TU München, Fakultät für Informatik
Lehrstuhl III: Datenbanksysteme
Prof. Dr. Thomas Neumann

Exercises for *Foundations in Data Engineering*, WiSe 23/24

Alexander Beischl, Maximilian Reif (i3fde@in.tum.de)
http://db.in.tum.de/teaching/ws2324/foundationsde

Sheet Nr. 04

Exercise 1

1. For the toy example from the lecture, illustrate how data moves through the machine. Consider two cases:
   a) The input file is read from disk.
   b) The input file is present in the operating system's file cache.

   Solution:

   a) Read with mmap
      i. Data is read from disk: The operating system allocates physical RAM frames for the corresponding virtual memory pages and transfers data from disk into the RAM frames. At the same time, virtual memory pages in the disk cache are mapped to the same RAM frames, so the data can be reused later without the need for additional transfers.
      ii. Data is already in the disk cache: The operating system checks the disk cache, finds the physical RAM frame corresponding to an offset in the file, then maps that RAM frame to a virtual memory page inside the process' address space.

   b) Read with system calls
      i. Data is read from disk: The kernel allocates a kernel buffer, where the data is copied from the disk controller, then it copies the data from the kernel buffer to the userspace buffer provided as argument to the system call.
      ii. Data is in the disk cache: The kernel check the disk cache, identifies the RAM frames containing the required data, then copies the data to the userspace buffer provided as argument to the system call.
2. With the illustration of the previous exercise, explain the maximum performance for the toy example that was estimated in the lecture.

   Solution:
   In the optimal case (e.g. with mmap) there is no unnecessary data copying. Thus we can just assume the program reads from DRAM at the DRAM bandwidth limit. We can use 20GB/s as an estimate for that.
3. Three implementations of the toy example were shown in the lecture: `awk`, `python` and a first attempt in `C++`. Why did they not reach the maximum estimated performance? How can the difference in runtimes between the solutions be explained?

Solution:
Computation on computer uses many different resources. The important resources for the above tasks are:

- Storage bandwidth
- Main memory bandwidth
- Within the CPU
  - Arithmetic and logic unit (ALU)
  - Registers
  - Instruction fetch and decode unit
  - Branching

The presented solutions utilize only one CPU core. In modern processors, with many cores, one processor can not saturate the memory bandwidth. Saturated memory bandwidth however was used to estimate the maximum runtime. Therefore, the estimated runtime can not be achieved.
The bottleneck in the examples is somewhere within the processor. The above tools are different in how efficient the are in utilizing the available resources. `Python` and `awk` use interpretation. This technique requires many more processor resources for executing programs than the other option: Compilation. We used a compiler for `C++` to create a machine executable binary. The compilers goal is to transform the program into a series of machine instructions that performs as efficient as possible. This more optimal usage of CPU resources accounts for the difference in runtime.

4. How does the toy example's first implementation in `C++` (as shown in the lecture) process text?
   Solution:
   The solution reads the file byte by byte. When it encounters a column separator, it increases the column count. When it reaches the fifth column, it parses the column into a string, adds the parsed value to the overall sum and proceeds to the next line.
5. In the lecture, we have seen a variant of the toy example that used a binary file as input instead of csv format. Why is this format more efficient for computing the sum of all quantities?
   Solution:
   - Only relevant information in file, file smaller
   - No searching for fifth column and newline
   - No parsing of numbers

Exercise 2 We are about to optimize the toy example for better performance. Before we start, let's take a step back and have a look at some experiences with optimization.

"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should

forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%." – Donald Knuth in *Structured Programming with go to Statements*.

"The First Rule of Program Optimization: Don't do it. The Second Rule of Program Optimization (for experts only!): Don't do it yet." – attributed to Michael A. Jackson

With this in mind, answer the following questions:

1. What is the goal of performance optimization?
   Solution:
   - run faster
   - run using less energy
   - use less memory
2. What are the downsides of manual optimization?
   Solution:
   - Unreadable code
   - Optimization may be to system specific, not portable
   - Danger of optimizing the wrong piece of code
3. What is a good strategy for optimization?

   Solution: Measure, change, measure!
4. Which tools are there to support optimization?
   Solution:
   - Profilers: Perf, VTune
   - Tracers: DTrace, STrace, PTrace

Exercise 3 In the lecture, a programming trick was shown that allows to search for a newline character in an array of characters. The purpose of the following series of questions is to demonstrate how this technique works.

1. What is the general idea of the programming trick? Why do we employ the trick and how does it help?
   Solution:
   - Load 8 bytes into one register
   - All operations work on 8 bytes at a time, therefore we may use less instructions
2. What is a bitwise operation and which operations are available on modern Intel CPUs?
   Solution:
   Bitwise operations work on sequences of bits. The the result for one bit does not influence that of the neighboring bit. Examples: NOT, AND, OR, XOR, left shift, right shift

3. Addition is not a bitwise operation. How is it possible to perform 8 byte-wise additions with one (non-SIMD) add instruction?

   Solution:
   This is possible if all operand bytes have values < 128. When those values are moved consecutively into an 8 byte register and added to a 8 byte register with the same properties, no value will be larger then 255 and thus not occupy more than 8 bits. This means, that the result of the addition will not leave the 1 byte boundary and not spill over into the next byte. Thus the result can be correctly interpreted as 8 individual bits.

4. Given 8 consecutive bytes in the variable `block` of type `uint64_t`, which bitwise operation on block sets all bits in a byte to 0 if the byte equals `'\n'` (or 1010 in binary representation or $0x0A$ in hexadecimal)?

   ```
   all0iffNl = <your answer here>;
   ```

   Solution:

   ```
   uint64_t pattern = 0x0A0A0A0A0A0A0A0A; \\ hex value of '\n' is 0A
   uint64_t all0iffNl = block ^ pattern;
   ```

5. Given the variable `searchResult` of type `uint64_t`, assume that the highest bit in each byte is 0 (or in other words, each byte is ≤ 127). Which operation on `searchResult` will set the highest bit to one in each byte iff the byte is not 0?

   ```
   highestBitSetIffByteNot0 = <your answer here>;
   ```

   Solution:

   ```
   uint64_t low = 0x7F7F7F7F7F7F7F7F;
   uint64_t highestBitSetIffByteNot0 = searchResult + low;
   ```

6. Given the previous result, which operations need to be performed so that the highest bit is 1 and all others are 0 if the highest bit is not set and all 0 if the highest bit is set?

   ```
   highestSetIf0 = <your answer here>;
   ```

   Solution:

   ```
   uint64_t high = 0x8080808080808080;
   uint64_t highestSetIf0 = ~highestBitSetIffByteNot0 & high;
   ```

7. Assuming all byte values in `block` are ≤ 127, i.e. their highest bit is 0, combine the results of the three previous questions to a chain of operations that takes `block` as input and produces a variable of type `uint64_t` in which every byte is 0 if the corresponding byte in `block` is something else than `'\n'` and has only the highest bit set if the corresponding byte equals `'\n'`.

   Solution:

   ```
   pattern = 0x0A0A0A0A0A0A0A0A; \\ hex value of '\n' is 0A
   uint64_t low = 0x7F7F7F7F7F7F7F7F;
   uint64_t high = 0x8080808080808080;
   all0iffNl = block ^ pattern;
   highestBitSetIffByteNot0 = all0iffNl + low;
   highestSetIf0 = ~highestBitSetIffByteNot0 & high;
   ```

8. Previously, we assumed that all byte values are smaller than 128. Adapt your solution so that it can also handle values greater 128.

   Solution:

```
pattern = 0x0A0A0A0A0A0A0A0A;
uint64_t low = 0x7F7F7F7F7F7F7F7F;
uint64_t high = 0x8080808080808080;
// add lowChar to know which one of the chars was <= 127
uint64_t lowChar = ~block & high
// added & low to install invariant: byte values <= 127
all0iffNl = block ^ pattern & low;
highestBitSetIffByteNot0 = all0iffNl + low;
highestSetIf0 = ~highestBitSetIffByteNot0 & high;
result = highestSetIf0 & lowChar;
```

Exercise 4 In this exercise, we want to optimize the `C++` based approach on the TPC-H dataset corresponding to the following query as seen on the previous exercise sheet:

```
select sum(l_extendedprice) from lineitem
```

First, you should perform simple performance measurements with `time` and `perf` in order to identify the main performance issue. To achive this investigate for each of the following components of your program how much time it requires:

- reading a row of the input file
- locating delimiters
- conversion of a price string to an integer

In the lecture you have learned about the `mmap` system call which may lead to a significant improvement in this situation. Clone the project from this repository and fill in the missing code fragments in all sections marked with `TODO`. Your implementation should exploit the aforementioned system call.

Analyze your current solution again with `perf` (documentation: `man perf`, `man perf stat` and `man perf record`). What conclusion can you draw now? Which part of your implementation still offers room for improvement? Apply the blockwise programming trick as shown in the lecture in order to improve the execution performance further.

Hint: You may want to add some padding bytes right after the mapped file's region in order to simplify your implementation. It is sufficient to increase the `len` parameter accordingly (bytes after the mapped file's region will be set to zero).

Hint 2: Using the `__builtin_ctzll` built-in function is an efficient way to count the number trailing zero bits in a 64-bit integer. This built-in function might be useful in the implementation of your `find_first` function.

Solution:

main.cpp:

```
//-- TODO exercise 3.5
int handle = open(argv[1], O_RDONLY);
auto size = lseek(handle, 0, SEEK_END);
// ensure that the mapping size is a multiple of 8 (bytes beyond
// the region of the file are set to zero)
auto mapping_size = size + 8;  // padding for the last partition
auto data = mmap(nullptr, mapping_size, PROT_READ, MAP_SHARED, handle, 0);
//--

//-- TODO exercise 3.5
// cleanup
munmap(data, mapping_size);
close(handle);
//--
```

sum.cpp:

```
// pattern : the character to search broadcasted into a 64-bit integer
// block : memory block in which to search
// return : 64-bit integer with all the matches as seen in the lecture
inline uint64_t get_matches(uint64_t pattern, uint64_t block) {
    //-- TODO exercise 3.5 part 2
    constexpr uint64_t high = 0x8080808080808080ull;
    constexpr uint64_t low = 0x7F7F7F7F7F7F7F7Full;
    uint64_t lowChars = (~block) & high;
    uint64_t foundChars = ~((((block & low) ^ pattern) + low) & high);
    uint64_t matches = foundChars & lowChars;
    return matches;
    //--
}

// pattern : the character to search broadcasted into a 64bit integer
// begin : points somewhere into the partition
// len : remaining length of the partition
// return : the position of the first matching character or -1 otherwise
ssize_t find_first(uint64_t pattern, const char* begin, size_t len) {
    // locate the position of the following character (you will have to
    // use the pattern argument directly in the second part of the exercise)
    const char to_search = pattern & 0xff;

    // Hint: You may assume that reads from "begin" within [len, len + 8)
    // yield zero
    //-- TODO exercise 3.5
    size_t i = 0;
    while (i < len) {
        uint64_t block = *reinterpret_cast<const uint64_t*>(begin + i);
        uint64_t matches = get_matches(pattern, block);
        if (matches != 0) {
            uint64_t pos = __builtin_ctzll(matches) / 8;
            if (pos < 8) {
                return (i + pos);
            }
        }
        i += 8;
    }
    //--
    return -1;
}
```

Exercise 5 With memory mapped IO, how is data transferred from the OS file cache to the reading process?

Solution:

Every process has its own virtual memory space which no other process can alter. The virtual memory space of each process goes from address 0 to the highest possible address. The real memory space however is one continuously addressable space. In order to bridge this gap, the memory management unit (MMU) translates addresses from user space addresses to hardware addresses. The MMU divides the hardware space into pieces, so called pages. For every page, there is a mapping from process address to hardware address. This mapping is determined by the operating system.
Now, if the OS wants to move data in the virtual address space, it may either copy data from location a to location b or change the mapping of virtual addresses. The latter allows to move data much faster than actually copying data. It is called a zero copy approach.