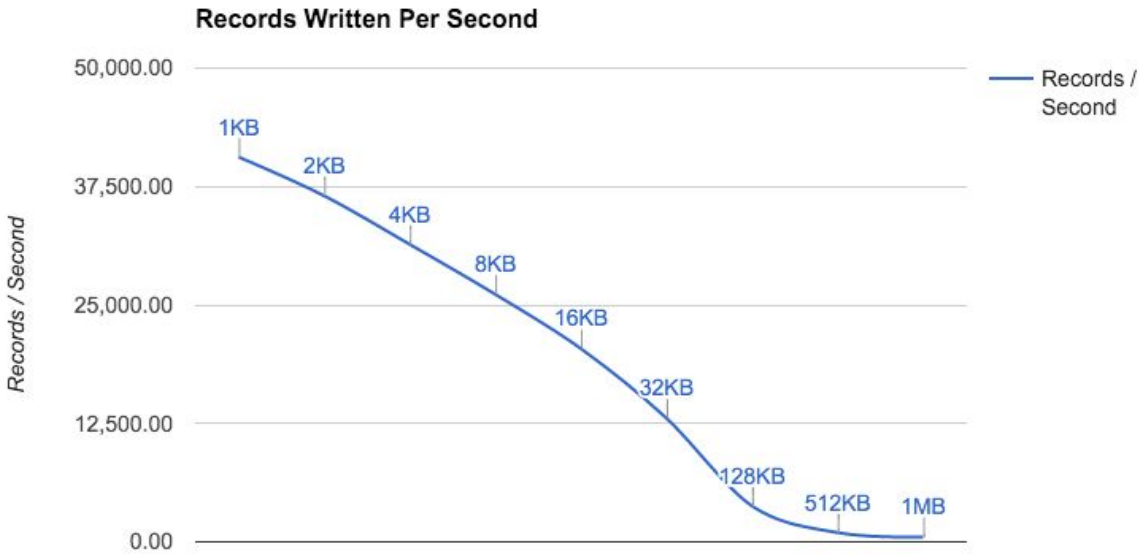
CSC443 Assignment 2

Data Layout

November 18, 2015

Binuri Walpitagamage - 999141780
Ammar Javed - 999058273

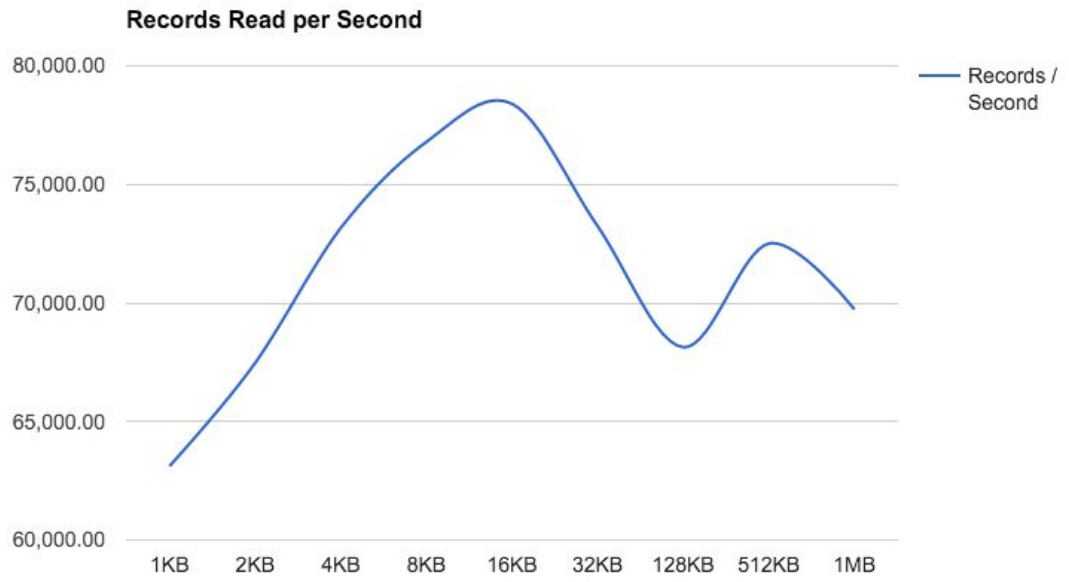
Section 3: Page Layout



Graph 3.1

This graph depicts the number of records written per second (write_fixed_len_pages) using different block sizes. We took the average time of 3 runs to ensure consistency.

We started timing as soon as the csv_file file pointer(to read from) and the page_file pointer(to write to) was ope. The code path timed included reading in individual records from the CSV file, initializing a page and filling it with that record, plus as many others which would fit, and once full, writing it to the disk.



Graph 3.2

This graph depicts the number of records read per second using different block sizes. We took the average time of 3 runs to ensure consistency.

The code path we timed for reading included reading in the metadata for the first page (to bring back consistent pages as they were updated), calculating the page data from the meta data, and reading in the records we had stored in that page. We did not include the time it took to dump the pages to the screen, as we were only interested in how long it took to read the pages and records.

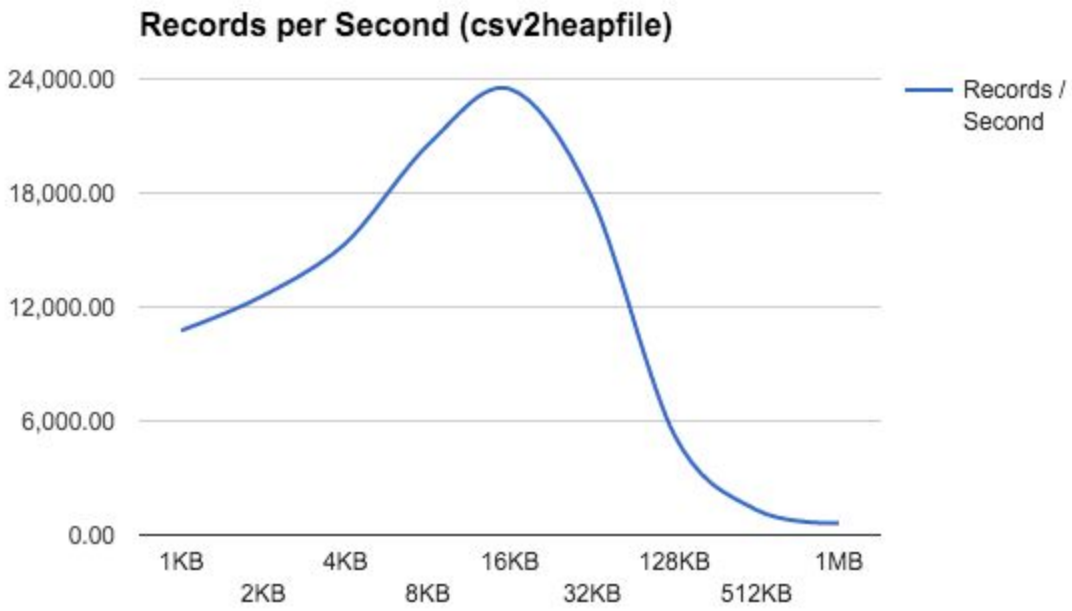
Our experiments confirm that the page based format of handling data is superior to the CSV format, especially when writing to disk. From assignment 1, we know that writing to the disk is the most expensive operation. From our current data, we can see how smaller block sizes being written to the disk would perform much faster than the larger block sizes, which tend to emulate the entire CSV file being written from memory onto the disk. Even though writing large block sizes may result in a lower number of write operations being performed, writing small chunks of memory is faster even with the larger overhead; and Graph 3.1 confirms that as the block sizes get larger, the write efficiency decreases rapidly.

Graph 3.2 confirms the same is true for reading records. There data hits a peak at 16kb block size at which we can read in a very large amount of records in a second, however as we increase the page sizes, you see performance dropping. Another smaller peak occurs at 512kb.

Compared to graph 3.1, read operations are much faster but also shows that, multiple small calls to it are still faster than fewer longer operations (to fill up the larger block).

A shortcoming of organizing pages this way is that without the meta data, you are forced to scan through the entire page block by block to try and see where there might be room to insert a new record into, or find a certain RID. In order to relieve the search overhead, we decided to write out the page meta data, so that we can decide whether to continue or to skip reading the page. Even with this implementation, you may still end up iterating to the end of the pages before finding the free slot required. This is the shortcoming that the heapfile addresses.

Section 4: Heap File



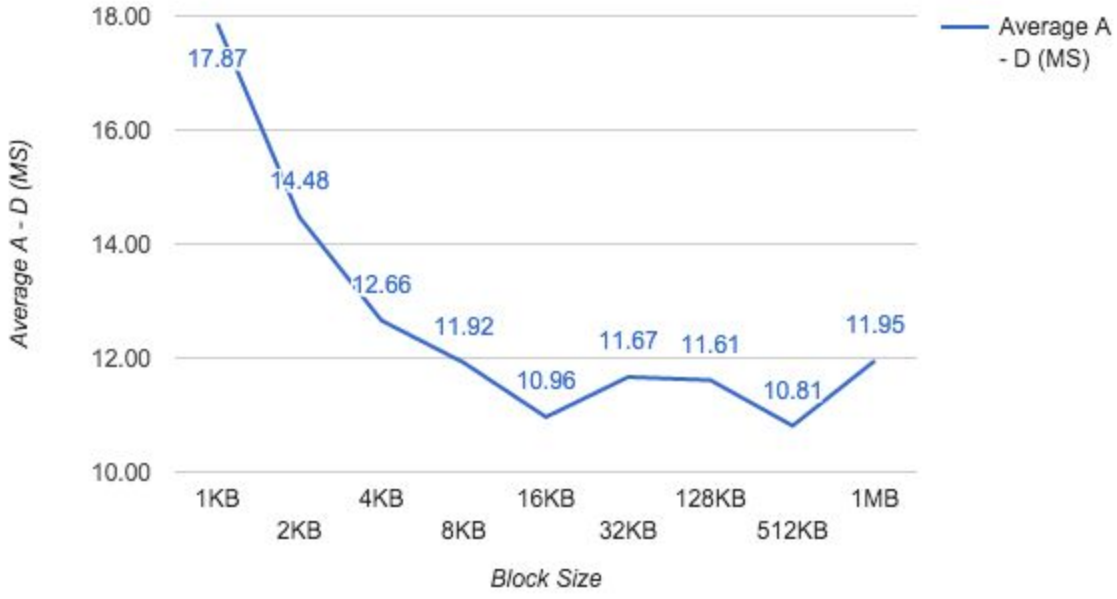
Graph 4.1

The records per second throughput of the csv2heapfile function using multiple different page_sizes. We took the average time of 3 runs to ensure consistency.

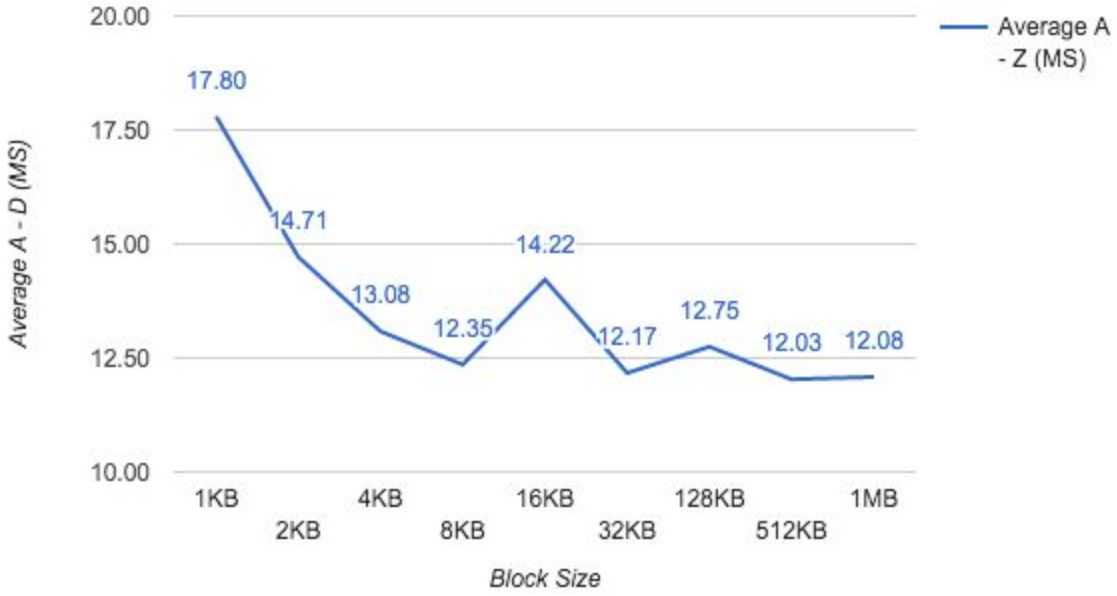
It was interesting to see a curve which resembled both the read and write to disk operations we tested earlier. As expected, for writing, the records per second dropped significantly as we got to the larger page_sizes. What's more interesting to note is that, because we are doing significantly more reads (since we have to travel back and forth in the file to update the heap directories and write out both the data and [additional] directory pages) we notice a peak in our

performance around the same page_sizes as we saw in Graph 3.2 . This supports the fact that a page size between 8KB to 32KB is the ideal pages size range for reading from a disk.

Select



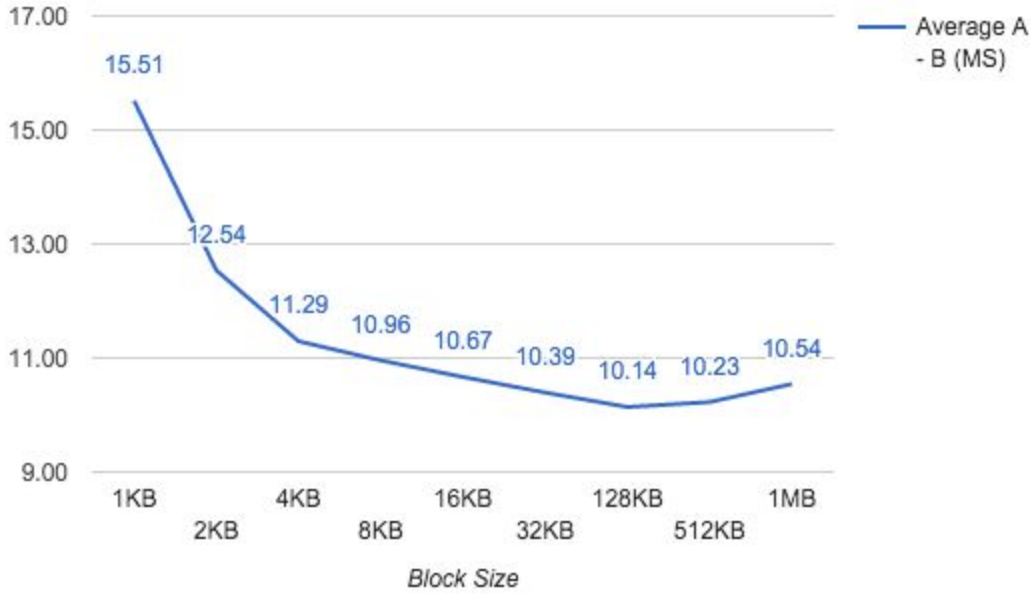
Graph 4.2
Start = A End = D
We took the average time of 3 runs to ensure consistency.



Graph 4.3

Start = A End = Z

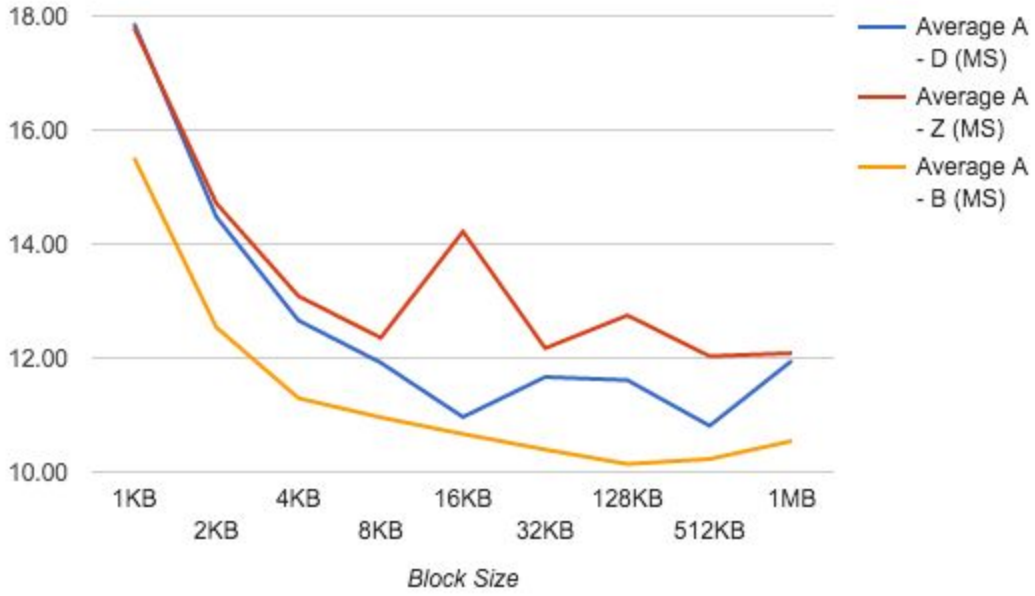
We took the average time of 3 runs to ensure consistency.



Graph 4.4

Start = A End = B

We took the average time of 3 runs to ensure consistency.



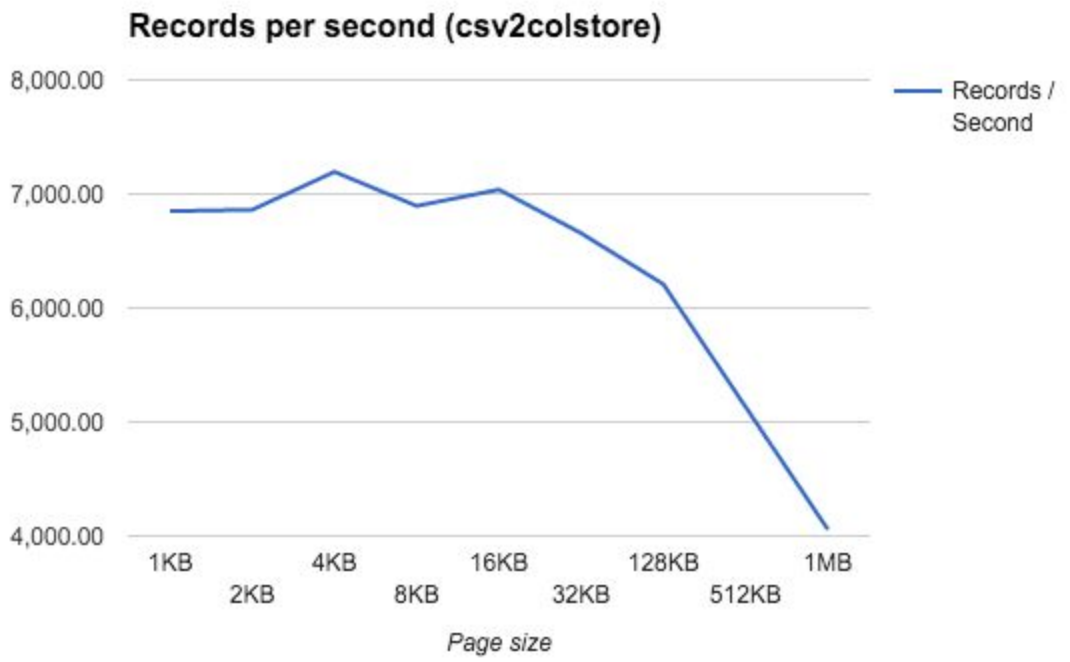
Graph 4.5

We took the average time of 3 runs to ensure consistency.

The Graphs 4.2, 4.3, 4.4, 4.5 measure the runtime of the select queries in milliseconds. A couple of things we noticed were that the run times of the queries got increasingly faster as your page sizes got bigger. Looking at Graph 4.5 however, which shows the runtimes of different block sizes along with different Start and End criteria, you see that the more ‘correct’ attributes you had to output, the slower your query ran, because of the increase in the number of calculations you had to do. So the smaller your Start-End range was, the faster your query ran.

Section 5: Column Store

csv2colstore

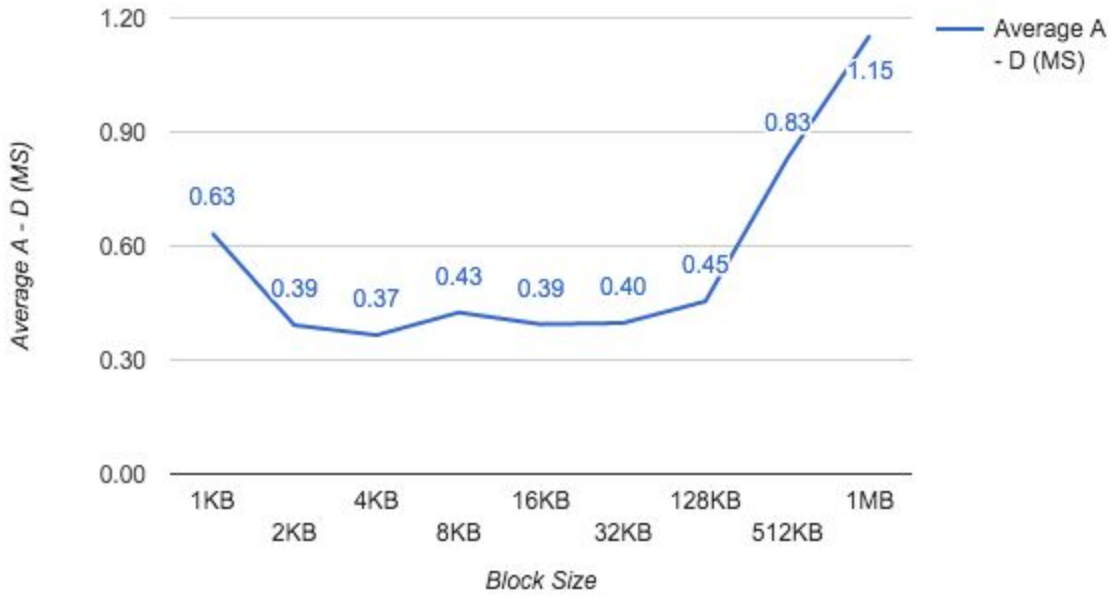


Graph 5.1

The records per second throughput of the csv2colstore function using multiple different page_sizes. We took the average time of 3 runs to ensure consistency.

While the trend between the Graph 4.1 and Graph 5.1 is similar, the number of records per second is vastly different. The reason for the throughput to be so much lower is that given any number of records with 100 attributes each, you are going to be writing out a minimum of 100 pages with any given block size. This does not include the added overhead of setting up the heap directory. With the row based implementation, you are only going to initialize and write out pages = records / (page_size / record_size). The trade off with a slow initialization however is a performance boost in query execution, as the following sections will show.

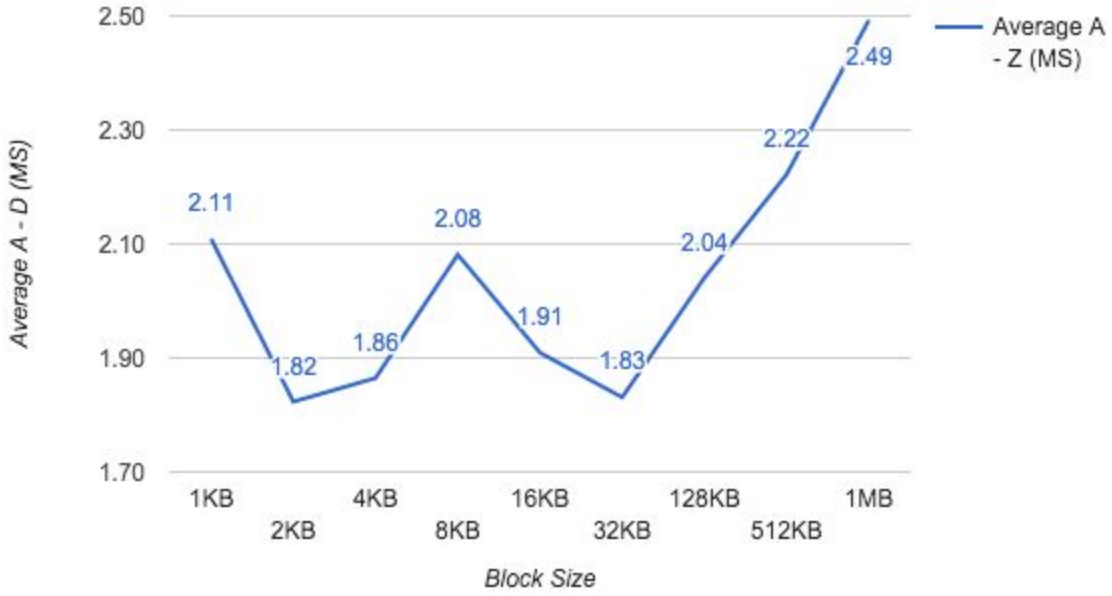
select2



Page 5.2

Start = A End = D

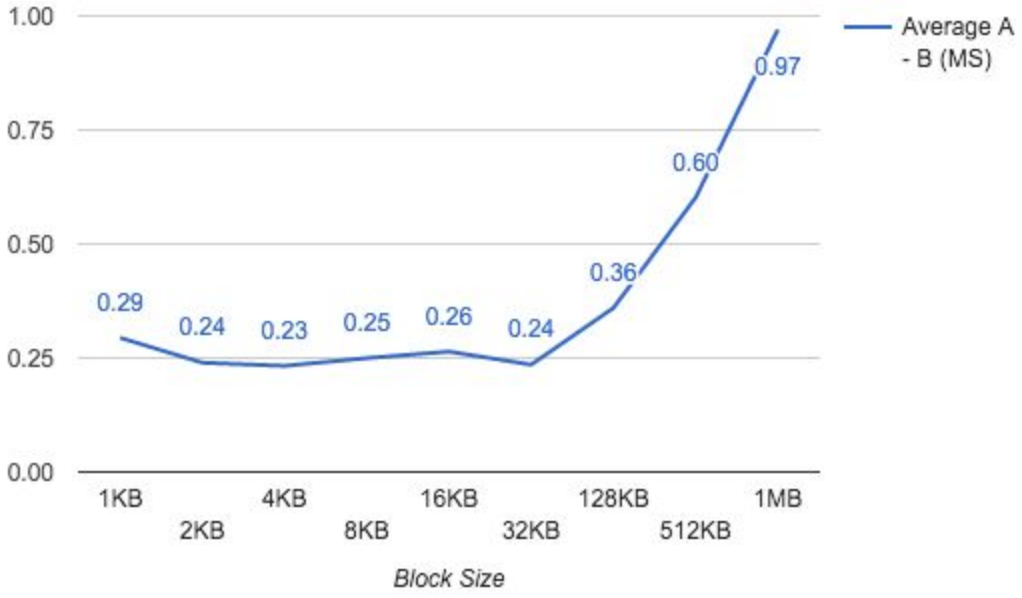
We took the average time of 3 runs to ensure consistency.



Graph 5.3

Start = A End = Z

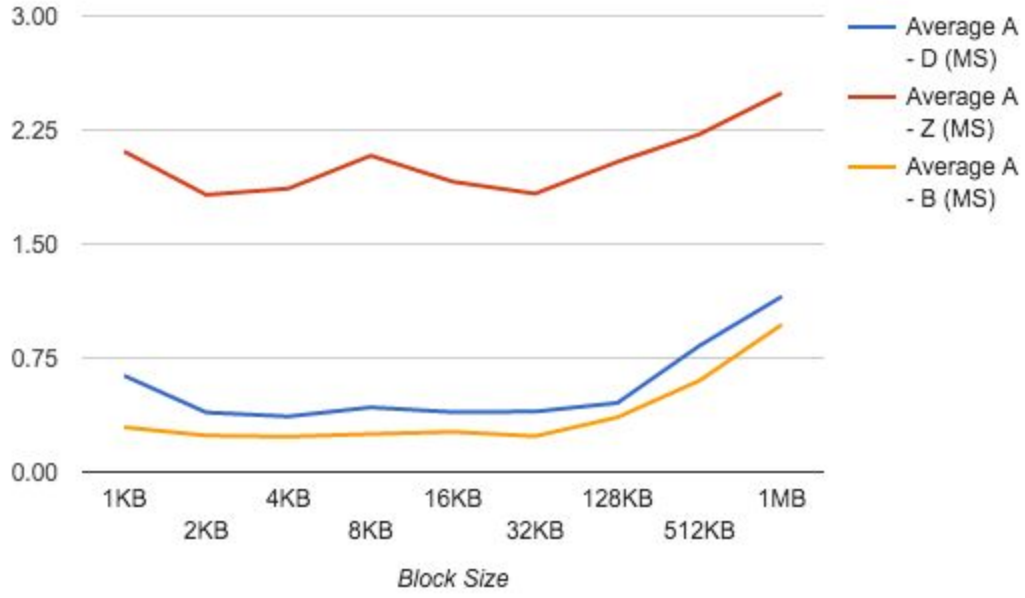
We took the average time of 3 runs to ensure consistency.



Graph 5.4

Start = A End = B

We took the average time of 3 runs to ensure consistency.

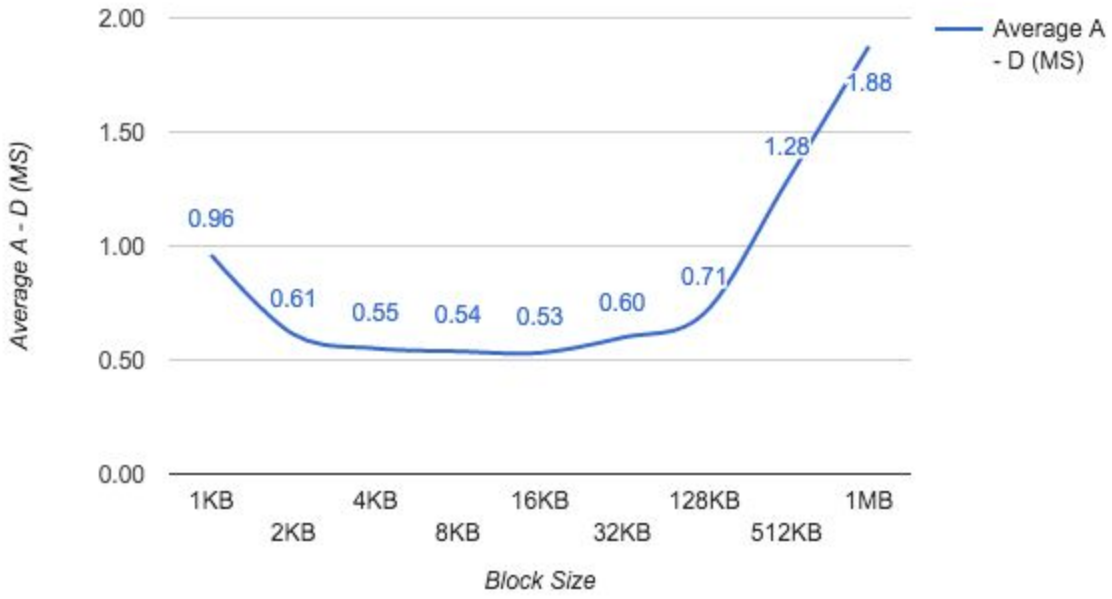


Graph 5.5

We took the average time of 3 runs to ensure consistency.

Interestingly, we saw a minimum of 8x performance boost in the select2 query compared to the select query on the row store database. It is easily explained by the fact that to perform string comparisons on one attribute/column, you had to read in all the records which included the other 99 attribute values that were not needed. However in the column store database, all of the records' attribute which you wanted to compare were stored in 1/100th of the number of pages.

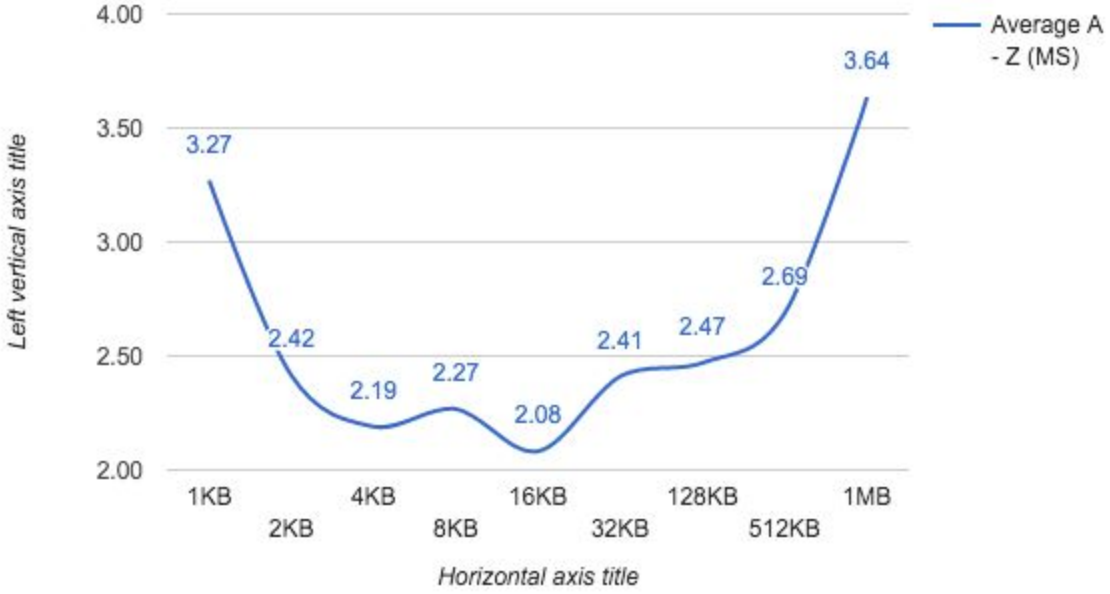
select3



Page 5.6

Start = A End = D

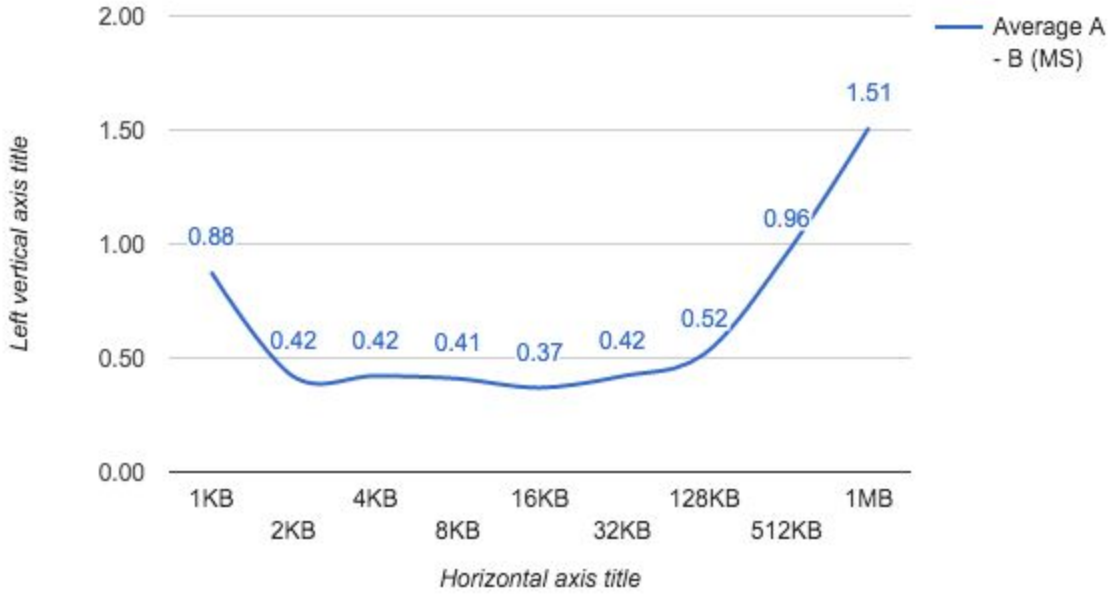
We took the average time of 3 runs to ensure consistency.



Graph 5.7

Start = A End = Z

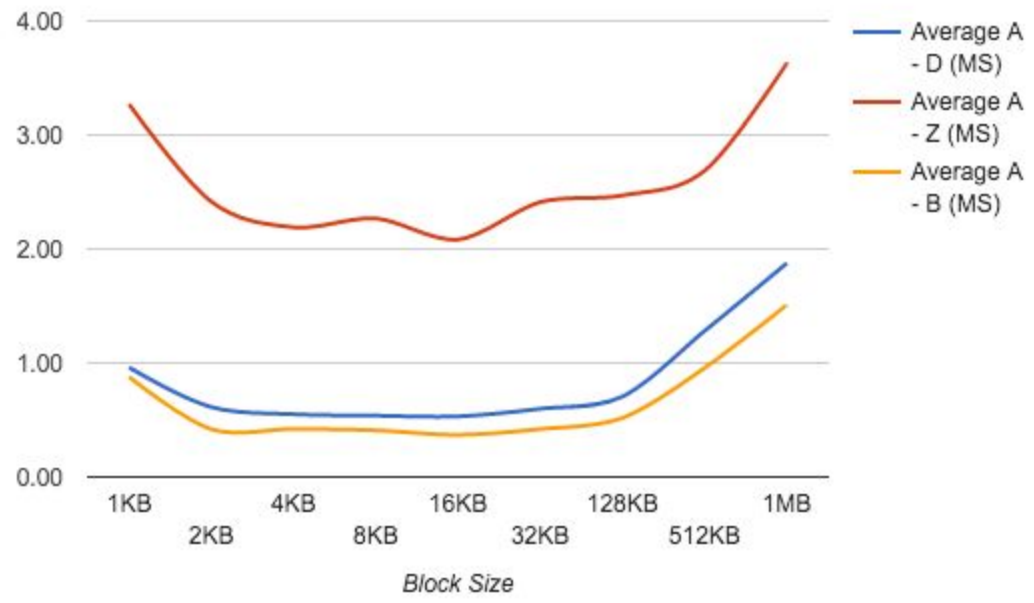
We took the average time of 3 runs to ensure consistency.



Graph 5.8

Start = A End = B

We took the average time of 3 runs to ensure consistency.



Graph 5.9

We took the average time of 3 runs to ensure consistency.

The trend of select3 across various block sizes is very similar to the run times of select2 and for the same reasons performed faster than select. The increase in select3's runtime is due to the fact that once we perform the string comparisons to calculate which records are valid to be selected, we read in another (potentially more than one) page[s] of the attribute which was specified by <return_attribute_id> that we have to project. There are certain queries in which select3 would actually perform the same as select2, and that is when the <attribute_id> and the <return_attribute_id> is the same, since we have to do the comparison and the projection on the same column. Unfortunately our code ends up reading in the <return_attribute_id> again regardless of whether it matches the <attribute_id> or not but would have been a nice improvement.