Changes from 3rd edition:
The major change from the previous edition is that the 3rd edition chapter on query processing has been split into two chapters.

Coverage of size estimation for different operations, which was earlier covered along with algorithms for the operations has now been separated out into a separate section (Section 14.2). Some of the formulae for estimation of statistics have been simplified and a few new ones have been added.

Pseudocode has been provided for the dynamic programming algorithm for join order optimization. There is a new section on optimization of nested subqueries, which forms an important part of SQL optimization. The section on materialized views is also new to this edition.

Exercises

14.1 Clustering indices may allow faster access to data than a nonclustering index affords. When must we create a nonclustering index, despite the advantages of a clustering index? Explain your answer.
Answer: There can be only one clustering index for a file, based on the ordering key. Any query which needs to search on the other non-ordering keys will need the non-clustering index. If the query accesses a majority of the tuples in the file, it may be more efficient to sort the file on the desired key, rather than using the non-clustering index.

14.2 Consider the relations $r_1(A, B, C)$, $r_2(C, D, E)$, and $r_3(E, F)$, with primary keys *A*, *C*, and *E*, respectively. Assume that r_1 has 1000 tuples, r_2 has 1500 tuples, and r_3 has 750 tuples. Estimate the size of $r_1 \bowtie r_2 \bowtie r_3$, and give an efficient strategy for computing the join.
Answer:

- The relation resulting from the join of r_1, r_2, and r_3 will be the same no matter which way we join them, due to the associative and commutative properties of joins. So we will consider the size based on the strategy of $((r_1 \bowtie r_2) \bowtie r_3)$. Joining r_1 with r_2 will yield a relation of at most 1000 tuples, since C is a key for r_2. Likewise, joining that result with r_3 will yield a relation of at most 1000 tuples because *E* is a key for r_3. Therefore the final relation will have at most 1000 tuples.
- An efficient strategy for computing this join would be to create an index on attribute *C* for relation r_2 and on *E* for r_3. Then for each tuple in r_1, we do the following:
  a. Use the index for r_2 to look up at most one tuple which matches the *C* value of r_1.
  b. Use the created index on E to look up in r_3 at most one tuple which matches the unique value for E in r_2.

14.3 Consider the relations $r_1(A, B, C)$, $r_2(C, D, E)$, and $r_3(E, F)$ of Exercise 14.2. Assume that there are no primary keys, except the entire schema. Let $V(C, r_1)$ be 900, $V(C, r_2)$ be 1100, $V(E, r_2)$ be 50, and $V(E, r_3)$ be 100. Assume that r_1

has 1000 tuples, r_2 has 1500 tuples, and r_3 has 750 tuples. Estimate the size of $r_1 \bowtie r_2 \bowtie r_3$, and give an efficient strategy for computing the join.
Answer: The estimated size of the relation can be determined by calculating the average number of tuples which would be joined with each tuple of the second relation. In this case, for each tuple in r_1, $1500/V(C, r_2)$ = 15/11 tuples (on the average) of r_2 would join with it. The intermediate relation would have 15000/11 tuples. This relation is joined with r_3 to yield a result of approximately 10,227 tuples (15000/11 $\times$ 750/100 = 10227). A good strategy should join r_1 and r_2 first, since the intermediate relation is about the same size as r_1 or r_2. Then r_3 is joined to this result.

14.4 Suppose that a B$^+$-tree index on *branch-city* is available on relation *branch*, and that no other index is available. What would be the best way to handle the following selections that involve negation?

a. $\sigma_{\neg(branch\text{-}city < \text{“Brooklyn”})}(branch)$
b. $\sigma_{\neg(branch\text{-}city = \text{“Brooklyn”})}(branch)$
c. $\sigma_{\neg(branch\text{-}city < \text{“Brooklyn”} \vee assets < 5000)}(branch)$

Answer:

a. Use the index to locate the first tuple whose *branch-city* field has value “Brooklyn”. From this tuple, follow the pointer chains till the end, retrieving all the tuples.
b. For this query, the index serves no purpose. We can scan the file sequentially and select all tuples whose *branch-city* field is anything other than “Brooklyn”.
c. This query is equivalent to the query

$$\sigma_{(branch\text{-}city \geq \text{“Brooklyn”} \wedge assets < 5000)}(branch)$$

Using the *branch-city* index, we can retrieve all tuples with *branch-city* value greater than or equal to “Brooklyn” by following the pointer chains from the first “Brooklyn” tuple. We also apply the additional criteria of $assets < 5000$ on every tuple.

14.5 Suppose that a B$^+$-tree index on (*branch-name, branch-city*) is available on relation *branch*. What would be the best way to handle the following selection?

$$\sigma_{(branch\text{-}city < \text{“Brooklyn”}) \wedge (assets < 5000) \wedge (branch\text{-}name = \text{“Downtown”})}(branch)$$

Answer: Using the index, we locate the first tuple having *branch-name* “Downtown”. We then follow the pointers retrieving successive tuples as long as *branch-city* is less than “Brooklyn”. From the tuples retrieved, the ones not satisfying the condition $(assets < 5000)$ are rejected.

14.6 Show that the following equivalences hold. Explain how you can apply then to improve the efficiency of certain queries:

a. $E_1 \bowtie_\theta (E_2 - E_3) = (E_1 \bowtie_\theta E_2 - E_1 \bowtie_\theta E_3)$.