Database Design and Implementation

Chapter 15 Conceptual Exercises

Questions	Answers
15.1.	To show that the product operator is associative, we need to show that for any three tables *A*, *B*, and *C*, the following holds: $A\times B)\times C = A\times(B\times C)$ Let's take tuples$a\in A$, $b\in B$, and $c\in C$. The product operation ×× combines tuples from two tables to form all possible combinations where each combination consists of the attributes of both tables. • In $(A\times B)\times C$, first *A* is combined with *B* to form a set of tuples where each tuple is an element of *A* followed by an element of *B*. Let's denote a tuple from $A\times B$ as *ab*. Then each tuple *ab* is combined with each tuple in *C* to form $(ab)c$. • In $A\times(B\times C)$, first *B* is combined with *C* to form a set of tuples where each tuple is an element of *B* followed by an element of *C*. Let's denote a tuple from $B\times C$ as *bc*. Then each tuple �*a* from *A* is combined with each tuple *bc* to form $a(bc)$. In both cases, the result is a set of tuples where each tuple contains all the attributes of *A*, *B*, and *C*. The order in which the combinations are formed does not matter; the resulting set of tuples will be the same. Thus, the product operation is associative.
15.2.	Consider a query that takes the product of several tables, $T1,T2,...,Tn$. There are many ways to construct a query tree that represents the product of these tables due to the associativity of the product operator. To transform one tree into another, we can use the associativity and commutativity properties of the product operator: • **Associativity**: The product operator is associative. Therefore, we can regroup the operations without changing the outcome. This allows us to move parentheses in the query tree. • **Commutativity**: The product operator is also commutative, meaning that $A\times B=B\times A$. This allows us to swap the order of operations at any level of the tree. Given any two equivalent query trees for the same product query, we can apply these equivalences to transform one tree into another step by step: 1. Use commutativity to reorder the tables in one of the trees to match the order in which they appear at the leaves of the other tree.

	2. Apply associativity to rearrange the parentheses so that the grouping of tables in the product operations matches between the two trees. Since these operations are equivalences, they preserve the semantics of the query, ensuring that the transformed tree is still a valid representation of the original query. As such, any two query trees can be transformed into each other using a series of these two basic operations.
15.3.a.	1. Apply a commutative transformation to swap the positions of COURSE and SECTION. 2. Apply a series of associative transformations to rearrange the nodes so that DEPT is at the top, followed by SECTION and COURSE on the right subtree.
15.3.b.	1. Start with the COURSE node. 2. Apply an associative transformation to make SECTION the left child of the COURSE node. 3. Repeat the process to make ENROLL the left child of the SECTION node. 4. Then, make STUDENT the left child of the ENROLL node. 5. Finally, make DEPT the left child of the STUDENT node, resulting in a left-deep tree.
15.4.a.	Notice that selection operations are commutative. As long as the selection predicates do not depend on each other, you can switch their order without affecting the result set.
15.4.b.	A select node can be moved above a project node if the attributes that the select predicate operates on are still present in the projection. In other words, the selection does not depend on any of the attributes that are being projected out.
15.4.c.	A select node can be moved above or below a groupby node if the selection predicate applies to attributes that are not being aggregated. If the predicate is on the grouping attributes or on attributes not affected by the aggregation, the selection can be pushed down to potentially reduce the number of tuples being grouped.
15.5.a.	The union operator is associative and commutative: • Associative: (R ∪ S) ∪ T = R ∪ (S ∪ T) • Commutative: R ∪ S = S ∪ R
15.5.b.	A selection can be pushed inside a union if it applies to both underlying tables. For example, σ_p(R ∪ S) can be transformed into (σ_p(R)) ∪ (σ_p(S)).
15.6.a.	Anti-join and semi-join operators are not generally associative or commutative. However, under certain conditions, like when dealing with subsets of data where the relationships are known to be associative, they might exhibit associative behavior. Commutativity might be seen in a semijoin under certain conditions as well.
15.6.b.	Selections can be pushed inside of an anti-join or semi-join if the selection predicates are relevant to the tables being joined and do not depend on the computation of the antijoin or semijoin itself.

15.7.	This reduces the number of tuples being joined, which can significantly improve query performance.
15.8.	The cost of a query plan is determined by the database's query optimizer, which estimates the resources needed for various operations like I/O, CPU, and memory. Two equivalent query trees could have different costs based on the order of join operations, the use of indexes, or the sizes of the intermediate result sets. Creating such trees typically involves understanding the specifics of the database schema, data distribution, and the cost functions used by the optimizer. However, a common approach is to compare a tree that performs selections early (thus reducing the size of intermediate results) to one that performs selections late. Despite being equivalent in result, the former can often be much less expensive to execute.
15.9.	The "bushy tree" structure could be considered most promising based on certain heuristics such as: 1. Selections are pushed down: This is beneficial because it filters the rows as early as possible, reducing the number of rows that need to be joined in later steps. 2. Joining smaller sets first: If **ENROLL** and **SECTION** tables after selection have fewer rows, joining them before the larger **STUDENT** and **COURSE** tables can reduce the overall cost of the query. These heuristics are commonly used to minimize I/O, CPU time, and memory usage in query execution plans. Without specific cost metrics, we base the evaluation on these principles. The tree's efficiency also depends on the indexes available and the cardinality of the tables involved.
15.10.	To list other join orders for the query tree in Fig. 15.6c, without requiring Cartesian products, we would keep **DEPT** as the last join to maintain referential integrity and avoid products. This gives us the following combinations with **DEPT** at the end. (STUDENT, ENROLL, COURSE, SECTION, DEPT) (STUDENT, SECTION, ENROLL, COURSE, DEPT) (STUDENT, SECTION, COURSE, ENROLL, DEPT) (STUDENT, COURSE, ENROLL, SECTION, DEPT) (STUDENT, COURSE, SECTION, ENROLL, DEPT) (ENROLL, STUDENT, SECTION, COURSE, DEPT) (ENROLL, STUDENT, COURSE, SECTION, DEPT) (ENROLL, SECTION, STUDENT, COURSE, DEPT) (ENROLL, SECTION, COURSE, STUDENT, DEPT) (ENROLL, COURSE, STUDENT, SECTION, DEPT)

	(ENROLL, COURSE, SECTION, STUDENT, DEPT) (SECTION, STUDENT, ENROLL, COURSE, DEPT) (SECTION, STUDENT, COURSE, ENROLL, DEPT) (SECTION, ENROLL, STUDENT, COURSE, DEPT) (SECTION, ENROLL, COURSE, STUDENT, DEPT) (SECTION, COURSE, STUDENT, ENROLL, DEPT) (SECTION, COURSE, ENROLL, STUDENT, DEPT) This enumeration assumes that **STUDENT**, **ENROLL**, **SECTION**, and **COURSE** can be freely reordered, as long as the final join to **DEPT** is preserved.
15.11.	• The actual size of the data returned by selections. • The distribution of key values (which affects the performance of join operations). • The presence of indexes that can drastically change the cost dynamics. An example query could be one where Heuristic 4 does not adequately account for the benefits of using an index on one of the tables, leading to a less optimal join order.
15.12.a.	The cost of the two trees in Figure 15.6 would be calculated based on the sizes of the intermediate results and the number of I/O operations required.
15.12.b.	Heuristic algorithms might prioritize reducing the size of intermediate results early on.
15.12.c.	Dynamic programming would evaluate all possible plans to find the one with the lowest cost.
15.12.d.	The lowest-cost plan would be identified by comparing the costs calculated for all considered query trees, possibly using a cost formula that incorporates factors such as CPU time, I/O operations, and memory usage.
15.13.a.	Show that one join order has a lower cost by calculating the sizes of the intermediate results for both join orders and the number of I/O operations required for each join.
15.13.b.	Apply the heuristic algorithm using Heuristic 5a and then 5b.
15.13.c.	Dynamic programming algorithm to enumerate all possible join orders and calculate their costs to find the most promising one.
15.13.d.	After identifying the most promising tree, you would calculate the lowest-cost plan by applying the cost metrics used in your environment.
15.14.	The dynamic programming algorithm for join enumeration typically only considers left-deep join trees, where each join combines a new table with the result of all previous joins. To extend it to consider all possible trees, including right-deep and bushy trees, the algorithm must be modified to iterate over all subsets of joins and not just sequential additions. This would involve: 1. Enumerating all possible subsets of the relations to be joined.

	2. For each subset, calculating the best way to join the tables within that subset. 3. Combining these subsets in different ways to form the full set of tables. 4. For each combination, calculating the cost and keeping track of the minimum cost join order. This extended algorithm would be significantly more complex and computationally expensive, as the number of possible trees grows exponentially with the number of tables to join. However, it would provide a more thorough exploration of the search space for the most efficient query plan. For practical implementation, database systems often impose limits or use approximations to keep the computation of the join enumeration manageable.