TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Department of Computer Science
Heterogenous Information Systems Group

Master's Thesis:

Bottlenecks Uncovered: A Component-Wise Breakdown of the Runtime of an OLTP System

by **Max Fabian Gilbert***

Day of Issue: February 1, 2020
Day of Release: June 1, 2020

Advisor: M. Sc. Caetano Sauer
First Reviewer: Prof. Dr.-Ing. Stefan Deßloch
Second Reviewer: Prof. Dr.-Ing. Dr. h. c. Theo Härder

*m_gilbert13@cs.uni-kl.de

Abstract

This page intentionally left blank.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

This page intentionally left blank.

1 Buffer Pool Pointer Swizzling

1.1 Introduction

1.2 Performance Evaluation

1.2.1 System Configuration

1.2.2 Benchmark

1.2.3 Results

1.2.4 Analysis

1.3 Conclusion

2 Page Eviction by the Buffer Management

2.1 DBMS Buffer Management

The buffer management of a typical disk-based DBMS serves the purpose of providing the higher layers of the DBMS with managed access to data and metadata pages of a database stored in files on the secondary storage. This managed access involves fetching specific pages into specific memory locations—so-called *buffer frames*—inside the buffer pool and write-back changes to pages in memory to database files in secondary storage.

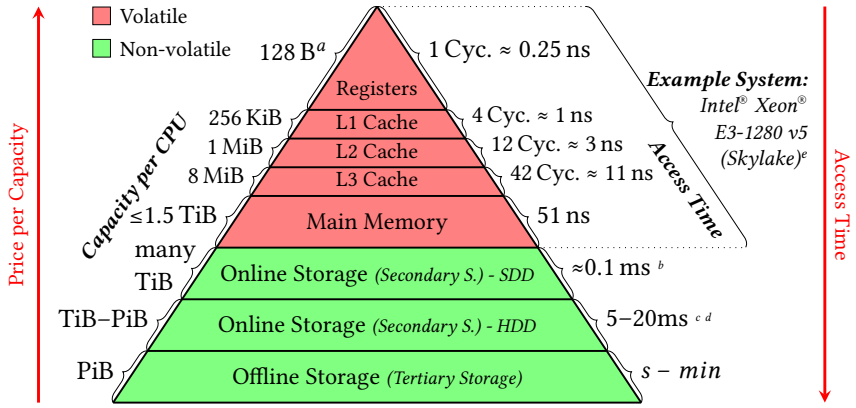
Today, there are server systems with 48 TiB of cache-coherent shared memory[1] that allow *in-memory* management of almost any database of any application. But such expensive systems (>2 000 000 €) do not pay off for most applications with large databases. Therefore, there will still be many situations in which the main memory of a system is significantly smaller than the database(s) managed on it. Accordingly, the number of buffer frames in the buffer pool is then smaller than the number of pages in the database.

Therefore, the buffer management must *evict* pages from buffer frames if currently not buffered pages are referenced while there are no more free buffer frames. For this purpose, each buffer manager has a page eviction module—implementing one of the many *page replacement algorithms* developed since the 1960s.

In *OLTP* applications, the majority of database accesses are random accesses, so the access latency of the underlying storage technology (Figure 2.1) is critical to performance. The main goal of buffer management is to maximize the *hit rate* in the DB buffer by keeping as many pages of the

[1] `https://bit.ly/37ZfGMH`

working set (pages referenced in the near future) as possible in relatively fast (51 ns) main memory to avoid expensive (>100 µs) secondary storage accesses.



[a]Only considered general purpose registers.
[b]https://bit.ly/2BaG1v9
[c]https://bit.ly/2YxdvN1
[d]https://bit.ly/2NxvMnj
[e]https://bit.ly/3dDBZbH

Figure 2.1: Storage hierarchy of computer systems

Bélády's optimal page replacement algorithm [Bél66] evicts the one page from the buffer pool that is not re-referenced for the longest time in the future. However, this algorithm cannot make this decision at runtime, since this would require knowledge about future requests to the database. Therefore, page replacement algorithms that can be implemented in DBMS must use heuristics to approximate the eviction decisions of Bélády's algorithm to achieve a high hit rate. These heuristics are usually based on the assumption that there is a temporal locality of the page references. The management of data in B-tree indices, which were developed 50 years ago to improve the spatial locality of references in order to reduce random disk access in database applications [BM70], increases the temporal locality of page references due to the hierarchical structure of B-trees in which higher-level pages are more likely to be referenced.

The expected performance of these possible page replacement algorithms relative to the buffer pool size is shown in Figure 2.2. In this diagram, MR_{CS} is the minimum miss rate, which is greater than 0 due to the page misses that occur when each page is first referenced after a cold start. The maximum miss rate $MR_{\max}$ is less than 1 due to the fact that there are always some random page hits, even with RANDOM eviction on workloads with no locality of references. D is the database size and $B_{\min}$ is the minimum buffer pool size that will still lead to a working system.

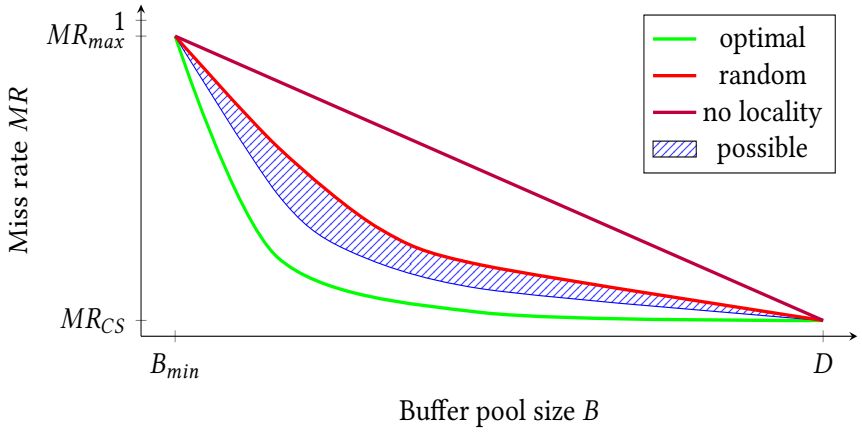


Figure 2.2: Connection between miss rate and buffer pool size [EH84]

Besides the impossibility to implement Bélády's optimal page replacement algorithm, there are also technical limitations in the use of the possible page replacement algorithms. The abstract algorithms can evict any page from the buffer pool at any time. However, in a real-world system, there are many reasons why a page that has been selected for eviction by the page replacement algorithm cannot be immediately evicted. One reason, for example, is already visible in the typical interface of a buffer pool—a page can be fixed and unfixed by the higher layers of the DBMS. Between fixing and unfixing a page, it is guaranteed that it remains in the same buffer frame, since the fixing thread processes the page during this time. Another problem for some page replacement algorithms is the possibility to explicitly evict pages from the buffer pool—independent from the decisions

of the page replacement algorithm.

The test system used for this work knows the following reasons why a page in the buffer pool is either temporarily or permanently unreclaimable:

- Metadata pages can never be evicted.
- B-tree root pages can never be evicted.
- Inner B-tree pages can never be evicted when pointer swizzling (like in [Gra+14]) is used in the buffer pool.
- B-tree pages with foster children (the Foster B-Tree from [GKK12] is used) cannot be evicted.
- Dirty pages cannot be evicted until they are written back.
- Pages pinned to the buffer pool by higher layers of the DBMS cannot be evicted.

All non-trivial page replacement algorithms collect statistics about page references to base their eviction decisions on. Simple implementations of these page replacement algorithms do not reflect in their statistics the fact that certain pages are temporarily or permanently unreclaimable. But treating pages found temporarily unreclaimable as a page reference and excluding pages that are permanently unreclaimable from the eviction could improve future eviction decisions or the runtime of the algorithm.

2.2 Page Replacement Strategies

There are two traditional classifications for page replacement algorithms—Bélády's classification from [Bél66] and the classification by Effelsberg and Härder from [EH84].

Bélády grouped the replacement algorithms into the three classes described in Table 2.1. Class 3

The classification by Effelsberg and Härder is two-dimensional as it takes into account if a page replacement algorithm considers the time since a certain reference of a page happened—the age—and the number of references of a page—the references.

The following discussion of various page eviction algorithms is not organized according to any of the two classifications

Class 1 Replacement algorithms of this class do not use statistics for their eviction decisions.

Class 2 Replacement algorithms of this class keep statistics about the latest references of *pages in the buffer pool* and use them for their eviction decisions.

Class 3 Replacement algorithms of this class keep statistics about each time *any page* was fetched from the database and each time it was evicted from the buffer pool and use these statistics for their eviction decisions.

Table 2.1: Bélády's classification of page replacement algorithms from [Bél66]

Consideration during selection decision		**Age**			
		No consideration	Since most recent reference	Since some recent reference	Since first reference
References	No consideration	RANDOM LOOP			FIFO FILO
	Most recent reference	ZCLOCK	LRU MRU CLOCK GCLOCK-V2 DGCLOCK-V2 LeanStore		
	Some recent references			LRU-K	
	All references	LFU	GCLOCK-V1 DGCLOCK-V1		

Table 2.2: Page replacement algorithm classification by Effelsberg and Härder from [EH84]

2.2.1 RANDOM

The RANDOM page replacement strategy is the simplest one as it does not maintain any statistics about past page references and

2.2.1.1 LOOP

2.2.2 First In, First Out (FIFO)

The FIFO page replacement strategy always evicts the oldest page first—it organizes the pages in a FIFO queue, which is normally implemented as a ring buffer. However, due to limitations caused by the fact that certain pages are temporarily or permanently unreclaimable, the replacement strategy must be modified for use in a DB buffer pool, and such a Quasi-FIFO cannot be implemented with a simple ring buffer, but needs more complex data structures for its page reference statistics and more complex control structures for finding replacement candidates.

The implementation evaluated here uses two queues, the *FIFO queue* and the *retry queue*. When a page is fetched into the buffer pool, it is placed in the *FIFO queue*. It will be moved to the tail of the *retry Queue* if it is found to be unreclaimable after being selected for eviction by the **function** SELECT from Algorithm 2.1.

2.2.3 First In, Last Out (FILO)

2.2.4 Least Recently Used (LRU)

The LRU page eviction algorithm is probably the most traditional approach for

Algorithm 2.1: Selection of replacement candidates with Quasi-FIFO.

```
1:  function SELECT
2:      selected ⟵ 0
3:      while true do
4:          if currentlyCheckingRetryQueue == false then
5:              if currentQueueChecks < initialQueueChecks ∨ |retryQueue| == 0 then
6:                  selected ⟵ initialQueue.pop()
7:                  currentQueueChecks ⟵ currentQueueChecks + 1
8:                  if notExplicitlyEvictedList [selected] == true then return selected
9:                  else
10:                     notExplicitlyEvictedList [selected] ⟵ true
11:                     continue
12:                 end if
13:             else
14:                 selected ⟵ retryQueue.pop()
15:                 if notExplicitlyEvictedList [selected] == true then
16:                     currentQueueChecks ⟵ 0
17:                     currentlyCheckingRetryQueue ⟵ true return selected
18:                 else
19:                     notExplicitlyEvictedList [selected] ⟵ true
20:                     continue
21:                 end if
22:             end if
23:         else
24:             if currentQueueChecks < retryQueueChecks ∨ |initialQueue| == 0 then
25:                 selected ⟵ retryQueue.pop()
26:                 currentQueueChecks ⟵ currentQueueChecks + 1
27:                 if notExplicitlyEvictedList [selected] == true then return selected
28:                 else
29:                     notExplicitlyEvictedList [selected] ⟵ true
30:                     continue
31:                 end if
32:             else
33:                 selected ⟵ initialQueue.pop()
34:                 if notExplicitlyEvictedList [selected] == true then
35:                     currentQueueChecks ⟵ 0
36:                     currentlyCheckingRetryQueue ⟵ false return selected
37:                 else
38:                     notExplicitlyEvictedList [selected] ⟵ true
39:                     continue
40:                 end if
41:             end if
42:         end if
43:     end while
44: end function
```

3 Component-Wise Performance Evaluation of an OLTP System

3.1 Introduction

3.2 Single-Threaded OLTP System Analysis

3.2.1 Read-Only YCSB

3.2.2 Write-Only YCSB

3.2.3 Read-Write YCSB

3.2.4 TPC-B

3.2.5 TPC-C

3.3 Multi-Threaded OLTP System Analysis

3.3.1 Read-Only YCSB

3.3.2 Write-Only YCSB

3.3.3 Read-Write YCSB

3.3.4 TPC-B

3.3.5 TPC-C

3.4 Conclusion

Bibliography

[BM70] R. Bayer and E. McCreight. "Organization and Maintenance of Large Ordered Indices". In: *Proceedings of the 1970 ACM SIGFIDET Workshop on Data Description, Access and Control.* SIGFIDET '70. Houston, Texas: Association for Computing Machinery, Nov. 1970, pp. 107–141. ISBN: 9781450379410. DOI: 10.1145/1734663.1734671.

[Bél66] László A. Bélády. "A Study of Replacement Algorithms for Virtual-Storage Computer". In: *IBM Systems Journal* 5 (2 June 1966), pp. 78–101. ISSN: 0018-8670. DOI: 10.1147/sj.52.0078.

[EH84] Wolfgang Effelsberg and Theo Härder. "Principles of Database Buffer Management". In: *ACM Transactions on Database Systems* 9 (4 Dec. 1984). Ed. by Robert W. Taylor, pp. 560–595. ISSN: 0362-5915. DOI: 10.1145/1994.2022.

[GKK12] Goetz Graefe, Hideaki Kimura, and Harumi Kuno. "Foster b-trees". In: *ACM Transactions on Database Systems* 37 (3 Aug. 2012). Ed. by Zehra Meral Özsoyoğlu, 17:1–17:29. ISSN: 0362-5915. DOI: 10.1145/2338626.2338630. URL: http://dl.acm.org/citation.cfm?id=2338630 (visited on Jan. 13, 2016).

[Gra+14] Goetz Graefe et al. "In-Memory Performance for Big Data". In: *Proceedings of the 41st International Conference on Very Large Data Bases.* Vol. 8: *Proceedings of the VLDB Endowment.* The 41st International Conference on Very Large Data Bases (Aug. 31–Sept. 4, 2015). Ed. by Chen Li and Volker Markl. founding H. V. Jagadish. In collab. with Kevin Chang et al. Proceedings of the VLDB Endowment 1. Very Large Data Base Endowment Inc. Kohala Coast, Hawaii, Sept. 2014, pp. 37–48. ISSN: 2150-8097. DOI: 10.14778/2735461.2735465. URL: http://

www.vldb.org/pvldb/vol8/p37-graefe.pdf (visited on Dec. 13, 2016).