Implementation Details

Misc

We have created a new class `SimpleWriter<T>` for the purpose of easily writing to a output file. A `SimpleWriter<T>` object can be used to write to a binary output file using the method `write(T record)`. It automatically takes care of allocating new pages as old pages get full. It is guaranteed to keep no more than 1 pinned page in the buffer. Lastly once the writing work is complete we call `fill(INT_MIN)` to fill the remaining portion of the last page with `INT_MIN`.

Similarly we have created a new function `record_at<T>(char* data, int offset)` to read the record of type `T` at offset `offset` of page data `data`.

Internally, we *always* use `memcpy` to read and write from a page. This is because the `page.GetData()` returns a `char*`. The CPP language standard does not guarantee that `char*` will be word aligned, therefore directly typecasting it into a `int*` is unsafe (as `int*` should necessarily be word(4-bytes) aligned). Hence, we stick with `memcpy`.

We do not stored the file in memory as mentioned in the spec, however, in join 1 and join 2 we read some numbers that are present in the currently pinned pages in the buffer into a vector to sort them and process efficiently. We have taken care to clear the vector as soon as those pages are unpinned. Since the buffer size is a compile time constant therefore the size of our in-memory vector also remains compile time constant (independent of file size).

We tested our implementations using randomly generated test cases. Tester code is, however, not included in this submission.

Algorithm Details

Linear Search

Implementation of linear search is fairly standard. We read the query file using `fstream` and for each integer in the query file we read each page of input file one by one and for each integer in the input file matching with the query integer we output it to the output file. Since, we read the input file page by page, and unpin each page before reading the next page – atmost one pinned page is used for scanning input file. Similarly we use `SimpleWriter<int>` for writing to the output file so it also uses at most one pinned page. Overall, we can complete the search in at most 2 pinned pages. As a special case we first also check whether the input file is empty or not, in which case we abort the search right away.

Number of page reads : $O(QN)$
Worst case number of page writes : $O(QN)$

Where Q is number of queries and N is number of pages in the input file.

Binary Search

For binary search in the implementation we maintain the following invariant (assuming we have to search for number `num` and the number of last page in the file is `lastpage`) – we maintain 2 pointers `lo` and `hi` such that all pages in the range [0, lo] have no number greater than equal to `num`, and all pages in the range [`hi` , `lastpage`] have at least one number greater than or equal to `num`. Initially we set `lo = -1` and `hi = lastpage + 1`. Then, we read the page with page num `mid = (lo+hi)/2` and accordingly adjust either the `lo` or `hi` pointer afterwards. The search terminates as soon as `lo >= hi-1`. After the search terminates we are guaranteed that `hi` is the first page number which has at least one number not smaller than `num`. So we then do a linear scan from this page until we stop finding any occurrences of `num`.

Here, we have also applied an optimization of early-stopping. Similar to linear search once again we unpin each page before pinning the next page so that at all times only one page of the input file is pinned, and for output we use simple writer which internally also ensures the same. So binary search also requires at most 2 pinned pages.

Number of page reads : $O(Q(\log N + N'))$
Number of page writes : $O(QN')$

Where Q is the number of queries, N is the number of page in input file, and N' is the avg. number of pages a query number spans in the input file (so for example if each query number spans 2 pages in the input file then $N' = 2$)

So as we can see that the binary search provides significant improvement over linear search in terms of number of page reads.

Deletion

For deletion, if we want to delete a number `num` from compacted input file, we first use the binary search algorithm from above to find the first occurrence of this `num` in the input file, and then use a linear search starting from this page to find the last occurrence of `num`. Clearly, we need to delete all pages/numbers between first and last occurrence of `num`. Here a few cases may arise. First case is that it is possible that last occurrence of the `num` is actually the last offset in `file`. In this case we may simply dispose all the pages from first occurrence to last occurrence to compact the file. In the other case, we first use a 2 pointer approach to copy all integers from beyond the last occurrence to the first occurrence so that file is now compacted and then finally dispose all the remaining pages at the end (also, the last indisposed page may need to be filled with `INT_MIN`). We repeat the algorithm for each query number. Adhering to the specifications, we compact the file properly after each query number instead of compacting only once at the end.

At any time at most 2 pinned pages can be in buffer (one for the first pointer, and one for the second pointer). So therefore deletion takes can complete in the minimum buffer size of 2. We reckon that it is also possible to do it one pinned page also but it would double the number of reads required.

Number of page reads : $O(Q(\log N + N/2))$
Number of page writes : $O(QN/2)$

Where Q is number of queries and N is the page in the input file. An explanation for number of page reads - first doing the binary search requires $O(\log N)$ page reads, then we may potentially need to shift upto $N/2$ pages on average so we need to read all these pages so that is why complexity is given as such.

Join 1

For join 1 the specifications requires us to use 1 page for input 1, and B-2 pages for input2 (the one other page is for output, of course). So to make most efficient use of buffer space we use the following approach. We first read $B-2$ pages from the input 2 and pin them in the buffer. We store all the integers in these $B-2$ pages into a vector to sort them. Then, we read each page from input 1, one by one, and for each integer in that page we search in the vector for possible matches and output all the matches. Once the input 1 is fully scanned, we clear the vector, as well as unpin all $B-2$ pages. Then, we repeat the above process for the next $B-2$ pages in the input 2, and keep repeating until input 2 is also fully scanned. Note that at any point of time we only store those integers in the vector which are already present in the pinned pages in the buffer, so we are not using any additional memory except for some constant amount (B is a compile-time constant). Also note that sorting the vector has not impact except for improving run-time performance (we can do a binary search on sorted vector instead of doing a linear scan).

Number of page reads : $O(N_1\frac{N_2}{B-2}+N_2)$
Number of page writes : $O(N_1N_2)$

Where N_1 is number of pages in input 1, N_2 is number of pages in input 2, B is the buffer size (constant).

Explanation for number of page reads : Note that we need to do a full scan of input 1 (full scan takes $O(N_1)$ reads) as many times as it takes to exhaust input 2. Since we read $B-2$ pages of input 2 at a time, this means we would need to do a full scan of input 1 approximately $N_2/(B-2)$ times.

Join 2

In join 2 we are allowed to use $B-2$ buffer slots for input 1, and only 1 slot for input 2 (of course 1 slot for output also). For join 2, we first read $B-2$ pages of input 1, and pin them in the buffer. Then we read all integers from these $B-2$ pages and store and sort them into a vector. Next for each integer thus obtained we do a binary search on input 2 (input 2 is sorted) to find the first occurrence of that integer in input 2 and then do a linear scan from that occurrence onwards to output each match found. Afterwards, we clear the vector and unpin all $B-2$ pages to read next $B-2$ pages from input 1 and repeat the process until input 1 is fully scanned. Here, we use 2 important optimizations, first is that we collapse duplicates in the $B-2$ pages of input 1 so as to make a single binary search even for multiple occurrence of the same number. Secondly, as the vector of $B-2$ pages is sorted in-memory, we can dynamically adjust the domain of binary search so that we don't have to perform the binary search in the entire file.

Number of page reads : $O(N_1(\log\frac{N_2}{\alpha}+N_2'))$
Number of page writes : $O(N_1N_2)$

Where N_1 is number of pages in input 1, N_2 is number of pages in input 2, α is some undetermined constant, N_2' is avg span of each number in sorted input 2.

Explanation for number of page reads : For each number in input 1 we first do a binary search in input 2. Binary search takes $O(\log N_2)$ time but because we are dynamically constraining domain of binary search the size of space is reduced (that is why the α). Secondly, after the binary search we do a linear search do discover number of occurrences, the extent of linear search will depend on number of occurrences which we have assumed to be N_2' on average.