Prof. Dr. T. Grust, Benjamin Dietrich, Denis Hirn SS 2020

DB2
Forum: https://forum-db.informatik.uni-tuebingen.de/c/ss20-db2

Assignment 5 (26.05.2020)

Submission: Tuesday, 09.06.2020, 10:00 AM

Relevant videos: up to `DB2 - Chapter 07 - Video #26`.
https://tinyurl.com/DB2-2020

1. [14 Points] **Classic Clock Sweep**

The **Clock Sweep algorithm** or Clock algorithm approximates the **LRU** replacement strategy. The algorithm structures the buffer like a *clock* such that 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** ($\Leftarrow$) initially points to the first buffer entry.
- Each buffer entry is annotated with an additional **recently used bit** which can be set (`1`) or reset (`0`). This bit indicates whether or not an entry has been used since the last clock sweep.

Replacement Strategy:
In the following, we will be using the set of positive integers $N = \{1,2,\ldots,n\}$ to represent *disk pages* and the *reference string* $R = r_1,\ldots,r_t,\ldots$, where each r_t denotes a page, to describe the sequence of page references.

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

1. If page p 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 step 2.

Example:
With the buffer size set to 4 and reference string $R = 1,2,3,4,5,3,6,7$, the Clock algorithms takes these steps:

$R = \mathbf{1},2,3,4,5,3,6,7$

Buffer Entry	Used	
1	1	
	0	$\Leftarrow_1$
	0	
	0	

$R = \mathbf{2},3,4,5,3,6,7$

Buffer Entry	Used	
1	1	
2	1	
	0	$\Leftarrow_1$
	0	

$R = \mathbf{3},4,5,3,6,7$

Buffer Entry	Used	
1	1	
2	1	
3	1	
	0	$\Leftarrow_1$

$R = \mathbf{4},5,3,6,7$

Buffer Entry	Used	
1	1	$\Leftarrow_1$
2	1	
3	1	
4	1	

$R = \mathbf{5},3,6,7$

Buffer Entry	Used	
5	1	
2	0	$\Leftarrow_5$
3	0	
4	0	

$R = \mathbf{3},6,7$

Buffer Entry	Used	
5	1	
2	0	$\Leftarrow_0$
3	1	
4	0	

$R = \mathbf{6},7$

Buffer Entry	Used	
5	1	
6	1	
3	1	$\Leftarrow_1$
4	0	

$R = \mathbf{7}$

Buffer Entry	Used	
5	1	$\Leftarrow_2$
6	1	
3	0	
7	1	

Note: During some 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:

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

When running the the program with a buffer size of 4 and the provided reference string file `example.ref`, 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. Print the hits and misses 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.

Suppose the buffer is of infinite size and every page 1 to 100 and 10001 to 20000 is referenced. Determine the least amount of misses in each scenario and use this as your baseline. Run the Clock algorithm for each scenario with a buffer size of 500, 100 and 50 pages. Based on the results, which of these scenarios is suitable for the Clock algorithm? Explain briefly.

2. [8 Points] **Shared Buffercache in PostgeSQL**

   We haven't looked at indexes yet. For now, assume that indexes support rapid value-based accesses to a table. Indexes are held in separate heap files just like tables.

   The *Clock Sweep algorithm* – which is implemented in PostgreSQL – is supposed to fail in a scenario of alternating *index* and *relation* page references.

   But how does *PostgreSQL* perform in this scenario? The extension `pg_buffercache`[1] allows us to inspect the shared buffer.

   Experimental setup:

   - Load `/assignments/assignment05/populate_orders.sql` to create a table `orders` with an index `orders_idx` on column `o_orderkey`.
   - Set the configuration variable `shared_buffers` in `postgresql.conf` to `400kB` (50 pages) for this experiment.
   - Restart the `postgresql` server before the experimental run to start with a *cold cache*.

   Experiment:

   (a) On a *random access* to a tuple in `orders`, first an index and subsequently a relation page (both possibly cached in the shared buffer) are accessed. Use system catalog information from `pg_class` to answer these questions:
   - How many pages does *relation* `orders` occupy?
   - How many pages does *index* `orders_idx` occupy?
   - What is the probability of a relation resp. index page to be the target of a *random access* to a single tuple in `orders`?

   (b) In `/assignments/assignment05/experiment.sql` you can find a *PLpg/SQL* function `run(n INT)` to run n subsequent random accesses to table `orders` using index `orders_idx`. In between each access, we are going to collect statistics on the buffer content and log them to the temporary table `results`.
   - Use `pg_buffercache` to complete function `run` in line

     ```
     -- YOUR QUERY ON pg_buffercache
     ```

     with a query that returns (1) how many pages of relation `orders` and (2) how many pages of index `orders_idx` are currently located in the shared buffer. **Note:** take care *not* to access `pg_class` in that query; use static `relfilenode`s instead.
   - Restart the postgres server and run the experiment with `run(300)`.

   (c) Draw and hand in a chart of the results. For each measurement (x-axis) draw the number of buffer slots occupied by index pages and the number occupied by relation pages (y-axis) into the diagram.
   Note: You may either use a `JPG`, `PNG` or a `PDF` file to hand in your chart.

   (d) Describe and analyze the results briefly. Looking at the page access probabilities in 2a, is the PostgreSQL's page replacement strategy a good fit for this scenario?

[1] https://www.postgresql.org/docs/current/pgbuffercache.html

3. [8 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) ii with `EXPLAIN (VERBOSE, ANALYZE)`. Compare the runtimes with the result of the simplified SQL query of (b) i. What happened? Explain briefly.