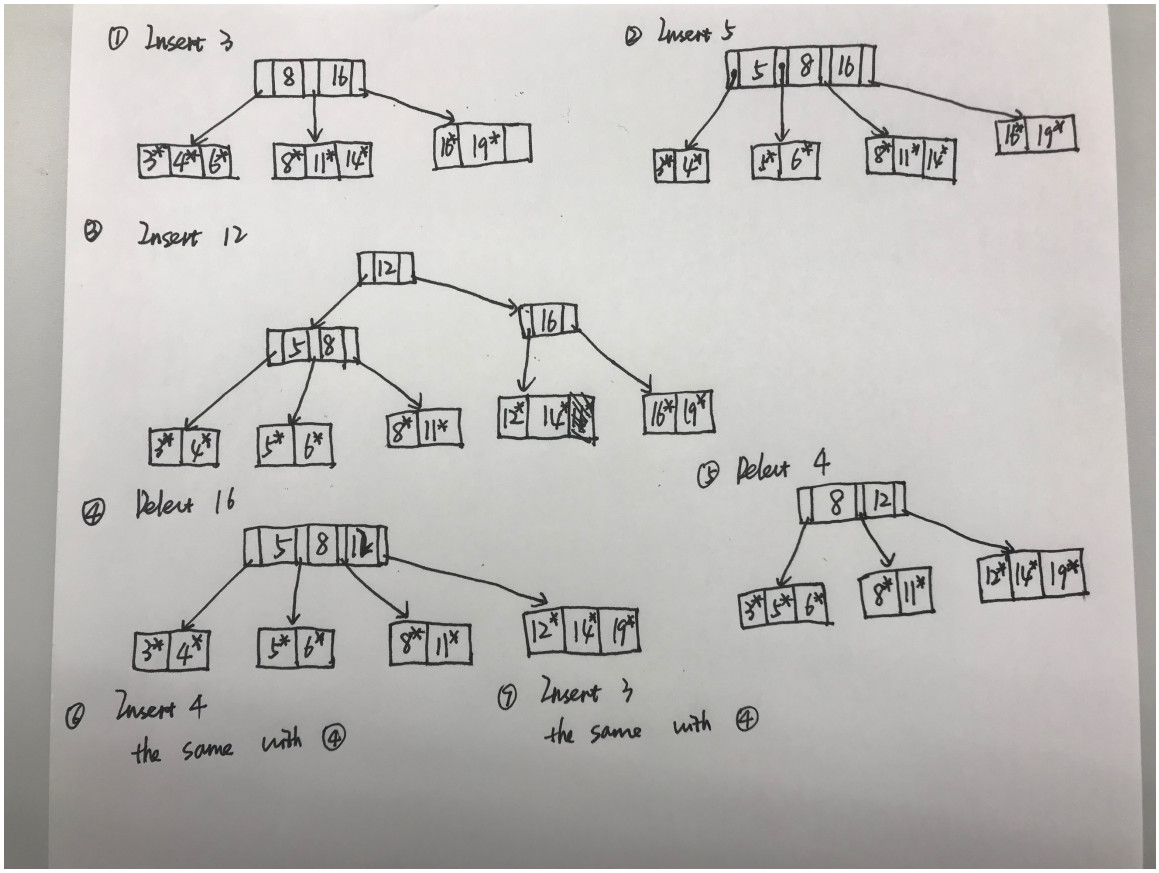
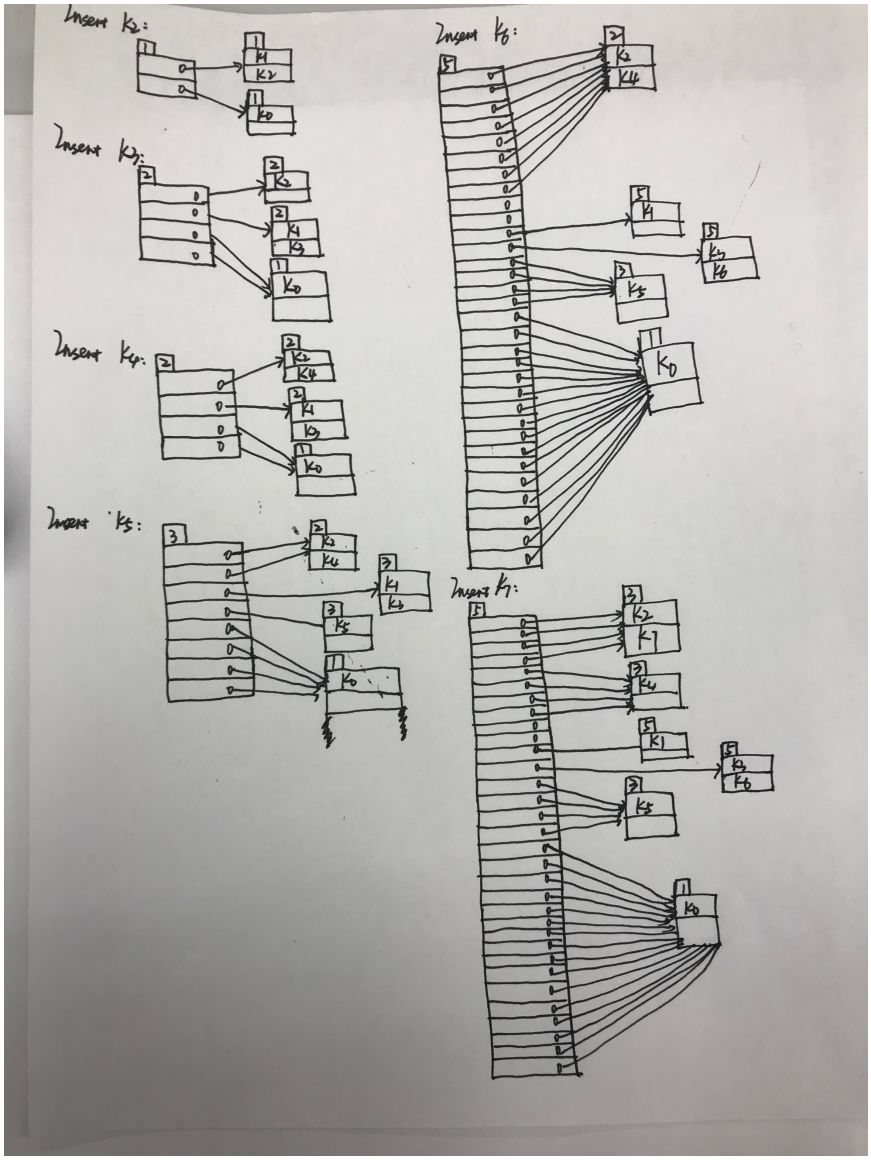
1.


① Insert 3
② Insert 5
③ Insert 12
④ Delete 16
⑤ Delete 4
⑥ Insert 4
the same with ④
⑦ Insert 3
the same with ④

2.



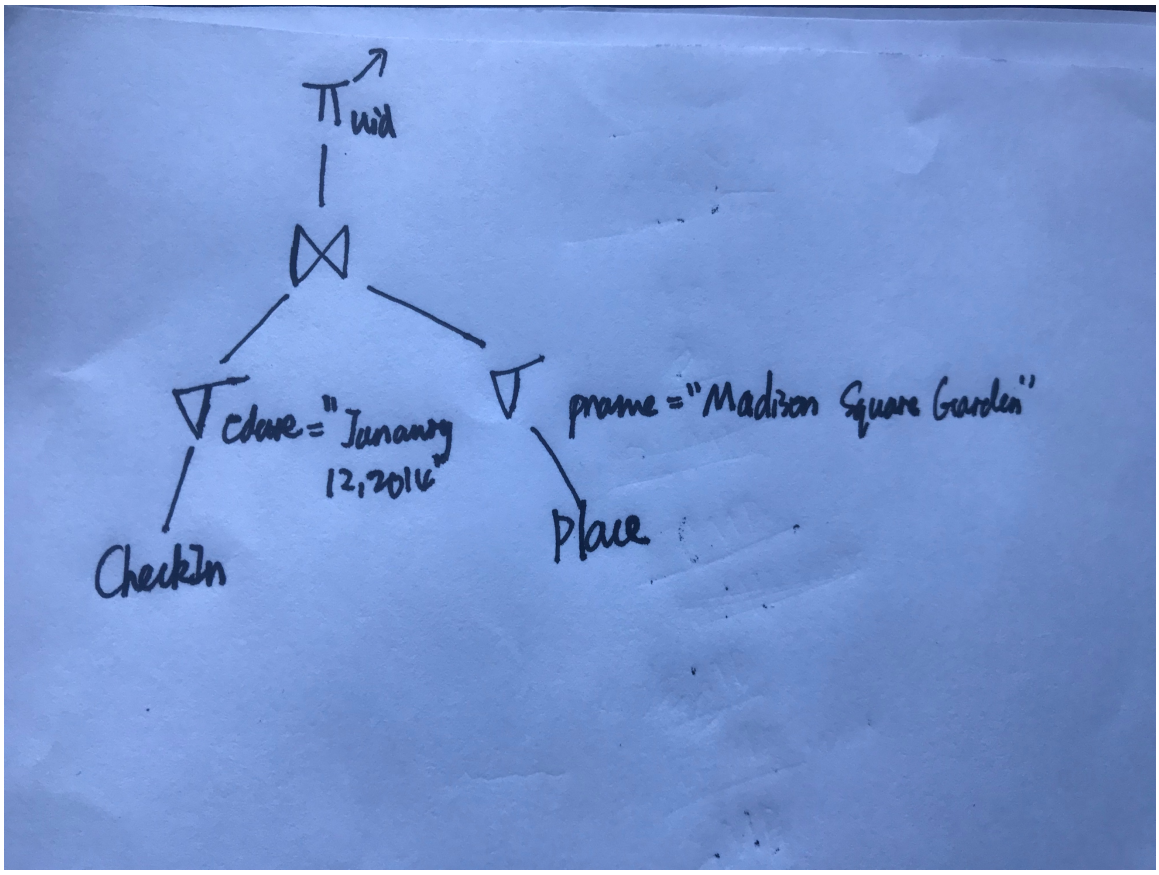
3.

(a)

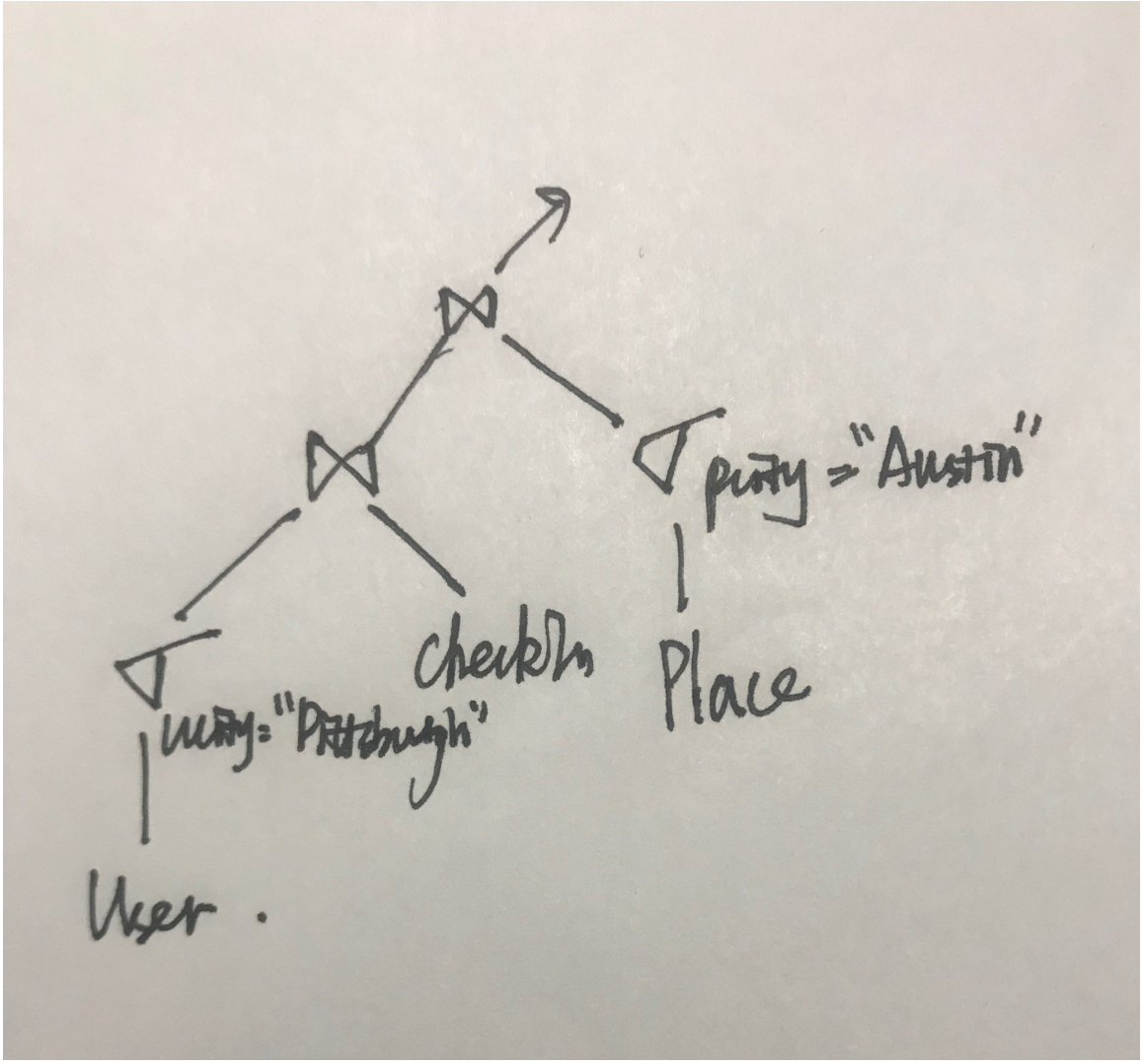
Query 1: List the IDs of all users who check into Madison Square Garden on January 12,2014.

Query2: List the uid and pid and check_in time of all users who live in Pittsburgh and checked into a Austin's place.

(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 can scan CheckIn to find the chech-ins which happen on January 12,2014. There are one million check-ins, taking 50MB size, that clearly fit in the main memory. So we can first can CheckIn to filter out check-ins on January 12,2014, and place those in main memory. Then we scan Place once, cheching which place is named Madison Square Garden, and then looking if those tuples match any of the surviving CheckIn tuples. Then we output the uid, thus we scan the CheckIn and Place once, for a total of 200MB+100GB, costing 1002s.



Query2: We scan User to find the users whose hometown is Pittsburgh. There are 50000 such users, and these tuples take about 5MB which clearly fit in the main memory. And there are 2 billion check-ins for a total of 100GB which would not fit in memory. So we first scan User to filter out the users living in Pittsburgh and place them in main memory. And then scan CheckIn once, looking if the tuples match any of the surviving User tuples. There are 10 million tuples, taking about 500MB which is larger than main memory, but actually we only need uid, pid and cdate of each check-in, which would probably fit in about 40 out of the 50 bytes of a checkin tuple. So this brings size down to 400MB, which fit in the main memory, so we can place it in the main memory. And then scan Place once to filter the places which is in Austin and look if the tuples match any of the surviving

CheckIn tuples, then output the uid, pid, cdate. So we we scan User, CheckIn and Place once, for a total of 1000MB+200MB+100GB, costing 1012s.

(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 250000 index entries. Since about 137 index entries are in each leaf node there will be about 250,000/137 ~ 1825 leaf nodes. On the next level there will be about 1825/137 = 13 nodes,and then we have the root of the tree. So the B+ tree has 3 levels of nodes: the root, one internal levels, and the leaf level. Thus is takes about 4 * 10 ms = 40 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 1825 * 4KB ~ 7300KB.

Dense unclustered index on (pid) in the CheckIn table:

Each index entry now has a 16-byte pid and an 8-byte RID, thus n is also 171. Assuming 80% occupancy per node, each leaf node will contain about 137 index entries and each internal node will have 137 children.

There are 2 billion index entries, and thus there are 14.6 million nodes at the leaf level, 106569 nodes at the next level, then 778 then 6, then the root. Thus the tree has 5levels of nodes and 6 * 10 ms = 60ms 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 60ms for a single record. The size of the tree is dominated by the leaf level, which is about 14.6 million * 4KB = 58.4GB.

(d)

Query1: We would choose a clustered index on cdate in CheckIn, to accelerate the fetching of checkins happening on January 12,2014 reducing that cost to a fraction of a second. We

could then add a clustered index on pname in Place. So we would not need to scan any table.

Query 2: We would choose a clustered index on ucity in User, to accelerate the fetching of users living in Pittsburgh, reducing that cost to a fraction of a second. We could then add a unclustered index on uid in CheckIn. So we would basically be left with the cost of scanning Place once in the final join, about 2 seconds.

4.(a)The capacity of the disk is 2*40000*1000*1024bytes ~800GB. The maximum rate of a read(using 15000RPM=250RPS) is 250*1000*1024bytes/s ~ 250MB/s, and the average rotational latency is $\frac{1}{250*2}$=2ms.

(b)Note that 250MB/s~250KB/ms, so it takes x/250ms transfer time to read x KB of data after the initial 5ms for seek and average rotational latency.

Block model:

Read 4KB: t=5+4/250ms=5.016ms

Read 100KB: T=25t=125.4ms

Read 100MB:T=25000t=125.4s

LTR model:

Read 100KB: T=5+100/250=5.4ms

Read 100MB:T=5+100000/250=405ms

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

(c)Phase1: Repeat the following until all data is read: Read 10GB of data and sort it in the main memory using any sorting algorithm. Write it into a new file until all data is read. The time to read 10GB is ~ 5ms+40s=40.005s. To read and write 25 such files takes 40.005*2*25~2000s

Phase2:Merge the 25 files created in Phase 1 in one pass. The main memory is divided into 26 buffers. Each buffer is of size 10000/26=384.6MB.For each buffer, the read and write time is 5ms+384.6/250s=1.54s. 250GB can be divided into 250*1000/384.6=650 pieces.The total time is 1.54*650*2~2002s.

(d) Now the degree is 5, so there are 2 passes for 25 files. The main memory is divided into 6 buffers. Each buffer is of size 10000/6=1707MB. For each buffer, the read and write time

is 5ms+1707/250s=6.83s. 250GB can be divided into 250*1000/1707=146pieces. Each pass costs 146*6.83*2~1994s. All 2 passes cost 1994*2=3988s.