Problem 1: Assume the following simple B+-tree with n=4:

A whiteboard with writing on it

Description automatically generated with medium confidence

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. (The fourth, rightmost pointer in each leaf points to the next leaf to the right, as often done.) Sketch the state of the tree after each step in the following sequence of operations:

Delete 18, Insert 4, Insert 2, Insert 1, Insert 0, Delete 20, Delete 13.

Delete 18

Diagram

Description automatically generated

Insert 4

Diagram

Description automatically generated

Insert 2

Diagram

Description automatically generated

Insert 1

Diagram

Description automatically generated

Insert 0

0 is inserted into node (1, 2, 3) which causes a split. 2 will be moved into node (4, 6, 13) which preempts a split as well.

Diagram

Description automatically generated

Delete 20

20 is at a leaf node, where the child node conditions still stand

Diagram

Description automatically generated

Delete 13

Diagram

Description automatically generated

Problem 2:

You are given a sequence of 7 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 resulting 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 in the order given. Sketch the bucket address table and the buckets after each insertion.

|  |  |
| --- | --- |
| Keys | Hash value |
| K0 | 10110110 |
| K1 | 01000101 |
| K2 | 01000111 |
| K3 | 01001100 |
| K4 | 10111010 |
| K5 | 10101011 |
| K6 | 00101000 |

Diagram

Description automatically generated

Insert K2

Diagram, engineering drawing

Description automatically generated

Insert K3

Diagram

Description automatically generated

Insert K4

Diagram, engineering drawing

Description automatically generated

Insert K5

Diagram, engineering drawing

Description automatically generated

Insert K6

Diagram, engineering drawing

Description automatically generated

Problem 3:

In the following, assume the latency/transfer-rate model of disk performance, where we estimate disk access times by allowing consecutive blocks on disk to be fetched with a single seek time and rotational latency cost (as shown in class). All index entries are of size 16 bytes (8 bytes for key and 8 bytes for record ID), and all tables stored with 100% occupancy.

You are given the following simple database schema, which models rides taken in a subway system. We assume that riders swipe their metrocard whenever they enter or leave the system. For each ride taken, we store the origin and destination station, the ID of the metrocard, and time+date stamps for when the rider swiped in and out. For each station, we store its station ID, name, and the borough and neighborhood where it is located. A metrocard is identified by a unique ID, and we also store the date it was purchased, and the date it expires. Our simplified schema does not keep track of the cost of the ride or the balance on the metrocard. We also do not store the identities of the riders.

MetroCard (mid, purchasedate, expiredate)  
Station (sid, sname, sborough, shood)  
Ride (mid, origsid, destsid, intimedate, outtimedate)  
// origsid and destsid reference sid in Station, and mid references mid in MetroCard

Assume there are 400 million metrocards, 500 stations, and 20 billion rides stored in the schema. For simplicity, assume that every attribute is of size 8 bytes, and the size of each tuple is thus 8\*x bytes for a table with x attributes, and that the size of a table is simply the size of a tuple multiplied by the number of tuples (i.e., 100% occupancy and no gaps or space for block metadata). Now consider the following two queries:

select R.mid, R.intimedate, R.destid  
from Ride R  
where R.origsid = 184

select M.mid  
from MetroCard M, Ride R, Station S  
where M.mid = R.mid and R.origsid = S.sid and and S.sname = “Columbus Circle” and  
year(M.purchasedate) = 2020 and date(R.intimedate) = '2021-03-21’

1. For each query, describe in one sentence what it does. (What task does it perform?) For the second query, assume that year() and date() return the year and date of a time+date stamp.

select R.mid, R.intimedate, R.destid  
from Ride R  
where R.origsid = 184

Lists the id of the metrocard, the destination, and swipe in time of all rides from id 184

select M.mid  
from MetroCard M, Ride R, Station S  
where M.mid = R.mid and R.origsid = S.sid and and S.sname = “Columbus Circle” and  
year(M.purchasedate) = 2020 and date(R.intimedate) = '2021-03-21’

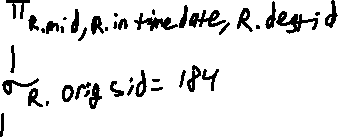
Get the ids of all metrocards that were purchased in 2020 and boarded a ride on March 21, 2021 coming from station Columbus Circle.

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. Provide estimates of the running times, assuming that 1GB of memory is available for query processing, that we have a hard disk with 5ms access time (seek time plus rotational latency) and a maximum transfer rate of 200 MB/s, and that query times are dominated by the cost of reading and writing disk-based data

1. Assume that there are no indexes on any of the relations, and that all relations are unsorted. Describe and justify how a database system would best execute the two queries. Draw a query plan and state what algorithms for selection and joins should be used. Assume that all 500 subway stations have about the same amount of riders entering and leaving (about 0.2% of the total), that each day is responsible for about 0.1% of the total ridership (say, assuming that the system keeps data for the last 1000 days), and that about 20% of all metrocards were purchased in 2020. Also, assume independence between different attributes, and that there is only one station called “Columbus Circle”. Also try to estimate the running times of the queries.

select R.mid, R.intimedate, R.destid  
from Ride R  
where R.origsid = 184

The first query will be drawn as



The estimated run time:

The Ride table has 5 attributes which are 8 bytes each. This means each row in the Ride table has 40 bytes. Since ride contains 20 billion entries, that means the entire ride table is 800 GBs.

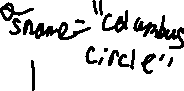
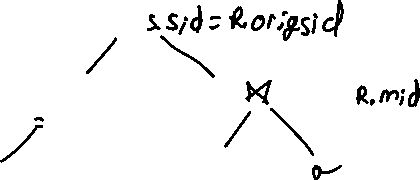
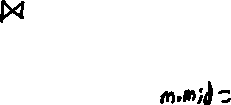
We would first scan the table Ride R and collect any entries where R.origsid = 184.

200 MB/s read time = 0.2 GB/s

select M.mid  
from MetroCard M, Ride R, Station S  
where M.mid = R.mid and R.origsid = S.sid and and S.sname = “Columbus Circle” and  
year(M.purchasedate) = 2020 and date(R.intimedate) = '2021-03-21’

The second query will be drawn as:

* Station S has 4 attributes, making each row 32 bytes each. Station S has 500 entries meaning the table Station has 16 KBs of data.
* Ride has 800 GBs of data \* 0.1% = 800 MBs for one day
* Metrocard has 3 attributes, giving each row 24 bytes each and 400 million entries. In total, the Metrocard table has 9.6 GB of data
* 9.6 GBs \* 20% = 1.92 gigabytes for 2020
* The optimal query join would be based off the smallest sets table we can join together first.
* Station S is the smallest table, so we can join it with the next smallest table.
* Ride table can be compressed into a smaller set if we do a select search, making it smaller than the Metrocard table.



The estimated run time:

* We start by scanning Station S. We’re looking for only one attribute Sname = Columbus Circle and storing it in main memory. Only one tuple needs to be stored.
* Then, we have to scan the Metrocard table and join it with Station.
* There are 20% of all metrocards in 2020: 400 million \* 0.2 = 80 million metrocards in 2020
* We’re using R.mid and storing M.purchase date which are two attributes of 8 bytes each.
* 16 bytes \* 80 million= 1.28 GBs to store in main memory
* Finally, we would have to scan the rides table and join it with the previous two tables:
* We are checking one day in rides which has 0.1% of all rides: 0.001 \* 20 billion = 20 million rides
* We’re storing R.origsid, R.mid, and R.intimedate for R which totals to 24 bytes
* 24 bytes \* 20 million rides = 480 MBs to store in main memory

The cost of reading all three tables:

1. Consider the order of the two joins in the second query. Should you first join M with R, and then the result with S, or should you first join S with R, and then join the result with M? Which would be faster for this query, or would it not make much of a difference in this case? Discuss.

* Station S has 16 KBs of data.
* Ride has 800 GBs of data \* 0.1% = 800 MBs for one day
* Metrocard has 1.92 gigabytes for 2020

The optimal method of making query joins is by joining the smallest table with the next smallest table in order to keep our current set of table matches as small as possible.

While the Ride table is the largest table, we are searching the set of rides within a specific day. We can do a search within Rides and grab only the necessary tuples, reducing the size of the Rides table to be smaller than the Metrocard table which searched for all entries within a year.

Because Ride is smaller than Metrocard, it would be optimal to search within Rides first, then join it with the Station table. The resulting subset would be smaller than joining Station with Metrocard. We would be saving approximately an extra gigabyte of tuples to match by joining S and R first, assuming we do a query search within R before joining the two together.

1. Consider a dense unclustered B+-tree index on purchasedate in the MetroCard table, and a dense clustered B+-tree index on origsid in the Ride table. For each index, what is the height and the size of the tree? How long would it take to fetch a single record with a particular purchasedate time+date stamp using the first index, assuming there is only one matching record? How long would it take to fetch all, say, 1000000 records matching a particular origsid in the case of the second index? (You may assume 80% occupancy ratio for the index nodes, and 100% for the underlying table, and that there is no caching of data in memory.)

**Dense, Unclustered on Metrocard**:

Each index would hold the key and pointer and the purchasedate of the Metrocard table = 16 bytes for each index entry + an additional 4 bytes for purchasedate = 20 bytes. There would be 400 million index entries for Metrocard.

With a node size being 4 KBs:

This means that each node will contain 164 children. With 400 million index entries, that would mean

This means that there are 3 levels in the B+ tree. The size of the tree would depend on the leaf level where each entry is 4KBs.

It would then take 4 \* 10 ms = 40 ms to fetch a single record from the table. When fetching multiple entries in an unclustered tree, we would have to make a search for each one individually. Fetching 1,000,000 records in an unclustered tree would cost 40 ms \* 1 million = 11.11 hours to fetch all records.

**Dense, clustered tree on Ride**

* 16 bytes for an index entry + 4 bytes for an entry for origsid
* With Ride containing 20 billion entries:

This leads to the root node, giving us a tree of height 5. The size of the tree would be 12.12 million \* 4 KBs = 48.48 GBs of data.

The time it would take to fetch an entry from the B+ tree would be: 10ms \* 5 = 50 ms. Since this is a clustered index, the time it would take to fetch multiple entries would be faster. The time it would take to fetch 1 million entries:

1. Suppose that for each query, you could create only one index structures to make this query faster. What index structure would you create, and how would this change the evaluation plans, algorithms used, and running times? (In other words, redo (b) for each query using your best choice of index for that query.)

select R.mid, R.intimedate, R.destid  
from Ride R  
where R.origsid = 184

Because this table is based off using only origsid, we can use a dense, clustered index on (origsid) in the rides table. Instead, this would require only a lookup in the general id range containing 184. This would change the runtime to be only 50 ms to fetch one entry in the structure.

select M.mid  
from MetroCard M, Ride R, Station S  
where M.mid = R.mid and R.origsid = S.sid and and S.sname = “Columbus Circle” and  
year(M.purchasedate) = 2020 and date(R.intimedate) = '2021-03-21’

This table runs based off multiple attributes including sid, origid, and mid. In addition, this table is focused on gathering from batch fetch requests using purchase date and intimeanddate.

We can use a sparse, clustered index on the Ride table based on origsid and intimidate. This would reduce the size of Ride R from 40 billion entries to about 20 million, turning the size of our fetch record pool from 800GBs to 800 MBs.

Problem 4:

1. Consider a hard disk with 12000 RPM and 2 double-sided platters. Each surface has 200,000 tracks and 2000 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 500 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?

Capacity: 2 platters \* 200,000 tracks \* 2000 sectors \* 500 bytes per sector = 0.4 TBs

Rate of read: 12,000 RPM = 200RPS

200 RPS \* 2000 sectors per track \* 500 bytes = **200 MB/s**

The average rotational latency: Half the time it takes to make a full rotation

1. Suppose we have 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 scan the entire MetroCard table from the previous problem?

The Metrocard table contains 400 million entries which, given each row is 8 bytes each, adds up to 3.2 GBs. It has a transfer rate of 200 MB/s = 200 KB/ms

4 ms (avg. seek time) + 3.2 GBs / 200 MB/s = 4ms + 3200 MBs / 200 MB/s = 4 ms + 160 s = 160.004 seconds

1. Suppose you want to sort the Ride table by mid, and you have 20GB of main memory available. How long would this take using an external memory merge sort, and how many merge phases would you use? Assume the same disk as in (b).

There are 20 billion rides which are 16 bytes each: 20 billion \* 16 byes = 320 GBs

Main memory contains 20GBs of memory and a transfer rate of 0.2 GB/s

D = 2: 4ms seek time + x/0.002 ms to read x GBs of data

200 MB/s = 0.2 GB/s

20 GBs / 0.2 GB/s = 100 seconds to read 20 GBs

Time it takes to read 20 GBs and sort it in memory= 4 ms + 100s = 100.004 s

Time it takes to sort the entire list of 320 GBs = 100.004 s \* 16 = 1600.064 seconds