Team23

賴怡惠 103011105 陳映竹 103062131 林宛萱 103062305

1. The basic information of the paper
   a. title:
      The LRU-K Page Replacement Algorithm For Database Disk Buffering
   b. conference
      Management of data
   c. published year
      1993
   d. authors
      Elizabeth J. O'Neil , Patrick E. O'Neill, Gerhard Weikum
2. The main idea of the paper

   LRUK即為一般常見的LRU變形版，在LRU的情況下其K = 1，意思是LRU只會考慮最近『一次』的access time，而LRUK會考慮最近的『第K次』access time！

   LRUK透過追蹤每個page最近第K(LRU-K的K)次被referenced的時間和目前的時間差，可於buffer滿時作為首先replace時間差最大的page。此方法可解決LRU1時，若join兩個table，一個table做外迴圈，另一個table做內回圈，會造成做外回圈的table一直被swap，而外回圈的record雖然會一段時間才使用一次，卻會一直被使用！有些page卻是剛剛才被access，不過很久以後都不會被access。若用LRUK(K>1)選擇被替換的page時，可檢視該page前K次和目前的時間差，若該page隔一段時間會被access，比久久access一次但才剛被access的page，機率較低被替換！所以LRUK其實是利用過去access的pattern，去預測未來access的方法，而不需要額外的hint，例如：目前在做的sql command種類。因為LRUK是依照目前的access pattern的較長時間歷史紀錄去預測未來pattern，又不需要太多overhead!

   但其缺點為一開始因為還要蒐集每個page access的pattern，所以一開始的perforamance會比較不好，但以長久來看(也就是LRUK適應aceess patter)performance會較LRU好！

   舉例：存取page的順序：p1 p1 p2 p3 p2 p3 p1 p3 p4

   LRU2-->p4會被替換掉，因為其最久沒被access

page	access interval
p1	8
p2	7
p3	4
p4	infinity

   LRU-->p2會被替換掉，因為其最久沒被access

page	access interval
p1	3
p2	5
p3	2
p4	1

3. A brief summarization of what you implement in VanillaDB

VanillaDB本來實作的Buffer Replacement的方法是採用Clock Strategy，並且實作在BufferPoolMgr裡面pin以及pinNew的method中，因此我們主要更動本來實作Clock Strategy的地方，改為LRU-K的做法，並且新增一些LRU-K所需要紀錄的東西。LRU-K的值我們透過static的方式存取紀錄在properties的值。

我們新增了兩個class，分別是LRUHistory以及BufferQueue，LRUHistory儲存blockId過去被pin以及unpin的時間（我們實作了兩種計算時間的方法作為比較），BufferQueue儲存blockId對應到在BufferPool中的index。

我們在BufferPoolMgr中建了一個historyMap（儲存BlockId跟LRUHistory的對應），一個buffQueue的list，以及兩個timer用來紀錄進入pin/pinNew和unpin的時間。

A. <u>BufferPoolMgr.pin</u>

每次進入pin時timer會+1，根據findExistingBuffer(blk)的結果我們有兩種可能性。

1. findExistingBuffer(blk)==null，表示目前bufferPool中並沒有該blk，所以必須做I/O將他從disk拿進來，這裡又分為兩種狀況，一種是剛開機時，bufferPool還沒被裝滿，我們採用本來的Clock Strategy，另一種是bufferPool被裝滿，必須做replacement把已經存在buffer的block flush回disk，然後將新的block放進該buffer。
   我們透過lastReplacedBuff來判斷是否裝滿bufferPool，在每次做Clock時，我們不是將lastReplacedBuff更新成currBlk，而是將他+1，所以當lastReplacedBuff超過bufferPool.length就代表需要有人被換掉。
   Swap的方式我們寫在findSwapBuffer的method中。
2. findExistingBuffer(blk)!=null，表示該blk存在bufferPool中，所以不用做replacement，但是historyMap需要新增該LRUHistory的access時間（以pin time計算時間的話需要）。

B. **<u>BufferPoolMgr.pinNew</u>**

作法幾乎同pin，只是沒有第二種findExistingBuffer(blk)!=null狀況。

C. <u>BufferPoolMgr.unpin</u>

在每個buffer被unpin的時timer2會+1，並且透過setTime新增historyMap中該buffer對應到的block的unpin時間。

D. <u>BufferPoolMgr.findSwapBuffer</u>

我們在這裏執行LRU-K的replacement strategy。buffQueue中儲存了所有bufferPool index跟blockId的對應，因此我們透過iterate buffQueue，得到blockId，再根據他去historyMap中找出他們倒數第K次access的時間，如果遇到倒數第K次時間是infinity的（我設為-1），則直接return該blockId對應到的buffer，並且更新historyMap以及該筆BufferQueue。如果遇到的時間不是infinity，則我們會將他加入我們創的validBuffQueue的hashMap中。

iterate完buffQueue中的所有BufferQueue之後，若沒有return則代表沒有infinity，這時我們會將加進validBuffQueue中的BufferQueue根據倒數第K次時間排序，將結果放入buffQueueOrder這個list，再去iterate 他，將沒有被lock住也沒有被pin住的block return回去。

值得一提的是，pin以及pinNew都會執行到這個method，因此在做buffer assign給某一個blk的動作時，我們是根據是否傳入要pin的blockId來決定執行哪個步驟。

```
if(newBlk == null)
    buf.assignToNew(fileName, fmtr);
else
    buf.assignToBlock(newBlk);
blockMap.put(buf.block(), buf);
```

E. **LRUHistory**

用來紀錄每一個曾經被access過的block的blockId，以及被pin以及unpin的時間。我們以list的方式紀錄每次該block被pin以及unpin的時間，當list中的時間超過K次時就更新倒數第K次的時間（否則維持初始化的-1）。

F. **BufferQueue**

每一個BufferQueue object我們在裡面紀錄blockId以及對應到的bufferPool的index，當該index在BufferPoolMgr中被swap進不同的block時，我們會透過setBlockId的方式更改紀錄的blockId的值。

4. Your evaluation and experiment
   A. enironment
      a. Intel Core i5-8250 CPU
      b. @ 3.4GHz
      c. 8 GB RAM
      d. 512 GB SSD
      e. windows 10
   B. evaluation

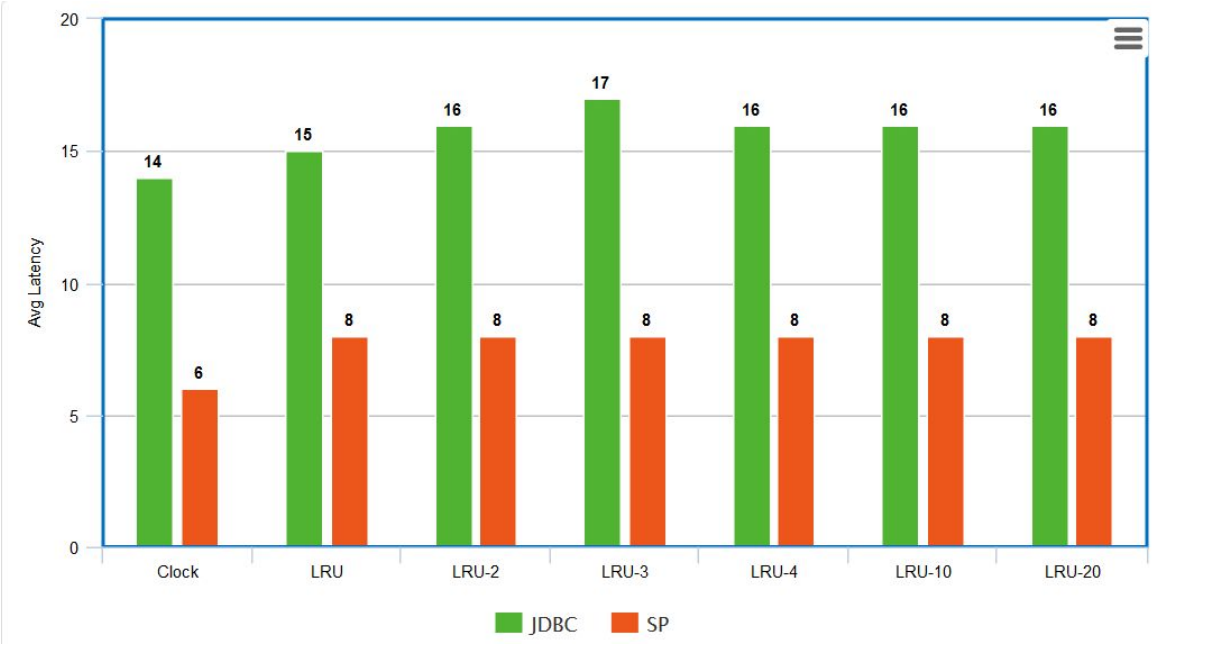
Buff_Pool_Size =1024	JDBC avg latency	avg throughput	SP avg latency	avg throughput
Orign(clock)	15	608	7	1351
LRU	14	637	8	1112

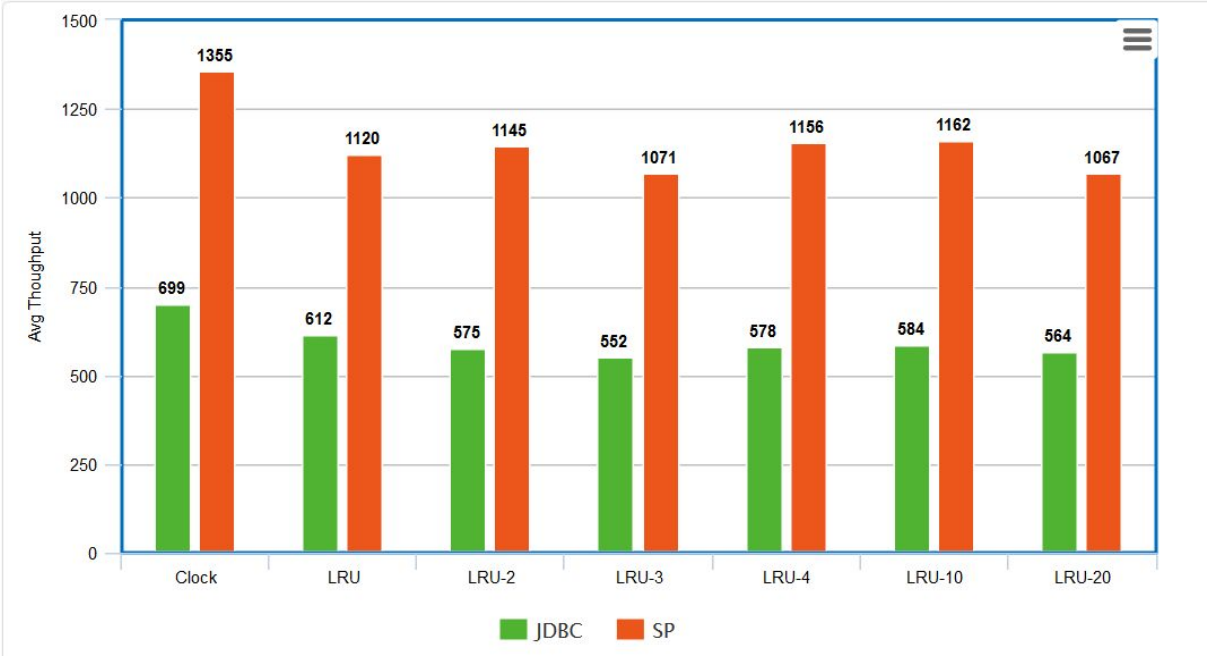
LRU-2	14	664	8	1224
LRU-3	14	655	8	1136
LRU-4	15	620	8	1123
LRU-10	16	587	8	1177
LRU-20	16	649	8	1078

Buff_Pool_Size =100	JDBC avg latency	avg throughput	SP avg latency	avg throughput
Orign(clock)	14	699	6	1355
LRU	15	612	8	1120
LRU-2	16	575	8	1145
LRU-3	17	552	8	1071
LRU-4	16	578	8	1156
LRU-10	16	584	8	1162
LRU-20	16	564	8	1067

C. experiment

以下是在buffer_pool_size=100時,avg lantency和avg throughput的長條圖





5. An analysis for your experiments

當buffer_pool_size=1024時，原來的Clock buffeMgr和我們寫的LRU相比下並不能看出顯著的差異，但當我們把buffer_pool_size調小到100時，可明顯看出在JDBC和SP下，orign(clock)版本的avg latency都較小，avg throughput也都較大。而在LRU-K值改變時，並無明顯差異。

6. The problems you occur during implementation and how you solve them

我們一開始因為對原文理解有困難就上網查LRU-K的原理，本來的理解是某一個page必須被access超過K次才會被放進buffer，因此實作上就遇到一個無解的問題，不被放進buffer要如何存取？（跑testcase的時候，tx會一直repin）向助教詢問之後才發現我們理解錯誤。

另外在剛開機的情況下buffer還沒有被裝滿所以需要default的replacement algorithm，無法一開始就使用LRU-K。因此我們一開始使用vanilla core所使用的clock replacement algorithm，來處理某些page不會被access到K次，和一開始仍在學習或適應access pattern的時間。

7. A conclusion

paper原文中最後有針對兩種page size做比較，但是因為VanillaDB並沒有看到可以修改page size的地方，因此我們只有針對bufferPool size以及K值做比較。paper的實驗中，針對不同bufferPool size，可以發現在bufferPool小的時候LRU-2的hit rate明顯高於LRU-1，但是當bufferPool變大時，兩者的差異就會縮小。LRU-2和LRU-3則沒有明顯差異，唯LRU-3似乎略優於LRU-2。

若將throughput高以及latency低當作hit rate的依據（因為少I/O理論上會比較快），我們的實驗卻沒有反映出paper的結果，原因可能是我們沒有implement好，或是與access pattern有關。

k 越大不代表performance越好，因為某個page會不會出現 K 次和access的pattern有關，K 越大表示要等到page出現 K 次其距離才有代表性，不然就relacement alogorithm即使是LRUK也跟LRU一樣了！換句話說，若某page未出現 K 次，表示其前K次出現的時間和現在的時間差為無限大，若出現時間差為無限大的page為常態！那麼我們就會依照access time越早優先替換，這也就是Least Recently Used的定義了！