Database Design and Implementation
Chapter 13 Conceptual Exercises

Questions	Answers
13.1.a.	If there is only one student in the class of 2005, the right-hand materialize node may not be particularly beneficial because the operation will not handle a large set of records. Materialization is useful when operations are expensive and involve large datasets where intermediate results are reused. If only one record qualifies the criteria, the overhead of materializing may outweigh its benefits.
13.1.b.	With two students in the class of 2005, the benefit of the right-hand materialize node starts to increase slightly, but it is still minimal due to the small dataset. The right-hand materialize node becomes more valuable as the number of records increases, as it can help avoid repeated computation on the dataset for subsequent operations.
13.1.c.	Swapping the right and left subtrees means we would be materializing the result of the **Select:GradYear=2005** operation. To calculate the savings, we would compare the cost of computing the select operation on-the-fly each time the subtree is accessed against the cost of accessing the materialized result. Savings can be quantified in terms of disk I/O operations, CPU cycles, or query time. If the number of students from 2005 is significantly less than the total number of records, this could save on computation time by avoiding full table scans on subsequent accesses.
13.2.a.	The final run produced by sequential merging will always require the same number of merges because each merge takes two runs and produces one. Whether merging iteratively or sequentially, the number of merges does not change as it depends on the total number of initial runs.
13.2.b.	Sequential merging requires more block accesses because each new merge involves a run that is potentially larger than in the iterative approach. In the iterative approach, each merge combines runs of approximately equal size, minimizing the number of block transfers. In sequential merging, runs become progressively larger, which means more block accesses are required as the runs grow.
13.3.a.	If the input records are already sorted, the algorithm in Fig. 13.6 will produce the fewest initial runs since it will be able to take advantage of the existing order to produce single, long runs instead of multiple one-block runs.
13.3.b.	If the input records are sorted in reverse order, both algorithms will end up producing the same number of initial runs. This is because the sorting criteria for both algorithms (ascending order) is the reverse of the input, so they will both start a new run for each record.
13.4.a.	The cost of sorting each table using 2, 10, or 100 auxiliary tables will depend on the number of records in the table (R(T)) and the number of blocks (B(T)). Sorting with more auxiliary tables generally reduces the number of passes required through the data, but this needs to be balanced against the overhead of managing more runs and the complexity of the merge operation.
13.4.b.	For joining tables, the cost of performing a merge join would similarly depend on the sizes of the tables and the number of auxiliary tables. Joining larger tables or

	those with more records will be more expensive in terms of I/O operations. The key is to perform joins in such a way as to minimize the number of records that need to be compared, which is where the statistics in Fig. 7.8 would be essential to estimate costs.
13.5.a.	If the database is very large, the list of **TempTable** objects could become a source of inefficiency due to memory consumption. If too many objects are created, it could lead to memory overflow or thrashing as the system tries to manage the available memory.
13.5.b.	A better solution could be to use a streaming approach where **TempTable** objects are not all stored in memory at once but are processed in a pipeline fashion. Alternatively, one could use a database cursor to iterate through the records, which allows the program to handle large datasets without keeping everything in memory.