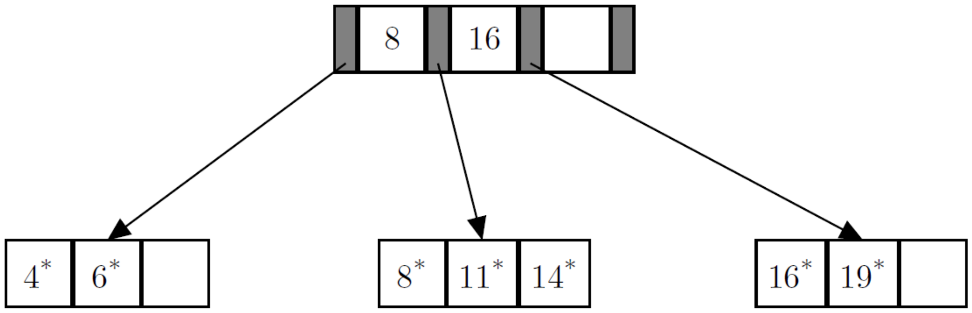
**NYU Tandon School of Engineering** **April 22, 2018**
**Computer Science and Engineering**
**CS6083, Spring 2018**

# **Problem Set 4** (due May 5)

**Note:** For simplicity, you may assume that KB, MB, and GB refer to $10^3$, $10^6$, and $10^9$ bytes, respectively.

## Problem 1

Assume the following simple $B^+$-tree with n=4:



This tree consists of only a root node and three leaf nodes. Recall that the root node must have between 2 and *n* child pointers (basically RIDs) and between 1 and *n*-1 key values that separate the subtrees. In this case, the values 8 and 16 mean that to search for a value strictly less than 8 you visit the left-most child, for at least 8 and strictly less than 16, you visit the second child, and otherwise the third child. Each internal node (none in this figure) has between 2 and 4 child pointers and between 1 and 3 key values (always one less than the number of pointers), and each leaf node has between 2 and 3 key values, each with an associated RID (to its left) pointing to a record with that key value in the underlying indexed table. Sketch the state of the tree after each step in the following sequence of insertions and deletions:

***Insert 9, Insert 20, Insert 21, Delete 6, Delete 21, Insert 12, Insert 13***

Note that for insertions, there are two algorithms, one that splits a full node without trying to off-load data to a direct neighbor, and one that first tries to balance with a direct neighbor in the case of a full node. Please use the first algorithm! For deletion, first try to merge, and if not possible, rebalance with a neighboring sibling.

## Problem 2

You are given a sequence of 8 key values and their 8-bit hash values that need to be inserted into an extendible hash table where each hash bucket holds at most two entries. The sequence is presented in Table 1 below. (You do not need to know what function was used to compute the hashes, since the hashes are already given.) In Figure 1 you can see the state of the hash table after inserting the first two keys, where we only use the first (leftmost) bit of each hash to organize the buckets. Now insert the remaining six keys ($k_2$ to $k_7$) in the order given. Sketch the bucket address table and buckets after each insertion.

| Keys | Hash values |
|---|---|
| k0 | 10101100 |

|  |  |
|---|---|
| k1 | 01110010 |
| k2 | 11111001 |
| k3 | 10100100 |
| k4 | 11010001 |
| k5 | 10011110 |
| k6 | 10100101 |
| k7 | 11100110 |

Table 1: Sequence of 8 keys and their corresponding 8-bit hashes.

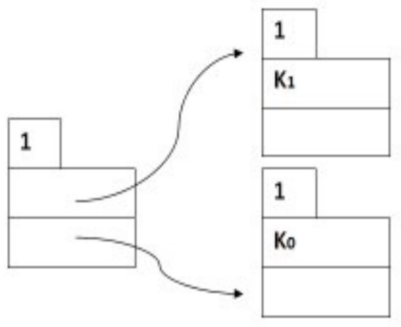


Figure 1: Hash Table state after insertion of the first two keys

## Problem 3

In the following, assume the latency/transfer-rate model of disk performance, where we estimate disk access times by allowing blocks that are consecutive on disk to be fetched with a single seek time and rotational latency cost (as shown in class). Also, assume a record ID of an index entry costs 8 bytes.

You are given the following very simple database schema that models a music website and stores information about users, tracks, plays, and artists. We store user information including user ID, username, the city a user lives in, and the date when the user registered on the website; artist information is also stored, including artist ID, artist name, and the artist's genre; we also store information about tracks, including track ID, artist id, track name, and the duration of the track; user play records are stored in the Play table, in which we store the uid, tid, and the date and time when the user played the track. The schema is as follows:

**User (uid, uname, ucity, joindate)**
**Artist (aid, aname, genre)**
**Track (tid, aid, tname, tduration)**
**Play (uid, tid, pdate, ptime)**

Assume there are 100 million users, 5 million tracks, 1 million artists, and 20 billion play records over a total period of 100 days. Each play tuple is of size 40 bytes, and all other tuples are 100 bytes. Now consider the following queries:

**select tid, aid**
**from Track t, Play p**
**where t.tid = p.tid and p.pdate = "2017-11-29"**

```
select tid, aid
from Artist a, Track t, Play p
where a.aid = t.aid and t.tid = p.tid and a.genre = “Jazz” and p.pdate = “2017-11-29”

select uid, uname
from User u, Track t, Play p
where u.uid = p.uid and t.tid = p.tid and t.duration < 10s and u.ucity = “Miami”
```

(a) For each query, describe in one sentence what it does. (That is, what task does it perform?)

In the following questions, to describe how a query could be best executed, draw a query plan tree and state what algorithms should be used for the various selections and joins. Also provide estimates of the running times, assuming these are dominated by disk accesses.

(b) Assume that there are no indexes on any of the relations, and that all relations are unclustered (not sorted in any way). Describe how a database system would best execute all three queries in this case, given that 1 GB of main memory are available for query processing, and assuming a hard disk with 5ms for seek time plus rotational latency (i.e., a random access requires 5ms to find the right position on disk) and a maximum transfer rate of 100 MB/s. Assume that 0.1% users live in Miami, 1% of all artists’ genre is Jazz, and 1% of all tracks have a duration shorter than 10 seconds. Otherwise, assume that data is evenly and independently distributed; e.g., 200 million playing records were made on November 28, 2017.

(c) Consider a unclustered B+-tree index on aid in the Artist table, and a dense clustered B+-tree index on tid in the Play table. For each index, what is the height and the size of the tree? How long does it take to fetch a single record with a particular key value using these indexes? How long would it take to fetch all, say, 50 records matching a particular tid value in the case of the second index? (Assume that aid and tid each take 8 bytes, and that a record ID or pointer takes 12 bytes. You may assume 80% occupancy ratio for the index nodes, and 100% for the underlying table.)

(d) Suppose that for each query, you could create up to two index structures to make the query faster. What index structures would you create, and how would this change the evaluation plans and running times? (In other words, redo (b) for each query using your best choice of up to two indexes for that query.)

## Problem 4

(a) Consider a hard disk with 12000 RPM and 3 single-sided platters. Each surface has 200,000 tracks and 1000 sectors per track. (For simplicity, we assume that the number of sectors per track does not vary between the outer and inner area of the disk.) Each sector has 1024 bytes. What is the capacity of the disk? What is the maximum rate at which data can be read from disk, assuming that we can only read data from one surface at a time? What is the average rotational latency?

(b) Suppose the same disk as in (a), where the average seek time (time for moving the read-write arm) is 4ms. How long does it take to read a file of size 200 KB? How about a file of 20 MB? How about a file of 2 GB? Use both the block model (4KB per block) and the latency/transfer-rate model, and compare. You can assume that KB, MB, GB are $10^3$, $10^6$, and $10^9$ bytes, respectively

(c) Suppose you have a file of size 256 GB that must be sorted, and you have only 4 GB of main memory to do the sort (plus unlimited disk space). Estimate the running time of the I/O-efficient merge sort algorithm from the class on this data, using the hard disk from part (b). Use the latency/transfer-rate model of disk performance, and ignore CPU performance. Assume that in the merge phase, all sorted runs from the initial phase are merged together in a single merge pass.

(d) Suppose you use two (instead of one) merge phases in the scenario in (c). What is the running time now?