Database Design and Implementation
Chapter 14 Conceptual Exercises

Questions	Answers
14.1.	Having an excess number of buffers in a database system can be advantageous beyond just preventing them from all being pinned at the same time. The additional buffers can accommodate larger portions of the database in memory, which can reduce disk I/O operations and increase the performance of query processing, particularly for operations that require scanning large amounts of data or complex joins.
14.2.	If a database system has more buffers than the number of blocks in the database, all the buffers can still be used effectively. The surplus buffers can be used to keep frequently accessed data in memory, store pre-computed query results, or hold temporary tables and indexes that can speed up query processing. However, after a certain point, additional buffers may not yield significant performance gains.
14.3.a.	If the entire database fits into memory, disk-based components like the buffer manager, which manages the pinning and unpinning of disk blocks, might become less critical or even unnecessary.
14.3.b.	Components like the query optimizer should function significantly differently, as the cost metrics would shift from I/O bound to CPU bound.
14.3.c.	A better function to model the cost of evaluating a query in a main-memory system could be the CPU time taken to process the query, which would consider factors like CPU cycles per operation, parallelism, and the complexity of operations.
14.4.a.	Having the **open** method determine **numbuffs** is less desirable because the number of buffers needed can change dynamically based on the current database load and the specifics of the data being processed. This decision is better made within each method that knows the context of its operation.
14.4.b.	Allocating buffers in the **SortPlan** constructor is even worse because it assumes that the buffer requirements will not change over the lifespan of the **SortPlan** object, which is not flexible and can lead to inefficient buffer usage.
14.5.a.	The number of buffers used in the scanning phase of a merge join would correspond to the number of blocks that can be simultaneously held in memory. In the context of the **ChunkScan** class, this would be the range of blocks between **startbnum** and **endbnum**. The exact number of buffers used would depend on the number of blocks spanned by the chunk.
14.5.b.	If only 100 buffers were available, the buffers would be allocated proportionally based on the size of the tables, with the majority going to the larger table, here assumed to be **ENROLL**.
14.5.c.	If buffers are allocated for **STUDENT** before **ENROLL**, then **STUDENT** would get the majority or all of the buffers if it is the larger table. If it's smaller, the remaining buffers would be allocated to **ENROLL**.
14.5.d	The cost of fully materializing either of the sorted tables would be the sum of the I/O operations to read the table into buffers, the CPU time to sort the table, and the I/O operations to write the sorted table back to disk if necessary.

Database Design and Implementation
Chapter 14 Conceptual Exercises

14.6.a.	The algorithm effectively partitions records based on their grouping fields into temporary tables (hash partitioning). This guarantees that all records with the same grouping field value are in the same partition, allowing for a group-by operation to be performed on each partition independently.
14.6.b.	The preprocessing cost includes hashing each record and writing it to the corresponding temporary table. The scanning cost involves reading each temporary table and performing the sort-based group-by operation on it. The exact costs would depend on the number of records, the number of group fields, and the costs of I/O operations.
14.6.c.	This algorithm may not be as good as a sort-based groupby (assuming one that works on the entire table at once) because it involves additional overhead from hashing and managing multiple temporary tables. The sort-based groupby can take advantage of sequential disk accesses and efficient in-memory sorting algorithms, while the hash-based approach can cause random I/O due to hashing.
14.6.d.	In a parallel-processing environment, the algorithm can distribute the load across multiple processors. Each processor can independently perform the group-by operation on a different partition, leading to potential performance gains due to parallelism.
14.7.a.	The number of block accesses required for the product would be the product of the number of blocks in T1 and T2 plus the number of blocks to read T1 once, assuming that T2 is read once per block of T1.
14.7.b.	This number is less than the basic product algorithm because the multibuffer product can keep a portion of T1 in memory, reducing the number of times T2 needs to be read from disk compared to a basic nested-loop join which would read T2 for each record of T1.
14.8.	Not materializing the LHS can cause buffer use problems because the LHS might not be fully available in memory, leading to repeated disk I/Os if the buffers are overwritten by RHS blocks. For efficiency, not materializing the LHS might mean that the join condition is evaluated repeatedly for tuples that have already been processed, resulting in computational overhead.
14.9.	The hash join algorithm can be rewritten to perform all hashing in a preprocessing stage and then perform merging in the scanning stage. This would involve creating a non-recursive loop that processes each partition and performs the join, likely using a hash table to find matching records from T1 and T2.
14.10.	Using different values of k for T1 and T2 would result in non-matching partitions for the same join field value, which means that records that should be joined might end up in different partitions, causing the join to fail.
14.11.a.	Choosing the value of k once and passing it into each recursive call would maintain consistency in the partitions, ensuring that matching records from T1 and T2 end up in the same partition.
14.11.b.	The trade-off is between the flexibility of adapting k to the size of partitions in recursive calls and the consistency of using a single k. Adapting k can optimize

	the size of partitions but can complicate the join logic. A consistent k simplifies the logic but might not be optimal for partition sizes. A consistent k is preferable for simplicity and predictability.
14.12.	Replacing the recursive hashjoin with mergejoin for joining individual buckets would alter the cost analysis significantly. Mergejoin is efficient for sorted buckets and would reduce the number of block accesses compared to a recursive hashjoin, which could potentially rehash records multiple times.

14.13.a.	**SELECT SName, DName FROM STUDENT, DEPT WHERE MajorId=DId**
	• **Mergejoin**: It would require sorting both tables on **MajorId** and **DId**, respectively, if they are not already sorted. The cost would be the sum of the sorting costs plus the cost of a single sequential scan of both tables.
	  • Sorting STUDENT: 45,000 log2(45,000) block accesses (assuming external sort).
	  • Sorting DEPT: Not needed as it only has 2 blocks.
	  • Sequential scan: 4,500 + 2 = 4,502 block accesses.
	• **Hashjoin**: It would require partitioning both tables into buckets based on **MajorId** and **DId**.
	  • STUDENT: 4,500 block accesses to read and partition.
	  • DEPT: 2 block accesses to read and partition.
	  • Joining: 4,500 + 2 = 4,502 block accesses to join partitions.
	• **Indexjoin**: If there's an index on STUDENT.MajorId, it can directly access the corresponding DEPT records.
	  • STUDENT: 4,500 block accesses to scan.
	  • DEPT: 40 distinct values, so at most 40 index lookups.
	  • Total: 4,500 + 40 = 4,540 block accesses.

14.13.b.	**SELECT SName, DName FROM STUDENT, DEPT WHERE MajorId=DId AND GradYear=2020**
	• **Mergejoin**: Similar calculation to (a), but only for STUDENT records with **GradYear=2020**.
	  • STUDENT: We have 50 distinct **GradYear** values, so 4,500 / 50 = 90 block accesses for sorting.
	  • DEPT: Not needed as it only has 2 blocks.
	  • Sequential scan: 90 + 2 = 92 block accesses.
	• **Hashjoin**: Similar to (a), but only for STUDENT records with **GradYear=2020**.
	  • STUDENT: 90 block accesses to read and partition.
	  • DEPT: 2 block accesses to read and partition.
	  • Joining: 90 + 2 = 92 block accesses to join partitions.
	• **Indexjoin**: If there's an index on STUDENT.MajorId and GradYear.
	  • STUDENT: 90 block accesses to scan.
	  • DEPT: 40 distinct values, so at most 40 index lookups.
	  • Total: 90 + 40 = 130 block accesses.

14.13.c.	**SELECT DName FROM STUDENT, DEPT WHERE MajorId=DId AND SId=1**

	• **Mergejoin**: Not efficient since we are looking for a specific **SId**.
	• **Hashjoin**: Not efficient for the same reason as mergejoin.
	• **Indexjoin**: If there's an index on STUDENT.SId, it would be the most efficient.
	  • STUDENT: 1 block access if SId=1 is in its own block.
	  • DEPT: Assuming index on DId, 1 block access.
	  • Total: 1 + 1 = 2 block accesses.

14.14.d.	**SELECT SName FROM STUDENT, ENROLL WHERE SId=StudentId AND Grade='F'**
	• **Mergejoin**: Not efficient for the same reason as (c).
	• **Hashjoin**: Not efficient for the same reason as (c).
	• **Indexjoin**: If there's an index on ENROLL.StudentId and STUDENT.SId.
	  • STUDENT: 4,500 block accesses to scan.
	  • ENROLL: 14 distinct values for **Grade**, so we can assume it's evenly distributed. Therefore, 1,500,000 / 14 ≈ 107,143 block accesses for 'F' grade.
	  • Total: 107,143 + 4,500 = 111,643 block accesses.