CS 186
Spring 2021

Hashing

1 Motivation

Sometimes, sorting is a bit overkill for the problem. In a lot of cases, all we want is to group the same value together, but we do not actually care about the order the values appear in (think GROUP BY or de-duplication). In a database, grouping like values together is called hashing. We cannot build a hash table in the standard way you learned in 61B for the same reason we could not use quick sort in the last note; we cannot fit all of our data in memory! Let's see how to build an efficient out-of-core hashing algorithm.

2 General Strategy

Because we cannot fit all of the data in memory at once, we'll need to build several different hash tables and concatenate them together. There is a problem with this idea though. What happens if we build two separate hash tables that each have the same value in them (e.g. "Brian" occurs in both tables)? Concatenating the the tables will result in some of the "Brian"s not being right next to each other.

To fix this, before building a hash table out of the data in memory, we need to guarantee that if a certain value is in memory, all of its occurrences are also in memory. In other words, if "Brian" occurs in memory at least once, then we can only build the hash table if every occurrence of "Brian" in our data is currently in memory. This ensures that values can only appear in one hash table, making the hash tables safe to concatenate.

3 The Algorithm

We will use a divide and conquer algorithm to solve this problem. The "divide" phase will be partitioning passes, and the "conquer" phase will be actually constructing the hash tables. Just like in the sorting note, we will assume that we have B buffer frames available to us.

The first partitioning pass will hash each record to $B-1$ **partitions**. A partition is a set of pages such that the values on the pages all hash to the same value (for the hash function used to construct the partition). We do this by using $B-1$ output buffers. When an output buffer fills up we flush the page to disk. When that buffer fills up the next time, we place it adjacent to the page we flushed to disk before from that same buffer. The most important property of each partition is that if a certain value appears in that partition, all occurrences of that value in our data appear in that partition. In other words, if "Brian" appears in that partition, "Brian" will not appear in any other partition. This is because "Brian" always hashes to the same value, so it cannot possibly end up in a different partition. We only have $B-1$ partitions because we need to save one buffer frame to be the input buffer.

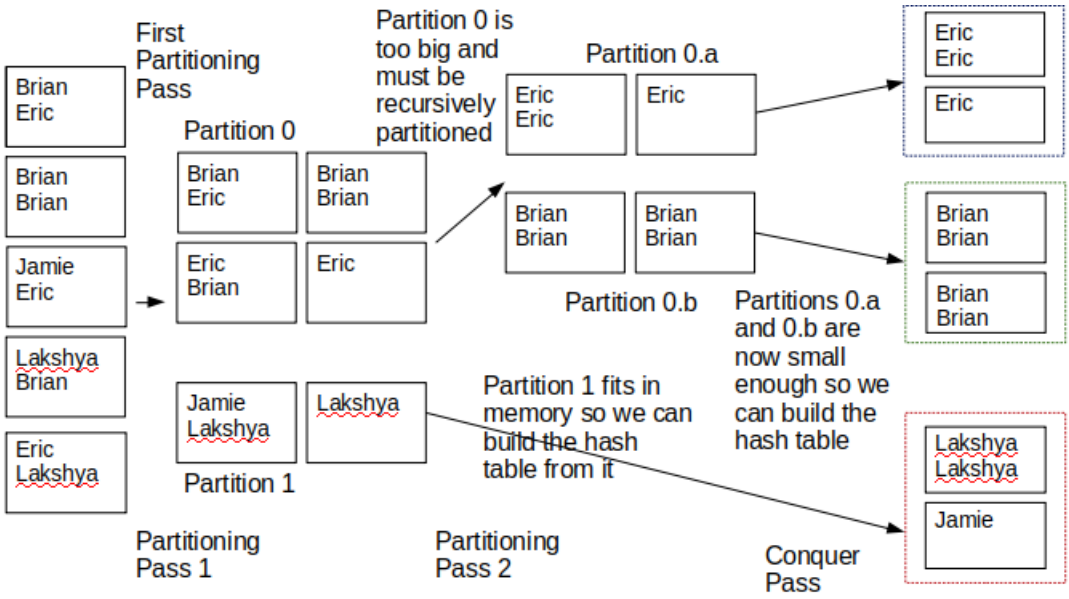
After this first partitioning pass, we can go right to the hash table building phase for the partitions that fit in memory. What does it mean to fit in memory? Fitting in memory means the partition

must be B pages big or less. For the partitions that are too big, we simply repartition them using a different hash function than we used in the first pass. Why a different hash function? If we reused the original function, every value would hash to its original partition so the partitions would not get any smaller. We can recursively partition as many times as necessary until all of the partitions have at most B pages.

Now all of our partitions can fit in memory, and we know that all like values occur in the same partition. The only thing left to do is build a hash table for each partition and write each hash table to disk.

4 Example

In the following example, we will assume that we have $B = 3$ buffer pages available to us. We also assume that Brian and Eric hash to the same value for the first hash function but different values for the second hash function, and Jamie and Lakshya hash to the same value for the first hash function.



You can see that Partition 0 is too big because it contains 4 pages, but we only have 3 buffer frames available to us. When it gets recursively partitioned, however, the subpartitions (0.a and 0.b) are both only 2 pages long, so they can now fit into memory. You can also see that after the final "conquer" pass, all of the like values are next to each other, which is our end goal.

5 Analysis of External Hashing

We're not going to be able to create a simple formula to count the number of I/Os like in the sorting algorithm because we do not know how large the partitions will be. One of the first things we need to recognize is that it is possible for the number of pages in the table to increase after a partitioning pass. To see why, consider the following table in which we can fit two integers on a page:

[1, 2] [1, 4] [3, 4]

Let's assume B=3, so we only divide the data into 2 partitions. Let's assume 1 and 3 hash to partition 1, and 2 and 4 hash to partition 2. After partitioning, partition 1 will have:

[1, 1], [3]

And partition 2 will have:

[4, 2], [4]

Notice that we now have 4 pages when we only started with 3. Therefore, the only reliable way to count the number of I/Os is to go through each pass and see exactly what will be read and what will be written. Let m be the total number of partitioning passes required, let r_i be the number of pages you need to read in for partitioning pass i, let w_i be the number of pages you need to write out of partitioning pass i, and let X be the total number of pages we have after partitioning that we need to build our hash tables out of. Here is a formula for the number of I/Os:

$$(\sum_{i=1}^{m} r_i + w_i) + 2X$$

The summation doesn't tell us anything that we didn't already know; we need to go through each pass and figure out exactly what was read and written. The final $2X$ part, says that in order to build our hash tables, we need to read and write every page that we have after the partitioning passes.

Here are some important properties:

1. $r_0 = N$
2. $r_i \leq w_i$
3. $w_i \geq r_{i+1}$
4. $X \geq N$

Property 1 says that we must read in every page during the first partitioning pass. This comes straight from the algorithm.

Property 2 says that during a partitioning pass we will write out at least as many pages as we read in. This comes directly from the explanation above - we may create additional pages during a partitioning pass.

Property 3 says that we will not read in more pages than what we wrote out during the partitioning pass before. In the worst case, every partition from pass i will need to be repartitioned, so this would require us to read in every page. In most cases, however, some partitions will be small enough to fit in memory, so we can read in fewer pages than we produced during the previous pass.

Property 4 says that the number of pages we will build our hash table out of is at least as big as the number of data pages we started with. This comes from the fact that the partitioning passes can only increase the number of data pages, not decrease them.

6 Practice Questions

1) How many IOs does it take to hash a 500 page table with B = 10 buffer pages? Assume that we use perfect hash functions.

2) We want to hash a 30 page table using B=6 buffer pages. Assume that during the first partitioning pass, 1 partition will get 10 data pages and the rest of the pages will be divided evenly among the other partitions. Also assume that the hash function(s) we use for recursive partitioning are perfect. How many IOs does it take to hash this table?

3) If we had 20 buffer pages to externally hash elements, what is the minimum number of pages we could externally hash to guarantee that we would have to use recursive partitioning?

7 Solutions

1) The first partitioning pass divides the 500 pages into 9 partitions. This means that each partition will have 500 / 9 = 55.6 $\implies$ 56 pages of data. We had to read in the 500 original pages, but we have to write out a total of 56 * 9 = 504 because each partition has 56 pages and there are 9 partitions. The total number of IOs for this pass is therefore 500 + 504 = 1004.

We cannot fit any partition into memory because they all have 56 pages, so we need to recursively partition all of them. On the next partitioning pass each partition will be divided into 9 new partitions (so 9*9 = 81 total partitions) with 56 / 9 = 6.22 $\implies$ 7 pages each. This pass needed to read in the 504 pages from the previous pass and write out 81 * 7 = 567 pages for a total of 1071 IOs.

Now each partition is small enough to fit into memory. The final conquer pass will read in each partition from the previous pass and write it back out to build the hash table. This means every page from the previous pass is read and written once for a total of 567 + 567 = 1134 IOs.

Adding up the IOs from each pass gives a total of 1004 + 1071 + 1134 = **3209 IOs**

2) There will be 5 partitions in total (B-1). The first partition gets 10 pages, so the other 20 will be divided over the other 4 partitions, meaning each of those partitions gets 5 buffer pages. We had to read in all 30 pages to partition them, and we write out 5 * 4 + 10 = 30 pages as well, meaning the first partitioning pass takes 60 IOs.

The partitions that are 5 pages do not need to be recursively partitioned because they fit in memory. The partition that is 10 pages long, however, does. This partition will be repartitioned into 5 new partitions of size 10/5 = 2 pages. We read in all 10 pages for this partition, and we wrote out 5 * 2 = 10 pages, meaning that this recursive partitioning pass takes 20 IOs.

The final conquer pass needs to read in and write out every partition, so it will take 4 * 5 * 2 = 40 IOs to conquer the partitions that did not need to be repartitioned and 5 * 2 * 2 = 20 IOs to conquer the partitions that were created from recursive partitioning, for a total of 60 IOs.

Finally, add up the IOs from each pass and get a total of 60 + 20 + 60 = **140 IOs**.

3) **381 pages**. Since B = 20, we can potentially hash upto B * (B - 1) = 20 * (20 - 1) = 380 pages without doing recursive partitioning. If we want to guarantee recursive partitioning, we need one more page then this, giving us 381 pages as our solution.

Even with a perfect hash function, one page would have B+1 pages, which would require recursive partitioning since B+1 can't fit into memory when there are B buffer pages.