Exercise Session 2

Data Processing on Modern Hardware
263-3502-00L — Fall Semester 2012

Cagri Balkesen
cagri.balkesen@inf.ethz.ch

Department of Computer Science ETH Zurich, Switzerland

27 September 2012

Part 1: Row Store vs. Column Store

- Last week → get a general understanding of the memory hierarchy in your machine
- This week → inspect cache characteristics with respect to SQL queries (row store vs. column store)
- Assignment 2 (part 1): you will implement a simple query on a *single relation* of the TPC-H benchmark database
- The query will be completely evaluated in main memory
- This time we provide a framework. You only need to
  - implement the query in the row store
  - implement the query in the column store
  - do the measurements

Generating the TPC-H Data Set

1. Download the TPC-H load generator
   → `http://www.tpc.org/tpch/spec/tpch_2_8_0.tar.gz`
2. Rename `makefile.suite` → Makefile and edit as follows:

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

3. Build `dbgen` tool
4. Create `lineitem.tbl` → `$ ./dbgen -T L -s 1`
   `-T L` : only create lineitem table of the entire data set
   `-s 1` : scaling factor 1 ≈ 6 million, 2 ≈ 12 million
5. Copy `lineitem.tbl` to the `src-handout` directory

LINEITEM schema

→ adapted from TPC-H Benchmark Specification 2.8.0, page 15

```
typedef struct {
 uint32_t          orderkey;
 uint32_t          partkey;
 ...
 int32_t           linenumber;
 char              shipmod[11];
 char_t            comment[45];
 uint32_t          shipdate;
} lineitem_t;
```

What is the memory layout of the lineitem_t struct? → Alignment
Hint: Use `sizeof(...)` and `offsetof(...)` (defined in `stddef.h`)

SQL Query on LINEITEM

You should implement the following (nonsense) query for the row store and the column store:

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

```
/* row store */
lineitem_t relation[N];
```

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

Before you implement → what do you expect for these two approaches?

Profiling: Hardware Performance Counters

- You can measure execution time using `gettimeofday(...)`
- **Profile** your the application to understand better what is going on
- Modern CPUs contain (a few) performance counter and monitoring registers
- Registers count events
  - Branch misprediction
  - Cache miss
  - TLB miss
  - ...
- After a number of events an interrupt is raised and the profiler, e.g., OProfile, can update statistics

OProfile: Full System Profiler

- System-wide profiler for Linux (Kernel+User Space)
- Sampling-based, no instrumentation
- Kernel module contains handler for interrupt and controls access to Performance Counter Registers

Example:

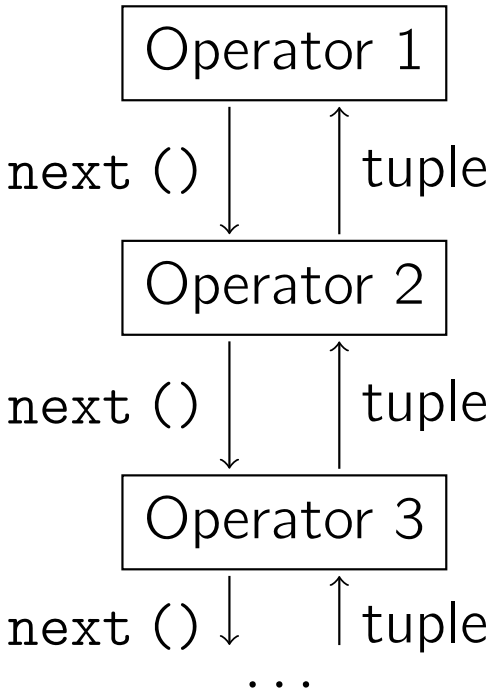
```
$ opcontrol --list-events
...
$ sudo su
# opcontrol --setup --no-vmlinux \
--event=L1D_PEND_MISS:10000
# opcontrol --start-daemon
# opcontrol --start
# ./column_store
# opcontrol --shutdown
# opreport -l
```

- **Installation (Ubuntu)**:
  `$ sudo apt-get install oprofile`
- **Hint**: use `--image` (opcontrol) option to filter resuts

Part 2: Volcano-based Pipeline

The query processing engine of most RDBMSs implement the **Volcano Iterator Model**:

- Operators request tuples from their input operator(s) using `next ()`
- Data is processed **tuple-at-a-time**
- Operators are **pipelined**
- Each operator keeps its own **state**
- How well is this model suited for main memory column store?



The Framework: Operators

- `engine.h`, `engine.c` → definitions and implementation of operators
  - `anyop` (base type), only `next ()`
  - `aggop` (AVG, COUNT, MAX, MIN, SUM), 1 input op
  - `arithop` (ADD, SUB, MULT, DIV), 2 input ops
  - `scanop`, pointer to relation

- Example: type definition of `anyop`:

```
typedef struct arithop
   /* function pointer to operator implementation:
    * return value  > 0  corresponds to the number of tuples returned
    *               == 0  means end of relation reached
    *               == -1 means error occurred */
   int         (*next)(int32_t *tuples, struct arithop *op);

   /* operator parameters */
   operation_t operation;  /* operation implemented in arithmetic operator */
   anyop_t     *leftop;    /* left child operator */
   anyop_t     *rightop;   /* right child operator */
 arithop_t;
```

The Framework: Combining Operators to a Query

`cs_iterator.c` → combine operators into a query plan

Default Query:

```
SELECT MAX(orderkey)
FROM   lineitem
```

```
aggop_t     op_agg;
scanop_t    scan_orderkey;
int32_t     agg_result;

init_aggop(&op_agg, MAX, (anyop_t*)&scan_orderkey);
init_scanop(&scan_orderkey, orderkey, numrows, \
  sizeof(uint32_t),0, sizeof(uint32_t));

if (op_agg.next(&agg_result, &op_agg) == -1)
  return -1;
```

Get Accustomed with the Frame Work

Implement different queries:

```
SELECT SUM(orderkey)
FROM   lineitem
```

```
SELECT SUM(linenumber*shipdate)
FROM   lineitem
```

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

Your Main Task

Task: Modify the iterator functions `next_*` such that instead of one single tuple a vector of up to N tuples is processed.

- The iterator functions `next_*` are in `engine.c`
- **Hint**: you may also want to edit the operator structs in `engine.h` and `init_*` functions in `engine.c` to support vectorized processing

Measurements: Where is the Sweet Spot

- Vary the vector length *N* from **1** to **1,000,000** elements
- You can measure execution time using `gettimeofday(...)`
- As you will do many measurements a shell script might help:

```
#!/bin/bash
EXECUTABLE="./foo"
OUTFILE="results.txt"
for sizelog2 in $(seq 0 22)
do
size=$((1<<$sizelog2))
make clean
make foo NEXT_VECTOR_SIZE=$size

echo -n $size"  " >> $OUTFILE
$EXECUTABLE  2>> $OUTFILE
done
```

NEXT_VECTOR_SIZE: Makefile Example

```
# Makefile Example
CFLAGS = -m64 -O3 -g -Wall

ifndef NEXT_VECTOR_SIZE
  NEXT_VECTOR_SIZE = 4096
endif

all: foo

foo: foo.c
  gcc $(CFLAGS) -DNEXT_VECTOR_SIZE=$(NEXT_VECTOR_SIZE) \
                -o $@ foo.c

.PHONY: clean
clean:
  rm -f foo
```

NEXT_VECTOR_SIZE: C Program Example

```
#include <stdio.h>

#ifndef NEXT_VECTOR_SIZE
#define NEXT_VECTOR_SIZE 4096
#endif

int main(int argc, char **argv) {
  printf("Print to STDOUT %d\n", NEXT_VECTOR_SIZE);
  fprintf(stderr, "Print to STDERR %d\n", NEXT_VECTOR_SIZE);
  return 0;
}
```

Hand in your Results

- E-Mail to cagri.balkesen@inf.ethz.ch
- Subject = DPMH:assignment2_{your netzname}
- Body = Description of CPU you tested on, e.g., Xeon QuadCore L5520,(4x 2267MHz)
- Attach plots + raw data
- Attach source code (optional)

Literature

This assignment is based on

- Peter Boncz, Marcin Zukowski, Nielw Nes. *MonetDB/X100: Hyper-Pipelining Query Execution*. Conference on Innovative Data Systems Research (CIDR), 2005.

- Goetz Graefe. *Volcano - An Extensible and Parallel Query Evaluation System*. IEEE Transactions on Knowledge and Data Engineering, 1994.