not supported by all join algorithms, as they cannot be expressed as a simple lookup operation. In this case, nested loop join can be used to loop over all possible combinations of rows in the two tables and evaluate the non-equi join condition for each pair.
4. Parallelizable: Since there are no dependent loop iterations, this task is very parallelizable so we can employ a number of threads to speed up the join.
5. Sequential I/O: Better locality

c.ii)

Yes, a similar optimization can be applied to sort-merge joins. The optimization is called partitioned sort-merge join, and it works by partitioning the input tables into smaller subsets, sorting each subset independently, and then merging the sorted subsets together using the standard sort-merge join algorithm.

The implementation of partitioned sort-merge join involves the following steps:

1. Partitioning: Partition both input tables into N subsets each. The partitioning can be based on a hash function or range partitioning.
2. Sorting: Sort each subset independently using a standard sorting algorithm. This can be done in parallel, which can provide a significant performance improvement.
3. Merging: Merge the sorted subsets together using the standard sort-merge join algorithm, except that instead of merging the entire tables, only the corresponding subsets are merged.

The benefits of partitioned sort-merge join are similar to those of partitioned hash-join. By breaking the input tables into smaller subsets, the sort-merge join can take advantage of parallelism and reduce the amount of data that needs to be sorted and merged at any one time. This can lead to significant performance improvements, especially when dealing with very large input tables.

However, partitioned sort-merge join has some limitations. First, it requires additional disk space to store the sorted subsets, which can be a problem if disk space is limited. Second, it may not be as effective as partitioned hash-join for highly skewed data distributions, as the data may not be evenly distributed among the subsets. Finally, the additional overhead of partitioning and sorting the data can outweigh the performance benefits in certain cases.

d)

* B-Tree Index: B-Tree is a balanced tree structure that allows efficient searching, insertion, and deletion of data. B-Tree indexes are commonly used for range queries and sorting, but they can also be used for supporting aggregation. B-Tree indexes can be used to efficiently group data based on one or more columns, but they are less efficient than hash-based indexes for grouping.
* Hash Index: Hash indexes are commonly used for joins, but they can also be used for supporting aggregation. Hash indexes can be used to efficiently group data based on one or more columns, but they are less efficient than B-Tree indexes for range queries and sorting.
* Bitmap Index: Bitmap indexes are a type of index that store a bitmap for each distinct value of a column. Bitmap indexes can be used to efficiently group data based on one or more columns, and they are especially useful for low-cardinality columns (columns with few distinct values). Bitmap indexes are not as efficient as B-Tree or hash indexes for range queries and sorting, but they can be very efficient for aggregation queries.