CS360 Homework #4

May 17, 2017

Instruction /Notes:

Read these carefully

- **Please read all the points of the "Notes" sections- they're important for this Homework!!!**
- You **are not required to do any plotting in this Homework- only in certain problems to provide the tuples that would generate a plot**. You can then optionally plot (in the notebook with matplotlib, in Excel, wherever works)
- You may create new IPython notebook cells to use for e.g. testing, debugging, exploring, etc.- this is encouraged in fact! - **just make sure that your final answer for each question is in its own cell and clearly indicated**
- See CS360_2017S@Facebook for submission instructions
- Have fun!

1. Problem 1: Double Trouble

[50% points]

In this problem, we'll explore an optimization often referred to as **double buffering**, which we'll use to speed up the external merge sort algorithm we saw in Ch. 9.

Although we haven't explicitly modeled it in many of our calculations so far, recall that sequential IO (i.e. involving reading from / writing to consecutive pages) is generally much faster that random access IO (any reading / writing that is not sequential). Additionally, on newer memory technologies like SSD reading data can be faster than writing data (if you want to read more about SSD access patterns look here).

In other words, for example, if we read 4 consecutive pages from file A, this should be much faster than reading 1 page from A, then 1 page from file B, then the next page from A.

In this problem, we will begin to model this, by assuming that 3/4 sequential READS are "free", i.e. the total cost of 4 sequential reads is 1 IO. We will also assume that the writes are always twice as expensive as a read. Sequential writes are never free therefore the cost of N writes is always 2N.

1.1 Other important notes:

- **NO REPACKING**: Consider the external merge sort algorithm using the basic optimizations we present in Ch. 9, but do not use the repacking optimization covered in Ch. 9.
- **ONE BUFFER PAGE RESERVED FOR OUTPUT**: Assume we use one page for output in a merge, e.g. a B-way merge would require B + 1 buffer pages
- **REMEMBER TO ROUND**: Take ceilings (i.e. rounding up to nearest integer values) into account in this problem for full credit! Note that we have sometimes omitted these (for simplicity) in lecture.
- **Consider worst case cost**: In other words, if 2 reads could happen to be sequential, but in general might not be, consider these random IO

1.2 Part (a)

[30% points]

Consider a modification of the external merge sort algorithm where **reads are always read in 4-page chunks (i.e. 4 pages sequentially at a time)** so as to take advantage of sequential reads. Calculate the cost of performing the external merge sort for a setup having B + 1 = 20 buffer pages and an unsorted input file with 160 pages.

Show the steps of your work and make sure to explain your reasoning by writing them as python comments above the final answers.

Part (a.i) What is the **exact** IO cost of splitting and sorting the files? As is standard we want runs of size B + 1.

```
In [ ]: io_split_sort =
```

Part (a.ii) After the file is split and sorted, we can merge n runs into 1 using the merge process. What is largest n we could have, given reads are always read in 4-page chunks? Note: this is known as the arity of the merge.

```
In [ ]: merge_arity =
```

Part (a.iii) How many passes of merging are required?

```
In [ ]: merge_passes =
```

Part (a.iv) What is the IO cost of the first pass of merging? Note: the highest arity merge should always be used.

```
In [ ]: merge_pass_1 =
```

Part (a.v) What is the total IO cost of running this external merge sort algorithm? **Do not forget to add in the remaining passes (if any) of merging**.

```
In [ ]: total_io =
```

1.3 Part (b)

[10% points]

Now, we'll generalize the reasoning above by writing a python function that computes the approximate* cost of performing this version of external merge

sort for a setup having B + 1 buffer pages, a file with N pages, and where we now read in P -page chunks (replacing our fixed 4 page chunks in Part (a)).

**Note: our approximation will be a small one- for simplicity, we'll assume that each pass of the merge phase has the same IO cost, when actually it can vary slightly... Everything else will be exact given our model*!

We'll call this function `external_merge_sort_cost(B, N, P)`, and we'll compute it as the product of the cost of reading in and writing out all the data (which we do each pass), and the number of passes we'll have to do.

Even though this is an approximation, **make sure to take care of floor / ceiling operations- i.e. rounding down / up to integer values properly!**

Importantly, to simplify your calculations: Your function will only be evaluated on cases where the following hold*:

- **(B + 1) % P == 0** (i.e. the buffer size is divisible by the chunk size)
- **N % (B + 1) == 0** (i.e. the file size is divisible by the buffer size)

Part (b.i) First, let's write a python function that computes the exact total IO cost to create the initial runs:

```
In [ ]: def cost_initial_runs(B, N, P):

            # YOUR CODE HERE
```

Part (b.ii) Next, let's write a python function that computes the approximate* total IO cost to read in and then write out all the data during one pass of the merge:

```
In [ ]: def cost_per_pass(B, N, P):

            # YOUR CODE HERE
```

*Note that this is an approximation: when we read in chunks during the merge phase, the cost per pass actually varies slightly due to 'rounding issues' when the file is split up into runs. . . but this is a small difference

Part (b.iii) Next, let's write a python function that computes the exact total number of passes we'll need to do

```
In [ ]: def num_passes(B, N, P):
```

```
        # YOUR CODE HERE
```

Finally, our total cost function is:

```
In [ ]: def external_merge_sort_cost(B, N, P):
            return cost_initial_runs(B,N,P) +
                   cost_per_pass(B,N,P)*num_passes(B,N,P)
```

1.4 Part (c)

[10% points]

For B + 1 = 100 and N = 900, find the optimal P according to your IO cost equation above. Return both the optimal P value (P opt) and the list of tuples **for feasible values of P** that would generate a plot of P vs. IO cost, at resolution = 1(every value of P), stored as points:

```
In [ ]: # Save the optimal value here

        P=

        # Save a list of tuples of (P, io_cost) here,

        # for all feasible P's

        points =
```

*Below we provide starter code for using matplotlib in the notebook, if you want to generate the graph of P vs. IO cost; however any other software that allows you to visualize the plot (Excel, Google spreadsheets, MATLAB, etc) is fine!

```
In [ ]: # Shell code for plotting in matplotlib

        %matplotlib inline

        import matplotlib.pyplot as plt

        # Plot

        plt.plot(*zip(*points))

        plt.show()
```

2. Problem 2: IO Cost Models

[30% points]

In this problem we consider different join algorithms when joining relations R(A,B), S(B,C), and T(C,D). We want to investigate the cost of various pairwise join plans and try to determine the best join strategy given some conditions.

Specifically, for each part of this question, we are interested determining some (or all) of the following variables:

- P_R: Number of pages of R
- P_S: Number of pages of S
- P_RS: Number of pages of output (and input) RS
- P_T: Number of pages of T
- P_RST: Number of pages of output (and input) RS
- B: Number of pages in buffer
- IO_cost_join1: Total IO cost of first join
- IO_cost_join2: Total IO cost of second join

Note:

- **The output of join1 is always feed as one of the inputs to join 2**
- **Use the "vanilla" versions of the algorithms as presented in lecture, i.e. without any of the optimizations we mentioned**
- **Again assume we use one page for output, as in lecture!**
- **The abbreviates for the joins used are Sort-Merge Join (SMJ), Hash Join (HJ), and Block Nested Loop Join (BNLJ).**

2.1 Part (a)

[15% points]

Given:

- P_R: 10
- P_S: 100
- P_T: 1000
- P_RS: 50
- P_ST: 500
- P_RST: 250

- B: 32

Compute the IO cost for the following query plans:

- IO_Cost_HJ_1 where only hash join is used, *join1 = R(a, b), S(b, c)* and *join2 = join1(a, b, c), T (c, d)*
- IO_Cost_HJ_2 where only hash join is used, *join1 = T (c, d), S(b, c)* and *join2 = join1(b, c, d), R(a, b)*
- IO_Cost_SMJ_1 where only sort merge join is used, *join1 = R(a, b), S(b, c)* and *join2 = join1(a, b, c), T (c, d)*
- IO_Cost_SMJ_2 where only sort merge join is used, *join1 = T (c, d), S(b, c)* and *join2 = join1(b, c, d), R(a, b)*
- IO_Cost_BNLJ_1 where only block nested loop join is used, *join1 = R(a, b), S(b, c)* and `join2 = join1(a, b, c), T (c, d)`
- IO_Cost_BNLJ_2 where only block nested loop merge join is used, *join1 = T (c, d), S(b, c)* and *join2 = join1(b, c, d), R(a, b)*

Note: again, be careful of rounding for this problem. Use ceiling/floors whenever it is necessary.

Include 1-2 sentences (as a python comment) above each answer explaining the performance for each algorithm/query plan.

```
In [ ]: IO_Cost_HJ_1 =
        IO_Cost_HJ_2 =
        IO_Cost_SMJ_1 =
        IO_Cost_SMJ_2 =
        IO_Cost_BNLJ_1 =
        IO_Cost_BNLJ_2 =
```

2.2 Part (b)

[15% points]

For the query plan where *join1 = R(a,b),S(b,c)* and *join2 = join1(a,b,c),T(c,d)* find a configuration where using HJ for *join1* and SMJ for *join2* is cheaper than SMJ for *join1* and HJ for *join2*. The output sizes you choose for P_RS and P_RS must be non-zero and feasible (e.g. the maximum output size of *join1* is P_R*P_S).

```
In [ ]: P_R =
        P_S =
        P_T =
```

P_RS =
P_RST =
B =

HJ_IO_COST_join1 =
SMJ_IO_COST_join2 =

SMJ_IO_COST_join1 =
HJ_IO_COST_join2 =

3. Problem 3: Sequential Flooding

[20% points]

Note: Before doing this question, it is highly recommended that you go through Pr_12, which covers eviction policies for buffer managers such as LRU, and why sequential flooding can sometimes occurs with LRU.

In the practice accompanying Ch. 12, we saw something called sequential flooding that can occur when a default eviction policy (for example LRU) is used by the buffer manager. We saw that we can achieve much lower IO cost by using a different eviction policy, MRU ("most recently used").

Note that "Most recently used" means most recently accessed, either from buffer or disk, consistent with what we showed in Pr_12.

For this problem, we will take a closer look at the IO cost of different eviction policies when reading the pages of a file sequentially multiple times.

3.1 Part (a.i)

[2% points]

Write a python function `lru_cost(N, M, B)` that computes the IO cost of the LRU eviction policy when reading in all the pages of an N-page file sequentially, M times, using a bugger with $B + 1$ pages. Assume that after reading the files, you don't need to write them out (you can just release them, so there is no write IO cost).

```
In [ ]: def lru_cost(N, M, B):

            # YOUR CODE HERE
```

3.2 Part (a.ii)

[4% points]

Write a python function `mru_cost(N, M, B)` that computes the IO cost of the MRU eviction policy when reading in all the pages of an N-page file sequentially, M times, using a bugger with $B + 1$ pages. Assume that after reading the files, you don't need to write them out (you can just release them, so there is no write IO cost).

```
In [ ]: def mru_cost(N, M, B):

            # YOUR CODE HERE
```

3.3 Part (a.iii)

[4% points]

Now that you have written these functions, provide the tuples which generate the plot of **M vs. the absolute value of the difference between LRU and MRU in terms of IO cost** for B = 6, N = 10, and M between 1 and 20 inclusive (saved as the variable p3_lru_points)

```
In [ ]: B = 6
        N = 10
        M = 20

        # Provide a list of tuple (m, difference between LRU
        # and MRU in terms of IO cost) here:
        p3_lru_points =
```

Again, you can optionally plot your answer to check that it seems reasonable- starter code for doing this in the notebook below:

```
In [ ]: # Shell code for plotting in matplotlib
        %matplotlib inline
        import matplotlib.pyplot as plt

        # Plot
        plt.plot(*zip(*p3_lru_points))
        plt.show()
```

3.4 Part (b)

[8 + 2% points]

Recall that the LRU eviction policy removes the least recently used page when the buffer is full and a new page is referenced which is not there in buffer. The basic idea behind LRU is that you timestamp your buffer elements, and use the timestamps to decide when to evict elements. Doing so efficiently, requires some serious book-keeping, this is why in practice many buffer managers try to approximate LRU with other eviction policies that are easier to implement.

Here we will focus on the CLOCK or Second Chance policy. In the CLOCK eviction

policy, the candidate pages for removal are considered left-to-right in a circular manner(with wraparound), and a page that has been accessed between consecutive considerations will not be replaced. The page replaced is the one that - considered in a circular manner - has not been accessed since its last consideration.

In more details the CLOCK policy proceeds maintains a circular list of pages in the buffer and uses an additional clock (or second chance) bit for each page to track how often a page is accessed. The bit is set

to 1 whenever a page is referenced. When clock needs to read in a new page in the buffer, it sweeps over existing pages in the buffer looking for one with second chance bit set to 0. It basically replaces pages that have not been referenced for one complete revolution of the clock.

A high-level implementation of clock: 1. Associate a "second chance" bit with each page in the buffer. Initialize all bits to ZERO (0). 2. Each time a page is referenced in the buffer, set the "second chance" bit to ONE (1). this will give the page a second chance. . . 3. A new page read into a buffer page has the second chance bit set to ZERO (0). 4. When you need to find a page for removal, look in left-to-right in a circular manner(with wraparound) in the buffer pages: - If the second chance bit is ONE, reset its second chance bit (to ZERO) and continue. - If the second chance bit is ZERO, replace the page in the buffer.

You can find more details on CLOCK here.

Part (b.i) Write a python function `clock_cost(N, M, B)` that computes the IO cost of the CLOCK eviction policy when reading in all the papges of an N-page file sequentially, M times, using a bugger with $B + 1$ pages. Assume that after reading the files, you don't need to write them out (you can just release them, so there is no write IO cost).

```
In [ ]: def clock_cost(N, M, B):

            # YOUR CODE HERE
```

Part (b.ii) Now that you have written the CLOCK cost function, provide the tuples which generate the plot of **M vs. the absolute value of the difference between LRU and CLOCK in terms of IO cost** for B = 6, N = 10, and M between 1 and 20 inclusive (saved as the variable p3_clock_points).

```
In [ ]: B = 6
```

```
N = 10
M = 20
p3_clock_points = [(m, abs(lru_cost(N, m, B)
                          - clock_cost(N, m, B)))
                   for m in range(1, M+1)]
```

Does the CLOCK eviction policy prevent sequential flooding? How does it perform against LRU? Write a short explanation in the field below.

```
In [ ]: # EXPLANATION GOES HERE
```