CSC443 Winter 2018

Assignment 1

Due: Sunday Feb 11, 2018 at 11:59 PM

Part I: Disk access characteristics

In this assignment, we investigate the data access characteristics of secondary storage devices. In the end, you will see the need of block based data transfer between main memory and the hard drive. Refer to the hints for some useful information on C language.

1. Sequential write to file (10%)

1.1. Random file generation

Implement a function:

```
1. /**
2.  * populate a random array (which is already
3.  * allocated with enough memory to hold n bytes.
4.  */
5. void random_array(char *array, long bytes);
```

random_array() populates the array with random characters from A to Z inclusively.

Now, use this function to implement the command ***create_random_file*** with the following syntax. Your program should allocate a fixed amount of memory ***char buffer[block_size]***, and repeatedly generate random content into buffer, and then write buffer to disk. A reasonable ***block_size*** is 1MB.

```
$ create_random_file <filename> <total bytes> <block size>
```

Your program will need to report the time it took to write the random bytes to the file using the specified block size.

1.2. Experiments

Write a script to experiment with different block sizes, and record the respective write data rate in bytes/second. You can use any scripting language (e.g. python, bash). Your scripts MUST run

on CDF. Your script has to measure the rate for 10 different block sizes in the range of 100B to 3MB

You have to prepare a very short report for the following three sections:

1. Plot the observation of write data rate versus block size. Provide a simple explanation of the observation.

2. Discuss the existence of the optimal block size for write.

3. Try it on different machines and different storage medium (hard drive, solid state drive, USB storage). Compare the plots (pick at least 2 different medium).

2. Sequential read from file (20%)

In this section, you are to experiment with the performance of sequential scan with different read block sizes.

2.1. Computing histogram using block read

You are to implement a function which scans through the file in blocks, and compute the distribution of different letters, i.e., count the occurrences of each letter: `A` to `Z`. Your function returns the status code.

```
/**
 * file_ptr : the file pointer, ready to be read from.
 * hist: an array to hold 26 long integers.  hist[0] is the
 *       number of 'A', and hist[1] is the number of 'B', etc.
 * block_size: the buffer size to be used.
 * milliseconds: time it took to complete the file scan
 * total_bytes_read: the amount data in bytes read
 *
 * returns: -1 if there is an error.
 */
int get_histogram(
    FILE *file_ptr,
    long hist[],
    int block_size,
    long *milliseconds,
    long *total_bytes_read);
```

Here is a sample of how your function can be used:

```
long hist[26];
long milliseconds;
long filelen;
FILE *file_ptr = fopen("mybigfile.txt", "r");

/**
 * Compute the histogram using 2K buffers
 */
int ret = get_histogram( file_ptr,
                         hist,
                         2 * 1024,
                         &milliseconds,
                         &filelen);
assert(! ret < 0)

printf("Computed the histogram in %d ms.\n", milliseconds);
for(int i=0; i < 26; i++) {
    printf("%c : %d\n", 'A' + i, hist[i]);
}
printf("Data rate: %f Bps\n", (double)filelen/milliseconds * 1000);
```

Create an executable with the arguments:

```
$ get_histogram <filename> <blocksize>
```

Your program is to display the histogram, the block size used, the total number of bytes read from the file, and the time took in milliseconds. This is the sample printout of the program:

```
A 43948
B 48299
C 43424
D 23852
...
Z 48603
BLOCK SIZE 1048576 bytes
TOTAL BYTES 104857600 bytes
TIME 72 milliseconds
```

2.2. Experiment

Write a script to experiment with different block sizes, and record the read data rate in bytes per second. You can use any scripting language (e.g. python, bash). Your scripts MUST run on CDF

machines. Your script has to measure the rate for 10 different block sizes in the range of 100B to 3MB.

You have to prepare a short report for following three sections:

1. Plot the observed read data rate versus block size. Explain the curve.
2. Discuss the existence of an optimal block size for read. Compare this with the case for write.
3. Run it on different machines and different medium. Compare the plots (pick at least 2 mediums)

3. Organize your code

Makefile

```
CC = g++
library.o: library.cc library.h
    $(CC) -o library.o -c library.cc

create_random_file: create_random_file.cc library.o
    $(CC) -o $@ $< library.o

get_histogram: get_histogram.cc library.o
    $(CC) -o $@ $< library.o
```

5. Hints and facts about C

5.1. Generating random characters:

```
#include <stdlib.h>

...
char c = 'A' + (rand() % 26);
...
```

5.2. Measuring system time in the units of milliseconds

```
1. #include <sys/timeb.h>
2. ...
3. struct timeb t;
4. ftime(&t);
5. long now_in_ms = t.time * 1000 + t.millitm;
6. ...
```

5.3. Perform block based write to disk

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4.
5. ...
6.
7. long block_size = ...;
8. char *buf = ...
9.
10. FILE *fp = fopen(filename, "w");
11. fwrite(buf, 1, block_size, fp);
12. fflush(fp);
13.
14. ...
15.
16. fclose(fp);
```

Note:

- Line 11: ***fwrite(buf, itemsize, itemcount, fileptr)*** writes in total ***itemsize * itemcount*** bytes to ***fileptr*** from ***buf***.
- Line 12: ***fflush(fileptr)*** forces a disk write. Modern operating system may have its own buffer manager, so your write may be deferred in the cache.

5.4. Perform block based read from disk

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

long block_size = ...;
char *buf = ...
FILE *fp = fopen(filename, "r");

...

bzero(buf);
fread(buf, 1, block_size, fp)

...

fclose(fp);
```

5. Deliverables

1. A tar ball named ***a1-part1.tar.gz*** that includes all project files i.e. c/c++ files, makefile, scripts, plots, report. Make sure that you can tar/untar your file on CDF successfully. if we cannot untar the file, your assignment will not be marked.

2. A maximum 4 pages report in pdf format.

Part II: Relational data layout on disk

In this assignment, we are to implement a library containing functions to store and maintain relational data on disk using the ***heap file*** data structure. We are asking you to implement and experiment two different file formats corresponding to a ***row store*** and a ***column store***.

1. Record serialization (10%)

For simplicity, we assume that records are maps mapping attribute names to values. The attribute names are stored as part of the schema, which will not be stored as part of the record serialization.

Therefore, we can abstract records as a tuple of values.

```
1. #include <vector>
2. typedef const char* V;
3. typedef std::vector<V> Record;
```

You need to implement serialization of fixed length records.

```
/**
 * Compute the number of bytes required to serialize record
 */
int fixed_len_sizeof(Record *record);

/**
 * Serialize the record to a byte array to be stored in buf.
 */
void fixed_len_write(Record *record, void *buf);
```

You will also need to implement the deserialization functions.

```
1. /**
2.  * Deserializes `size` bytes from the buffer, `buf`, and
3.  * stores the record in `record`.
4.  */
5. void fixed_len_read(void *buf, int size, Record *record);
```

1.1. Experiments

We assume that there is only one table schema. There are 100 attributes, and each attribute is 10 bytes each. So, records in the table are fixed length.

1. Calculate the size of fixed length serialization of records in the table.
2. Use ***fixed_len_sizeof()*** to check if it agrees with your calculation.

2. Page layout (20%)

Recall that it's critical for all disk I/O to be done in units of blocks, known as pages. In this section, we experiment with storing serialized records in pages.

2.1. Storing fixed length records in pages

Use slotted directory based page layout to store fixed length records.

```
typedef struct {
    void *data;
    int page_size;
    int slot_size;
} Page;
```

```
/**
 * Initializes a page using the given slot size
 */
void init_fixed_len_page(Page *page, int page_size, int slot_size);

/**
 * Calculates the maximal number of records that fit in a page
 */
int fixed_len_page_capacity(Page *page);

/**
 * Calculate the free space (number of free slots) in the page
 */
int fixed_len_page_freeslots(Page *page);

/**
 * Add a record to the page
 * Returns:
 *   record slot offset if successful,
 *   -1 if unsuccessful (page full)
 */
int add_fixed_len_page(Page *page, Record *r);
```

```
23.
24. /**
25.  * Write a record into a given slot.
26.  */
27. void write_fixed_len_page(Page *page, int slot, Record *r);
28.
29. /**
30.  * Read a record from the page from a given slot.
31.  */
32. void read_fixed_len_page(Page *page, int slot, Record *r);
```

Use these functions to implement the following executables.

Load as many records from a comma separated file to fill up a page, and append the page to a file. Repeat until all the records in the CSV files are written to the page file. Your program should follow the following syntax, and produce the output containing record count, page count, and time took, similar to as follows:

```
$ write_fixed_len_pages <csv_file> <page_file> <page_size>

NUMBER OF RECORDS: 1000
NUMBER OF PAGES: 32
TIME: 43 milliseconds
```

Write another program to load the ***page_file***, and print out all records in the page in CSV format.

```
$ read_fixed_len_page <page_file> <page_size>
```

2.2. Experiment

1. Plot the performance (records / second) versus page size for write and read.
2. Discuss why page based format is superior to storing records using a CSV file.
3. Discuss the short comings of the way we organize pages.

Note: You can use the provided simple csv generator (***mkcsv.py***) to generate test inputs.

3. Heap file (20%)

Finally, we are able to build the code to generate and maintain heap files.

Again, we assume a fixed table schema. All records have 100 attributes, and the values of each attribute are 10 bytes.

A heap file is just a paginated file. Each page is to store a series of records.

```
1. typedef struct {
2.     FILE *file_ptr;
3.     int page_size;
4. } Heapfile
```

We assume the following way to assign unique identifiers to records in the heap file:

- Every page p has an entry in the heap directory of `(page_offset, freespace)`. The page ID of p can be the index of its entry in the directory. We call this: $\text{ID}(p)$.
- Every record r is stored at some slot in some page p. The record ID, $\text{ID}(r)$ is the contenation of $\text{ID}(p)$ and the slot index in p.

You can use the following structures to abstract page ID and record ID.

```
1. typedef int PageID;
2.
3. typedef struct {
4.     int page_id;
5.     int slot;
6. } RecordID;
```

3.1. Heap file functions

We are to implement a directory based heap file in which we have directory pages (organized as a linked list), and data pages that store records.

```
/**
 * Initalize a heapfile to use the file and page size given.
 */
void init_heapfile(Heapfile *heapfile, int page_size, FILE *file);
```

```
/**
 * Allocate another page in the heapfile.  This grows the file by a page.
 */
PageID alloc_page(Heapfile *heapfile);
```

```
/**
 * Read a page into memory
 */
void read_page(Heapfile *heapfile, PageID pid, Page *page);
```

```
/**
 * Write a page from memory to disk
 */
void write_page(Page *page, Heapfile *heapfile, PageID pid);
```

The central functionality of a heap file is enumeration of records. Implement the record iterator class.

```
class RecordIterator {
    public:
    RecordIterator(Heapfile *heapfile);
    Record next();
    bool hasNext();
}
```

3.2. Heap file operations

Write the following executables to allow basic operations on the heap file. Your program must rely on page based disk I/O.

```
# Build heap file from CSV file
$ csv2heapfile <csv_file> <heapfile> <page_size>

# Print out all records in a heap file
$ scan <heapfile> <page_size>

# Insert all records in the CSV file to a heap file
$ insert <heapfile> <csv_file> <page_size>

# Update one attribute of a single record in the heap file given its record ID
# <attribute_id> is the index of the attribute to be updated (e.g. 0 for the first attribute, 1
 for the second attribute, etc.)
# <new_value> will have the same fixed length (10 bytes)
$ update <heapfile> <record_id> <attribute_id> <new_value> <page_size>

# Delete a single record in the heap file given its record ID
$ delete <heapfile> <record_id> <page_size>
```

3.3. Experiment

Measure the performance of csv2heapfile, comment on how the page size affects the performance of load.

Write an executable:

```
# Select single attirbute with parameters <start> and <end>
# <attribute_id> is the index of the attribute to be selected (e.g. 0 for the first attribute, 1
 for the second attribute, etc.)
$ select <heapfile> <attribute_id> <start> <end> <page_size>
```

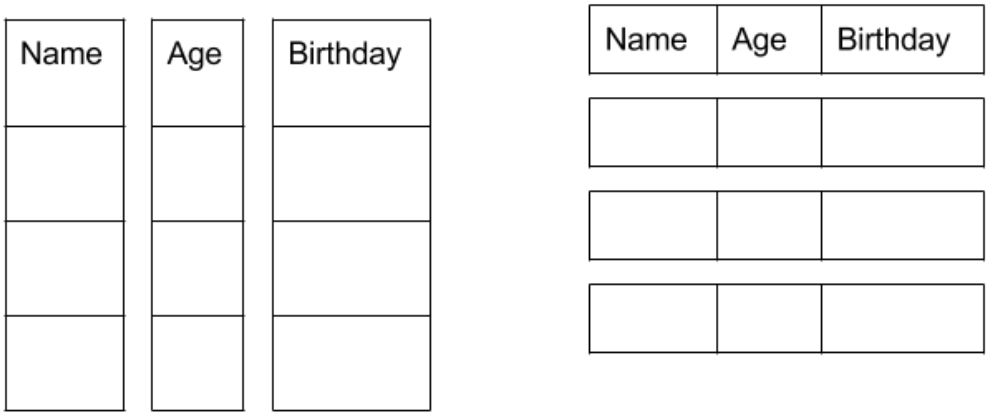
that performs the following parametrized SQL query:

```
SELECT SUBSTRING(A, 1, 5) FROM T
WHERE A >= start AND A <= end
```

1. Measure the performance of the query versus page size.
2. Comment on the choice of ***page size*** and the effects of the range from ***start*** and ***end*** on the performance of the query.

4. Column Store (20%)

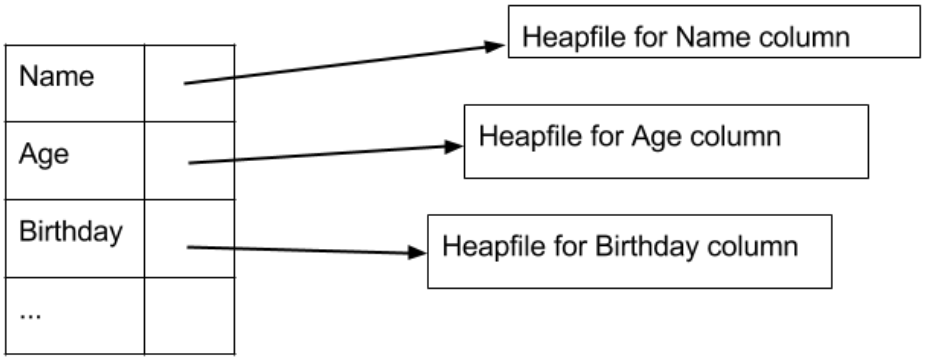
A column-oriented DBMS (or Column Store), stores all values of a single attribute (column) together, rather than storing all values of a single record together. Hence, the main abstract they use is of columns-of-data rather than rows-of-data. Most DBMS are row-oriented, but depending on the workload, column-oriented DBMS may provide better performance. Below is an illustration of column store (left) and row-store (right).



Consider a query that returns all values of the attribute Name. In a column store you only need to retrieve Name values from the file. In constract, in when using row-oriented storage, to find the Name value of a record, you are necessarily retrieving all attribute values in the record. Hence, to return all values of the Name attribute, you need to read the entire table. The drawback of column store is you that you need to do extra work to reconstruct the tuple. Suppose you wish to return the Name and Salary values from a table. To be able to reassemble records, a column-oriented DBMS will store the tuple id (record-id) with each value in a column.

Column-oriented storage has advantages for queries that only access some of the attributes of a table. However, insertion and deletion now may require multiple page accesses as each tuple is no longer stored on a single page

For this assignment, we will implement a simplified version of column store using your existing heap file implementation. Our implementation will have a separate heap file for each column of a table.



We will use the same fixed table schema (100 attributes, 10 bytes each). For each attribute, you need to create a separate heap file. You can name the heap file with the same name as the attribute id. You should put all attribute heap files in a single file directory. This is a simplification for this assignment to make the bookkeeping on what files are in a relation simpler. Think about the limitations of the simplification.

Tuple reconstruction: Different attributes of a tuple will be in different heap file. So, we need to reconstruct the tuple (part of the tuple) to get the result of a query. We can store the tuple-id with each field. Two attributes will have the same tuple-id if they belong to the same tuple.

Tuple ID	Name
1	X
2	Y
3	Z

Tuple ID	Age
1	19
2	20
3	21

Tuple ID	Birthday
1	11/11/95
2	2/2/94
3	1/1/93

4.1. Column-Oriented file operations

Implement an executable to create a column store from a CSV file:

```
# Build a column store from CSV file
# <colstore_name> should be a file directory to store the heap files
$ csv2colstore <csv_file> <colstore_name> <page_size>
```

Implement an executable:

```
# Select a single attribute from the column store where parameter
# <attribute_id> is the index of the attribute to be project and returned (e.g. 0 for the first
 attribute, 1 for the second attribute, etc.)
# the value of the attribute must be between <start> and <end>
$ select2  <colstore_name> <attribute_id> <start> <end> <page_size>
```

that performs the following parametrized SQL query. Note that the selection predicate is on the same attribute that is returned by the query.

```
SELECT SUBSTRING(A, 1, 5) FROM T
WHERE A >= start AND A <= end
```

Implement an executable:

```
# Select only a single attribute from the column store where parameter
# <return_attribute_id> (B) is the index of an attribute to be projected and returned
# <attribute_id> (A) is the index of (a possibly different) attribute whose value must be betwee
n  <start> and <end>
$ select3  <colstore_name> <attribute_id> <return_attribute_id> <start> <end> <page_size>
```

that performs the following parametrized SQL query:

```
SELECT SUBSTRING(B, 1, 5) FROM T
WHERE A >= start AND A <= end
```

4.2. Experiment

Measure the performance of ***csv2colstore*** against different page sizes.

1. Compare the result with that of ***csv2heapfile*** in the previous section. Comment on the difference.

2. Compare the performance of ***select2*** with that of ***select*** in the previous section. Comment on the difference.

3. Compare the performance of ***select3*** with that of ***select*** and ***select2***. Comment on the difference especially with respect to different selection ranges (different values of start and end).

5. Organize your submission

```
# Makefile
CC = g++

library.o: library.cc library.h
	$(CC) -o $@ -c $<

csv2heapfile: csv2heapfile.cc library.o
	$(CC) -o $@ $< library.o

scan: scan.cc library.o
	$(CC) -o $@ $< library.o

insert: insert.cc library.o
	$(CC) -o $@ $< library.o

update: update.cc library.o
	$(CC) -o $@ $< library.o

delete: delete.cc library.o
	$(CC) -o $@ $< library.o

select: select.cc library.o
	$(CC) -o $@ $< library.o

csv2colstore: csv2colstore.cc library.o
	$(CC) -o $@ $< library.o

select2: select2.cc library.o
	$(CC) -o $@ $< library.o

select3: select3.cc library.o
	$(CC) -o $@ $< library.o
```

6. Deliverables

1. A tar ball named ***a1-part2.tar.gz*** that includes all project files i.e. c/c++ files, makefile, scripts, plots, report. Make sure that you can tar/untar your file on CDF successfully. if we cannot untar the file, your assignment will not be marked.
2. A maximum 4 pages report in pdf format.