Databasesystems 2

Forum: https://forum-db.informatik.uni-tuebingen.de/c/ss18-db2

Assignment 5 (29.05.2018)

Submission: Tuesday, 05.06.2018, 10:00 AM

1. [20 Points] **Classic Clock Sweep**

   The **Clock Sweep algorithm** or Clock algorithm is an approximation of the **LRU** replacement strategy. The algorithm works by structuring the buffer like a *clock* where the first and last entry are considered to be adjacent to each other. We also have to keep track of the following:

   - The **clock pointer** ($\Leftarrow$) which initially points to the first buffer entry.
   - For each buffer entry, we keep track of a **recently used bit** which is either set (`1`) or reset (`0`).

   Replacement Strategy:
   Given a set of positive integers $N = \{1, 2, \ldots, n\}$ representing *disk pages*. Below, a sequence of page references $R = r_1, \ldots, r_t, \ldots$ where each r_t denotes some page, will be referred to as *reference string*.

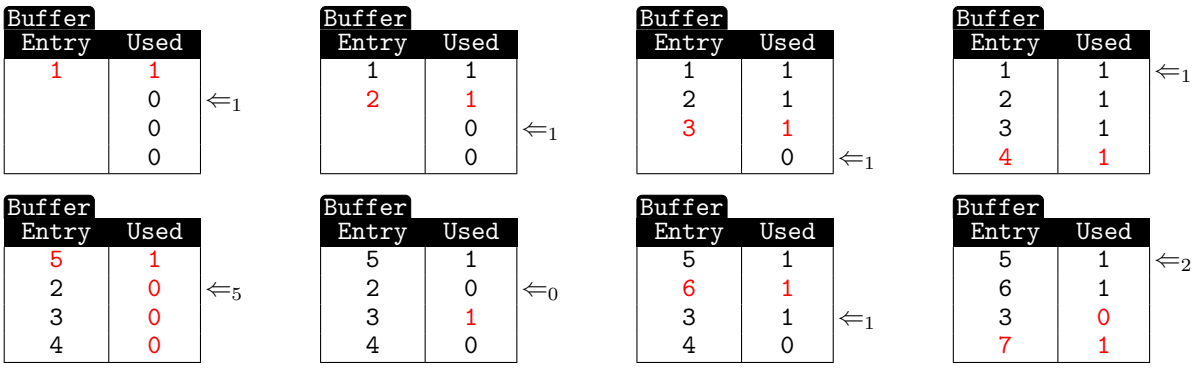
   For each page p referenced by R, the Clock algorithm works as follows:

   1. If p already exists in the buffer, set its recently used bit and continue with the next page in the reference string. Otherwise, continue with step 2.
   2. If the recently used bit at the current clock pointer position is reset, set it and replace the buffer entry with p. Then, move the clock pointer to the next buffer entry and continue with the next page in the reference string. Otherwise, continue with step 3.
   3. Reset the recently used bit at the current clock pointer position. Then, move the clock pointer to the next buffer entry and continue with the next page in the reference string.

   Example:
   We read the following reference string in order:

   $$R = 1,2,3,4,5,3,6,7$$

   The buffer size is set to 4. Thus, the Clock algorithm takes the following steps:

   Note: In some of the steps of this example, the clock pointer may move multiple times. This is indicated by $\Leftarrow_i$ where i is the amount of steps the pointer has taken to reach its position.

(a) You are given the **incomplete** C program `clock.c`. Complete and hand in the working implementation of the Clock algorithm by writing the function body of the following function:

```
int clock_sweep_step(int clock_pointer, buffer *buffer_state, int
    buffer_size, int current_reference) {
   // YOUR CODE HERE
}
```

Running the the program with a buffer size of 4 and the provided reference string file `example.ref` based on the example discussed before, the program should terminate with the following output:

```
Reference: 1, Buffer: (1,1) ( , ) ( , ) ( , ) Pointer Position: 1
Reference: 2, Buffer: (1,1) (2,1) ( , ) ( , ) Pointer Position: 2
Reference: 3, Buffer: (1,1) (2,1) (3,1) ( , ) Pointer Position: 3
Reference: 4, Buffer: (1,1) (2,1) (3,1) (4,1) Pointer Position: 0
Reference: 5, Buffer: (5,1) (2,0) (3,0) (4,0) Pointer Position: 1
Reference: 3, Buffer: (5,1) (2,0) (3,1) (4,0) Pointer Position: 1
Reference: 6, Buffer: (5,1) (6,1) (3,1) (4,0) Pointer Position: 2
Reference: 7, Buffer: (5,1) (6,1) (3,0) (7,1) Pointer Position: 0
```

(b) Modify `clock.c` to count the hits and misses of the Clock algorithm. The hits and misses have to be printed in the following format:

```
Hits: 1 Misses: 7
```

(c) In the lecture, we discussed two scenarios (1) and (2) in which LRU may fail (see: slide 18 in slide set 6). We provided you with two files `scenario-1.ref` and `scenario-2.ref` usable as input for the Clock algorithm you implemented in (a) and (b).

`scenario-1.ref`:
Provides a reference string with 100000 random generated references as described in the scenario. Pages referenced by I are numbered 1 to 100. Pages referenced by R are numbered 10001 to 20000.

`scenario-2.ref`:
Provides a reference string where T_0 sequentially scans 10000 distinct pages numbered 10001 to 20000 once. For each page referenced by T_0, the transactions T_1, T_2 and T_3 reference random pages numbered 1 to 100.

Modify `clock.c` to determine the page ratio between pages numbered 1 to 100 and 10001 to 20000 in the buffer after the Clock algorithm concludes.
Determine the optimal amount of hits and misses in each scenario and write them down. Run the Clock algorithm for each scenario with a buffer size of 500, 100 and 50 pages. Based on the results, determine which of these scenarios is more suitable for the Clock algorithm? Explain briefly.

2. [10 Points] **Simplify Expressions**

For this task, we will examine the following SQL query:

```
SELECT (((1 - (h.v/h.v)) * h.v) +
  (h.v * (h.v - h.v + (5*4 - 19))/h.v)
    ) / (
  (h.v - h.v + 1)/(h.v - h.v + 1))
FROM huge AS h;
```

create and populate table `huge(v FLOAT NOT NULL CHECK (v <> 0))` to hold 10^7 random rows.

(a) **MonetDB:**

i. Examine the MAL program of the SQL query with `EXPLAIN`. Which automatic simplifications have been done to the SQL query?

ii. Simplify the MAL program manually. Write down the most simplified MAL program with the smallest amount of MAL statements while still producing the same result as the original SQL query. Print the result using the built-in `io.print(...)` function.

(b) **PostgreSQL:**

i. Examine the ouput of the SQL query with `EXPLAIN (VERBOSE, ANALYZE)`. Which automatic simplifications have been done to the SQL query?

ii. Simplify the SQL query manually. Write down the most simplified SQL query with the least amount of arithmetic operators while still producing the same result as the original SQL query.

iii. Examine the manually simplified SQL query of (b) with `EXPLAIN (VERBOSE, ANALYZE)`. Compare the runtimes with the result of the simplified SQL query of (a). What happened? Explain briefly.