CSE 444: Database Internals. Section 3: Indexing.

1. CLUSTERING

(based on Exercise 15.6.2 and 15.6.4, pp. 745 in the textbook)

Consider a relation R with $500,000$ tuples stored on disk in $10,000$ pages. Let there be an index on $R.a$, and let the number of distinct values in the column a be k, for some number k. Give the cost of $\sigma_{a=0}(R)$ as a function of k, under the following circumstances. You may neglect the disk I/O's needed to access the index itself.

[We assume that the number of tuples with $a = 0$ is $T_0 = \frac{T(R)}{k}$ spread over $B_0 = \lceil \frac{B(R)}{k} + 1 \rceil$ pages. The extra page is to account for the case when tuples with $a = 0$ start appearing in the middle of a page, when the relation is ordered by the attribute a.]

(1) The index is clustered (the book says the index is clustering; it is the same thing). [Answer: B_0, since we can use the index to sequentially read the pages that contain tuples with $a = 0$.]

(2) The index is not clustered. [Answer: T_0, since we have to read each tuple at a time since the index is unclustered.]

(3) The relation R is clustered on the attribute a, and no index is used. [Answer: We can scan the whole relation at a cost of $B(R)$ pages. We can do slightly better by performing a binary search and then reading in the pages containing $a = 0$ for a cost of $\log_2 B(R) + B_0$.]

(4) The relation R is clustered but the index on $R.a$ is not clustering. For what values of k is it preferable to perform a table-scan of R over using the index? [Since R is clustered, we can read it all by reading in $B(R)$ pages. Reading from the index takes T_0 page reads. Thus, scanning is preferable when $B(R) < T_0$, that is when $k < \frac{T(R)}{B(R)} = 50$].

2. PERFORMANCE TUNING

Suppose we have a relation $R(a, b, c, d)$, where attribute a is the key. The relation is clustered on a, and there is a single B+ tree index on a. Consider each of the following query workloads, independently of each other. If it is possible to speed it up significantly by adding a single additional B+ tree index to relation R, specify which attribute or set of attributes that index should cover. You may only add at most one new index. If two or more possible indices would do an equally good job, pick one of them. If adding a new index would not make a significant difference, you should say so. Give a brief justification for your answers.

[We will have to use composite keys.]

(1) All queries have the form: **select** $*$ **from** R **where** $b >?$ **and** $b <?$ **and** $d =?$
[An index on the search key (d, b) works the best since with such a key, the index entries for $(d, b_1), (d, b_2), \ldots$ appear sequentially. The search key (b, d) does not work that well since $(b_1, d), (b_1, d), \ldots$ do not appear sequentially in the index.]

(2) 100,000 queries have the form: **select** $*$ **from** R **where** $b =?$ **and** $c =?$ and 100,000 queries have the form: **select** $*$ **from** R **where** $b =?$ **and** $d =?$
[Any one of a search key index on (b, c) or (b, d) would speed up half of the queries. It does not seem likely that a single index can speed up the entire workload.]

3. B+ TREE: INSERTIONS AND DELETIONS

Walk-through an example on the board. This example is the one found in the textbook by Ramakrishnan et al.

Let each node has m entries and $m + 1$ pointers, where m is constrained such that $d \leq m \leq 2d$. d is called the tree's order.

There are three layers:

(1) **Root:** Has at least two pointers. Is allowed to have $1 \leq m \leq 2d$ entries.
(2) **Intermediate layer:** Let the keys be $K_1, \ldots K_m$. Then the first pointer points to **some** of the records with value less than K_1, while the last one points to **some** of the records with value greater than or equal to K_{j-1}. The intermediate pointers point to **all** the records with keys that are greater than equal to K_{j-1} and less than Kj.
(3) **Leaves:** The rightmost pointer points to the next leaf. Of the remain m pointers, the i^{th} pointer points to the i^{th} key.

Insertion. Insertion is done recursively, starting from a leaf node and moving up, as needed.

Search for the leaf that would have stored the key we want to insert and insert the key if there is free space in the leaf node. If not, we have $2d + 1$ entries and they can be split into two nodes with the left node and three in the right.

Note that for leaf pages, the first entry of the newly split page is "copied up"; while for internal nodes, the first entry of the newly split page is "pushed up". We push up the non-leaf entries to prevent redundant copies of the key-values. We can not push up for the leaf entries since we need all key-values to be present in the leaves.

For the splitting of non-leaf node, think of pushing the 3 entries through the new root, and using the last pushed value as the new root key-value.

Deletion. Deletes are also performed recursively, starting from a leaf node and then moving up, as needed.

Find the leaf node that contains the key and delete the key. If, after the deletion, there are still m entries, nothing more needs to be done. Otherwise, we look at a sibling leaf node to redestribute entries to balance the nodes. If we redistribute, we "copy up" the first entry of the second node. Otherwise, we merge the siblings by copying the entries from the right node to the left one, and copying up a blank, and then deleting it. If the non-leaf, parent node has more than d entires, we stop; otherwise, We redistribute keys for the non-leaf node, by "pushing" keys through the parent from a the sibling. This might create a vacancy, and we recursively go up.