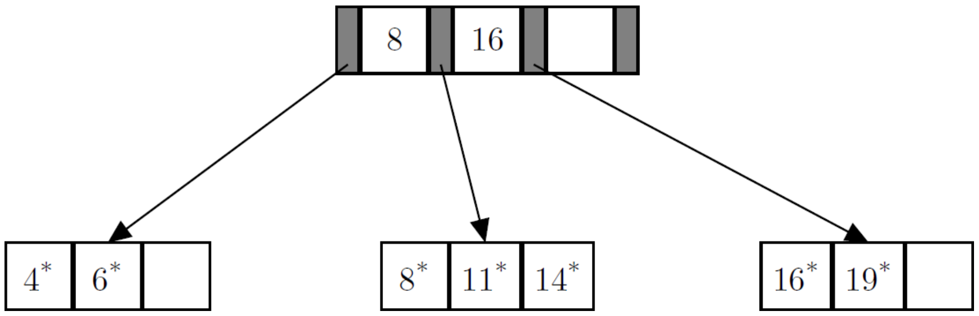
Name: Rahul Purushottam Gaonkar (rpg283)

Problem Set 4

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:



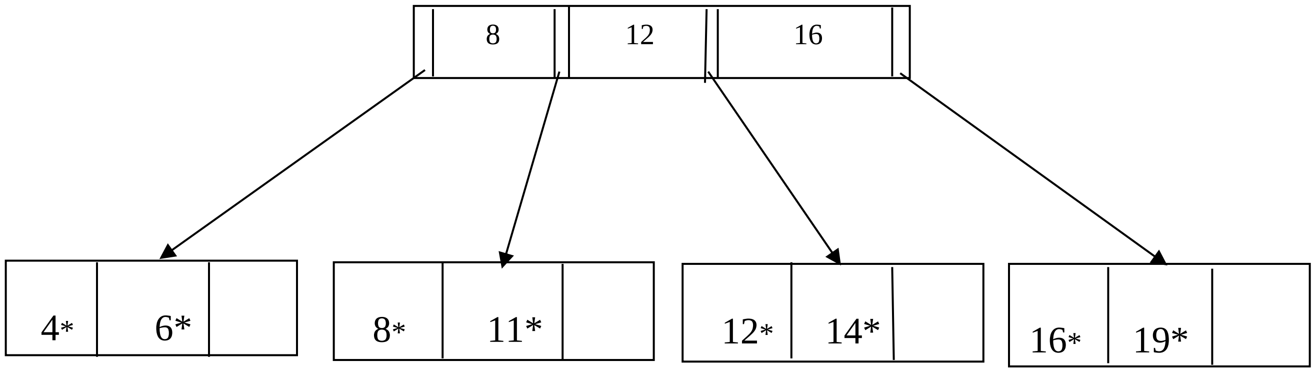
This tree consists of only a root node and three leaf nodes. Recall that the root node must have between 2 and *n* child pointers (basically RIDs) and between 1 and *n*-1 key values that separate the subtrees. In this case, the values 8 and 16 mean that to search for a value strictly less than 8 you 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 12, Insert 14, Insert 13, Delete 12, Insert 25, Insert 30, Insert 27, Delete 8, Delete 16

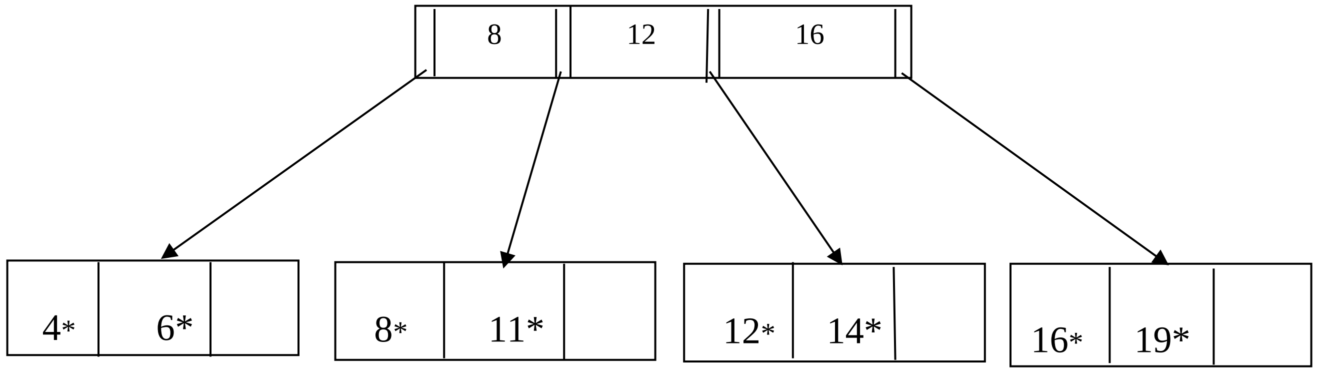
Note that for insertions, there are two algorithms, one that splits a full node without trying to off-load data to a direct neighbor, and one that first tries to balance with a direct neighbor in the case of a full node. Please use the first algorithm!

Solution:

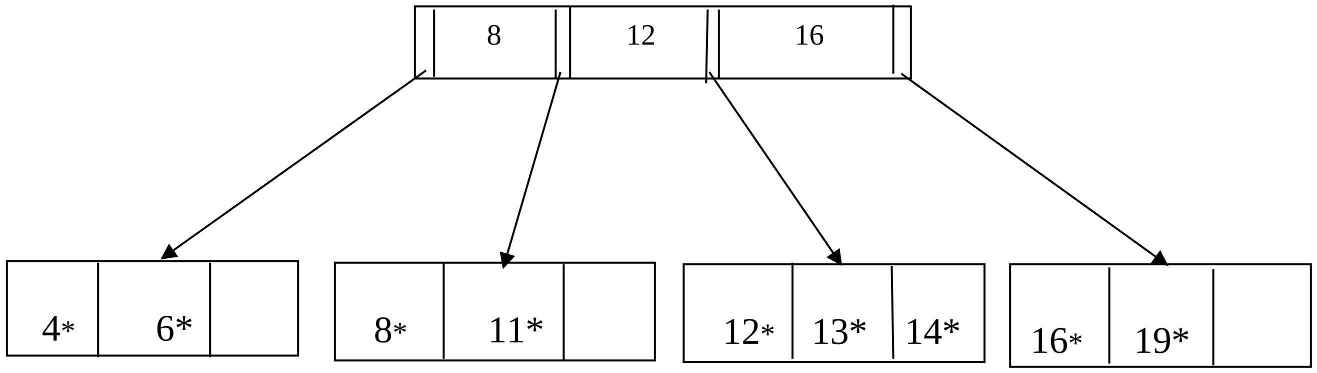
Step 1: Insert 12



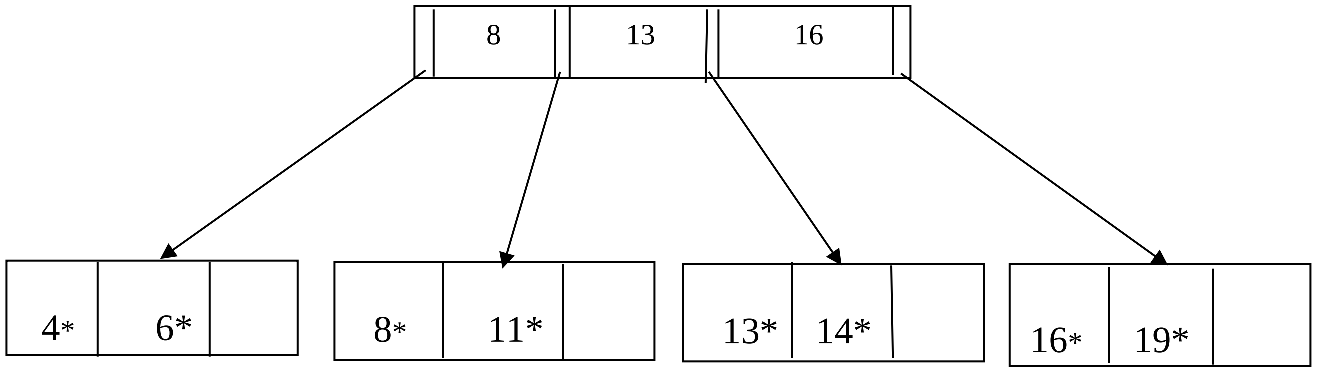
<u>Step 2</u>: Insert 14
Since 14 is already present in the B+ tree therefore no change in the index structure. A new record would be added though in the file(bucket) which is pointed by the key 14 in the index structure.



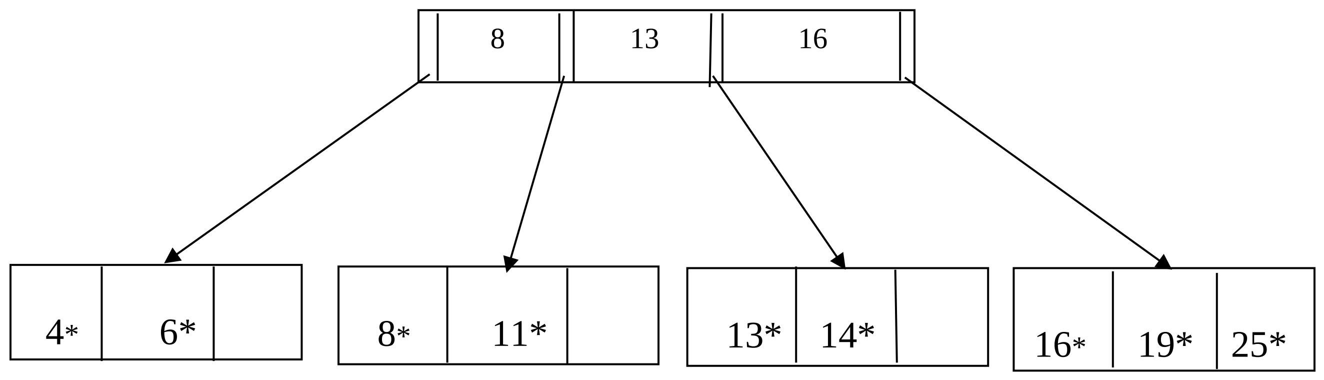
<u>Step 3</u>: Insert 13



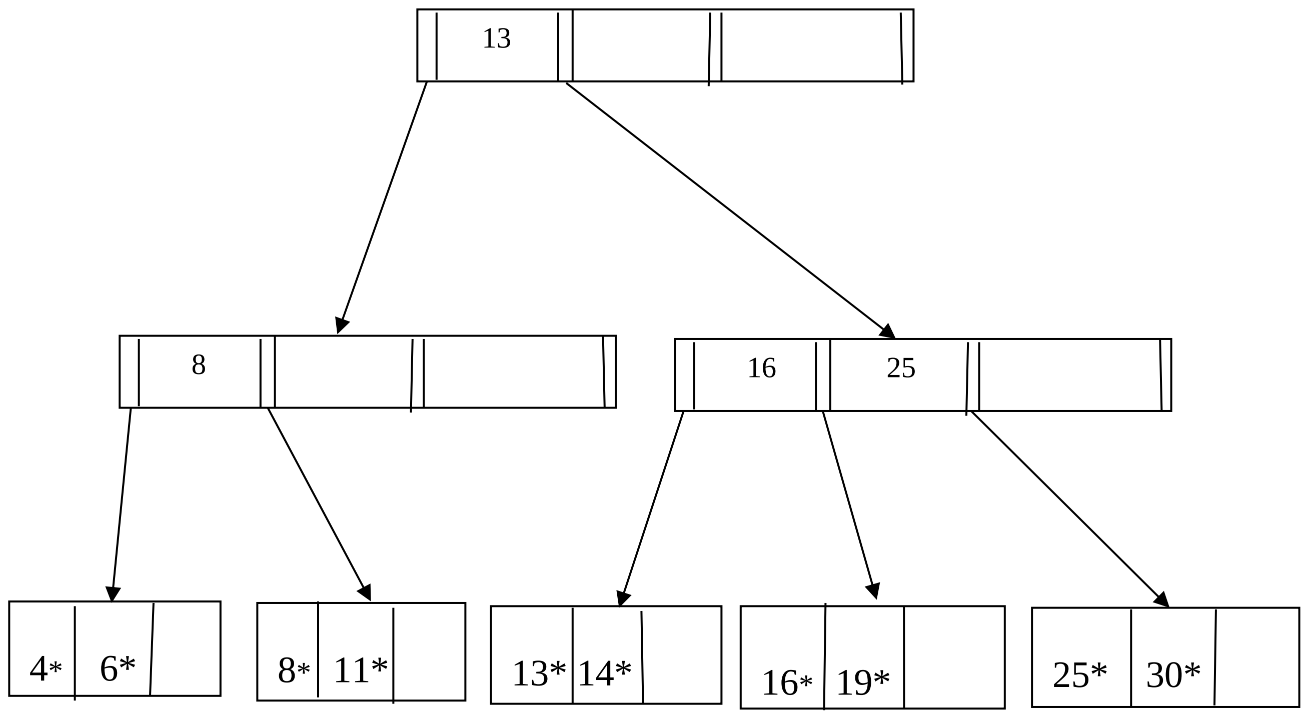
<u>Step 4</u>: Delete 12



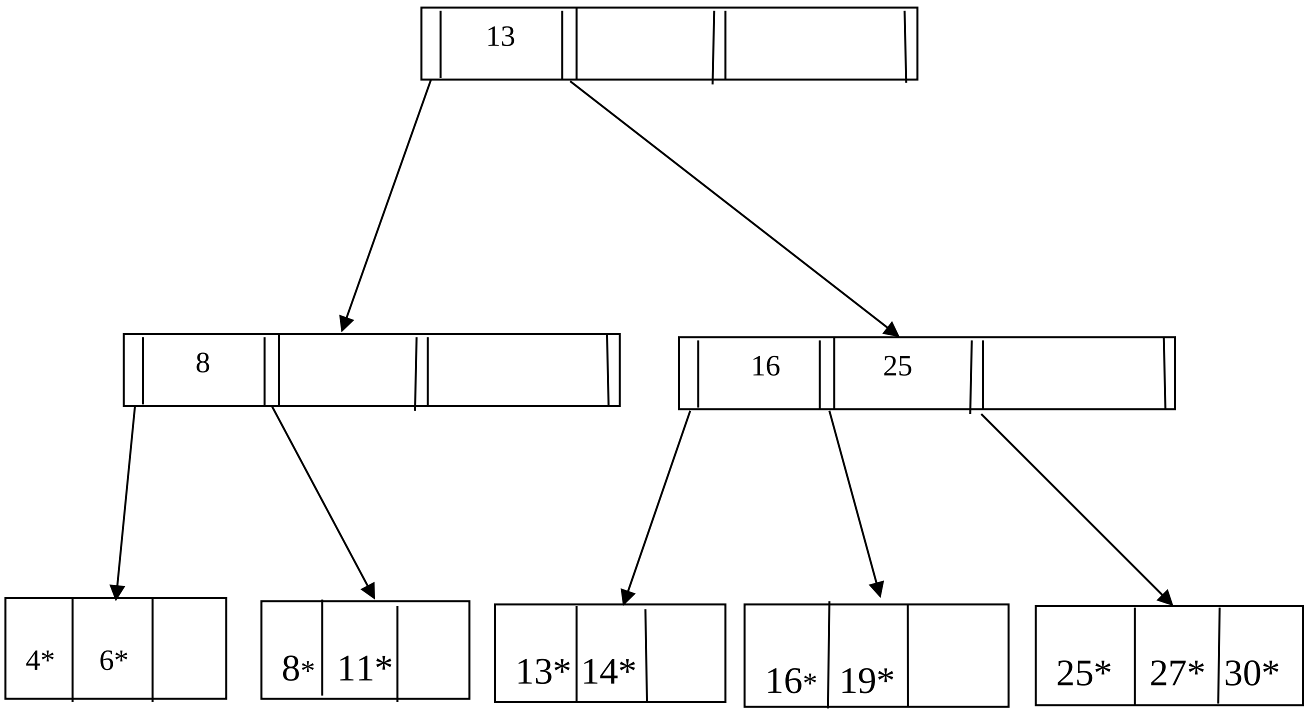
Step 5: Insert 25



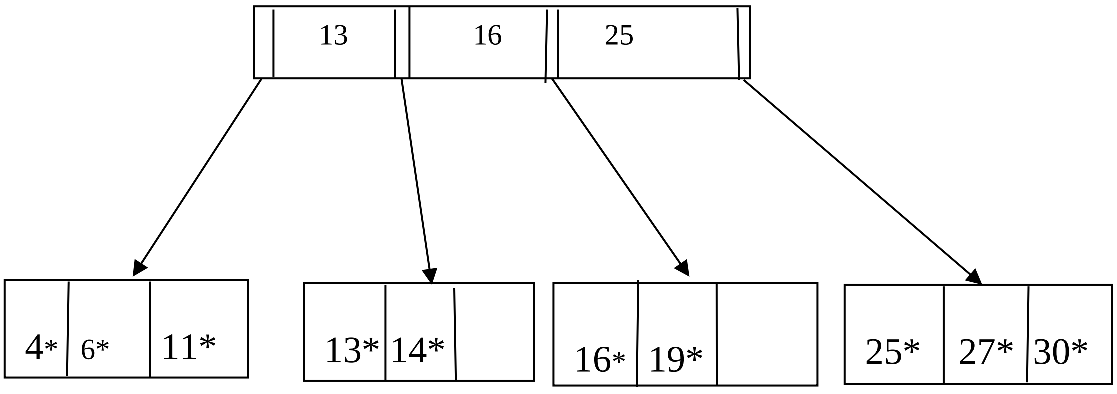
Step 6: Insert 30



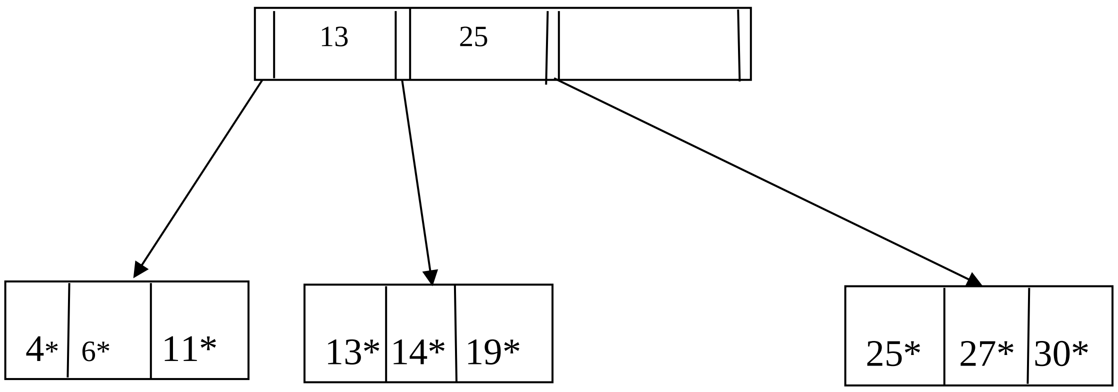
<u>Step 7:</u> Insert 27



<u>Step 8:</u> Delete 8



<u>Step 9</u>: Delete 16



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	00100100
k1	10100101
k2	01110010
k3	10101010
k4	10011110
k5	11010001
k6	11100110
k7	11111001

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

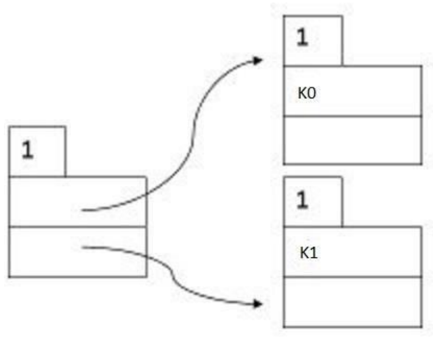
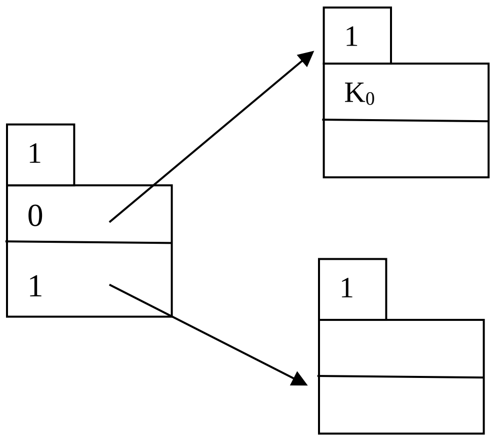


Figure 1: Hash Table state after insertion of the first two keys

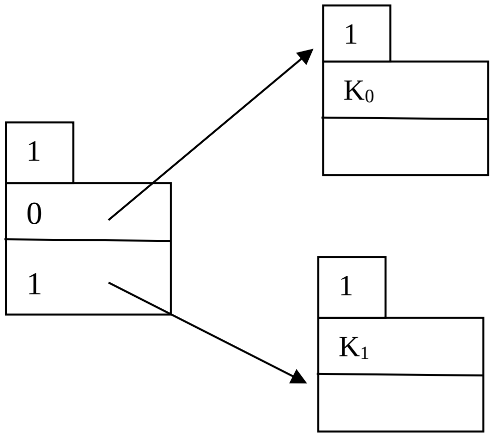
Solution:

Starting by assigning values to bucket by checking only one leftmost bit of the hash value (bucket size is 2)

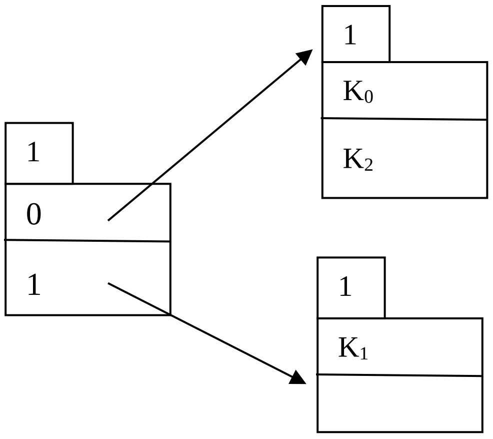
Step1: Inserting key K_0



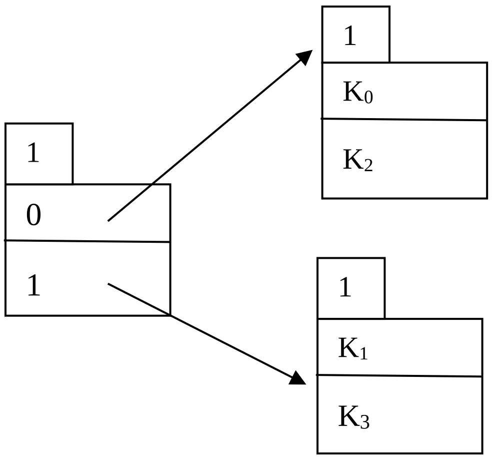
Step2: Inserting key K_1



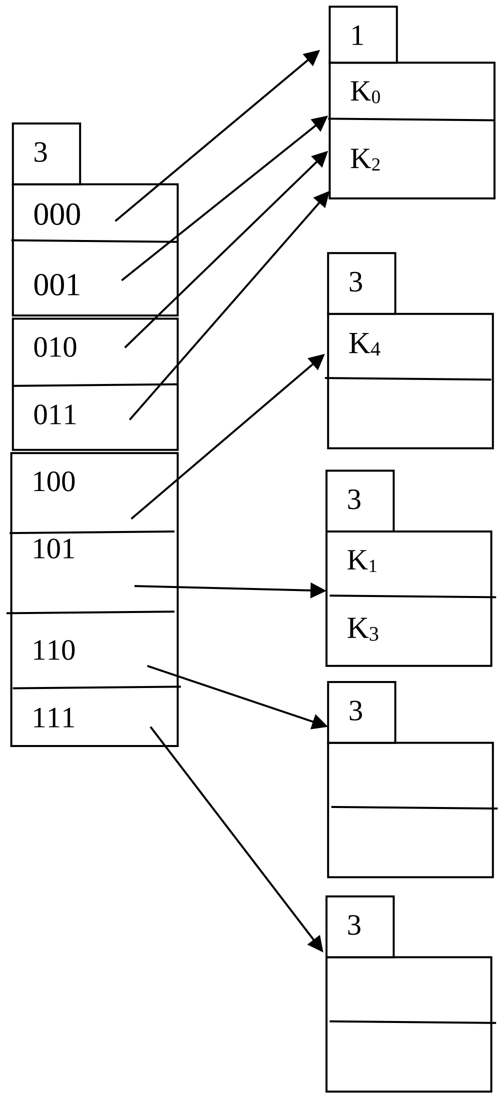
Step3: Inserting key K_2



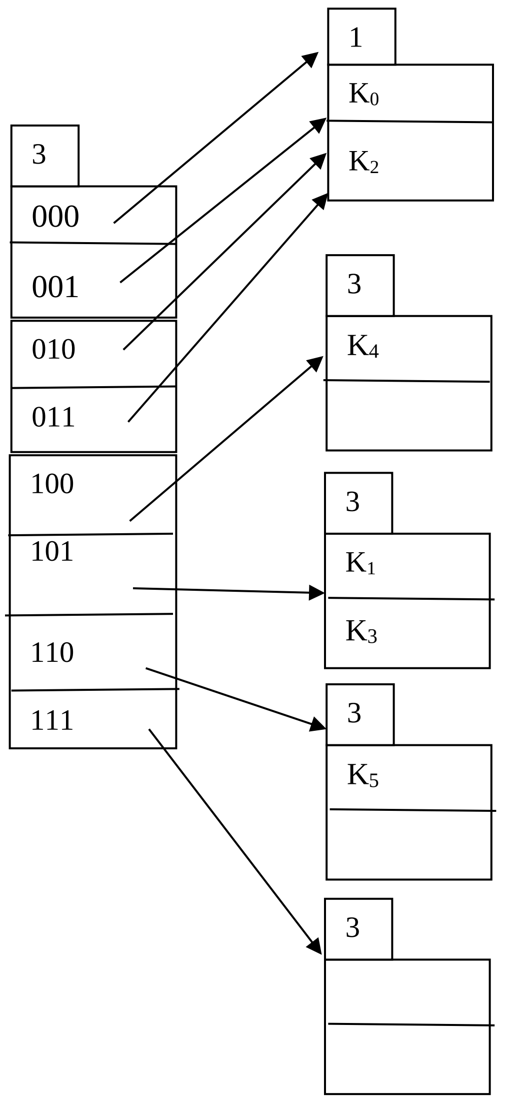
Step4: Inserting key K_3



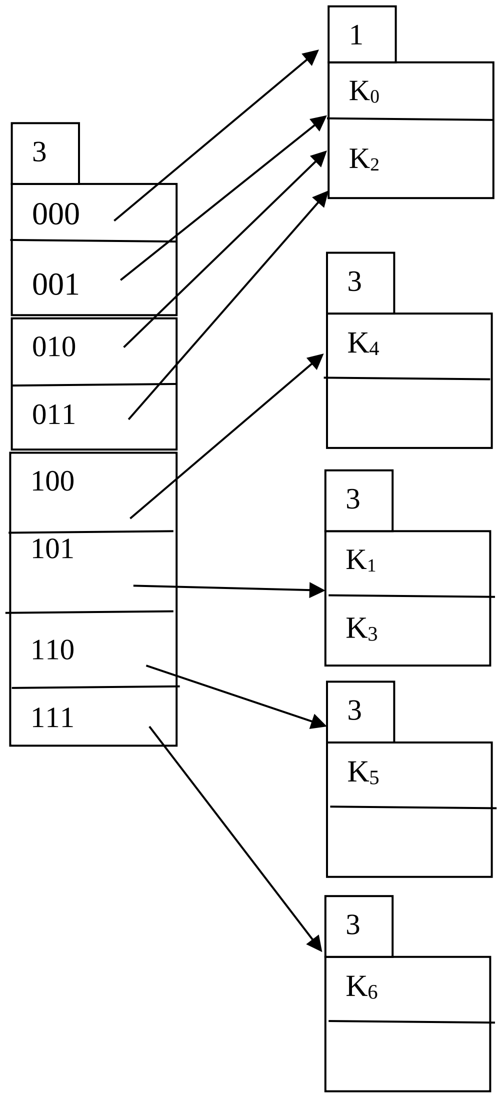
<u>Step5</u>: Inserting key K_4
Since K_4 will be assigned to bucket 1 which is full, we will have to split it into buckets 100,101,110 and 111 i.e. hash values will be assigned to these buckets if the first three leftmost bits of the hash value are either 100,101,110 or 111 respectively.



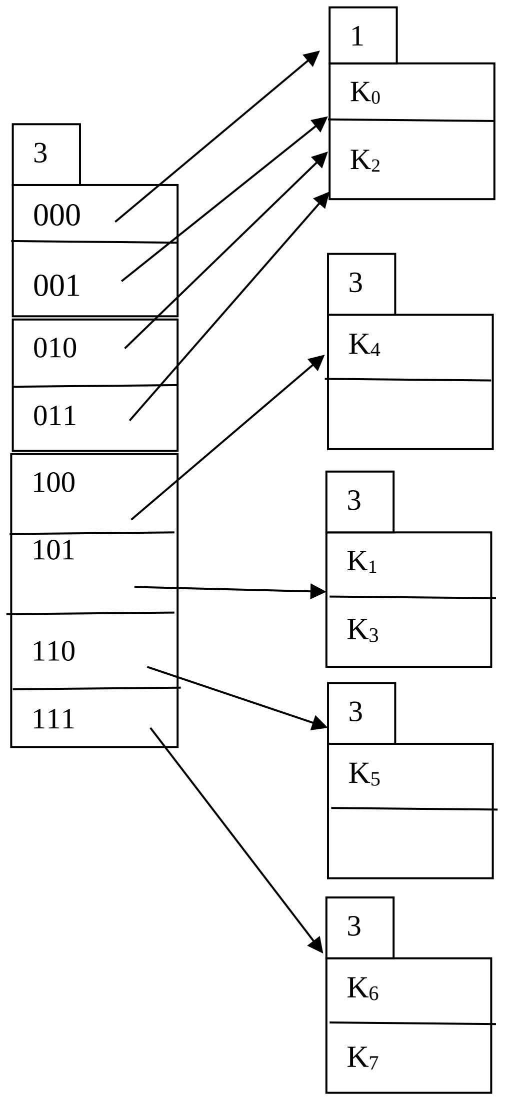
Step6: Inserting key K_5



Step6: Inserting key K_6



Step7: Inserting key K_7



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, we use the term RID (Record ID) to refer to an 8-byte "logical pointer" that can be used to locate a record (tuple) in a table.

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 (<u>uid</u>, uname, ucity, joindate)
Artist (<u>aid,</u> aname, genre)
Track (<u>tid</u>, aid, tname, tduration)
Play (<u>uid, tid, pdate, ptime</u>)

Assume there are 50 million users, 10 million tracks, 1 million artists, and 10 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 uid, uname
from User u, Play p
where u.uid = p.uid and pdate = "2017-11-29"

select uid, uname
from User u, Track t, Play p, Artist a
where u.uid = p.uid and t.tid = p.tid and a.aid = t.aid and genre = "Jazz"

select tid, tname
from User u, Track t, Play p
where u.uid = p.uid and t.tid = p.tid and ucity = "Chicago"

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

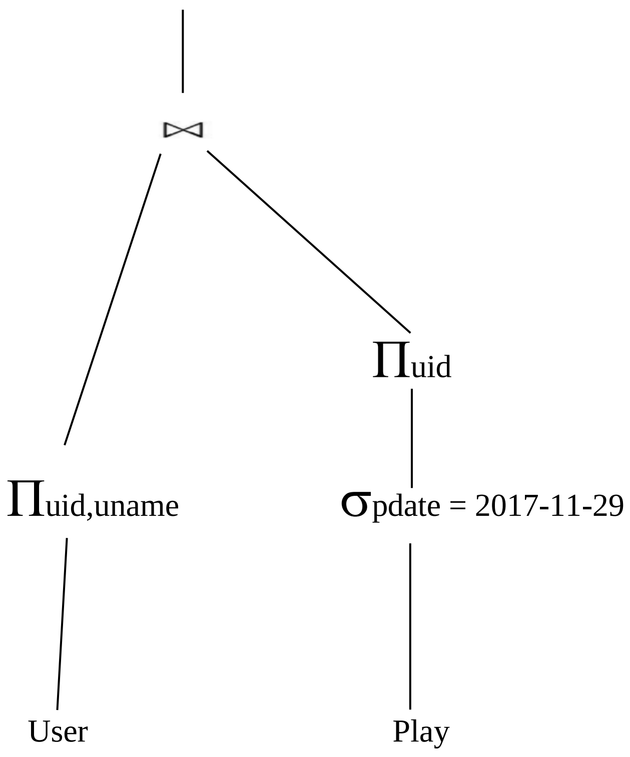
1. The first query displays the user ID and username of the users who have played tracks on 2017-11-29.
2. The second query displays the user ID and username of the users who have played tracks of Artist with genre Jazz.
3. The third query displays track ID and track name played by the users who live in Chicago.

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 500MB of main memory are available for query processing, and assuming a hard disk with 10ms for seek time plus rotational latency (i.e., a random access requires 10ms to find the right position on disk) and a maximum transfer rate of 100 MB/s. Assume that 1% users live in Chicago and 0.1% of the artists' genre is Jazz. Otherwise, assume that data is evenly and independently distributed; e.g.: 100 million playing records were made on November 28, 2017

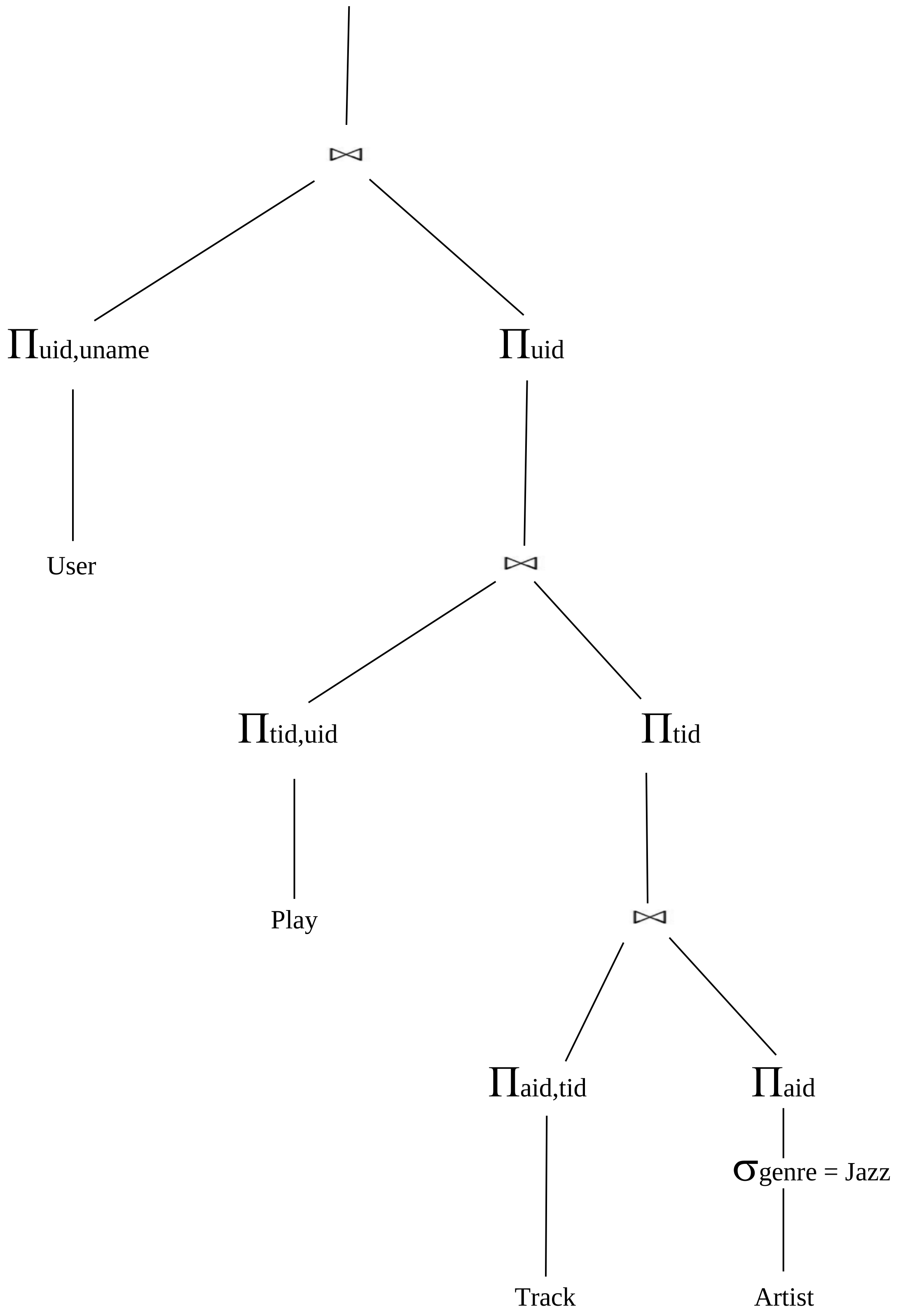
Ans:
Users table has 50 million records each of 100 bytes. Total Size = 5GB
Tracks table has 10 million records each of 100 bytes. Total Size = 1GB
Artist table has 1 million records each of 100 bytes. Total Size = 100MB
Play table has 10 billion records each of 40 bytes. Total Size = 400GB
Main Memory Size = 500MB
Access Time of Hard Disk = 10ms
Maximum Transfer Rate = 100MB/s
Number of records of Users living in Chicago in the User table = 1% of 50 million = 5,00,000
Total Size of the Tuples of Users in the User table who live in Chicago = 50 MB
Number of records of Artist in the Artist table with genre Jazz = 0.1% of 1 million = 1000
Total Size of the tuples of Artist in the Artist table with genre Jazz = 100KB
As the other data is evenly and independently distributed and 100 million playing records were made on November 28,2017. Therefore, 100 million records were also made on November 29,2017.
Total Size of the tuples of play with pdate as 'November 29,2017' = 4GB

1. **select uid,**
   uname from
   User u, Play p
   where u.uid = p.uid and pdate = “2017-11-29”



Explanation: We first scan the play table to find all the tuples with pdate = ‘2017-11-29’. There are 100 million such tuples. On an average every user has 200 records in play table therefore we can expect 5,00,000 tuples of User table to join with the 100 million tuples of the play table. We just need the uid attribute from the play table which occupies 10 bytes of every 40-byte play record. Therefore, total size of 100 million tuples is 1GB which is twice that of the main memory size. So here we use block nested loop join where initially the first half of the required data of Play table is scanned which fits in the main memory and then the User table is scanned once after that the second half of the required data of Play table is scanned and then the User table is scanned again. So total cost will be scanning the Play Table once and scanning the User table twice.
Time taken to read the first half of the Play table (200GB data) = 10 + ((200*10^6)/100) = 10 + 2000000 = 2000010ms = 2000.01 s
Time taken to read the entire Play table (400GB data) = 2000.01*2 = 4000.02 s
Time taken to read the entire User table (5GB data) = 10 + ((5*10^6)/100) = 10 + 50000 = 50010ms = 50.01 s
Time taken to read the entire User table twice = 50.01*2 = 100.02 s
Total cost to scan the data required by the query is 4000.02 + 100.02 = 4100.04 s

2. select uid, uname
from User u, Track t, Play p, Artist a
where u.uid = p.uid and t.tid = p.tid and a.aid = t.aid and genre = “Jazz”

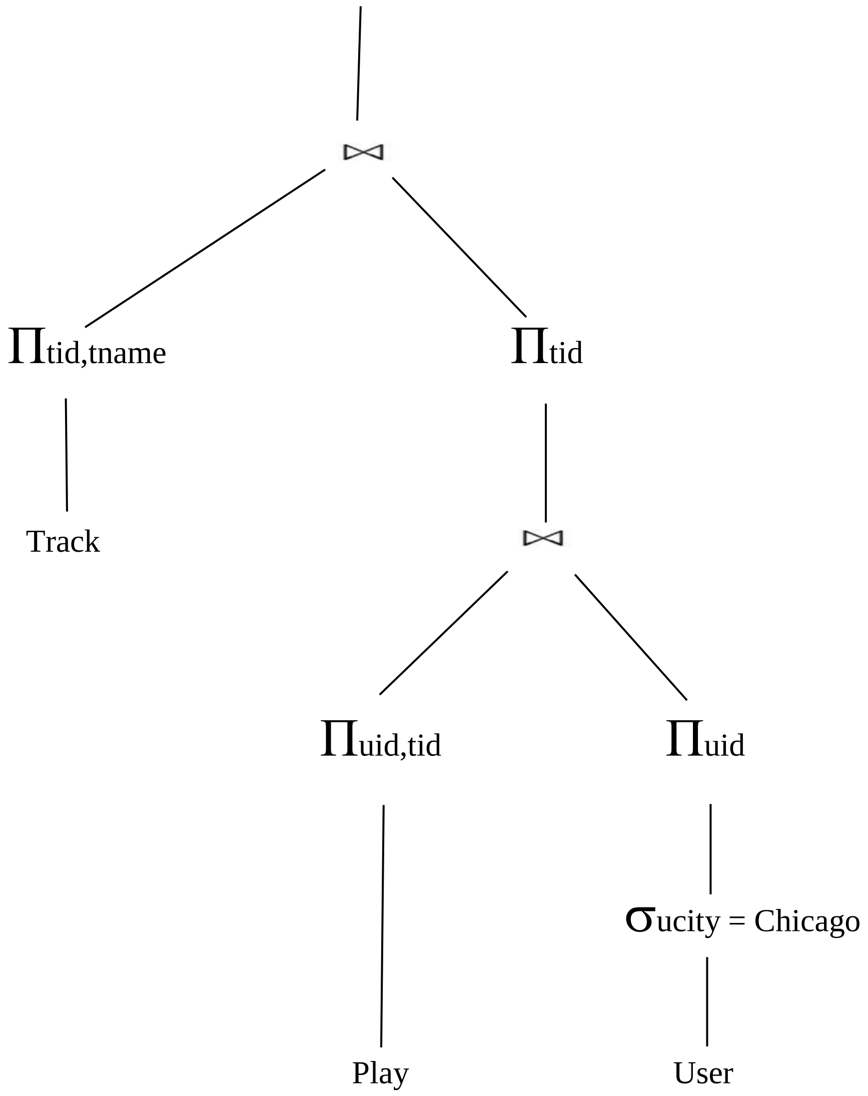


Explanation: On an average one artist will have 10 track records.1000 records of artist having genre Jazz will join with 10000 records from Track table. So, we first scan the Artist table for the required data (aid) which occupies say 20 bytes of the 100-byte artist record. So, the total size of 1000 records is 20KB which fits in the main memory. Now the track table is also scanned once to join with the records of the Artist table since the required data from the Track table (aid, tid) occupies 40 bytes of the 100-byte track record. So, the total size of

10000 records is 400KB which also fits in the main memory. This is then joined with the play table. The required data from the play table (tid, uid) which occupies 14 bytes of 40 byte play record. On an average 1 track will have 1000 play records.10000 track records will have 10 million Play records. So, the total size of the Play Table will be 140MB which will fit in the main memory. So, the Play table is also scanned once in the join operation. This result is then joined with User table by scanning it once. So, total cost will be scanning Artist table once, Track table once, Play table once and User table once.

Time taken to read the entire User table (5GB data) = 10 + $((5*10^6)/100)$ = 10 + 50000 = 50010ms = 50.01 s
Time taken to read the entire Track table once (1GB data) = 10 + $((1*10^6)/100)$ = 10 + 10000 = 10010ms = 10.010s
Time taken to read the entire Artist table once (100MB data) = 10 + 1000 = 1010ms = 1.010s
Time taken to read the entire Play Table once (400GB data) = 10 + 4000000 = 4000010ms = 4000.010s
Total cost to scan the data required by the query is 50.01 + 10.010 + 1.010 + 4000.010 = 4061.04s

3. **select tid, tname**
   from User u, Track t, Play p
   where u.uid = p.uid and t.tid = p.tid and ucity = "Chicago"



Explanation: On an average every user has 200 records in play table therefore we can expect 5,00,000 tuples of User table having city as 'Chicago' to join with the 100 million tuples of the play table. Therefore, we scan the User table once and the required data (uid) fits in the main memory because uid just occupies say 20 bytes of the 100 byte user record, so the total size of the required data (5,00,000 tuples) from the User table will be around 10MB.The required data from the play table (uid, tid) will occupy say 14 bytes of the 40 byte play record, so the total size of the required data (100 million tuples) from the Play table will be around 1.4 GB which is about triple the size of the main memory. So, we just scan the one third of the required data from the Play table while performing join operation with User table and this is then joined with the Track table by scanning it. This process is repeated twice for the other two third part of the Play table. In this process the track table is scanned two more times. So, the total cost is to scan User table once, Play table once and Track table thrice.

Time taken to read the entire User table (5GB data) = 10 + ((5*10^6)/100) = 10 + 50000 = 50010ms = 50.01 s
Time taken to read 1/3rd of the Play table (133.333GB data) = 10 + ((133.333*10^6)/100) = 10 + 1333330 = 1333340ms = 1333.34s
Time taken to read the entire Play table (400GB data) = 1333.34*3 = 4000.02 s
Time taken to read the entire Track table once (1GB data) = 10 + ((1*10^6)/100) = 10 + 10000 = 10010ms = 10.010s
Time taken to read track table thrice = 10.010*3 = 30.030 s
Total cost to scan the data required by the query is 50.01 + 4000.02 + 30.030 = 4080.06 s

(c) Consider a sparse clustered B+-tree index on uid in the User table, and a dense unclustered 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?
Ans:
1.
<u>Sparse clustered B+-tree index on uid in the User table:</u>
Each node of the tree contains n-1 key values and n pointers. Each record pointer is of 8 bytes and let's consider each key value (uid) is of 20 bytes. Each node is of size 4096 bytes, therefore total number of keys and pointers that fit in a node can be calculated as follows (n-1) *20 + n*8 = 4096 => n=147.Now considering 80% occupancy per node, n = 118.So we have 118 index entries in the leaf node and each internal node will have 118 children.
For the user table with size of each record as 100 bytes, we can fit 40 records in a disk page (disk block) assuming 100% occupancy by the relation. Therefore, in a sparse index, we will have one index entry for every 40 records in a disk page. So, we will have 1.25 million index entries. Since every leaf node contains 118 index entries, therefore total number of leaf nodes in the tree is 1250000/118 = 10,593 leaf nodes. The next internal level will have 10593/118 = 89 nodes and then we have the root of the tree. So, the B+ tree has three levels: root, one internal level and the leaf level. The height of the tree is 3. Thus, the time taken to fetch a single item is 4* (Time to read each 4KB block) = 4* (10 + ((4*10^3)/10^5)) = 4*(10 + 0.04) = 40.16ms using index structure assuming no caching.
The size of the tree: (10,593+89+1) *4 = 42732KB = 42.732MB

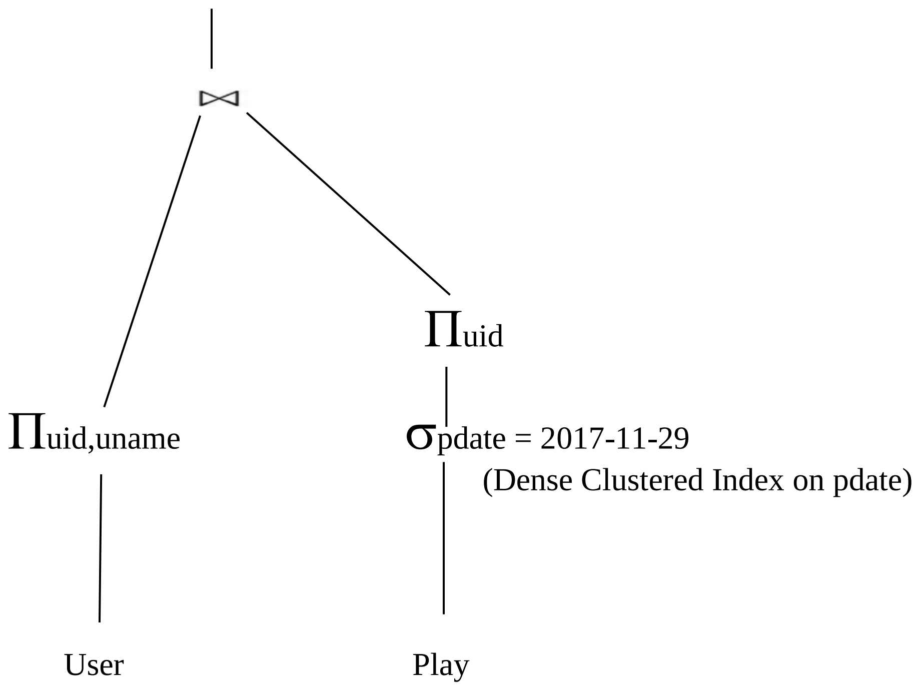
<u>Dense unclustered B+-tree index on tid in the Play table:</u>
Each node of the tree contains n-1 key values and n pointers. Each record pointer is of 8 bytes and let's consider each key value (tid) is of 10 bytes. Each node is of size 4096 bytes, therefore total number of keys and pointers that fit in a node can be calculated as follows (n-1) *10 + n*8 = 4096 => n=228.Now considering 80% occupancy per node, n = 182.So we have 182 index entries in the leaf node and each internal node will have 182 children.
We have 10 billion records in the Play table. Therefore, in a dense index, we will have one index entry for every record. So, we will have 10 billion index entries. Since every leaf node contains 182 index entries, therefore total number of leaf nodes in the tree is 54,945,055. The next internal level will have 301896 nodes, the next internal level will have 1659 nodes, the next internal level will have 9 nodes and then we have the root of the tree. So, the B+ tree has five levels: root, three internal level and the leaf level. The height of the tree is 5. Thus, the time taken to fetch a single item is 5*(10 + ((4*10^3)/10^5)) + 10 = 5*(10 + 0.04) + 10 = 60.2ms using index structure assuming no caching. The cost of fetching say 50 records would be the sum of 49 fetches of 49 other different records + Time taken to fetch a single item = 49*(10) + 60.2 = 550.2ms
The Size of the tree: (54,945,055 + 301896 + 1659 + 9 + 1) *4 = 220.994GB

(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.)

Ans:

1.

Evaluation Plan:

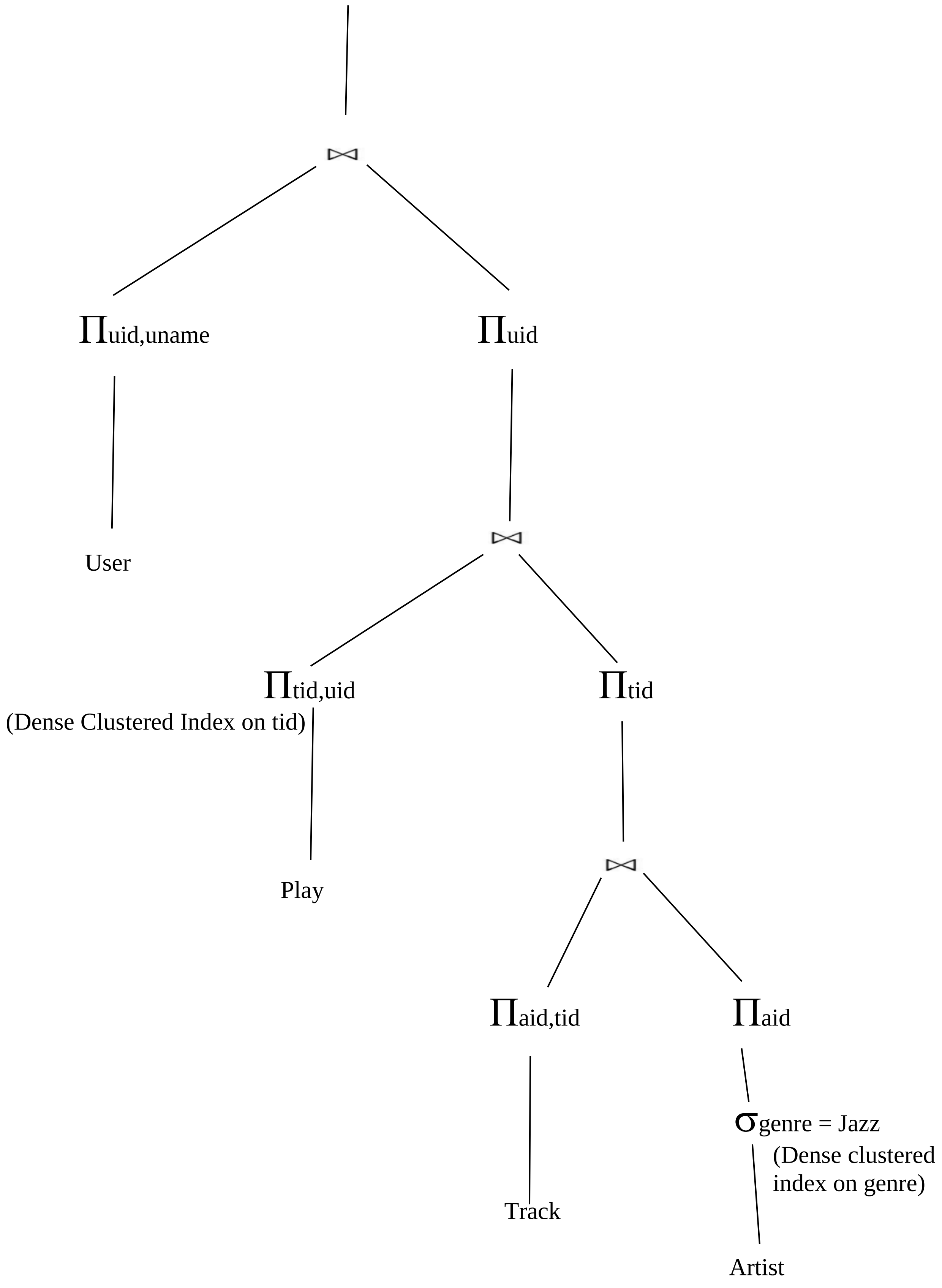

⋈
Π_{uid}
$\Pi_{uid,uname}$
$\sigma_{pdate = 2017-11-29}$
(Dense Clustered Index on pdate)
User
Play

Explanation:

We are using a dense clustered index on pdate in Play relation. This would result in faster lookup as we would be scanning 1% of the total data in Play relation which would take approx 8s. We will not create any dense index for the User table as it will involve 100 million lookups which will result in a cost of 25080s for fetching. So, we will just create a dense clustered index on pdate in Play relation.

2.

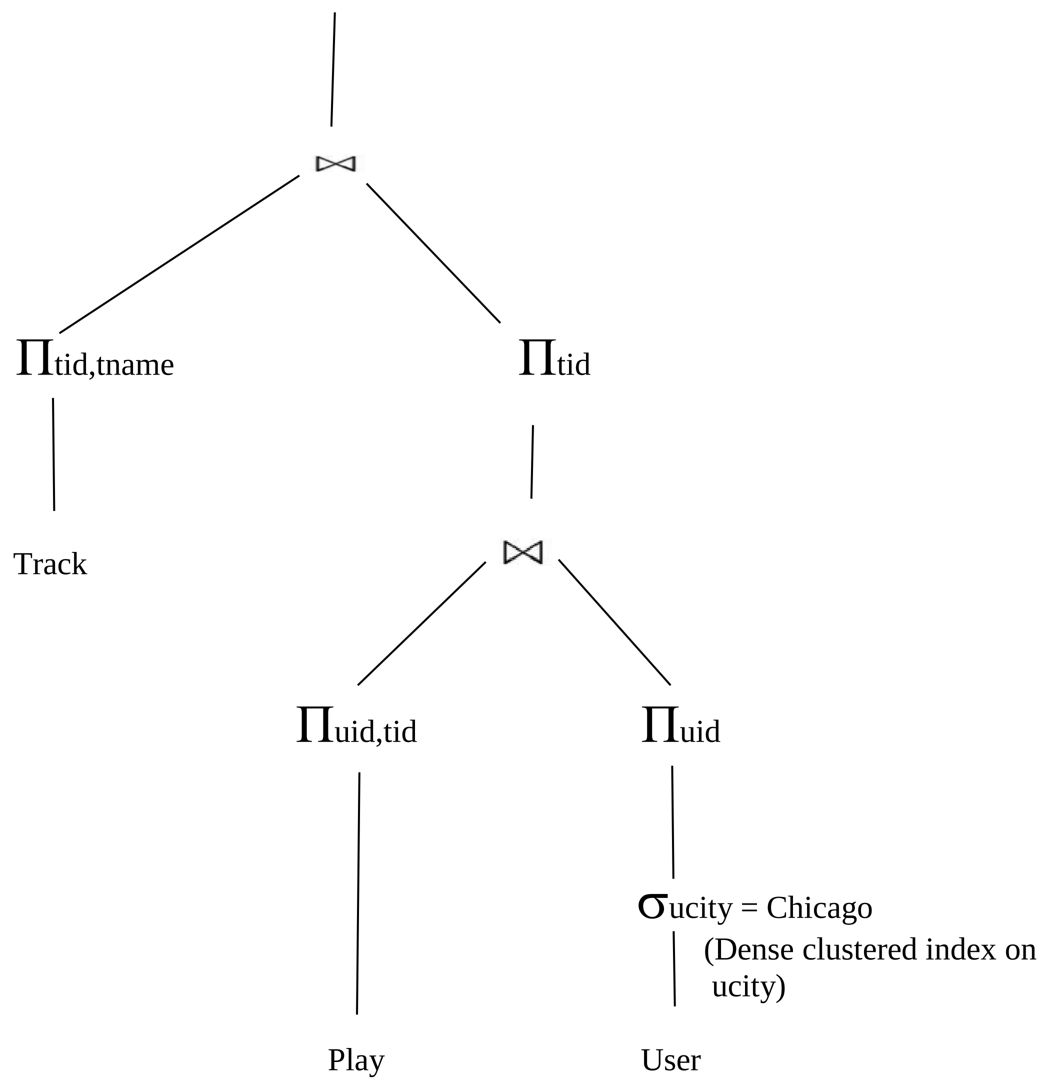
Evaluation Plan:



Explanation:
We will be using a dense clustered index on genre in Artist relation. This will result in faster fetching of records with genre =Jazz as we must read only 0.1% of the total data in Artist relation. We will also be

creating a dense clustered index on tid in Play relation. This will result in increasing the performance of join operation by reducing the time of fetching the records from Play relation to approximately 503 s. The cost of fetching records from other relations in the above query using index structure would be greater than then simply scanning the entire table. Therefore, the two index structures dense clustered index on genre in Artist relation and dense clustered index on tid in Play relation will make the execution of the query faster.

3.
Evaluation Plan:



Explanation:
We are using a dense clustered index on ucity in User relation. We will be getting faster lookups as we will be reading only 1% of the total data in User relation i.e. Users living in Chicago. We will then simply scan the Play table once and the track table twice to complete the join as creating a second index on any of these relations would be costly as compared to simply scanning the tables. Therefore, we will only create one index i.e. dense clustered index on ucity in User relation for the above query to make it faster.

Problem 4

(a) Consider a hard disk with 10000 RPM and 3 double-sided platters. Each surface has 300,000 tracks and 3000 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?

Ans: 10000 RPM -> 10000/60 RPS -> 60/10000 S/Rotation -> 6 ms/rotation

1. Capacity of the disk = 2*3*300000*3000*1024 = 5529.6 GB

2. Maximum Rate at which data can be read from disk = Total data on a single track / Time taken for one rotation = 1024*3000 / 6*10^{-3} = 1024*3*10^{6} / 6 = 512 MB/s

3. Average rotational latency is time taken to complete half rotation = Time taken to complete one rotation/2 = 6ms/2 = 3ms

(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 100 KB? How about a file of 1000 KB? How about a file of 100 MB? Use both the block model (4KB per block) and the latency/transfer-rate model, and compare.

Ans: Average seek time: 4ms
Average Rotational Latency = 3ms
Maximum Transfer Rate = 512MB/s

Block Model (4KB per block):

Time to read a single block = Average seek time + Average Rotational Latency + Time taken to Transfer a single block

$= 4 + 3 + ((4*10^3)/ (512*10^6)) = 4 + 3 + 0.0078 = 7.0078$ms

1. File of Size 100KB

Time taken to read entire file = (7.0078*100)/4 = 175.195 ms = 0.175195 s

2. File Size 1000KB

Time taken to read entire file = (7.0078*1000)/4 = 1751.95 ms = 1.75195 s

3. File Size 100MB

Time taken to read entire file = (7.0078*100*10^3)/4 = 175195 ms = 175.195 s

Latency/Transfer-rate model:

1. File of Size 100KB

Time taken to read entire file = Average seek time + Average Rotational Latency + Time taken to Transfer the entire file

$= 4 + 3 + ((100*10^3)/ (512*10^6)) = 7.1953$ ms

2. File Size 1000KB

Time taken to read entire file = Average seek time + Average Rotational Latency + Time taken to Transfer the entire file

$= 4 + 3 + ((1000*10^3)/ (512*10^6)) = 8.95312$ ms

3. File Size 100MB

Time taken to read entire file = Average seek time + Average Rotational Latency + Time taken to Transfer entire file

$= 4 + 3 + ((100*10^6)/ (512*10^6)) = 202.3125$ ms

As we can see that the time taken to read a file using the LTR model is less as compared to the block model because in block model disk access time is added each time to read a block even if it is in the same file. The performance of both the models will be same only in case of small random reads. For long sequential data reads as in files LTR would be a better model.

(c) Suppose you have a file of size 36 GB that must be sorted, and you have only 1 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.
Ans:
Time taken to read 1GB of data into main memory = Average seek time + Average Rotational Latency + Time taken to Transfer 1GB data
= 4 + 3 + ((1*10^9)/(512*10^6)) = 1960.125 ms = 1.960125 s
Time taken to read 36GB of data into main memory = 36*1.960125 = 70.56 s
Total time taken to sort data into 36 files (read + write) = 70.56*2 = 141.129 s
d-way Merge Phase: (Single Merge Pass)
d = 36
There are 37 buffers (36 input buffers and 1 output buffer) of 27.77 MB each.
Time taken for reading/writing one buffer of data = 4 + 3 + 54.253 = 61.253 ms
Time taken to read entire 36GB of data = (36*61.253*10^3)/27.77 = 79.406 s
Total time taken to complete the Merge Phase (read + write) = 79.406*2 = 158.813s
Total estimated running time of the I/O-efficient merge sort algorithm = 158.813 + 141.129 = 299.94 s

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

Ans:
d-way Merge Phase: (Two Merge Pass)
d = 6
There are 7 buffers (6 input buffers and 1 output buffer) of 142.85 MB each.
Time taken for reading/writing one buffer of data = 4 + 3 + ((142.85*10^3)/512) = 4 + 3 + 279.003 = 286.003 ms
Time taken to read entire 36GB of data = (36*286.003*10^3)/142.85 = 72.076 s
Total time taken to complete a first pass of the Merge Phase (read + write) = 72.076*2 = 144.152 s
Total time taken to complete both the passes of the Merge Phase = 144.152*2 = 288.305 s
Total estimated running time of the I/O-efficient merge sort algorithm = 288.305 + 141.129 = 429.434 s