CS143 Fall 2017 HW3 Q2 and Q3 Solutions
These solutions are far more detailed than we expect from students on homeworks or exams. Crucial information is **bolded** - on an exam, we would expect to see at least these points. In practice there's probably one or two pages of actual *required* information to answer these questions (so don't get intimidated by the ~10 pages here)
Some extra/supplementary information is provided for curious students - these are labeled as EXTRA.
Some questions have common mistakes that I've detailed as well.

Problem 2: Indexes

Problem recap:
`taken(StNo, CourseID, Year, Quarter, Sec, Grade, Remarks)`
20 years, 10K new students each year so 200K unique StNo
40K students enrolled each quarter, 4 classes/student/quarter
160K classes taken (by students) each quarter
3 quarters per year -> 480K classes taken every year
9.6M tuples for grades over the entire 20 years
Assume avg number of students per class is 100 (thus, 4.8K courses offered per year)
Two indexes:

- Sparse on `StNo`
- Dense on (`CourseNo, Year, Quarter, Sec, StNo`)

Assume each index field takes 8 bytes (`CourseNo, Year, Quarter, Sec, StNo`)
Assume B+ tree pointers take 10 bytes.

B1: If the file blocks have 4096 bytes and each tuple in `taken` requires 100 bytes, how many blocks will be needed to store the unspanned tuples of this relation?

floor(4096 (B/block) / 100 (B/tuple)) = 40 tuples per block
9.6M total tuples / 40 tuples per block = <u>240K blocks</u>
EXTRA: Here the tuples fit perfectly into blocks; in practice you might need to round up because of a partially used block (eg 9,600,001 tuples would require 240,001 blocks)

B2: Compute the levels and the number of blocks at each level of the B+ tree, assuming a worst-case scenario.

We start with the sparse index calculations first. I explain the calculations in detail; in practice, showing the math and brief words describing the result is sufficient.

B2 (Sparse)

First we need to identify the fanout

Key: 8 bytes (just StNo), pointer: 10 bytes, block: 4096 bytes
block >= n*ptr + (n-1)*key
4096 >= 10n + 8n - 8; 4104 >= 18n; **n = 228**
(ie n = floor[(block + ptr)/(key + ptr)])
If you prefer to calculate number of keys (I'll call it k):
block >= (k+1)*ptr + k*key
k = floor[(block - ptr) / (key+ptr)]
Just keep in mind that fanout is in terms of number of pointers, n = k+1

In worst-case, we assume nodes are filled minimally so that the tree is taller (larger depth) and takes longer to traverse.

Leaf node min: **ceil((228+1)/2) = 115 pointers, or 114 keys**
(some of you got these equations mixed up - make sure you understand them!)
Internal node min: **ceil(228/2) = 114 pointers, or 113 keys**
Root min: 2 pointers or 1 key (constant)

As this is a sparse index, we know the data is clustered around StNo and is thus sequentially ordered. Each leaf pointer (except the last tail pointer) points to a data block, so with 115 pointers per leaf node that becomes 114 data block pointers per leaf node. We have 240K distinct StNo: **240K / (115-1)=** 2105.26 (approximate). In this case we have to round down in order to ensure each leaf node meets the minimum pointer constraint of 115 (114 excluding its tail pointer), so we arrive at **2105 leaf nodes**. (These node calculations are far wordier than needed, you can just say *floor(a/b) = c*)

The internal nodes require at least 114 pointers each (*this is different from the leaf nodes, which require 115 but always have 1 tail pointer to the next leaf node). In order to have sufficient internal nodes for each leaf node, we calculate **2105/114** = 18.46 (approx). Again, we round down to ensure each internal node meets the minimum pointer constraint (114), so we end up with **18 1st level nodes***.

Calculating the 2nd level, we note that 18 < 114 so we cannot insert another layer of internal nodes. Thus the final level must be the root node (which has a minimum of 2 pointers). There is always exactly **1 root node**. While not required for the problem, the root node has 18 pointers.

In total there are **3 levels, with 1754 leaves, 15 1st level nodes, and 1 root node.**

**According to lecture slides, 1st level is the one directly on top of the leaf nodes. In other courses 1st level may refer to the root node or the root's children, so be clear with which you are referring to if you use n-th level terminology on the homework or exams.*

Common mistake: Forgetting that one of the leaf pointers is the tail pointer to the next leaf (which means subtract 1 from the fanout before dividing to get number of data block pointers).

Common mistake: Calculating min leaf pointer and min internal node pointers the same. They are different! (Write down the chart on your cheat sheet if you can't remember)

Common mistake: using one leaf pointer per *StNo* value because Jason mentioned it in a Piazza post - that's fine for hw credit. We will not be considering that for a sparse index in the future though; for this class, **assume sparse means one leaf pointer per data block** in the future.

EXTRA: The sparse index is used as a 'directory' for the data blocks, hence why one pointer per block. One per distinct value would require fewer entries (200K) but there is then a question of how to read the data blocks with no pointers.

B2 (Dense)

See the sparse tree example for detailed steps - these are identical with the exception of dense vs sparse pointer count.
Key: 8 bytes per field, with 5 fields -> 40B. Similar to above, 4096 >= 10n + 40n - 40; **n = 82**
Leaf: 42 pointers. Internal: 41 pointers. Root: 2 pointers (constant).
The same worst-case assumption applies here; we want minimally filled nodes.
A dense index means we can't make assumptions about how the data is ordered, and must have a *leaf pointer for every individual data record*. With 9.6M tuples, that means we'll need 9.6M leaf pointers (not counting the tail leaf pointers).
floor(9.6M / (42-1)) = 234146. So we have **234,146 leaf nodes.**
floor(234146 / 41) = 5710. So we have **5,710 1st level nodes.**
floor(5710 / 41) = 139. So we have **139 2nd level nodes.**
floor(129 / 41) = 3. So we have **3 3rd level nodes**.
3 < 41 (minimum # of pointers for an internal node), so we end with **1 root node.**
In total there are 5 levels @ 234146, 5710, 139, 3, and 1 node each.

Common mistake: Forgetting that one of the leaf pointers is the tail pointer to the next leaf (which means subtract 1 from the minimum to get number of data block pointers)

EXTRA**:** It might look like you can do some sort of log calculation, but it's not entirely that straightforward because:

1. # of minimum leaf pointers to files may differ from # of minimum internal node pointers (when **n** is odd, which was not the case here).
2. We're applying floor in between every layer.

You might end up with the correct answer by accident, but to be entirely precise you should avoid doing so.

B3 How many blocks of B+ tree and file will the DBMS retrieve from disk to answer the following query: *Find the average grade in a given class (e.g. find the average grade for: CS143, 2010, Fall, sec. 1).* Assume the **worst-case** scenario, and that all the buffers are initially empty.

We can use the dense index because the search key `(CourseID, Year, Quarter, Sec)` is a **prefix of the index key** `(CourseID, Year, Quarter, Sec, StNo)`. This is similar to computing a range query on (CS143, 2010, Fall, sec. 1, *).
As calculated in B2, the dense index has 5 levels. Thus, 4 B+ tree blocks will be loaded before we start reading the first leaf node (it'll take 5 to bring in the first leaf node).
Once at the first leaf node, we can traverse the leaf node (and any subsequent ones) until we've read 100 pointers (assuming 100 students in the class, as stated in the problem definition).
From B2 we know each leaf node has 41 pointers, so we'll need at least 3 leaf nodes to include all of the 100 students. In the worst case, the tuples aren't aligned within the leaf nodes and we end up spanning one extra node for a total of 4 leaf nodes in total for all 100 tuples. Since leaf nodes have tail pointers to the next leaf node, reading the 2nd through 4th leaf nodes requires one disk operation each, for a total of **8 B+ tree blocks**. (Some people said 5+3, some said 4+4, they're both equivalent ways of thinking about the problem)
As this is a dense index, we don't know anything about the ordering of records within the data blocks. Thus we must assume that each record points to a random block not already in memory, and requires a disk read. With 100 records, this means **100 file blocks**. In total: 108 blocks (8 index + 100 file)

We cannot use the sparse index on StNo, because the search key isn't a prefix of StNo and the index cannot be used to narrow down the applicable records.

Using an index might not always be optimal, especially on large range searches. Thus **we should also consider the alternative of using a full table scan**. From B1 we know there are 240K file blocks for the whole table, which is much larger than the 100 calculated above. Technically a full scan would also require reading the sparse index leaf nodes too, but this is not required for the comparison. As a result, we can conclude that using the index in this case is definitely more efficient than a full scan.

Common mistake: Assuming that the 100 tuples will always fit in 3 leaf nodes (capacity 41 each). There could be a split across 4 leaf nodes of 1;41;41;17 (basically starting at the very end of the 1st block). This particular detail is very tricky, so it would not make a very big impact on your score if you miss it and say 3 leaf nodes. (I'll let it slide for HW3)

Common mistake: Forgetting that leaf nodes have tail pointers to the next leaf node. For range searches, you only need to find the first leaf node in the range and follow tail pointers from there; no need to traverse the whole index to get to the next leaf node!

B4 We now want to compute the average grade over the (480,000 or so) classes taken in year 2011 (assume that they all have the same credit). Explain how the DBMS will go about searching and retrieving blocks from disk for this query, and estimate the number of blocks the system will have to fetch if those 200,000 students each took 48 classes on the average. Assume the **worst-case** scenario, and that all the buffers are initially empty.

Our search key is by year (2011), but we don't have any indexes that start with year.

Sparse index is by StNo only.

Dense index does not start with Year. So we cannot use it here either.*

Furthermore we're only working with one table (no joins), so it doesn't make sense to create an index just for the query. That would require reading the whole table just to build the index, at which point we could just do a full table scan instead.

Thus the DBMS will opt for a **full table scan.** This requires using the primary (sparse) index to read all leaf nodes and corresponding data blocks. The general idea is that the DBMS will use the primary index (recall tables must have a primary key; the DBMS will build the primary index automatically) to find each data block and index it. We calculated the number of data blocks to be **240K** from B1, so that is the number of blocks retrieved from disk for a full table scan. In addition, we need to traverse the tree to the first leaf node and sequentially read all subsequent leaf nodes: **2 non-leaf levels + 2105 leaf nodes**. Final: **2 + 2105 + 240,000**

Common mistake: Trying to use the dense index here. Because we're estimating 480K records, that's at least 480K data blocks retrieved due to random placement. That number is already about 2x the cost of doing a full scan, and doesn't even account for index blocks retrieved.

Common mistake: Jason miscommunicated and gave an oversimplification that the DBMS can directly access its data blocks and read them all sequentially; in practice, we still need to rely on the sparse index here to read the data. This was concluded after a discussion with the professor and is the official answer for for this course.

*EXTRA: There are special optimizations you can do to extend index applicability, but we don't use them in this class.

Problem 3: Joins and Optimizations

In addition to the table `taken` whose schema and index have been described previously, we also have the table `student(StNo,Level,FirstName,LastName,Major)` describing our 200,000 students. This table has a primary B+ index on `StNo` which is the key for this relation and a foreign key for `taken`.There are five different levels: freshman, softmore, junior, senior, others.

C1 How many blocks will `student` use, if each tuple requires 100 bytes and each block contains 4096 bytes?

floor(4096/100) means **40 tuples per block** (this should look familiar from B1)
We have 200K student tuples (given in problem statement).
200K tuples / (40 tuples / block) = 5K blocks

C2 For the query $\Pi_{StNo}(\sigma_{Level="others"}(student)) \bowtie taken$ estimate the size of the results (i.e., how many tuples). Also estimate the cost of implementing the query measured by the number if blocks read from disks. You can assume that the join takes advantage of the sparse index on `taken.StNo` (Also, for the sake of simplicity, assume that all indexes used are already in main memory).

With no information on the distribution of data over the levels, we assume it is uniform; thus, the selection filter on student has a selectivity of 20% (for 'others').
After the selection filter, we expect 200K * 0.2 = **40K** tuples on the left hand side. The projection does not change the number of expected results. I refer to this as the left hand side (LHS).
Then we do a natural join with `taken`, which results in a join on the StNo field. This should yield 20% of the `taken` relation, which was originally 9.6M records. **We expect 1.92M records**.

If this is confusing, note that the natural join is just an inner join on StNo after filtering down to 20% of the original StNo values.

One SQL interpretation (which closer resembles pseudocode below) is:

```
SELECT *
    FROM taken
    WHERE StNo IN
      (SELECT StNo
       FROM student
       WHERE LEVEL='others')
```

Note that this isn't a direct translation of the RA, but may make more intuitive sense.

For the join: The LHS can be computed after reading in a block from the `student` table (5K blocks). The LHS should be 40K tuples large as calculated earlier. We have a sparse index on `taken.StNo` that is used. Thus `taken` is the inner relation for the index join.

The general approach for the index join is as follows:

```
For each block of student:
    for each tuple of student that meets 'level="others"':
        Lookup tuple.StNo in the index(taken.StNo)
```

Thus we load the blocks of the student table, and for each record in the block that passes our condition we do a lookup in the provided index.

All that's left is to identify how much the lookup operation takes. We're told to ignore index block retrieval. Furthermore we know the index is sparse so blocks should be clustered around `StNo`, meaning that we can sequentially read all data records for a given `StNo` once we've used the index to find the first one. At 4 classes per quarter per student, 3 quarters per year, and 4 years we expect each student record to have 48 tuples from `taken` (this is all given in the problem definition). From B1 we know that each block can hold 40 tuples, so this should take at least 2

tuples. Note that 5 sets of 48 tuples can fit perfectly into 6 blocks, and thus we will never have a 48-tuple set (for a given StNo) span more than 2 blocks.

Numbers for our final calculation:
5K blocks (loading the `student` table)
40K tuples (after filtering the `student` table, yielding LHS)
2 blocks per `StNo` lookup on the `taken.StNo` index

Combining it all:
5K blocks + (40K tuples * 2 blocks/tuple) = 85K blocks
(recall we don't consider index blocks in this problem, otherwise that would be added to the lookup number above)

Common mistake: Forgetting to include the cost of reading the student table blocks and only accounting for the tuples * lookup calculation.

Common mistake: Not realizing that records for a single `StNo` value will span more than one data block. Remember we still need to read all data blocks containing records for the index value, not just the very first one that the index points to. If you made this mistake, you would have arrived at a much lower 45K tuples.

Common mistake: Just adding two terms together… not quite sure how these answers came up. Note that the index lookup has to be done for each qualifying record.

EXTRA: We specifically told you that the DBMS uses the index here. If not instructed as such, you should also compare against other options. Consider NBLJ: at 5K for the smaller table and 240K for the larger one, `taken` would be the inner table and `student` would be the outer one. The formula for that would be 5K + 5K*240K = 1.205M file blocks, which is far more than the 125K calculated above. Normally an index join will perform better than an NBLJ unless the index lookup is very expensive (eg if each lookup would yield a large range of the table).

EXTRA: Just as NBLJ is an improvement on the naive tuple-based NLJ, there's an optimization to NBLJ that accounts for the maximum number of pages that can be stored in memory. The idea is an extension of NBLJ: just as NBLJ improves upon NLJ by comparing one block of each table rather than one record of each table, the optimization will compare multiple blocks of each table at once.