Assignment 4

Indexing Large Datasets with Hash Tables: A Case Study in Memory Efficiency - 12 points

Pedram Pasandide

Due Date: 2025, 8 April

Introduction

When working with large datasets, it's often necessary to quickly retrieve records based on specific attributes — for example, searching for all properties sold on a specific street. One way to improve search performance is by using **indexing**.

Rather than scanning through all records linearly, an index provides a faster lookup mechanism, typically based on a data structure like a **hash table** or a **B+ tree**. Importantly, an index does not duplicate the full dataset. Instead, it stores only the *key* (such as a street name) and a **pointer to the original data** in memory. This approach reduces memory usage and avoids unnecessary duplication.

Creating a separate copy of the dataset for every searchable column would be highly inefficient — both in terms of memory and consistency. Indexes allow us to efficiently support search operations while referencing the existing dataset in memory.

In this assignment, you will explore the concept of indexing by building a hash-based index on a large dataset, stored in memory. The dataset is taken from the publicly available Land Registry Price Paid Data (UK Residential Property Sales), and includes millions of records. For simplicity and focus, we will build an index on only one column: the street name.

- Through this project, you will practice:
- Loading large datasets into memory
- Creating and using a hash table index
- Understanding cache behavior and memory layout
- Comparing indexed search vs. full linear scan

Programming Logic (9.5 points)

This programming assignment is divided into four parts:

1. Reading CSV files and storing data in memory
2. Creating a hash index on a specific column
3. Performing a search case study (comparing index vs. linear scan)
4. Analyzing index efficiency using unused slots and load factor

You are provided with three files:

- `main.c` (Do NOT modify this file)
- `myDSlib.c`
- `myDSlib.h`

You will implement all your logic in `myDSlib.c`, using the structures and constants defined in `myDSlib.h`.

Part 1: Reading CSV Files into Memory

In this part, you will implement the `read_file()` function, which reads records from a CSV file and stores them in memory using the provided `Record` structure.

The dataset files (e.g., `pp-2024.csv`, `pp-2023.csv`) follow the format of the UK Land Registry Price Paid Data. Each line in the file represents a sale transaction and contains multiple fields, such as transaction ID, price, date, postcode, street name, town, and more. You can find more information about the data on here. You can check your results either by searching within CSV files or check here.

You must not change the following constants or structures:

```
#define MAX_FIELD_LEN 100
#define MAX_TRANSACTION_ID_LEN 39
#define MAX_POSTCODE_LEN 9

typedef struct {
```

```c
 int year, month, day;
} Date;

typedef struct {
 char transaction_id[MAX_TRANSACTION_ID_LEN];
 unsigned int price;
 Date date;
 char postcode[MAX_POSTCODE_LEN];
 char property_type;
 char old_new;
 char duration;
 char paon[MAX_FIELD_LEN];
 char saon[MAX_FIELD_LEN];
 char street[MAX_FIELD_LEN];
 char locality[MAX_FIELD_LEN];
 char town[MAX_FIELD_LEN];
 char *district;
 char county[MAX_FIELD_LEN];
 char record_status;
 char blank_col;
} Record;

// Global table pointer
extern Record* table;
extern size_t table_size;
```

Your job is to parse each line of the input CSV and save the result into an array of `Record` structs, called `table` with dynamically adjusted size `table_size`. Memory for this table must be allocated dynamically using `malloc()` or `realloc()` as needed. Each time `read_file()` is called, new data must be appended to the existing global `table`.

The `main.c` file will call your function like this:

```c
read_file("pp-2024.csv");
printf("Total records appended: %zu\n", table_size);

read_file("pp-2023.csv");
printf("Total records appended: %zu\n", table_size);
```

This means:

- Your `read_file()` function must preserve earlier data
- New records must be added at the end of the table array
- You must update `table_size` correctly
- You must dynamically allocate memory for the district field (since it is a `char*`)

This is what you should have so far:

```
Total records appneded: 666013
Total records appneded: 1500230
```

Using GDB to set a breakpoint after reading both CSV files, I can print the table to make sure reading the data is functional:

```
(gdb) p table[1000000]
$1 = {
 transaction_id = "{2ACACE8D-0C4A-295E-E063-4804A8C0B0EB}",
 price = 104950,
 date = {year = 2023, month = 10, day = 9},
 postcode = "L6 1AX\000\000",
 roperty_type = 70 ’F’,
 old_new = 89 ’Y’,
 duration = 76 ’L’,
 paon = "88", ’\000’ <repeats 97 times>,
 saon = "APARTMENT 413", ’\000’ <repeats 86 times>,
 street = "LOW HILL", ’\000’ <repeats 91 times>,
 locality = ’\000’ <repeats 99 times>,
 town = "LIVERPOOL", ’\000’ <repeats 90 times>,
 district = 0x555557447a10 "LIVERPOOL",
 county = "MERSEYSIDE", ’\000’ <repeats 89 times>,
 record_status = 65 ’A’,
 blank_col = 65 ’A
 }
```

Part 2: Creating a Hash Index on a Column

In this part, you will implement the function:

```
IndexEntry** createIndexOnStreet(Record* table, size_t table_size);
```

This function builds a hash index table with size `INDEX_SIZE` on the `street` field of each `Record` in the dataset. The goal is to create a fast-access structure that allows you to find records by street name **without scanning the entire dataset** or saving data all over again based on `street` column.

The `INDEX_SIZE` constant determines the number of buckets (slots) in your hash table. A small value can lead to many collisions and long linked lists, slowing down search. A very large value can waste memory, especially if many buckets remain unused. To balance performance and memory usage, you must choose an appropriate `INDEX_SIZE` defined in `myDSlib.h`:

```
#define INDEX_SIZE 10
```

An index is a data structure — in this case, a hash table — that maps a key (the `street` name) to a location in memory (a pointer to a `Record` in the dataset). This avoids scanning every record when performing lookups.

Instead of duplicating data, the index simply stores:

- A copy of the key (string)
- A pointer to the original record
- A link to the next matching entry in case of a hash collision

This is defined for you in the `IndexEntry` struct:

```
typedef struct IndexEntry {
 char* key; // The street name (copied from table[i].street)
 Record* record_ptr; // Pointer back to original record
 struct IndexEntry* next; // For handling collisions
} IndexEntry;
```

The hash table is implemented as an array of pointers (`IndexEntry*`) of size INDEX_SIZE. What `createIndexOnStreet()` does? For each record in the `table`:

1. Get the `street` name.
2. Compute its hash value: `hash = hash_string(street) % INDEX_SIZE`. The hash function called `hash_string()` is already `myDSlib.c`.

3. Allocate a new IndexEntry:
    - Copy the street name into `key`.
    - Set `record_ptr` to point to that `Record`.
4. Insert the entry at the **beginning** of the list at `index_on_street[hash]`.

Now, searching for a `street` string only requires computing its hash and scanning a short linked list, instead of the entire table.

NOTE: Before moving to the next part make sure `index_on_street` hash table is properly pointing the data in `table`.

Part 3: Search Case Study

In this part of the assignment, you will implement and compare two different search strategies:

1. A linear search that scans the full dataset stored in `table`
2. A hash-based search that uses the index you built in Part 2 (`index_on_street`).

You will implement the following two functions:

```
void searchStreetLinear(Record* table, size_t table_size, const
   char* target_street, bool printFlagLinearSearch);
void searchStreet(IndexEntry** index, const char* target_street,
   bool printFlagHashIndexSearch);
```

`searchStreetLinear(...)` performs a brute-force linear search through the entire `table`. For each record, it compares the `street` field to the `target_street`. If they match, it prints the record (depending on the `printFlag`). Time complexity is $O(n)$ where n is the number of records in the dataset.

`searchStreet(...)` performs a fast lookup using the hash table index, hashing the `target_street` to find the correct bucket. Then it walks a short linked list of matches (if any). Time complexity is ideally equal to $O(1)$ with good hash function and table size. It also depends on data.

Both functions include a `bool printFlag` parameter. If `printFlag == true`, your functions must print for every matching record:

- The street name

- The price
- The postcode
- The date (in YYYY-MM-DD format)

With print flag `true` only testing `searchStreet()`:

```
 Total records appneded: 666013
 Total records appneded: 1500230

 Hash index on street created.
 Match found: Street = GEORGE STREET | Price = 152500 | Postcode = CF40 1QP | Date = 2023-03-28
 Match found: Street = GEORGE STREET | Price = 145500 | Postcode = NP11 7QG | Date = 2023-01-27
 Match found: Street = GEORGE STREET | Price = 140000 | Postcode = CV7 8HJ | Date = 2023-02-01
<note: the matching data is truncated>
```

With print flag `true` only testing `searchStreetLinear()`:

```
 Total records appneded: 666013
 Total records appneded: 1500230

 Hash index on street created.
 Match found (linear): Street = GEORGE STREET | Price = 320000 | Postcode = MK40 3SG | Date = 2024-07-26
 Match found (linear): Street = GEORGE STREET | Price = 340000 | Postcode = B21 0EG | Date = 2024-03-08
 Match found (linear): Street = GEORGE STREET | Price = 320000 | Postcode = NG5 6LP | Date = 2024-05-17
<note: the matching data is truncated>
```

With both flags `false` and testing both functions:

```
Total records appneded: 666013
Total records appneded: 1500230

Hash index on street created.

Time (Linear Search): 0.046075 seconds
Time (Hash Index)   : 0.000176 seconds
```

Part 4: Load Factor Analysis

In this final part of the assignment, you will evaluate the efficiency of your hash table index by analyzing its load factor and the distribution of used slots.

The load factor of a hash table is a measure of how full the table is — that is, how many slots are being used. It is defined as:

```
load_factor = (number of used slots) / (total number of slots)
```

- A high load factor (close to 1) means most of the table is used, but could lead to more collisions (longer chains).
- A low load factor means many buckets are empty, which reduces collision chances but wastes memory.

A well-balanced hash table has a load factor somewhere between 0.5 and 0.9, depending on your needs. You are provided with the following code to run in `main.c`:

```c
int unused = count_unused_slots(index_on_street);
printf("\nUnused hash slots: %d out of %d (%.2f%% unused)\n",
   unused, INDEX_SIZE, 100.0 * unused / INDEX_SIZE);
int used = INDEX_SIZE - unused;
double load_factor = (double)used / INDEX_SIZE;
printf("Hash table load factor: %.3f\n", load_factor);
```

This code will:

- Count how many slots in the hash table are still NULL (unused)
- Print the number of unused and used slots
- Calculate and display the load factor as a floating-point number

You have to implement `count_unused_slots()`. With a `INDEX_SIZE=100000`, my program results:

```
Total records appneded: 666013
Total records appneded: 1500230

Hash index on street created.

Time (Linear Search): 0.046075 seconds
Time (Hash Index)   : 0.000176 seconds

Unused hash slots: 11504 out of 100000 (11.50% unused)
Hash table load factor: 0.885
```

Discussion Questions (1 point)

1. In terms of memory usage, we know there is no waste with `*district` since the memory is allocated exactly based on the size of the string. However, in `street`, there are many cases where the number of `char`s used is less than `MAX_FIELD_LEN`, which causes a waste of memory. In terms of memory access speed, which one is faster and why?

2. Although `searchStreet()` has an expected time complexity of $O(1)$, the observed CPU timing results show that it is not significantly faster than `searchStreetLinear()`, which has a time complexity of $O(n)$. Considering factors such as caching, memory layout, and real-world performance characteristics, explain why the speed difference is not as dramatic as the theoretical complexity suggests.

Report and Makefile (1.5 points)

Create a PDF file named `Report.pdf` that includes the documentation for your program. The report should cover essential information, such as how to compile and run the program. Create a section called Appendix in your report and include the codes **only** from `myDSlib.c` and `myDSlib.h` as text (not screenshots). A report file without following the above structure will **not** be accepted.

Additionally, write a `Makefile` to compile your program. The `Makefile` should define the rules for compiling. Include a short description about your `Makefile` in your report, describing the `Makefile` including every flag being used and all the rules.

Submission On Avenue to Learn

Submit your assignment as:

1. `myDSlib.c`,

2. `myDSlib.h`,

3. `Report.pdf`,

4. `Makefile`.

Please do **NOT** submit any zip files. Do not submit CSV files.