Chair of Applied Computer Science III
UNIVERSITY OF MANNHEIM
Prof. Dr. Guido Moerkotte
Email: moerkotte@uni-mannheim.de

Daniel Flachs
Email: daniel.flachs@uni-mannheim.de

Database Systems II — Exercise Sheet 4
Spring Semester 2019 — Created March 21, 2019

Exercise 1

Download the `zip` archive for this exercise sheet from the website.
Compile `experiment4.cc` with the output name set to `experiment4`, e.g., by compiling as `g++ experiment4.cc -o experiment4`.
Run the script `runExperiment4.sh` and analyze the output. `runExperiment4.sh` is a bash script.
If you run Windows, you are able to run this file in case your Windows version is 10 (or newer), cf. `https://www.howtogeek.com/261591/how-to-create-and-run-bash-shell-scripts-on-windows-10/`
The script `runExperiment4.sh` runs `experiment4` for different arguments. If you're having trouble to run the script, you can also perform the instructions of the script by hand.

Exercise 2

This exercise deals with compression.

Exercise 2 a)

In the `zip` file for this exercise sheet, you find a file named `country_or_area.csv`. This file contains the attribute values for the column *country_or_area* of the *International Financial Statistics* data set `https://www.kaggle.com/unitednations/international-financial-statistics`.
Implement a dictionary that allows you to compress the *country_or_area* attribute values given in `country_or_area.csv`.
You are free to choose on the implementation details. Your dictionary may be based on a hash table, a tree, or something simple (= inefficient), like an unsorted vector.

Exercise 2 b)

What is the compression rate and the percentage of space savings of your compression? The compression ratio is computed as

$$compRatio = \frac{uncompressedSize}{compressedSize},$$

where *compressedSize* is the size of the data plus the size of the dictionary.

The space savings is computed as

$$spaceSavings = 1 - \frac{compressedSize}{uncompressedSize}.$$

Exercise 3

This exercise deals with the difference in memory access costs for cache-aligned and unaligned accesses. It simulates tuples of a database relation that are stored in row store format, cf. Script, p. 53 ff. (especially the figure on p. 56).

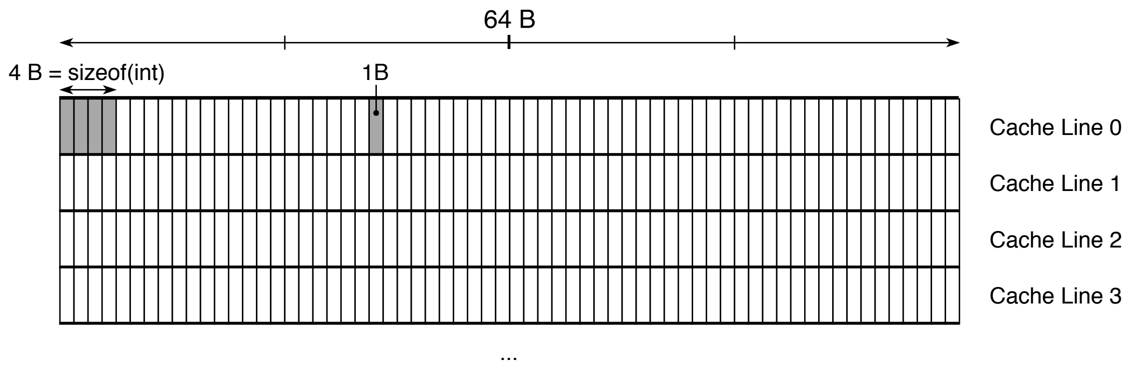
Consider the following piece of code. The `sum` function sums up `m` elements of integer array `B`, starting at `B[0]` in steps of `s` integers, i. e., `s = 3` would sum up `B[0]`, `B[3]`, `B[6]` etc. The main method uses the `sum` function to sum up the same array `A` with different access patters (step size and offset), denoted by `(0)`, `(1)` and `(2)` in the code.

```
int32_t sum (int32_t* B, const uint m, const int s) {
  int32_t lSum = 0;
  for(uint32_t i = 0; i < m; ++i) { // m: number of elements
    lSum += *B;
    B += s;       // s: step size
  }
  return lSum;
}


int main() {
  const uint32_t n = 1000*1000*100;
    // number of ints in array
  const uint32_t m = (n - 16) / 16;
    // 16*4 = 64 = sizeof(cacheline)

  void* A = 0;
  int lRc = posix_memalign(&A, 64, (n * sizeof(int32_t)));
  if(lRc) {
    printf("memalign failed.");
    return 1;
  }

  int32_t* B = nullptr;

  // *** (0) ***
  B = (int32_t*) A;
  for(uint32_t i = 0; i < n; ++i) { B[i] = 1; }
  const int32_t lSum0 = sum(B, m, 1);

  // *** (1) ***
  B = (int32_t*) A;
  for(uint32_t i = 0; i < n; ++i) { B[i] = 1; }
```

```
    const int32_t lSum1 = sum(B, m, 16);

    // *** (2) ***
    B = (int32_t*) (A + 62);
    for(uint32_t i = 0; i < n - 15; ++i) { B[i] = 1; }
    const int32_t lSum2 = sum(B, m, 16);

    printf("sum0/1/2: %d, %d, %d\n", lSum0, lSum1, lSum2);

    assert((void*) B < A + n);
    free(A);
}
```

Exercise 3 a)

In the following figure, for each access pattern `(0)`, `(1)` and `(2)`, indicate the location of the first array element that is summed up when calling the `sum` function. Suppose that `int* A` points to the leftmost integer in cache line 0.



Exercise 3 b)

Use your findings of the previous subtask to determine which of the access patterns is cache-aligned and which is not.

Exercise 4

Note: This exercise was discussed in class, but was originally not on the sheet. It was taken from Cormen et al., 3e, p. 277, Ex. 11.4-1.

Consider inserting the keys 10, 22, 31, 4, 15, 28, 17, 88, 59 into a hash table of length $m = 11$ using open addressing with the auxiliary hash function $h'(k) := k$. For inserting, use

a) linear probing,

b) quadratic probing with $c_1 := 1$, $c_2 := 3$,

c) double hashing with the two auxiliary hash functions $h_1(k) := k$ and $h_2(k) := (k \bmod (m-1)) + 1$.