EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

DB2
Sommersemester 2022
Prof. Torsten Grust, Denis Hirn
WSI — Database Systems Research Group

Assignment 5

Handin until: Friday, 03.06.2022, 09:00

The Summer Semester 2022 lecture evaluation is coming — Please give us feedback. Thank you!

Please take a few minutes and report back. Every bit of feedback counts — this is especially true for text comments. The insights gained are worth their weight in gold and allow us to assess whether we are steering the *DB2* lecture in the right direction and also how we can make the course even better in the remaining weeks of the semester.
The feedback forms can be filled in **after May 30th**. Once again: Thank you very much!

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

   The **Clock Sweep algorithm** approximates the **LRU** replacement strategy. The algorithm structures the buffer like a *clock*, so that the first and last entries are considered to be adjacent. In addition, the following must be noted:

   - The **clock pointer** (⇐) 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 an entry has been used or not since the last clock sweep.

   Replacement Strategy:

   In what follows, we use the set of positive integers 1,2, ... to denote *heap pages*. A *reference string* R is a sequence of pages.

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

   1. If page p is present 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 $R = 1,2,3,4,5,3,6,7$, Clock Sweep performs the following steps:

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

Buffer Entry	Used	
1	1	
	0	⇐1
	0	
	0	

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

Buffer Entry	Used	
1	1	
2	1	
	0	⇐1
	0	

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

Buffer Entry	Used	
1	1	
2	1	
3	1	
	0	⇐1

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

Buffer Entry	Used	
1	1	⇐1
2	1	
3	1	
4	1	

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

Buffer Entry	Used	
5	1	
2	0	⇐5
3	0	
4	0	

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

Buffer Entry	Used	
5	1	
2	0	⇐0
3	1	
4	0	

   $R = \mathbf{6},7$

Buffer Entry	Used	
5	1	
6	1	
3	1	⇐1
4	0	

   $R = \mathbf{7}$

Buffer Entry	Used	
5	1	⇐2
6	1	
3	0	
7	1	

   Note: During some steps in this example, the clock pointer may move several times. This is indicated by $\Leftarrow_i$ where i is the number of steps the pointer takes to reach its position.

(a) The file **clock.c** contains an incomplete C program. Complete the implementation Clock Sweep by writing the body of the following function:

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

```

If you run the program with a buffer size of 4 and the specified 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 Clock Sweep. 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 can fail (see: slide 18 in slide set 6). We provided you with two files **scenario-1.ref** and **scenario-2.ref** which can be used as input for the Clock Sweep algorithm you implemented in (a) and (b).

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

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

Run Clock Sweep for both scenarios with buffer size of 500, 100, and 50 pages. Based on the hits and misses you see, which scenario is suitably handled by Clock Sweep? Explain briefly.

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

We have not yet dealt with indexes. 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 table `orders` with 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 an (almost) empty shared buffer.

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 will find a *PLpg/SQL* function `run(n INT)` that will perform `n` consecutive random accesses to table `orders` using the index `orders_idx`. The function will collect statistics about the buffer contents between each access and log them in a temporary table `results`.

- Use `pg_buffercache` to complete the `run` function in line

```
1 -- 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 in the shared buffer.
Note: Use the PL/pgSQL variables `idx_filenode` and `heap_filenode` to formulate your query.

- Restart the postgres server and run the experiment with `run(300)`.

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

(d) Briefly decribe and analyze the results. Considering the page access probabilities in (a), is PostgreSQL's page replacement strategy suitable 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:

```sql
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`. What automatic simplifications were made to the SQL query?

      ii. Simplify the MAL program manually. Write the most simplified MAL program with the smallest number of MAL statements that still produces the same result as the original SQL query. Print the result with the built-in function `io.print(...)`.

   (b) **PostgreSQL:**

      i. Examine the output of the SQL query with `EXPLAIN (VERBOSE, ANALYZE)`. What automatic simplifications were made to the SQL query?

      ii. Simplify the SQL query manually. Write the most simplified SQL query with the least number of arithmetic operators that still produces the same result as the original SQL query.

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