2 Cache Awareness in Databases

In this assignment we are looking at the cache characteristics of row and column store database systems. As an example we are using a simple artificial query applied on a single relation of the TPC-H [4] benchmark database. TPC-H is a widely used performance benchmark for data warehouse (decision support) setups. Here we consider the *lineitem* relation which we model in a main-memory row store using the following C struct:

```
/* LINEITEM schema adapted from the
 * TPC-H Benchmark Specification 2.8.0, page 15 */
typedef struct {
   uint32_t orderkey;         /* identifier */
   uint32_t partkey;          /* identifier */
   uint32_t suppkey;          /* identifier */
   int64_t  quantity;         /* decimal */
   int64_t  extendedprice;    /* decimal */
   int64_t  discount;         /* decimal */
   int64_t  tax;              /* decimal */
   char     returnflag;       /* fixed text, size 1 */
   char     linestatus;       /* fixed text, size 1 */
   uint32_t commitdate;       /* date */
   uint32_t receiptdate;      /* date */
   char     shipinstruct[26]; /* fixed text, size 25 */
   int32_t  linenumber;       /* integer */
   char     shipmode[11];     /* fixed text, size 10 */
   char     comment[45];      /* variable text, size 44 */
   uint32_t shipdate;         /* date */
} lineitem_t;
```

The N row relation corresponds to an array of type `lineitem_t`. In the column store implementation there is an array for each attribute.

```
/* row store */                /* column store */
lineitem_t relation[N];        uint32_t orderkey[N];
                               uint32_t partkey[N];
                               ...
                               char     comment[45*N];
                               uint32_t shipdate[N];
```

In this exercise, we ask you to implement the following SQL query:

```
SELECT sum(orderkey+linenumber*shipdate)
FROM   lineitem
```

2.1 Analysis: Row vs. Column Store

For the data layout given by store type and the data access pattern of the query estimate the execution time of the query in the previous section.

Questions:

1. What is the memory layout of the `lineitem_t` struct?
2. With your knowledge about main memory and caches from the lecture and assignment 1, what do you expect for these approaches on your machine?

2.2 Experiment: Row vs. Column Store

We now ask you to implement the query in a row store using the struct and in a column store using the corresponding column arrays.

2.2.1 Generating the TPC-H data set

First, download the TPC-H load generator [5] and untar it in a separate directory. Rename the existing makefile template `makefile.suite` to `Makefile` and edit it as follows:

```
CC      = gcc
DATABASE= DB2
MACHINE = LINUX
WORKLOAD= TPCH
```

These settings will allow you to build the `dbgen` tool under Linux. Build the tools by typing `make`. Next, create the `lineitem.tbl` input file for the *lineitem* table.

```
$ ./dbgen -T L -s 1
```

By specifying `-T L` only the `lineitem.tbl` out of the entire data set is generated. The option `-s` is the scaling factor of the data set. For a scaling factor 1 `dbgen` will generate approximately 6 million rows. For a factor 2 approximately 12 million rows, etc.

2.2.2 Implement the query

We provide a skeleton [1] for the implementation in a column and a row store. Download and untar it in a new directory. Also copy previously generated data file `lineitem.tbl` into that directory.

Row Store. Implement the row store in the provided `row_store.c` file. The rows are loaded into an array of type `lineitem_t` (see `schema.h`) by the function `load_lineitems_rows`. `numrows` will contain the number of elements in the array, i.e., the number of rows loaded.

Column Store. Implement the column store in the provided `column_store.c` file. The function `load_lineitems_columns` allocates and loads the data from the file into 16 arrays, each representing a single column.

To keep matters simple you can implement the query in a *for* loop that iterates over all `numrows` rows. For measuring execution time, you can use the POSIX function `gettimeofday`. See the man page for details. For higher time resolution you can execute the queries multiple times.

Important: Note that since the data is loaded into main memory you have to choose the number of rows, i.e., the scaling factor of the data set, according to the RAM size of your system. Use `sizeof(lineitem_t)` to determine the size of a row and then calculate how many rows can fit into the main memory of your system. For the scaling factor also decimal fractions can be used.

Questions:

1. What is the difference in response time between the two implementations?
2. What is the speed up of the column store implementation?

2.3 Vectorized Volcano-based Pipeline

In the code skeleton [1] we implemented a few simple operators according to the Volcano Iterator model [3] for a column store. As an example to illustrate how to use the operators (implemented in `engine.c`) we combined them in `cs_iterator.c` into a plan corresponding to the following query:

```
SELECT MAX(orderkey) FORM lineitems .
```

Task: Implement the vectorized Volcano iterator pipeline [2]. That is, modify the iterator functions `next_*` in `engine.h` such that instead of one single tuple a vector of up to N tuples is processed. You can choose an implementation where N is set as a compile-time parameter. Measure the execution time for the query shown in Section 2. Vary the vector length from 1 to 1,000,000 elements.

Questions:

1. How does the vector size affect the performance?
2. What is the optimal vector length on your system and the given query?
3. Repeat the experiment for other queries such as:

```
SELECT sum(orderkey)          SELECT sum(linenumber*shipdate)
FROM   lineitem               FROM   lineitem

SELECT sum(orderkey+linenumber*shipdate)
FROM   lineitem
```

   How does the execution time change for different vector sizes? Explain!

References

[1] TPC-H data loading modules and code skeletons. http://www.systems.ethz.ch/sites/default/files/file/dpmh_Fall2012/src-handout-02_tar.bz2.

[2] Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-pipelining query execution. In *Conf. on Innovative Data Systems Research (CIDR)*, pages 225–237, 2005.

[3] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[4] TPC. TPC-H benchmark specification 2.8.0. http://www.tpc.org/tpch/spec/tpch2.8.0.pdf, 2008.

[5] TPC. TPC-H load generator: DBGEN and QGEN v2.8.0. http://www.tpc.org/tpch/spec/tpch_2_8_0.tar.gz, 2008.