CS145 Fall 2019 Homework 2

October 16, 2019

This homework is to help familiarize you with hashes, joins, and indexes. This homework, as with the rest of the homework in the course, is graded on completion ($>70\%$ correct for full credit). Your score on Gradescope will be a raw score based on correctness; however, when processing scores, we will use the 70% threshold.

For this homework, submit your work and answers as a PDF to Gradescope.

Assume the following numbers for hardware performance:

- Hard disk access (seek) takes 10ms
- Disk transfer speed is 100MB/sec

HW 2 is due 10/29 at 11:59PM (No Late Days Allowed).
Section 2 will be on Friday, 10/18 from 9:30 AM — 10:20 AM in NVIDIA Auditorium.

Question 1 [25 points] - Hashing

As we saw in lecture, hashing is very important for managing large data systems. For example, it is used to map from data/search keys to the location where that data is stored (memory address, DB block, machine/disk). Another common application is evenly dividing a set of inputs into buckets in order to process them in parallel, for example when counting large numbers of tuples. In this problem, we'll get a little more intuition about hash functions and collision resolution by looking at some examples.

Question 1.1 [5 points] - Some Intuition about Hash Collisions

The details of how different hash functions work, including how they are implemented and their statistical properties, are mostly beyond the scope of this class. However, a feature common to all hash functions is collisions. When dealing with hashing, it's helpful to have some intuition about the frequency of collisions. Recall that a collision occurs whenever for two distinct elements a and b and a hash function $h(x)$, $h(a) = h(b)$.

Assume you have a hash function (the actual process by which it operates is unknown) with the following properties:

- The outputs (hash values) are of size 8 bits (there are 256 possible hash values/buckets).
- The hash function distributes its outputs evenly across the entire output range (this property is very desirable in hash functions and is called **uniformity**).

What is the probability of having at least one collision if we are hashing 5 inputs? Give your answer as a numerical value (not an equation) but show some work/reasoning. A simple numerical answer with no reasoning will not count for full points.

What about for 10 inputs? 20 inputs? 50 inputs?

NOTE: One of the interesting (and somewhat counter-intuitive) facts about hash functions is that the probability of collision rises much faster than intuition might suggest. Even when hashing relatively small numbers of inputs (w.r.t the output range), the collision probability rapidly approaches 1. See https://en.wikipedia.org/wiki/Birthday_problem for more interesting discussion on this problem.

Question 1.2 [6 points] - Thinking about Collision Resolution

In most applications of hashing, how collisions are handled is an important part of the implementation. One example of this is in the implementation of hash partition join, which we discussed in lecture. Consider the case where we wish to join two relations R and S on a shared attribute A. The first step in the process, partitioning, is done using a hash function. We hash tuples from both relations using their attribute A in order to divide them up evenly into a finite number of buckets B.

In this process of partitioning, hash collisions may occur. Explain why having a large number of hash collisions in our buckets would harm the efficiency of the hash partition join algorithm.

In light of this, a method is needed to address collisions. As described in lecture, this can done with **double hashing**. If the initial partition resulted in collisions, these can be resolved by doing a second pass and re-hashing each collided element with a new hash function (on attribute A). The result of this second hash is used as an offset to determine how many bins to move the tuple over. For example, if a tuple is hashed initially to bin 3 and there is a collision, if $h_2(tuple) = 1$ then the tuple will be moved to bin $3 + 1 = 4$ (if this again results in a collision, the tuple can be shifted again by $h_2(tuple)$ until the collision is resolved). In practice, a common choice for this second hash function is $Hash_2(key) = K - (key \mod K)$, where K is a prime smaller than the number of buckets B.

You are given $B = 5$. You are also given the following relation R.

A	C	E
5	93	Red
7	28	Purple
3	70	Orange
11	545	Blue
6	88	Brown

The initial hash function is $h_1(A) = A \mod B$. The second hash function is $h_2(A) = 3 - (A \mod 3)$.

As part of the first step of a hash partition join, partition R using the given hash function. Assume that the tuples are hashed into buckets in the order they appear in the table (from top to bottom). Resolve any collisions with the provided second hash function. You can indicate your answer in the form of a table or by listing the tuple(s) in each bucket (B0, B1, ...).

Bonus Note: There are many other collision resolution strategies beyond those mentioned here, each with different advantages and disadvantages which can be analyzed probabilistically. If you're interested, CS166 covers hash tables and collision resolution in more depth.

Question 1.3 [4 Points] - Computing Counts of Pairs

As seen above, hash functions are useful whenever we need to (relatively) evenly distribute data. Another such application is dividing across multiple machines to process in parallel.

Imagine you have created a music app. Users of your app can login, starting a session, and then play songs. The database which forms the backend to your app contains a table with tuples (user_id, session_id, song_id) which represent every time a user has played a song. You wish to see which pairs of songs are most frequently listened to together in the same session.

You are given the following information:

- There are 10 million users
- There are 1 thousand songs
- The user IDs are 3 bytes
- The song IDs are 2 bytes

- The session IDs are 4 bytes
- The avg. number of songs played per session is 5

Assume all your data is stored on hard disks and use the values for disk seek/scan times from the top of the assignment. Assume that data is stored sequentially on disk.

Calculate the total time required to compute the counts for each pair. Assume that it takes 1 hr. to sort all pairs using the external merge sort algorithm. Please show some work/reasoning. A simple numerical answer with no reasoning will not count for full points.

Question 1.4 [6 Points] - Parallelizing Counting with Hashing

Now imagine you have a hash function which maps from any 64-bit input uniformly to a 32-bit hash value.

Explain how you could use it to parallelize the counting across n machines.

What would the total required time to compute the counts for each pair be once you've used your hash function to divide the pair tuples across 6 separate disks (which can write in parallel)? Please show some work/reasoning. A simple numerical answer with no reasoning will not count for full points.

Question 1.5 [4 Points] - Thinking a Little More About Hashing

Consider the application of hashing where we want to parallelize a task across n machines (where n is reasonably small, say 100) versus the task of using hashing as a way to map keys to a location (e.g. generating IDS).

For each of these two applications, is a low collision rate important? For each of these two applications, is uniformity important? Why?

Question 2 [25 points] - External Merge Algorithm

This problem explores a different optimization referred as **double buffering**, which we will use to speed up the **external merge sort algorithm**. Below in the subproblems, you will calculate the cost of performing the external merge sort.

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.

For example, if we read 8 consecutive pages from file **A**, this should be much faster than reading 2 pages from **A**, then 4 pages from file **B**, then 2 pages from **A**.

Please note the following:

- **NO REPACKING**: Consider the external merge sort algorithm using the basic optimizations we present in class, but do not use the repacking optimization covered in class.
- **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. 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.

Question 2.1 [12 points]

Consider a modification of the external merge sort algorithm where reads are always read in 8-page chunks (i.e. 8 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=40** buffer pages and an unsorted input file with **320** pages.

Show the steps of your work and make sure to explain your reasoning if necessary. You will need to fill in the python functions.

Question 2.1.1 [3 points]

What is the exact IO cost of spliting and sorting the files? As is standard we want runs of size **B+1**.

Question 2.1.2 [3 points]

After the file is split and sorted, we can merge **n** runs into 1 using the merge process. What is the largest **n** we could have, given reads are always read in 8-page chunks? Note: this is known as the arity of the merge.

Question 2.1.3 [3 points]

How many passes of merging are required?

Question 2.1.4 [3 points]

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.

Question 2.2 [13 points]

Now, we'll generalize the reasoning above by writing formula that compute 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 **8** page chunks in the previous section.

*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.

We'll calculate the IO cost for each merge phase and compute the total cost 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, you can assume:

- $(B+1)\%P == 0$ (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)

Question 2.2.1 [4 points]

First, write the formula that computes the exact total IO cost to create the initial runs in terms of B, N, and P.

Question 2.2.2 [4 points]

Next, write the formula that computes the approximate total IO cost to read in and then write out all the data during one pass of the merge in terms of B, N, and P.

Question 2.2.3 [3 points]

Next, write the formula that computes the exact total number of passes we'll need to do in terms of B, N, and P.

Question 2.2.4 [2 points]

Finally, write the formula that computes the total cost in terms of B, N, and P.

Question 3 [19 points] - B+ Trees

We've seen how B+ Trees can be used to build indices for efficient access to data. In this question, we'll strengthen our present understanding of B+ trees and look at a quick-yet-effective optimization.

Let's assume you've inherited a database of **2 billion** rows from your relatives. The rows contain a string field called *'unique name'* and a few other fields. Given the enormous size of the database, you decide to build a B+ tree indices on the *'unique name'* field.

Use the following values for calculations:

- Each page is exactly **4KB** (4096 Bytes)
- The size of each key in your B+ tree is **128 bytes**
- The size of each pointer in your B+ tree is **8 bytes**
- Your system has **48GB** of free RAM and **infinite** hard disk space
- The fill-factor of the B+ tree is set to be **100%**. Fill-factor denotes the percent of filled slots in the index pages.
- Assume a B+ tree node must fit into a single page.

Parameters of our B+ Tree [4 points]

Let's look at how our B+ tree would look with the amount of data we need to index.

Question 3.1 [3 points]

As discussed in the lectures, every non-root node of the B+ tree has between d and $2d$ keys. What's the maximum value of d you can choose for your B+ tree?

Question 3.2 [1 point]

How does the maximum value of d change when the fill factor is decreased to 67%?

Fit it in the system [15 points]

Assume that the fill-factor is 100% for subsequent calculations.

Note: *In order to avoid cascading of errors, please use $d = 16$ for subsequent calculations in this section. This value* ***may not*** *be the expected answer of Q3.1*

Question 3.3 [3 points]

Fan-out factor is defined as the number of pointers to child nodes coming out of a node. What is the fan-out factor using the above defined value of d?

Question 3.4 [6 points]

Let's plan out the space required for our index. How many levels do we need in our B+ tree? Compute the space required by each index level.

Question 3.5 [6 points]

Assume that each level must either be completely on RAM or disk. Note that all data pages stay on the disk. What is the worst case IO requirement (number of disk accesses) to access a record?

Question 4 [21 points] - Join Implementation

This problem will explore different join implementations and the associated IO costs for each model. Let R(a, b), S(b, c), and T(c, d) be tables. For the purpose of this question, use the values provided below.

- P(R) = number of pages of R = 20
- T(R) = number of tuples of R = 1600
- P(S) = number of pages of S = 200
- T(S) = number of tuples of S = 15000
- P(T) = number of pages of T = 2000
- P(R, S) = number of pages in output RS = 100
- P(S, T) = number of pages in output ST = 1000
- P(R, T) = number of pages in output RT = 500
- B = number of buffer pages = 32

Question 4.1 [1 point]

Let us start by considering a simple nested loop join. Compute the IO cost for a simple nested loop join if R is the "outer loop" and S is the "inner loop."

Question 4.2 [1 point]

Compute the IO cost for a simple nested loop join if S is the "outer loop" and R is the "inner loop."

Question 4.3 [4 points]

Now consider using a block nested loop join. Compute the IO cost for joining R, S and then joining the result with T. Then compute the IO cost for joining S, T and then joining the result with R.

Question 4.4 [2 points]

Now consider using a sort-merge join. Compute the IO cost for joining R, S and then joining the result with T. Assume that the tables are not sorted before starting. Also assume that we do not need to do any back up as described in lecture.

Question 4.5 [2 points]

Again, using a sort-merge join, compute the IO cost for joining S, T and then joining the result with R. Assume that the tables are not sorted before starting. Also assume that we do not need to do any back up as described in lecture.

Question 4.6 [2 points]

Now suppose we only want to join R and S (with sort-merge join) but this time all values for the join attribute are the same. What would be the IO cost now?

Question 4.7 [2 points]

Now consider using a hash join. Compute the IO cost for joining R, S and then joining the result with T.

Question 4.8 [2 points]

Again, using a hash join, compute the IO cost for joining S, T and then joining the result with R.

Question 4.9 [5 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 hash join for $join1$ and sort-merge join for $join2$ is cheaper than sort-merge join for $join1$ and hash join for $join2$ by adjusting buffer sizes and the number of pages in each table. Provide your answer in terms of the variables listed at the start of this question (such as P(R), P(R, S), B, etc.). The output sizes you choose for P(R, S) and P(R, S, T) must be non-zero and feasible (e.g. the maximum output size of $join1$ is P(R)*P(S)).