Exercise Session 5

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

18 October 2012

Assignment

Task 1: Row vs. Column Store

- Estimate execution time
  - Analyze memory layout
- Generate the TPC-H data set
- Implement row store
- Implement column store
- Compare the two approaches

Task 2: Vectorized Pipeline

- Implement the vectorized Volcano iterator pipeline
- Measure how the vector size affects the performance
- Implement various queries

Alignment Rules for `struct`

- Some CPU can only access aligned data (Alpha, SPARC)
- IA32/x86-64 can access unaligned data (slower)
- MMU has to "compose" unaligned access using two aligned accesses
- Example:
  - 32-bit `int`: last 2 bits of address always zero
- member of `struct` are aligned

Rule:

- Each member of the struct has to be aligned according to its size.
- Also when accessing members through an array of structs.

- Linux IA32: `short` multiples of 2. `int`, `float`, `doubles` multiple of 4.
- Linux x86-64: `long` and `double` multiple of 8.
- Windows: *k*-byte objects are aligned to addresses of multiple of *k*. *Always!*

SQL Query on LINEITEM

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

LINEITEM schema

```
typedef struct
  uint32_t orderkey;          char     linestatus;
  uint32_t partkey;           uint32_t commitdate;
  uint32_t suppkey;           uint32_t receiptdate;
  int64_t  quantity;          char     shipinstruct[26];
  int64_t  extendedprice;     int32_t  linenumber;
  int64_t  discount;          char     shipmode[11];
  int64_t  tax;               char     comment[45];
  char     returnflag;        uint32_t shipdate;
 lineitem_t;
```

Alignment of Row Structure

- Either apply rule or use `offsetof()` (defined in `stddef.h`)
- Example: `offsetof(lineitem_t, orderkey)`

- Linux IA32:
  - `offsetof(lineitem_t, orderkey) = 0`
  - `offsetof(lineitem_t, linenumber) = 84`
  - `offsetof(lineitem_t, shipdate) = 144`
  - `sizeof(lineitem_t) = 148`

- Linux x86-64:
  - `offsetof(lineitem_t, orderkey) = 0`
  - `offsetof(lineitem_t, linenumber) = 88`
  - `offsetof(lineitem_t, shipdate) = 148`
  - `sizeof(lineitem_t) = 152`

- A row spans **3 64-byte cache lines**.

Row Store Implementation

```
int32_t i;
int32_t agg_result = 0;
gettimeofday(&start, NULL);

for(i=0; i<numrows; i++) {
  agg_result += firstrow[i].orderkey +
                firstrow[i].linenumber *
                firstrow[i].shipdate;
}

gettimeofday(&end, NULL);
printf("result: %d\n", agg_result);
printf("query exeuction: %e sec\n",
       (end.tv_sec-start.tv_sec) +
       1e-6*(end.tv_usec-start.tv_usec));
```

result = 172092783
(scaling factor = 1)

Don't forget μs!

Column Store Implementation

```
int32_t i;
int32_t agg_result = 0;
gettimeofday(&start, NULL);

for(i=0; i<numrows; i++) {
  agg_result += orderkey[i] +
                linenumber[i] *
                shipdate[i];
}

gettimeofday(&end, NULL);
printf("result: %d\n", agg_result);
printf("query exeuction: %e sec\n",
       (end.tv_sec-start.tv_sec) +
       1e-6*(end.tv_usec-start.tv_usec));
```

Row vs Column Store

- Size of data set: 6 M tuples
- Row store: 152 B/tuple $\times$ 6 M tuples $=$ 912 MB scanned
- Column store: 3 attr $\times$ 4 B/attr $\times$ 6 M tuples $=$ 72 MB scanned

- Not only more memory scanned, row store also cache-inefficient.

- **Experiment** (Core2 Quad, QX9300, 2.53 GHz, 64 bit)
  - Row store: 133 ms
  - Column store: 18 ms
  - **Speedup of column store:** 7.4$\times$

Hardware Performance Counters

- Modern CPU contain several performance counter and monitoring registers.
- Register count events
  - Branch misprediction
  - Cache miss
  - TLB miss
  - ...
- After a number of events an NMI is raised $\rightarrow$ Kernel reads program counter and updates statistics.
- Choice of counters trade off sampling accuracy and overhead.

Events for Core2

Warning: different events with different names on different microarchitectures → here events for Core microarchitecture!

Event	*Description*
`L1D_PEN_MISS`	Outstanding L1 data cache misses
`L1I_MISSES`	Number of instruction fetch misses
`LLC_MISSES`	L2 cache demand requests that missed L2
`DTLB_MISSES`	DTLB miss events (TLBD0, STLB, load/store)
`ITLB`	Number of ITLB misses
`CPU_CLK_UNHALTED`	Clock cycles when not haltet
`INST_RETIRED`	Number of instructions retired

OProfile — Full System Profiler

- System-wide Profiler for Linux (Kernel+User Space)
- Sampling-based, no instrumentation
- Consists of Kernel module and User Space tools
- Kernel Module (in main tree) contains handler for NMI and controls access to Performance Counter Registers

Example:

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

OProfile — Row Store

```
# opcontrol --setup --no-vmlinux \
  --event=LLC_MISSES:10000 --event=L1D_PEND_MISS:10000 --image=[path to binary]
# opcontrol --start-daemon
# opcontrol --start
# ./row_store
# opcontorl --stop
# opreport -l

CPU: Core 2, speed 1600 MHz (estimated)
Counted L1D_PEND_MISS events (Total number of outstanding L1 data cache
misses at any cycle) with a unit mask of 0x00 (No unit mask) count 10000
Counted L2_RQSTS events (number of L2 cache requests) with a unit mask
of 0x41 (multiple flags) count 10000
samples  %        samples  %        symbol name
24131    98.9138  78       100.000  main
262       1.0739  0              0  load_lineitems_rows
2         0.0082  0              0  .plt
```

→ **24,131** $\times 10,000$ L1 misses in `main`

→ **78** $\times 10,000$ L2 misses in `main`

OProfile — Column Store

```
# opcontrol --setup --no-vmlinux \
  --event=LLC_MISSES:10000 --event=L1D_PEND_MISS:10000 --image=[path to binary]
# opcontrol --start-daemon
# opcontrol --start
# ./column_store
# opcontorl --stop
# opreport -l

CPU: Core 2, speed 1600 MHz (estimated)
Counted L1D_PEND_MISS events (Total number of outstanding L1 data cache
misses at any cycle) with a unit mask of 0x00 (No unit mask) count 10000
Counted L2_RQSTS events (number of L2 cache requests) with a unit mask
of 0x41 (multiple flags) count 10000
samples  %        samples  %        symbol name
2708     94.8844  3        100.000  main
126       4.4149  0              0  load_lineitems_columns
20        0.7008  0              0  .plt
```

→ **2,708** ×10,000 L1 misses in `main` (8.9 times less)

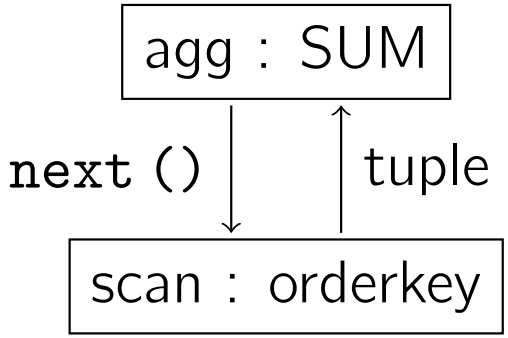
→ **3** ×10,000 L2 misses in `main` (26 times less)

Resource Stalls & IPC

- **Resource stalls**: These occur when the processor's pipeline cannot continue processing. Resource stalls increase as the number of cache misses increase.
- Oprofile: `EVENTS="RESOURCE_STALLS:10000"`
- Row Store $\rightarrow 16560 \times 10^5$ stalls, Column Store $\rightarrow 1669 \times 10^5$ stalls
- **IPC** : Instructions per Cycle
- Can we measure this with Oprofile?
- IPC = `INST_RETIRED` $\div$ `CPU_CLK_UNHALTED`
- Row Store $\rightarrow 0.14$, Column Store $\rightarrow 1.30$

Vectorized Volcano-based Pipeline : Queries

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




```
aggop_t op_sum;
scanop_t op_scan_orderkey;
init_aggop(&op_sum, SUM, (anyop_t*)&op_scan_orderkey);
init_scanop(&op_scan_orderkey, orderkey, numrows,
sizeof(uint32_t), 0, sizeof(uint32_t));
if (op_sum.next(&agg_result, &op_sum) == -1) return -1;
```

Vectorizing engine.h

```
#ifndef NEXT_VECTOR_SIZE
  #define NEXT_VECTOR_SIZE 4096
#endif

typedef struct aggop {
  ...
  int32_t *input_vector;
} aggop_t;

typedef struct arithop {
  ...
  int32_t *left_input;
  int32_t *right_input;
} arithop_t;
```

Vectorizing engine.c : init() and free()

```
int init_aggop(aggop_t *op, aggregate_t agg, anyop_t* input) {
 op->input_vector = (int32_t*)malloc(sizeof(int32_t)*NEXT_VECTOR_SIZE);
 ... }

int init_arithop(arithop_t *op, operation_t o, \
                 anyop_t *leftop, anyop_t *rightop) {
 op->left_input = (int32_t*)malloc(sizeof(int32_t)*NEXT_VECTOR_SIZE);
 op->right_input = (int32_t*)malloc(sizeof(int32_t)*NEXT_VECTOR_SIZE);
 ... }

void free_aggop(aggop_t *op) {
 free(op->input_vector);
}

void free_arithop(arithop_t *op) {
 free(op->left_input);
 free(op->right_input);
}
```

Vectorizing engine.c : next_scanop()

Tuple-at-a-time

```
int
next_scanop(void *tuples, scanop_t *op)
{
   uint8_t *bytes;
   if (op->row < op->numrows) {
      bytes = ((uint8_t*)op->rows) +
              op->row*op->rowwidth +
              op->offset;

      switch (op->width) {
      case 1:
         *((uint8_t*)tuples) = *bytes;
         break;
      ...
      }
      op->row++;
      return 1;
   } else {
      return 0;
   }
}
```

Vectorized

```
int
next_scanop(void *tuples, scanop_t *op)
{
   uint8_t *bytes;
   uint32_t n=0;
   while ((n<NEXT_VECTOR_SIZE)
          && (op->row < op->numrows)) {
      bytes = ((uint8_t*)op->rows) +
              op->row*op->rowwidth +
              op->offset;

      switch (op->width) {
      case 1:
         ((uint8_t*)tuples)[n] = *bytes;
         break;
      ...
      }
      n++;
      op->row++;
   }
return n;}
```

Vectorizing `engine.c` : next_aggop()

Tuple-at-a-time

```
int
next_aggop(int32_t *tuples, aggop_t *op)
{
  uint32_t r;
  int tuple;
  for (;;) {
    r =
    op->input->next(&tuple,
    op->input);
    if (r > 0) {
      op->count++;
      ...
    } ...
  }
}
```

Vectorized

```
int
next_aggop(int32_t *tuples, aggop_t *op)
{
  uint32_t n, numtuples;
  for (;;) {
    numtuples =
    op->input->next(op->input_vector,
                    op->input);
    if (numtuples > 0) {
      for (n=0; n<numtuples; n++) {
        op->count++;
        ...
      }
    } ...
  }
}
```

Throughput vs Vector Size