CSC 263 Assignment 1

Nikola Danevski Vladimir Maksimovski

Introduction

For this assignment, we implemented a database with a single, fixed schema, and some querying utilities. We finished all the assigned parts, and our report is shown below.

Experimental Part

Results were gathered on the cycle1.cs.rochester.edu machines. These machines run a magnetic hard drive. The log of the experiment is stored in "logs.tar.gz". All result graphs are shown in the Result Tables section.

Record Experiments

a) Each attribute is 10 characters. For easier interoperability with C, we also included null characters, to ensure strings are null-terminated. We have 100 attributes in the schema, resulting in a calculated record serialization size of 100*11 = 1100 bytes.

b) We created a simple C++ function printing the size of a serialization. To run the experiment, execute

```
make clean
make all
./serialization_experiment
```

Running this experiment prints that the serialization length is 1100 bytes, as we expected.

Page Experiments

a) To run experiments, execute "./page_experiment.sh". The experiments compute write and read performance in terms of write_fixed_len_page and read_fixed_len_pages respectively for varying page sizes, and for a constant record count of 20,000.

b) The y-axes of the tutorial graphs and our write graph are reciprocally related, which explains why the curve in opposite directions. In our graph, the performance for small block sizes (up to 65536 bytes per page, or approximately 60 records per page) is very bad. The performance afterwards tapers off, and peaks at a page size of approximately 1MB, or around 900 records. Increasing page sizes above that point reduces performance.
Comparing this with the graph in the tutorial, the bad performance ranges up to 100 records/page, it tapers off afterwards, and reaches an optimum at 100,000 records/page.
These results are in line with our expectations. Two factors impact the performance: the number of I/O requests, and the fraction of pages that gets unused after allocating all 20,000 records.
At smaller page sizes, we incur more disk I/O requests, which explains the slowdown at small page sizes. Meanwhile, for larger page sizes, the performance overhead of the I/O requests becomes insignificant. The second factor becomes relevant, however. If the page size is large enough that the data can be fit into two pages, there might be a lot of unused data in the second page. We expect that this is the reason why the graph tapers off.

The table for read performance does not show any meaningful patterns. We believe this is because 20,000 records get read so fast that the overhead is higher than the read time.

c) A CSV delimits tuples by a newline character, with no padding involved. This means that if we modify the first tuple e.g. modifying the first attribute from "AB" to "ABC", all succeeding bytes will get pushed to the right. This means that there is no way to modify an attribute in a CSV file without updating the whole file. With our page-based format, we can update single attributes easily, as long as the new attribute values use <= 10 bytes.

d) Our implementation always stores a record in one single page. This results in wasted space when a page has extra storage that isn't enough for another record e.g. a page with 1600 bytes inside, of which at most 1105 would be used for a record, the bit array, and the maximal capacity integer.

Heap File Experiments

a) The performance of `csv2heapfile` (write performance) can be seen in the graph below. The performance is the worst when the page size is the least (it starts at 1105B) and steadily increases up until around 211228B = 2×10^5B = 200 kB when it flattens out and continues with more or less the same performance until the biggest page size of 13518340B = 13×10^6B = 13 MB.

b) Our experiments showed that as the page size increases from 1kB to 16kB, we have an increase in performance of the query. After that, the speed does not change much as the page size increases.

c) In order to compare on how `start` and `end` affect the performance of the query, we decided to test select with 3 different pairs of parameters: (NIK, NIK), (N, V) and (A, ZZZZZ), where the first one is most restrictive, and the last one is least restrictive. Our results here vary and our assumption is that this is due to the C function `strncmp` which we use to lexicographically compare strings. Given that the records have attributes that are of size 10, the running time of `strncmp` is constant (and extremely small), thus the results vary and it is not the case that the most restrictive choice of `start` and `end` will necessarily mean best performance.

Column Store Experiments

a) `csv2colstore` behaves similarly to `csv2heapfile` according to our experiments. The only exception is that after its peak performance of 1024 rec/s at a page size of 1MB, it actually has a decline in performance until the maximum page size of 13MB.

b) The performance of `select2` and `select3` versus page size is fairly similar as can be seen from the graphs which look almost identical. This is due to the fact that our implementation of `select2` just calls `select3` with the same value for the selection and return attribute. When compared to `select` which has row-oriented storage, they seem to perform better, but there is a key reason why this is the case;
Namely, when testing `select` on the row-oriented storage implementation, we used 20000 records, while when testing `select2` and `select3` we used 1000 due to memory limitations on the testing machines. Given that the write time was magnitudes bigger when writing column-oriented storage, it is expected that this is the case for the reading time. Thus, we

decided to lower the number of records, while still having performance on the y-axis as records/second.

c) Just as in the situation with `csv2heapfile`, the choice of `start` and `end` did not have much effect on the performance. One thing we noticed is that the least restrictive range happens to be the fastest the most, but this is not enough to make a valid conclusion regarding this as the performance is not much better.

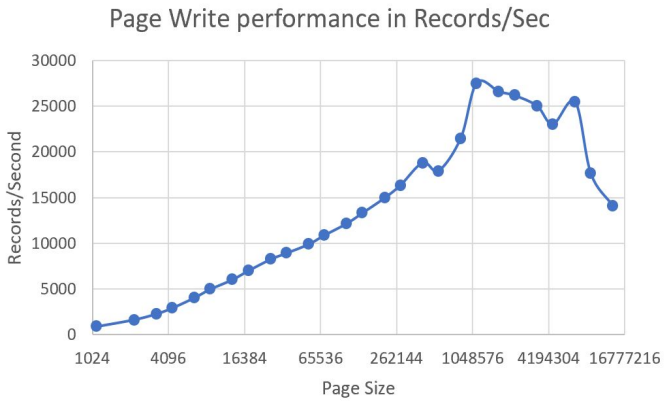
Documentation

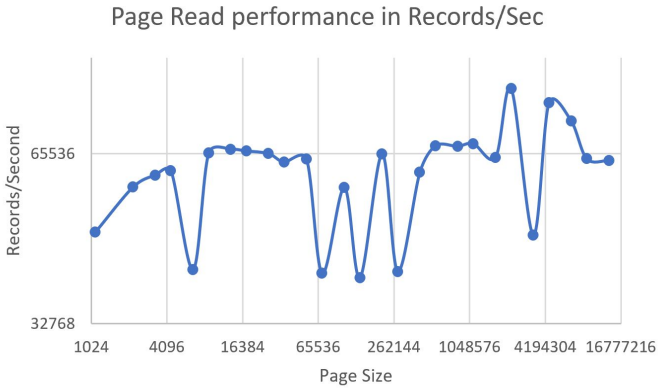
- src/ - directory containing the source code for all files
    - utils.cpp, utils.hpp – Header/Implementation files for utility functions in this project, like converting a C++ String to a dynamically-allocated char*.
    - CSVUtils.cpp, CSVUtils.hpp - Header/Implementation files for functions which read in CSV files.
    - Record.hpp, Record.cpp – Header and implementation files for interacting with Records.
    - Page.cpp, Page.hpp – Header and implementation files for Pages.
    - HeapFile.cpp, HeapFile.hpp – Header/Implementation files for HeapFiles
    - ColStore.cpp, hpp - Header/Implementation files for Column-based relational data stores.
    - Serialization_experiment.cpp - Source code file for the record experiment.
    - WriteFixedLenPages.cpp, ReadFixedLenPages.cpp - Source files for generating the respective executables for Page operations.
    - Csv2Heapfile.cpp, Scan.cpp, Insert.cpp, Update.cpp, Delete.cpp, Select.cpp - Source files for generating the respective executables for Heapfile operations.
    - CSV2Colstore.cpp, Select2.cpp, Select3.cpp - Source files for generating the respective executables for Column Store operations.
- Makefile - make file containing compilation targets. Run “make all” to compile all executables in the assignment.
- mkcsv.py - The CSV file generator provided in the handout.
- page_experiment.sh, heapfile_experiment.sh, column_store_experiment.sh - Scripts which run the data gathering steps for the Page Experiments, Heapfile Experiments, and ColStore Experiments respectively.
- logs.tar.gz - TAR file containing execution logs for all three experiments, which were used to generate the plots shown below.

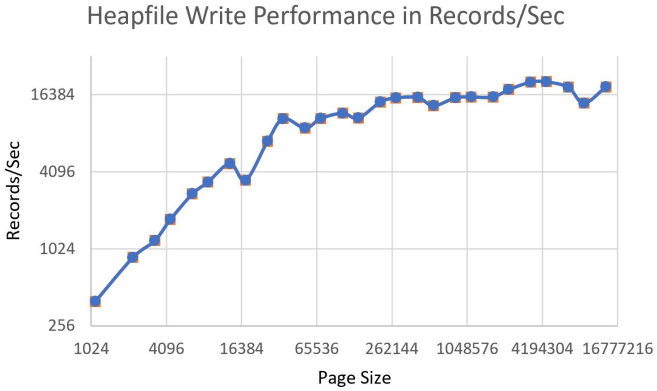
Compilation + Running Examples

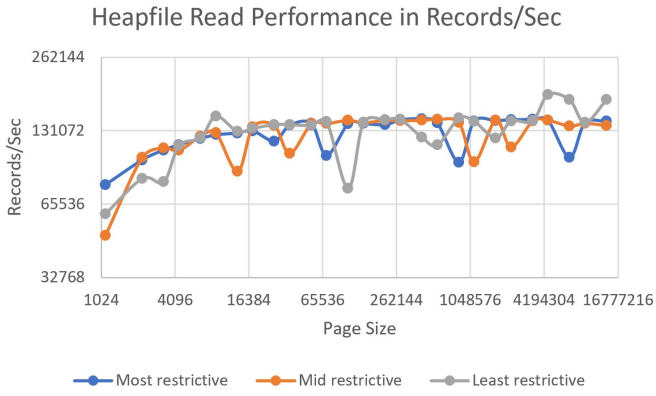
```
make clean

make all

./write_fixed_len_pages <csv_file> <page_file> <page_size>

./read_fixed_len_page <page_file> <page_size>
```

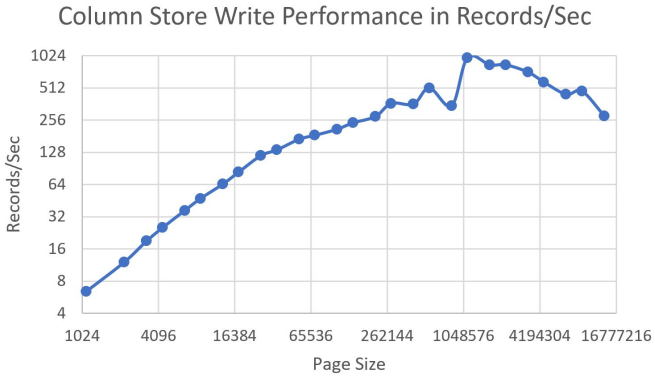
Result Tables

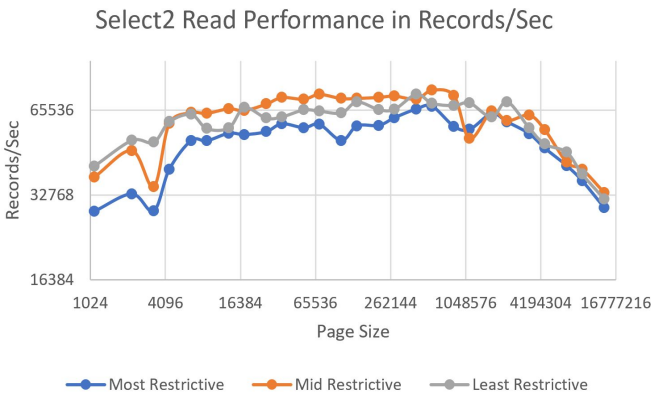


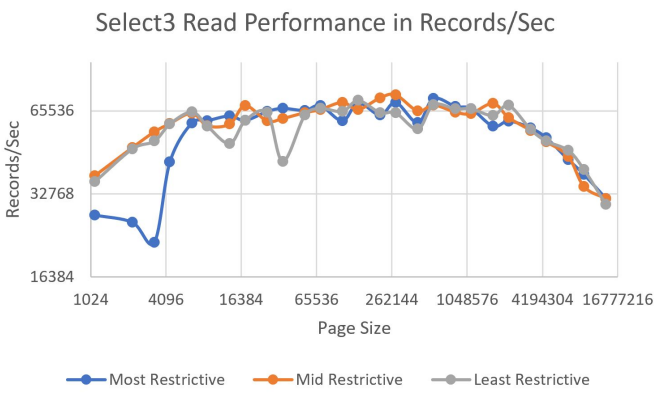









Select2 Read Performance in Records/Sec
Records/Sec
65536
32768
16384
1024
4096
16384
65536
262144
1048576
4194304
16777216
Page Size
Most Restrictive
Mid Restrictive
Least Restrictive


Select3 Read Performance in Records/Sec
Records/Sec
65536
32768
16384
1024
4096
16384
65536
262144
1048576
4194304
16777216
Page Size
Most Restrictive
Mid Restrictive
Least Restrictive