COSC-4411: Assignment #2

Due: Friday 15 October 2004

1. **I/O.** *I/O, I/O, it's off to work we go.*

   Consider file **R** which has 10,000,000 records, and that we have a B+ tree index on this file with the search key as **R**'s primary key. Assume that the order (d) of the B+ tree is 75, and that the index pages are 70% full, on average.

   a. Assume that the B+ tree index above for **R** is of alternative *one*; that is, the leaf pages of the B+ tree contain the data records themselves (at 20 data records per page).

      You want to find all records with search key $\geq x$ and search key $\leq y$. Say that 1,000 data records qualify. How many I/O's does it cost you to retrieve these?

      There are 20 records on each data-record page. So, the index needs to distinguish over $\lceil 10^7/20 \rceil = 500,000$ *leaf pages. (Some assumed the data-record pages also were 70% full, with 20 as the maximum. In this case,* $\lceil 10^7/(.7 \cdot 20) \rceil \doteq 714,286$ *need to be indexed.) Index pages have between 75 amd 150 keys. (Remember, the root index page is exempt to the half-full rule in B+ trees.) The index pages are 70% full, so have 105 keys, on average, and 106 pointers. So the fan-out is 106.* $\lceil \log_{106} 500,000 \rceil = 3$. *(*$\lceil \log_{106} 714,286 \rceil = 3$ *too.) Thus, we need to read 3 index pages to find the right leaf (data-record) page.*
      Since 1,000 records qualify, this will span 50 or 51 data-record pages. (Probably 51, since the first qualifying record is probably not at the beginning, but in the middle, of that first relevant data-record page.) So $3 + 51 = 54$ *I/O's.*

   b. Assume that the B+ tree index above for **R** is of alternative *two* and is unclustered; that is, the leaf pages of the B+ tree contain data entries that are $\langle \mathsf{key}, \mathsf{rid} \rangle$ pairs and not the data *records* themselves. Say that 50 data entries fit per page, and say that your buffer pool is absurdly small with just five frames.

      Again, you want to find all records with search key $\geq x$ and search key $\leq y$, and 1,000 data records qualify. How many I/O's does it cost you to retrieve these?

      There are 50 data-entries, on average, on each data-entry page. So, the index needs to distinguish over $\lceil 10^7/50 \rceil = 200,000$ *leaf pages. (Some assumed the data-entry pages also were 70% full, with 50 as the maximum. In this case,* $\lceil 10^7/(.7 \cdot 50) \rceil \doteq 285,715$ *need to be indexed.) Again, the index pages are 70% full, so have 105 keys, on average, and 106 pointers. Again,* $\lceil \log_{106} 200,000 \rceil = 3$. *(*$\lceil \log_{106} 285,715 \rceil = 3$ *too.) Thus, we need to read 3 index pages to find the right leaf (data-entry) page.*

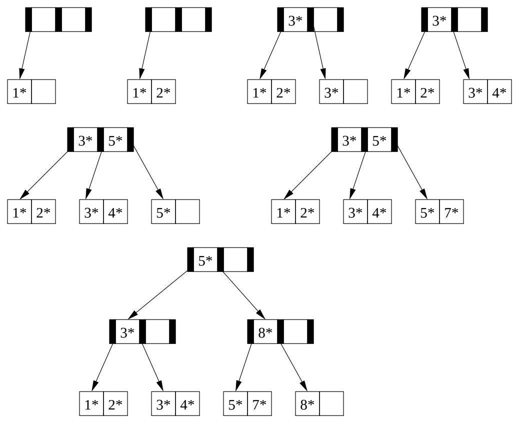
Since 1,000 records qualify, this will span 20 or 21 data-record pages. For each data-entry, we need to fetch the actual data record. Since the index is unclustered, this tends to one I/O per record to fetch. Since there are 500,000 pages of data records and we are fetching 1,000 records "randomly" with respect to those pages, it is predominantly the case no more than one qualifying record exists per page. As well, we have little buffer pool space, just five pages. So even if there were several qualifying records per page, the likelihood that same page will be in the buffer-pool come time we need the second qualifying record off of it is vanishingly small. Thus, in this case, it is not only worst-case that we will need to spend 1,000 I/O's to fetch the 1,000 records; it is also pretty much our average-case.
We spend 3 I/O's on index-page reads, 21 I/O's on data-entry page reads, and 1,000 I/O's on data-record page reads, for a total of 1,024.

c. Now assume that we have an index on **R**'s primary key instead that is an extendible hash index. Assume that the index is of alternative *two* and is unclustered. We want to find the record with search key value x. How many I/O's does it cost you to retrieve the record?

One I/O to read the directory page (it is an extendable hash), one to read the bucket page with the data-entry in it (it is unclustered), and one to read the data-record page, for a total of 3 I/O's.

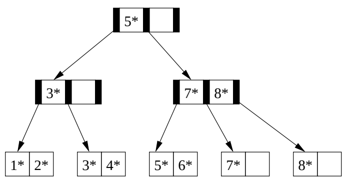
2. **B+ Trees.** *Root-bound.*

a. Starting from scratch, show how a B+ tree of order one would appear after adding keys 1, 2, 3, 4, 5, 7, and 8 in that order.



This was not the only correct way to do it! The results depend on whether you place the two keys ($d = 2$ here, so $d + 1 = 2$ too) on the left page (as I have done) or on the right page consistently after a split. Either policy follows the B+ tree rules just fine.

b. Now add key 6 to the tree from 2a. Do not do any redistribution.



Again, mileage may vary. If you composed your tree legally, but differently, in Question 2a, you will have a different result here.

c. If you did a bulk build in 2a instead, would the tree have looked different?

It would look the same as the solution here in 2a. This is not surprising, since we added the keys in order, which would be unusual in most incremental composition of B+ trees.
However, the result here could be different from that in 2a, if one built the tree in 2a legally, but differently, as discussed. Also, the tree here could have come out different if we were controlling the fill-factor on the leaves, say, to leave some space.
Having d = 2 *is quite artificial, so it is not so easy to see the distinctions. (Having* d = 2 *is easier for working through examples! Imagine if we had* d = 100 *instead as will be on the test. Just kidding.)*

3. **Page Layout.** *Open-ended.*

The page layout scheme described in the textbook for variable length records places a limit in advance on how many records can be placed on the page. This is because the slot array is of fixed length.

Briefly describe a new, reasonable scheme that would eliminate this restriction.

What are the disadvantages of your scheme compared with the textbook's?

The scheme the textbook discusses does *allow for more slot-directory cells to be added. If all cells are in use and a new record is to be placed on the page, a new cell can be added at the end of the slot directory. A potential problem with this approach is that the slot directory will have at least as many cells as the maximum number of records that ever occupied the page. No provisions are made for shrinking the slot directory ever; only growing it.*
One might think to eliminate cells from the slot directory when the associated record is deleted. However, this would be bad! The rid *is often, by design, the page# and the slot# together. If we eliminate a slot-directory cell, we will have effectively* changed *the* rid*'s of all the records in subsequent cells.*

A way around this would be to add yet another layer of indirection. Presently, a slot-directory cell contains two pieces of information: the record's offset (where on the page it is physically located) and a length. We could keep a third piece of information: a slot#. Now slot# is not synomymous with the position in the slot-directory array.
We would probably keep the slot-directory array sorted by slot# so that we could find a given record quickly via binary search over the array. (It was even easier before, since the rid *gave us the slot# which was the array index.) Of course, if we reuse a slot#, say 3, presently not in use on the page on a record insert, we will have to shift the tail of the array to make space. Really, this is not so bad. The slot directory is several hundred bytes at worst. Main memory operations (e.g., memcpy) are very fast for move shuch small amounts of data around. So it is not the same as if the slot directory were a large array where shifting would be a substantial expense.*

4. **Buffer Pool & Replacement Strategies.** *Pinned and pinned again.*

   Some pages in the buffer pool have at some time or another a pincount of more than one. Dr. Dogfurry has noted that one can consider pages pinned more times than other pages as more "popular". So we should favour them, leaving them in the buffer pool longer.

   Design a replacement strategy that does that. How long a page remanins in the buffer pool should be correlated with how many times it has been pinned while in the buffer pool *this* time. Hint: Think about the *clock* strategy.

Consider a modification of the clock strategy as follows. Instead of a reference bit, we could maintain a reference count. Every time a page in the buffer pool is pinned, we increment its frame's reference count. However, when the page is unpinned, we leave the reference count untouched.
When clock considers a page for replacement, it passes it by if the page has a positive pincount, of course. If the pincount is zero, clock checks the reference count. If it is positive, clock decrements the reference count, leaves the page, and goes on to check the next.
Thus a page will remain in the pool until clock has passed the page reference-count number of times, leaving pages with higher reference counts in the pool for longer.

5. **Compression.** *Paradox?*

   Dr. Dogfurry is consulting for Applications, Inc. They are exploring ways to save on disk-space usage for their data-intensive applications. So naturally, Dr. Dogfurry suggested that the applicaton server use a compression algorithm on programs' data whenever it is to be written to disk. Then, of course, the applicaton server would need to uncompress the data on the fly when it reads it back into main memory.

   Dr. Dogfurry said this was a trade-off: you save disk-space; but you lose time. Applications are slowed down because of the compression and decompression that must be done.

However, once Applications, Inc., implemented this, they found it sped up the application server! Why did it speed up?

Apparently, the I/O's saved in reading and writing the compressed data instead of the raw data (the compressed data takes fewer pages) outweighs the extra CPU time needed to decompress and compress the data. It is not so surprising really that this might be the case, since I/O's are quite costly time-wise compared with CPU-cycles. And the gap between I/O speeds and CPU speeds continues to grow in current technological trends.