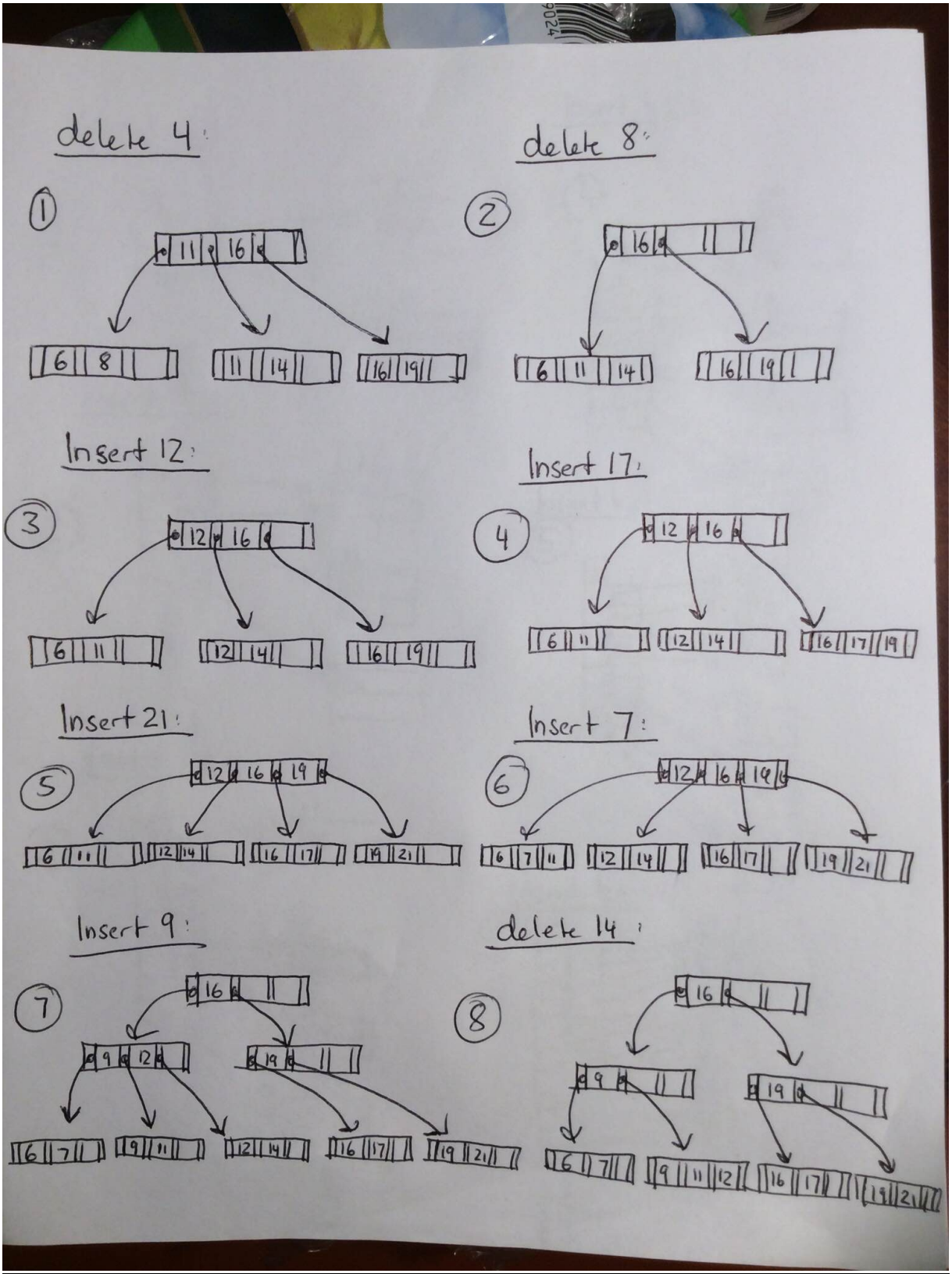
NYU School of Engineering
Computer Science and Engineering
CS6083, Fall 2015

December 19, 2015

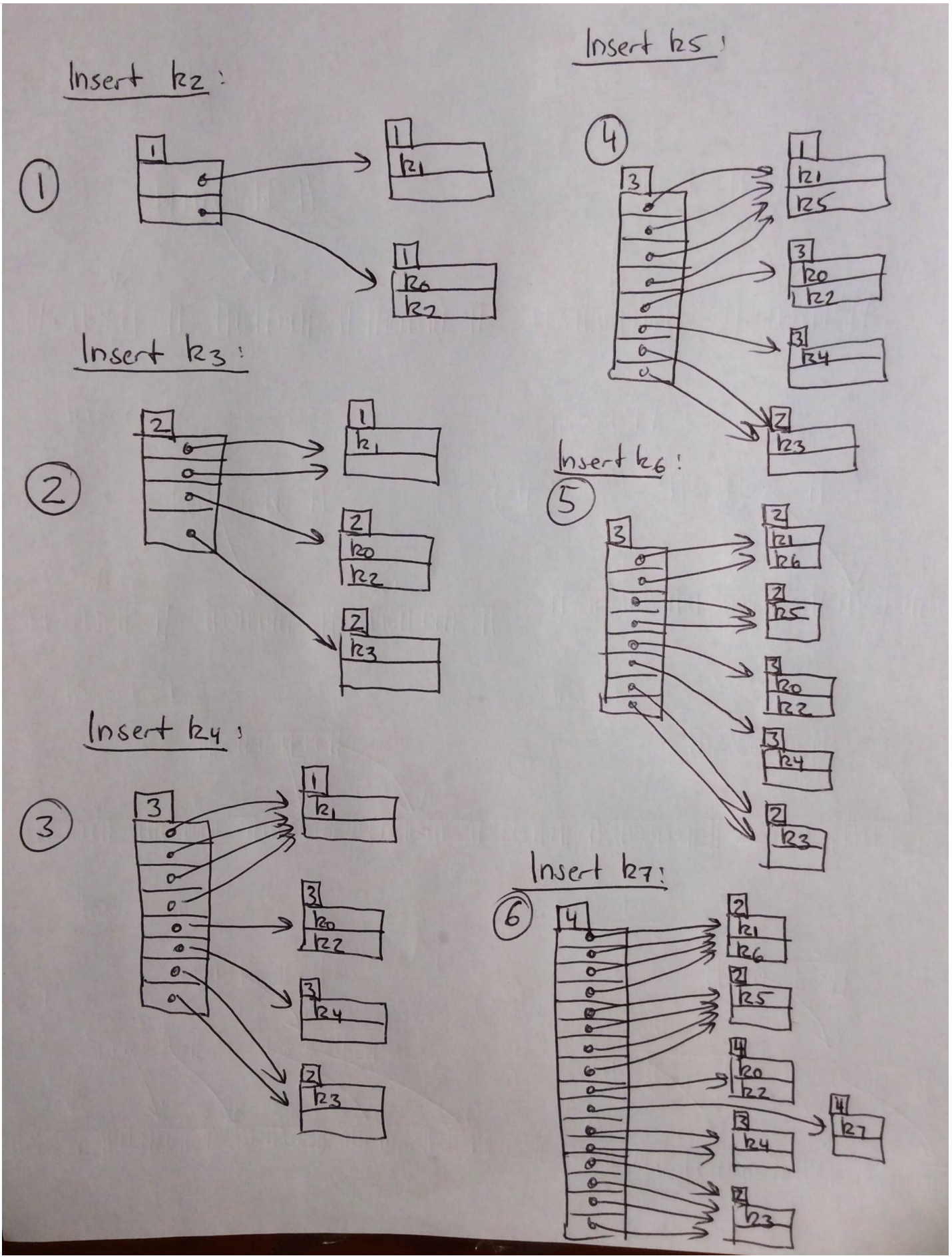
Sample Solution for Problem Set #4

Problem 1:



Note that when we insert 9, we put 3 leaves into the left child of the root, and 2 in the right. Could also be the reverse.

Problem 2:


Insert k2:
1
1
k1
1
k0
k2
Insert k3:
2
1
k1
2
k0
k2
2
k3
Insert k4:
3
1
k1
3
k0
k2
3
k4
2
k3
Insert k5:
4
3
1
k1
k5
3
k0
k2
3
k4
2
k3
Insert k6:
5
3
2
k1
k6
2
k5
3
k0
k2
3
k4
2
k3
Insert k7:
6
4
2
k1
k6
2
k5
4
k0
k2
4
k7
3
k4
2
k3

Problem 3:

a)

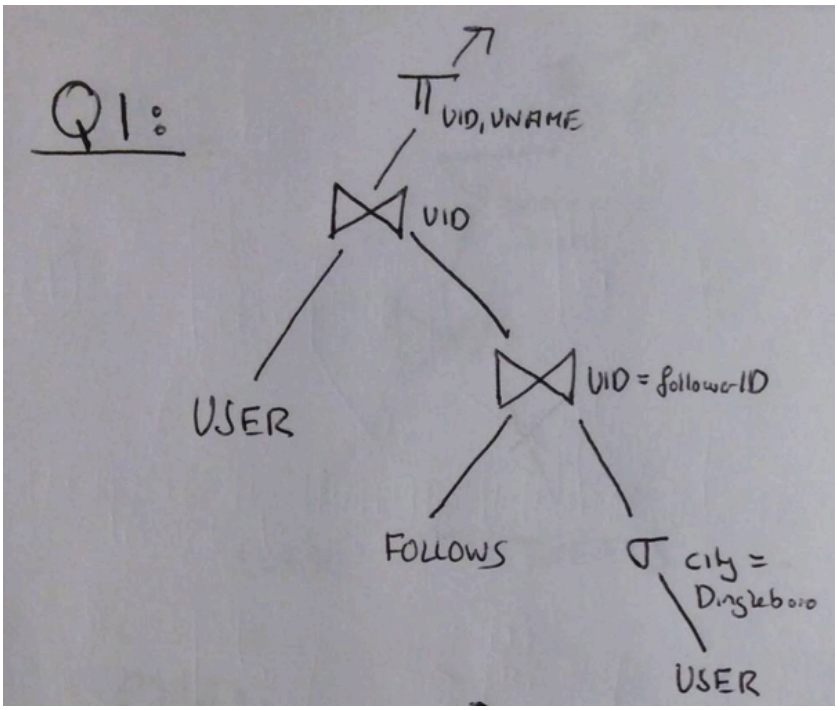
Query1: “List the IDs and names of all users who have a follower who lives in Dingleboro”.

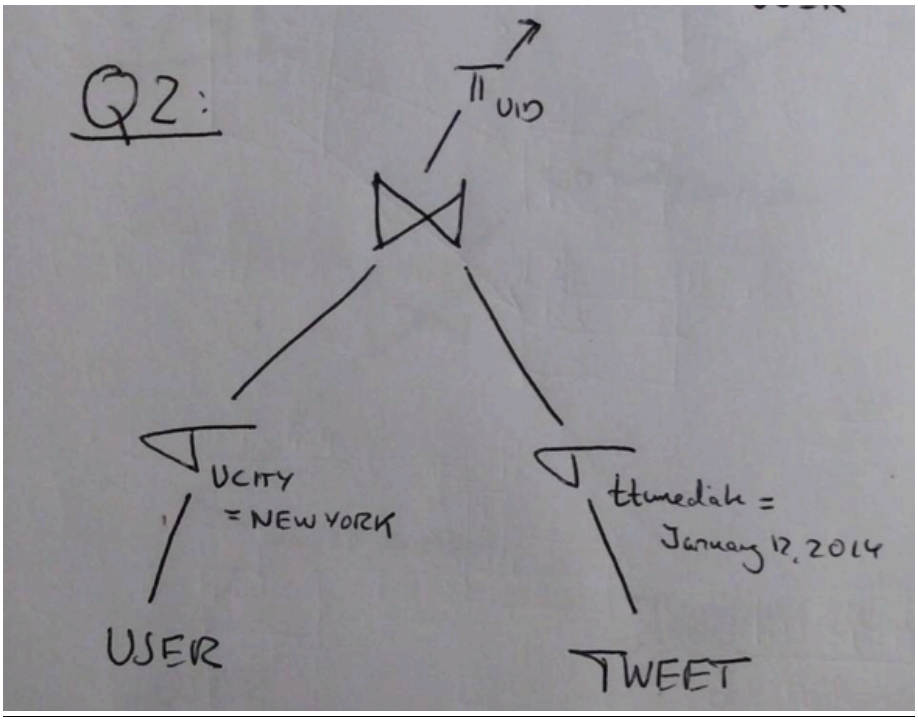
Query2: “List the IDs of all users who live in New York and made a tweet on January 12, 2014”.

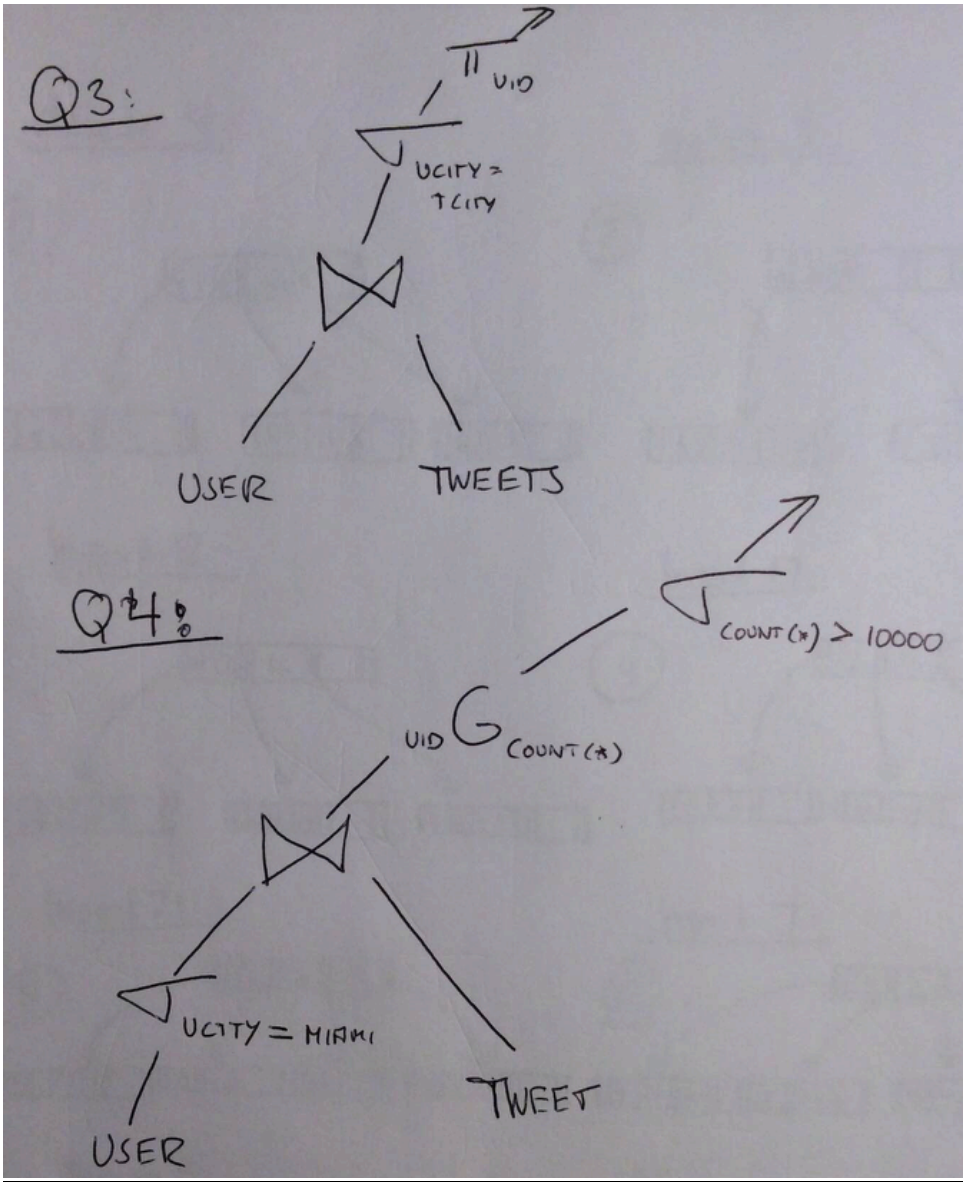
Query3: “List the IDs of all users who made a tweet in the city they live in”.

Query4: “List the IDs and number of tweets of all users living Miami who have made more than 10000 tweets”.

b) The query processing plans are shown below. Note that sort and hash-based joins are only used when even the smaller of the two inputs is much larger than memory, which in this case never happens. Thus, all joins are blocked nested-loop joins.







Query 1: We scan User to find users living in Dingleboro. There are 100 such users. On average, every user follows 10 other users, so we would expect about 1000 tuples in the Follows table to join with these 100 users. We do the join by placing the 100 user tuples for Dingleboro in main memory and then scanning Follows once, giving us 1000 tuples that clearly also fit in main memory. (Note that we actually only need the uid attributes of these tuples for the next join.) So, we place these in main memory and then join User again to do the join. So total cost is that of scanning User twice and Follows once. Thus, we scan a total of 6 billion tuples of 100 bytes, or 600GB, which takes about 6000s with our disk.

Query 2: We first check how many tuples will survive the selections: About 5 million users live in New York, and these tuples take about 500MB size. However, about 500 million tweets, taking 100GB in size, were done on January 12, 2014, which would not fit in memory. Thus, we first scan User to filter out users living in New York, and place those in main memory. (Actually, we only need their uids.) Then we scan Tweet once, checking which tweets were done on January 12, 2014, and then looking if those tuples match any of the surviving User tuples. Then we output the uids. Thus, we scan User and Tweet once, for a total of 50GB + 100,000GB, costing 1,000,500 seconds, or about 12 days.

Query 3: This query is tough to execute since the selection can only be done after the join. The User table is of size 50GB and thus larger than main memory. However, we only need the uid and ucity of each user, which would probably fit in about 20 of the 100 bytes of a User tuple. So this brings size down to maybe 20GB, meaning that half of the required data from User fits in main memory. Thus, we perform a block-nested loop join where we read half of User, then scan Tweet once, and then read the other half of User, and scan Tweet one more time. The subsequent selection can be done

on the fly and does not require any additional I/O. So the total amount of data to be read is 50GB + 2*100,000GB, costing 2,000,500 seconds, or about 24 days.

Query 4: We first scan the user table for those users who live in Miami, which leaves us with 500,000 tuples. These fit in memory. We then scan Tweet once to join users in Miami with their tweets, and then do the group-by and aggregation on the fly, by having one counter for each user in Miami (clearly 500,000 counters fit in memory), followed by selection on the counter values. So the cost is that of scanning User and Tweet, which is 1,000,500 seconds, as in the second query.

c)

Sparse clustered index on uid in the User table:

Each tree node contains n-1 keys and n pointers. With 16 bytes per key, 8 bytes per pointer and a node size of 4096 bytes, we can find how many keys and pointers can fit in each node: (n-1) * 16 + 8 * n = 4096 => n = 171. Assuming 80% occupancy per node, each leaf node will contain about 137 index entries and each internal node will have 137 children.

For the User table, 40 records fit into each disk page, assuming 100% occupancy in the disk blocks used by the relation. Thus a sparse index on the User table will have one index entry for every 40 records, or a total of 12.5 million index entires. Since about 137 index entries are in each leaf node there will be about 250,000/137 ~ 91240 leaf nodes. On the next level there will be about 91240/137 = 666 nodes, and then the next level has 5 nodes, and then we have the root of the tree. So the B+ tree has 4 levels of nodes: the root, two internal levels, and the leaf level. Thus is takes about 5 * 10 ms = 50 ms to fetch a single record from the table using this index assuming no caching. The size of the tree is dominated by the leaf level, which is about 91240 * 4KB ~ 365MB.

Dense unclustered index on (uid, tcity) in the Tweet table:

Each index entry now has a 16-byte uid, a (approximately) 20-byte city, and an 8-byte RID, and thus takes 44 bytes. Thus n ~ 93, and with 80% occupancy we get about 75 entries per node.

There are 500 billion index entries, and thus there are 6.666 billion nodes at the leaf level, 88.888 million nodes at the next level, then 1,185,185, then 15802, then 211, then 3, then the root. Thus the tree has 7 levels of nodes and 8 * 10 ms = 80ms is needed to fetch a single record from the table. The cost of fetching 50 records would involve 49 additional seeks into the underlying table, adding 49*10 = 490ms to the 80ms for a single record. The size of the tree is dominated by the leaf level, which is about 6.666 billion * 4KB = 26.66 TB.

d)

Query 1: We would choose a clustered index on ucity in User, to accelerate the fetching of users living in Dingleboro, reducing that cost to a fraction of a second. We could then add a clustered index on followerID in Follows, to reduce the cost of the join to that of 100 lookups. So we would basically be left with the cost of scanning User once in the final join, about 500 seconds. If we could have a third index, we would choose an unclustered index on uid in User, to also reduce the cost of the final join, performing instead about 1000 lookups for the 1000 users followed by users in Dingleboro.

Query 2: We choose clustered indexes on ucity in User and on ttimedate in Tweets. We would use the first index to fetch users in New York, which would take only 1% of the cost of a complete scan of User. Then we would "scan" Tweets during the join, as before, but only fetching tweets made on January 12, 2014, so the cost of accessing Tweets would be reduced by a factor of 1000, to 1005 seconds, or about 17 minutes.

Query 3: There seems to be no way to reduce the cost of this query using index structures. Maybe if we assume user is sorted by ucity and Tweet is sorted by tcity (which can be done by assuming clustered indexes), we could accelerate things a little bit by then doing a sort-based join on ucity=tcity, and then do the join on uid during that join. But this was not covered.

Query 4: For the fourth query, we can choose a clustered index on ucity in User, to accelerate the initial selection by a factor of 1000 (since 0.1% of all users live in Miami), to 0.5 seconds. Then we can accelerate the join with Tweet using a clustered index on uid in Tweets. This would result in 50000 lookups into the index, which is expensive, costing about 80ms per lookup, or about 4000 seconds total, but this is still much cheaper than scanning Tweets.

Problem 4

(a) The capacity of the disk is: 3*2*200,000*2000*512 bytes ~ 1,200,000 MB ~ 1.2 TB . The maximum rate of a read (using 6000RPM = 100RPS) is 100*2000*512 byte/s ~ 100 MB/s, and the average rotational latency is 5ms.

(b) Note that 100MB/s ~ 100KB/ms, so it takes x/100ms transfer time to read x KB of data after the initial 10ms for seek and average rotational latency.

Block Model:

Read 4KB: $t = 10 + 4/100\text{ms} \approx 10.04\text{ms}$

Read 20KB: $T = 5t = 50.2\text{ms}$

Read 200KB: $T = 50t = 502\text{ms}$

Read 20MB: $T = 5000t = 50.2\text{ s}$

LTR Model:

Read 20KB: $T = 10 + 20/100\text{ms} = 10.2\text{ ms}$

Read 200KB: $T = 10 + 200/100 = 12\text{ ms}$

Read 20MB: $T = 10 + 20000/100 = 210\text{ ms}$

Thus, the predictions by the LTR model are much faster (and more accurate) than those for the block model when reading large files.

(c) Phase 1: Repeat the following until all data is read: Read 1GB of data and sort it in main memory using any sorting algorithm. Write it into a new file until all data is read. The time to read 1GB is ~ 10ms + 10s = 10.01s. To read and write 80 such files takes 10.01 * 2 * 80 ~ 1600 s.

Phase 2: Merge the 80 files created in Phase 1 in one pass. The main memory is divided into 81 buffers. Each buffer is of size 1024/81 = 12.64 MB. For each buffer, the read and write time is 10ms + 12.64/100s = 136ms. 80 GB can be divided into 80*1024/12.64 = 6481 pieces. The total time is 136 * 6481 *2 ~ 1763s.

The total time for sorting the 240GB file in a single pass is about 1600 + 1763 = 3363s, or almost one hour.

(d) For an 8-way merge, we need 9 buffers, so each buffer is of size 114MB. Reading one buffer takes 1.14s, and reading all 80GB will take very close to 800 seconds, since the cost of seeks is very small compared to the cost of transfer. The same is also true for 10-way, 5-way, and 16-way merges, so the two options have almost exactly the same cost. A precise analysis would show the first option to be slightly better, but all options are worse than the one taken in part (c) where we have a single merge. (Note also that the cost of, say, a 5-way merge step does not depend on the ordering of the merge steps, so a 5-way followed by a 16-way has the same cost as 16-way followed by 5-way.