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

Exercises for *Foundations in Data Engineering*, WiSe 22/23

Alexander Beischl, Maximilian Reif (i3fde@in.tum.de)
http://db.in.tum.de/teaching/ws2223/foundationsde

Sheet Nr. 04

Exercise 1 This exercise takes a closer look at which machine instructions are generated in different variants for the newline finding algorithm. To answer the following questions, you may find it helpful to use the Godbolt Compiler Explorer at https://godbolt.org/. Create a function in `C++` in the source window. Then, discover which instructions the compiler generates from your source code. (Note: Don't forget to choose appropriate optimization settings for the compiler at the top of the right-hand side window. E.g. use `-O3 -std=c++20`.)

1. Which instructions are necessary when searching for a newline character in a byte-array with a regular byte-wise for loop?

   Solution: add, cmp, jump

2. How many instructions are issued per non-newline character in the search loop?
   Solution:
   5 Instructions https://gcc.godbolt.org/z/bbfE9b
   Tips: Identify the loop which is executed when a byte array is scanned and no newline is found. Hover over the instructions to discover their documentation. Right-click on them for more options.
   Explanation: The total generated code contains 12 instructions. However, only 5 instructions (basic block `.L9` & `.L3`) are used for the search loop over non-newline characters.
   The register `rsi` contains the value of variable `limit`, `rdi` contains `iter`. In line 2-5 we write `limit` into `rsi` and then compare `limit` to `iter` to decide if we even need to check the second condition, which is the newline comparison.
   If `iter` is smaller than limit we jump to the basic block `.L3` to check the content of `iter` against newline. On success we now enter the search loop the first time by jumping to `.L9`. In this basic block `.L9` `iter` is incremented, then compared to `limit` (residing in register `rax`). If the incremented value of `iter` is smaller than `limit` we check if the new char is a non-newline character (`.L3`) and then continue the loop with `.L9`. In case of a newline character we continue to `.L5` to write `iter` to `rax` and return it.
   Therefore, the search loop itself is executed by the basic blocks `.L9` & `.L3`, in total 5 instructions.
3. How many instructions per byte are used in the bitwise-operation based solution presented in the lecture?

Solution:
12 Instructions per loop iteration
12 / 8 [Instr./Byte] = 1.5 [Instr./Byte]
https://gcc.godbolt.org/z/qGcTf9
Explanation: In the first line of the assembly code we check if we need to enter the loop by comparing `rdi` (`iter`) and `rsi` (`limit`). Then, we either return `limit` (`.L6`) or initialize our variables `lowBits` in `rcx`, `0x0A0A...` in `r8` and `highBits` in `rdx`.
Next, we continue in `.L4`, which contains the loop operations in line 8-16, jump to `.L8` to execute the `if` statement and continue the loop. If the `if` fails we execute the `C++` code of line 13-14 in the assembly lines 17-23.
Therefore, the loop itself contains 12 instructions (line 8-16 and line 25-27).

4. Why does the bitwise operation based solution execute faster than the byte-wise solution?
   Solution: As discussed earlier, the execution bottleneck was in the CPU. The CPU is only able to execute a certain number of instructions per second. Reducing the number of instructions that need to be performed for the task therefore helps to reduce the overall execution time.

Exercise 2 Given a specific workload, which options are there to reduce the execution time?
Solution:

- More efficient algorithms
- More efficient implementations
- Use faster hardware (scale up)
- Use machines with more power (scale up)
- Use more machines (scale out)

Exercise 3

Please write SQL queries to answer the following questions on the TPC-H dataset, you can use the WebInterface to test your queries:

1. How many customers have an order whose comment contains the word packages?
   Solution:
   ```
   SELECT count(distinct o_custkey)
   FROM orders
   WHERE o_comment like '%packages%';
   ```
2. What is the average number of digits in l_orderkey?
   Solution:
   ```
   SELECT avg(char_length(l_orderkey::text))
   FROM lineitem;
   ```
3. What are the names of all customers and suppliers?
   Solution:
   ```
   SELECT c_name FROM customer
   UNION ALL
   SELECT s_name FROM supplier;
   ```
4. Retrieve 10 customers and the corresponding nation. As output, create a JSON object for each customer with the custkey, name and the nation as embedded json object. The nation object shall contain the nationkey and name.
   Solution:
   ```
   SELECT '{"custkey":' || c_custkey || ', "name":"' ||
      c_name || '", "nation":{"nationkey":' || n_nationkey ||
       ', "name":"' || n_name || '"}}' as custjson
   FROM customer, nation
   WHERE c_nationkey = n_nationkey
   LIMIT 10;
   ```

Exercise 4 Transform the following usage of coalesce into a usage of the case statement:

```
SELECT coalesce(description, 'None')
FROM sometable;
```

Solution:

```
SELECT CASE WHEN description IS NOT NULL
       THEN description
       ELSE 'None' END
FROM sometable;
```

Exercise 5 Decorrelate this SQL query:

```
-- TPC-H Query 17
SELECT sum(l_extendedprice) / 7.0 as avg_yearly
FROM lineitem,
     part
WHERE p_partkey = l_partkey
  and p_brand = 'Brand#23'
```

```
    and p_container = 'MED BOX'
    and l_quantity < (
          SELECT 0.2 * avg(l_quantity)
          FROM lineitem
          WHERE l_partkey = p_partkey
        )
```

Solution:

```
SELECT sum(l_extendedprice) / 7.0 as avg_yearly
FROM lineitem l,
     part,
     (
       SELECT 0.2 * avg(l_quantity) avg, l_partkey
       FROM lineitem
       group by l_partkey
     ) quant
WHERE p_partkey = l.l_partkey
  and p_brand = 'Brand#23'
  and p_container = 'MED BOX'
  and quant.l_partkey = p_partkey
  and l_quantity < quant.avg
```

(Optional) Exercise 6 We continue with our `C++` exercise from the previous sheets. Again consider this simple query on the TPC-H dataset:

```
SELECT sum(l_extendedprice) FROM lineitem
```

Now, we want to improve our solution further by utilizing multiple processing cores. Clone the project from this repository and implement the missing code fragments in the section marked by `TODO`. Partition the input data into multiple chunks and compute the sum of each partition individually.

Once the implementation is done, measure the execution time and compare it with the single threaded approach. How well does your implementation scale, that means how much does it gain with each additional processing core? What new bottlenecks can you identify?

Hint: You can instantiate a `C++` template function by specifying parameters between angle brackets right after the function name e.g.`sum_extendedprice<true>(...)`. This page also provides an introduction to template functions.

Solution:

```
int64_t sum_parallel(const void* data, size_t size) {
    const size_t num_threads = std::thread::hardware_concurrency
       ();
    results.resize(num_threads, 0);
    std::thread threads[num_threads];

    size_t remaining = size;
    size_t partition_size = (size + num_threads - 1) /
       num_threads;
    const char* data_start = static_cast<const char*>(data);
    const char* partion_start = data_start;
    //-- TODO exercise 4.3
    for (size_t i = 0; i < num_threads; ++i) {
        size_t size_hint = std::min(remaining, partition_size);
        remaining -= size_hint;
        int64_t* result = &results[i];
        if (i == 0) {
            threads[0] = std::thread(sum_extendedprice<true>,
               data_start,
                                     partion_start, size_hint,
                                        result);
        } else {
            threads[i] = std::thread(sum_extendedprice<false>,
               data_start,
                                     partion_start, size_hint,
                                        result);
        }
        partion_start += size_hint;
    }
    for (size_t i = 0; i < num_threads; ++i) {
        threads[i].join();
    }
    //--

    // aggregate results of all workers
    int64_t price_sum = std::accumulate(results.begin(), results
       .end(), 0ul);
    return price_sum;
}
```