Database Design and Implementation
Chapter 12 Conceptual Exercises

Questions	Answers
12.1.	All fields that are not ID of the table (e.g. Grade) are not appropriate to be indexed on.
12.2.a.	• **MajorId** in STUDENT: Since it's used in a join condition with DEPT. • **DName** in DEPT: Since it is a search condition for 'math'.
12.2.b.	• **SectId** in ENROLL: Because it's matched with SectionId. • **CourseId** in COURSE: Since it's used in a join with the SECTION table. • **Grade** in ENROLL: Although there are only 14 unique values, the specific search for 'F' might make it beneficial. • **Title** in COURSE: To quickly find 'calculus'.
12.3.a.	Assuming students are evenly distributed over 50 years: • Index search would likely require 1 block access to find the index entry, plus the number of block accesses to retrieve the records. • With 45,000 records and 50 years, we would have 900 students per year. • The number of blocks per record is 45,000/4,500 = 10. • So, for 2020, we'd access 900/10 = 90 blocks.
12.3.b.	Changing the distribution to 2, 10, 20, or 100 years: • For 2 years: 22,500 records per year, 22,500/10 = 2,250 blocks. • For 10 years: 4,500 records per year, 4,500/10 = 450 blocks. • For 20 years: 2,250 records per year, 2,250/10 = 225 blocks. • For 100 years: 450 records per year, 450/10 = 45 blocks. Each scenario reduces the number of block accesses because the records per year decrease.
12.4.	An index on field A is useless if the number of different A-values is less than the number of table records that fit in a block. This is because if every block contains unique values of A, then every search on A will likely return at least one block. This, in turn, does not reduce the search space effectively.
12.5.	Creating an index for another index (secondary index) could make sense in some database systems, especially for multi-level indexing systems where the first level index can grow very large and might benefit from having its own index for faster search. Or this can be useful in database management systems for scalable microservice architecture, where different DBMSs are used for each different microservice.
12.6.	Total number of blocks required = (Total number of records in DEPT table) / (Block size / (Key size + Pointer size))
12.7.	A **deleteAll** method could be useful for batch deletions where all records with a certain value need to be removed, for instance, when deleting all students enrolled in a cancelled course section. This would be more efficient than calling **delete** for each individual record.
12.8.	**Statistics given in Fig 7.8:** • **STUDENT** table has 4,500 blocks (**B(T)**) and 45,000 rows (**R(T)**). • **DEPT** table has 2 blocks and 40 rows. • The index on **MajorId** in **STUDENT** has 40 distinct values (**V(T,F)**).

- The index on **DId** in **DEPT** has 40 distinct values as well.

Cost of Plan Using STUDENT Index on MajorId:

1. For each distinct **MajorId** value, we have to read the corresponding index entry. There are 40 distinct values, so we assume 40 index reads.
2. Each index read for **MajorId** will point to a number of **STUDENT** rows. Since there are 45,000 students and 40 majors, on average there are 45,000 / 40 = 1,125 students per major.
3. For each student found, we then look up the **DEPT** table. Since **DEPT** is very small (2 blocks), we can assume that reading the whole **DEPT** table would be the cost for each student. But we will use the index on **DId** for the match, which should be significantly less than reading the whole table, so we will consider it to be a single block read per student. Therefore, the cost would be 1,125 reads per **MajorId**.

So, the total cost using the **STUDENT** index would be approximately: 40 index reads for **MajorId** + (1,125 student reads per **MajorId** * 40 **MajorId**) = 40 + (1,125 * 40) = 45,040 block reads.

Cost of Plan Using DEPT Index on DId:

1. Since **DEPT** only has 2 blocks, we can read the entire **DEPT** table with 2 block reads.
2. After reading **DEPT**, for each department, we look up all students in that department using the index on **MajorId**.
3. Since there are 45,000 students and 40 departments, we have 1,125 students per department as calculated before.

This time, however, since we're starting from **DEPT**, we read the entire **DEPT** table once and then perform the index read into **STUDENT** for each department.

As we have 40 departments, we will perform 40 index reads into **STUDENT**.
The total cost using the **DEPT** index would be: 2 block reads for the whole **DEPT** table + 40 index reads into **STUDENT** = 2 + 40 = 42 block reads.

It is clear from the calculation that the cost of the plan using the **DEPT** index on **DId** is significantly less than using the **STUDENT** index on **MajorId**. This is primarily due to the much smaller size of the **DEPT** table compared to the **STUDENT** table.

When performing an index join, if one of the tables is significantly smaller than the other and both have indexes on the join condition, it is generally more cost-effective to iterate over the smaller table and use its index to join with the larger table.

12.9.	We will be inserting records for employees having IDs 28, 9, 16, 24, 36, 48, 64, and 56 into the bucket file structure shown in Fig 12.7, assuming the same hash function $\mathrm{h}(x) = x \bmod 8$. Here's how to proceed with each insertion: 1. **Inserting ID 28:** The hash value is 28mod 8=428mod8=4. This goes to the same block as IDs 4 and 12, which is block 0. If we still have space in block 0, it will be inserted there; otherwise, we'd need to split the block. 2. **Inserting ID 9:** The hash value is 9mod 8=19mod8=1. This should go to block 1 according to the directory. 3. **Inserting ID 16:** The hash value is 16mod 8=016mod8=0, which would direct it to block 0. However, block 0 may need splitting if it's full due to the previous insertion of ID 28. 4. **Inserting ID 24:** The hash value is 24mod 8=024mod8=0, which again points to block 0, potentially causing a split if the block is already full. 5. **Inserting ID 36:** The hash value is 36mod 8=436mod8=4, pointing to block 0, which could be full by now, necessitating a split. 6. **Inserting ID 48:** The hash value is 48mod 8=048mod8=0, aiming for block 0, which is getting crowded if we haven't already split it. 7. **Inserting ID 64:** The hash value is 64mod 8=064mod8=0, once again targeting block 0. 8. **Inserting ID 56:** The hash value is 56mod 8=056mod8=0, and this also is meant for block 0. As the buckets become full, we'd need to split them. Each split would involve doubling the size of the directory and reassigning the records based on a new hash function, which considers one more bit of the hash code each time a split occurs (this is because we are dealing with extendable hashing).
12.10.	In an extendable hash index, the local depth (L) indicates the number of bits used to identify a block within the bucket file. Because extendable hashing dynamically adjusts the number of buckets (and the corresponding directory size) when buckets split or are coalesced, the blocks that result from a split will have their local depth increased by 1. Each bucket that points to a block with local depth L will have the same L rightmost bits in their hash values. This is because, during the splitting process, all records are redistributed based on these L bits.
12.11.	An algorithm for deletion in extendable hashing would involve the following steps: 1. Locate the block that contains the record to be deleted. 2. Remove the record from the block.

	3. Check if the block is underutilized (i.e., it has fewer records than a certain threshold). 4. If it is underutilized, check if it can be coalesced with its buddy block (the block with the same rightmost L-1 bits if local depth L>1). 5. If coalescing is possible (the buddy block exists and the combined records fit into a single block), then coalesce the blocks, decrease the local depth, and update the directory accordingly. 6. If coalescing is not possible or not practical (due to data distribution), no further action is needed. The practicality of this operation depends on the workload and the cost of updating the directory. In systems where deletions are rare, the cost of maintaining the structure for coalescing may not be justified. If deletions are frequent, coalescing can help in reducing wasted space.
12.12.a.	Before the global depth of the index becomes 1, only one record can be inserted because an empty index starts with a global depth of 0, meaning all records go into the same block until it is full.
12.12.b.	After the global depth becomes 1, we can have two blocks. Since each block can hold 100 records, we can insert an additional 200 records before the global depth needs to increase to 2 (100 for the already filled block and 100 for the new one).
12.13.a.	When the global depth increases from 3 to 4, the number of entries in the bucket directory doubles. If there were 2^3=8 entries at depth 3, there will now be 2^4=16 entries.
12.13.b.	Assuming an ideal distribution of records and that only the last insertion caused the split and increase in global depth, then only one block will have a single directory entry pointing to it, which is the new block created as a result of the split.
12.14.	Any extendable hash index can be accessed in two block accesses due to its structure: one block access is for reading the bucket directory to find out which block contains the record (since the directory is often kept in memory, this might not even count as a block access in some implementations), and the second block access is for reading the block that contains the actual record.
12.15.	The drawing process involves following the rules of B-tree insertion, starting with a single node (root) and then splitting nodes once they exceed the maximum number of records they can hold (in this case, 3 index records per block).
12.16.a.	B = 45,000 / 100 = 450 blocks
12.16.b.	For the directory file, it depends on the structure of the B-tree. The directory file is made up of the internal nodes of the B-tree.
12.17.a.	Since each block can hold 100 records, the first split will occur after the 101st insertion, as each B-tree block (or node) needs to be full before splitting.
12.17.b.	After the first split, we will have 2 blocks at the next level, each capable of holding 100 pointers to blocks. Therefore, we would need 100 blocks full of

	records before requiring a split to the next level (since each block can hold 100 records). That gives us 100 × 100 = 10,000 records. But, considering we already have 200 records after the first split (two full blocks), we would need an additional 9,800 insertions to cause the root to split again.
12.18.a.	During an index scan, the maximum number of buffers pinned simultaneously is usually equal to the height of the B-tree because the path from the root to a leaf is accessed.
12.18.b.	During an insertion, the maximum number of buffers pinned simultaneously would also correspond to the height of the B-tree. Additionally, there might be a need for an extra buffer during the split of a node
12.19.a.	To use a B-tree index for a range query, we would start at the root and traverse down to the leaf nodes that contain the entries where **GradYear > 2019**. The search would begin at the smallest key that satisfies the condition and continue scanning the leaves until there are no greater keys.
12.19.b.	The revisions to the SimpleDB B-tree code would include: • Implementing a range search function in the B-tree leaf pages to find the starting point of the range. • Modifying the IndexSelectScan to handle a range of values instead of a single value.
12.19.c.	The B-tree index on **GradYear** might not be useful if: • **GradYear** is not selective enough (e.g., most students graduate in the same year or a few years, resulting in a large range). • The distribution of **GradYear** is such that the range query would have to scan a significant portion of the entries. It would be useful if the range of **GradYear > 2019** includes a small portion of the total entries, making the range scan efficient.
12.19.d.	The SimpleDB IndexSelectPlan and IndexSelectScan classes are designed to work with selection predicates that are based on equality comparisons. When it comes to range predicates, such as "GradYear > 2019," the utility of different types of indexes varies. Static Hash Indexes: 1. **Hashing Mechanism**: Static hash indexes map a key value to a particular location in an index through a hash function. The result is that all entries with the same key value are grouped together in one bucket. 2. **Range Queries**: A range query, such as "GradYear > 2019," does not specify a single key value but rather a range of values. Since the hash function distributes key values based on their hash and not their logical ordering, values that are logically sequential (like years in a range) may not be stored close together in the index. 3. **Inefficiency**: To find all records where "GradYear > 2019," the index would need to be searched entirely, as there's no guarantee that values after

2019 are hashed to consecutive or nearby buckets. This is akin to a full table scan, thus negating the benefits of using an index for range queries.

Extendable Hash Indexes:

1. **Dynamic Growth**: Extendable hash indexes are a type of hash index that can dynamically grow and shrink. They maintain a directory that points to buckets where records are stored, and the directory grows as more unique hash values are added.
2. **Hash Function Limitation**: Like static hash indexes, they use a hash function to determine the location of records. The hash function is based on the value of the key, without regard to the logical order of the keys.
3. **Range Query Issues**: For a range query, extendable hash indexes encounter the same problem as static hash indexes: there's no way to efficiently retrieve all records within a range because logically sequential keys can be widely distributed across different buckets.

In summary, both static and extendable hash indexes are not useful for range queries because their underlying hash functions do not preserve the order of the key values. For such queries, indexes that maintain some sort of order, like B-trees or tree-based indexes, are more appropriate, as they allow for efficient traversal in a sorted order to find all keys within a given range.