Data Processing on Modern Hardware
Assignment 2

Handout: **5th May 2021**
Due: **12th May 2021 by 9am**

Introduction

For this assignment we have provided a sample code-base WeeDB of a minimal variant of an in-memory database system, which is suitable for an analysis of different execution models. The code-base contains the operators: scan, selection and aggregation for two of the processing models we covered in class: *iterator* and *materialization* models. In order to execute the queries, the query plans are first specified as a hierarchy of relational operators. In a full-fledged RDBMS, these plans would be generated with the help of the SQL parser and an optimizer.

The following query plan reads the relation R, filters the elements with values 15 and 42, and sums up the other elements.

$$\begin{tikzcd}
\underset{\mathrm{sum}(x)}{\Gamma} \arrow[d, dash] \\
\underset{x \neq 15}{\sigma} \arrow[d, dash] \\
\underset{x \neq 42}{\sigma} \arrow[d, dash] \\
\text{SCAN R}
\end{tikzcd}$$

```
//build query plan
RelOperator* root;
root = new AggregationOp ( AggOp::SUM,
   new SelectionOp ( PredType::EQUALS_NOT, 15,
        new SelectionOp ( PredType::EQUALS_NOT, 42,
            new ScanOp ( R )
        )
    )
);
```

The operators support the `open()`, `next()`, and `close()` interfaces for the iterator processing model and `setRelation()` interface for the materialization processing model. The following pseudo-code illustrates how to use the APIs to implement simple query plans for the two models.

```
//execute tuple-at-a-time
root->open();
Tuple* t = root->next();
while ( t != nullptr) {
    t = root->next();
    print ( t );
}
root->close();
```

```
//execute operator-at-a-time
Relation resultRelation =
    root->get();
```

To do the required analysis you should use a profiler. Most modern processors have so-called ”performance counters”, with which different cache- and CPU-events can be directly measured in hardware. These are counters with which certain events, e.g., “All requests that missed L2” can be counted. There are various tools and libraries to access the counters. For instance: Intel PCM[1] is a library that is available as open source software, works on Linux and supports very modern features of the Intel Processors.

[1] `https://github.com/opcm/pcm`

Perf[2][3] is a tool that emerged from the Linux kernel development and also works as a standalone program. It is widely used, and has decent community support with many tutorials. For Intel CPUs you can find the list of events that can be monitored in the Intel manuals [4].

1. Task – Analysis

Execute different queries with tuple-at-a-time and operator-at-a-time execution models. You can either use the given queries, or come up with your own. Analyze the runtime behaviour of your system and the use of the system resources like CPU, caches, main memory and memory bus to answer the following questions:

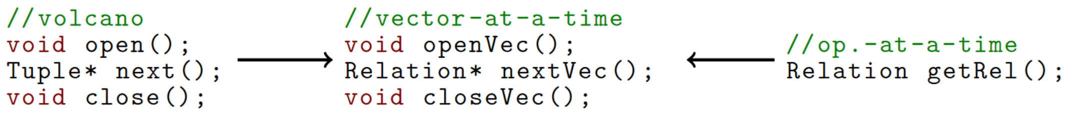
1. How heavily do the two processing models use the system's resources?
2. What changes in resource usage do you observe when you vary the number of operators? Do the differences meet your expectations? Perform some back of the envelope calculations.

The following metrics can be useful for your analysis:

- the number of memory accesses, e.g., cache lines that are moved between the main memory and the LLC (L3 cache).
- the total number of instructions that were executed.
- the number of cache hits and cache misses

2. Task – Vectorization model

The vectorization model aims to increase the efficiency of the materialization model with a better use of the CPU caches. Similar to the iterator model, the data of the relations are read and passed on from operator to operator without further I/O. However, instead of individual tuples, the operators pass blocks (i.e., vectors) of tuples to the next operator. With a suitable block size, the data remains in the CPU cache. Vector-at-a-time can thus be seen as a combination of the two processing models. This is also reflected in the interfaces of the different processing models, as shown in the figure:



```
//volcano
void open();
Tuple* next();
void close();
```

```
//vector-at-a-time
void openVec();
Relation* nextVec();
void closeVec();
```

```
//op.-at-a-time
Relation getRel();
```

Extend WeeDB with the vectorization model. Implement the vector-at-a-time interface for the DBMS operators in BaseOperator.h, Operators.h, OperatorsVector.cpp and WeeDB.cpp. You can use the implementations of the other processing models as a guide.

3. Task – Analysis

Evaluate your implementation of the vector-at-a-time processing model. Expand on the previous analysis and the cost-model. Use experiments to answer the following questions:

- What influence does the block size have on the execution of vector-at-a-time?
- Can vector-at-a-time increase the cache efficiency compared to operator-at-a-time processing?

[2]https://perf.wiki.kernel.org/index.php/Main_Page
[3]http://sandsoftwaresound.net/perf/perf-tutorial-hot-spots/
[4]http://www.intel.com/products/processor/manuals