Thread-specific Database Buffer Management in Multi-core NVM Storage Environments

Tsuyoshi Ozawa, Yuto Hayamizu, Kazuo Goda, Masaru Kitsuregawa
Institute of Industrial Science, The University of Tokyo, Japan

Abstract

Database buffer management is a cornerstone in modern database management systems (DBMS). So far, a *shared buffer* strategy has been widely employed to improve the cache efficiency and reduce the IO workload. However, it involves a significant processing overhead induced by the inter-thread synchronization, thus failing to exploit the potential bandwidth that recent non-volatile memory (NVM) storage devices offer. This paper proposes to employ a *separated buffer* strategy. According to this strategy, the database buffer manager is allowed to achieve significantly higher throughput, even though it may produce an extra amount of IO workload. In recent multi-core NVM storage environments, separated buffer performs faster in query processing than shared buffer. This paper presents our experimental study with the TPC-H dataset on two different NVM machines, demonstrating that separated buffer achieves up to 1.47 million IOPS and finally performs up to 637% faster in query processing than shared buffer.

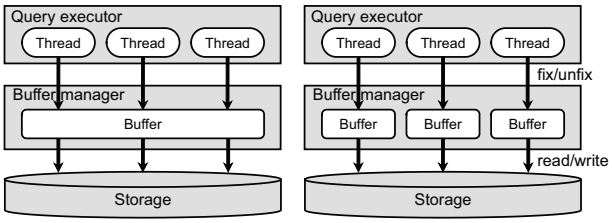
1 Introduction

The emerging non-volatile memory (NVM) has the great potential to accelerate query processing in database management systems (DBMS). Redesigning and optimizing DBMS for NVM is now a key problem.

Aiming at exploiting the potential bandwidth of NVM, this paper proposes a new design strategy for a database buffer manager, a cornerstone component being responsible for handling IOs in DBMS. So far, the mainstream study of a database buffer manager was directed to improving the cache efficiency and optimizing the IO sequence in order to reduce IO time and then improve query processing performance [2, 3, 7]. This strategy was practically acceptable because IOs were approximately six orders of magnitude slower than memory, allowing many processor cycles to be utilized for carefully controlling every IO. Now, due to NVM, the latency gap between IO and memory is narrowing (one to four orders of magnitude). A database buffer manager is only allowed to utilize a much smaller amount of processor cycles for IO control. Actually, a database buffer manager is often a performance bottleneck [8]. This paper proposes a *separated buffer* strategy, which simplifies the buffer management to reduce inter-thread synchronization and improve IO throughput even though it sacrifices the cache efficiency. Our experimental study clarifies that this simplification finally improve query processing performance in recent multi-core NVM storage environments.

2 Database Buffer Management

In modern DBMS, a database buffer manager is an essential abstraction layer that handles all IOs between a query executor and persistent storage. While processing a query, the query processor requests a *fix* of a page when needing the page; upon the request, the database buffer manager reads the page content from the storage and places the page into a buffer (allocated in main memory), so that the query processor can use the page for the query processing [5]. Similarly, the query processor requests an *unfix* when no longer needing the page; the database buffer manager releases the page from the buffer after writing the page content to the storage, if necessary, according to page replacement policies such as LRU, Generalized CLOCK [2], TinyLFU [3] and ARC [7].



(a) Shared buffer (b) Separated buffer

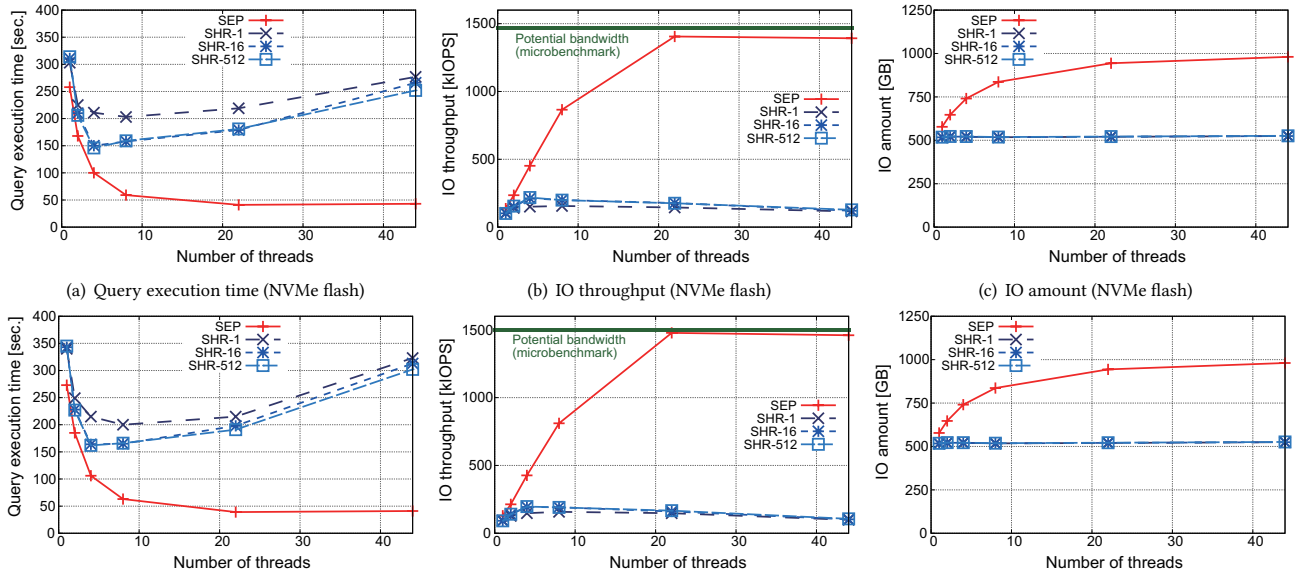
Figure 1: Comparison of database buffer strategies.

Recent DBMS eagerly utilizes multi-threading in order to achieve query execution parallelism [4]. The query processor invokes multiple threads, each of which concurrently fixes and unfixes pages during query processing. Conventionally, as illustrated in Figure 1(a), all the threads share a single buffer in order to improve the cache efficiency. However, this strategy involves mutual exclusion at the buffer for every fix/unfix operation, thus inducing significant processing overhead for inter-thread synchronization [8]. Particularly, in recent multi-core NVM storage environments, the database buffer is likely to be a performance bottleneck [6].

Contrary to the conventional practice, we propose to separate the buffer into thread-specific pieces, as illustrated in Figure 1(b). This strategy removes the necessity of mutual exclusion at the buffer, thus having the great benefit of reducing the inter-thread synchronization and improving the IO throughput significantly. A possible concern is that this strategy may produce an extra amount of IO workload since different threads may invoke IOs for an identical page. The next section experimentally demonstrates that the benefit overcomes the concern; specifically speaking, the separated buffer strategy improves the query performance.

3 Experiment

We experimentally verified the performance benefit of the separated buffer strategy in comparison with the conventional shared buffer strategy by using a multi-threaded database engine [4] with a TPC-H dataset (scale factor: 100) [1]. We implemented four configuration options: (1) separated buffer (SEP), (2) shared buffer with a single giant lock (SHR-1), (3) shared buffer with 16 partitioned locks (SHR-16) and (4) shared buffer with 512 partitioned locks (SHR-512). In all cases, Generalized CLOCK [2] was utilized for buffer replacement. Page size and buffer size were set to 16,384



(a) Query execution time (NVMe flash) (b) IO throughput (NVMe flash) (c) IO amount (NVMe flash)

(d) Query execution time (Optane) (e) IO throughput (Optane) (f) IO amount (Optane)

Figure 2: Separated buffer (SEP) achieves significant speedup (up to 637% faster with 44 threads) over shared buffer (SHR-1, SHR-16 and SHR-512) on both *NVMe flash* (a-c) and *Optane* (d-f) machines for simplified Query 3 on TPC-H dataset.

bytes and 65,536 pages. Query execution was tested with one to 44 threads, each being assigned to a separate processor core. This paper reports a result of the simplified Query 3 (three-way nested-loop index join) as a representative case, since quite similar results were observed for other queries.

We performed the test on two different machines: *NVMe flash* was composed of two Intel Xeon E5-2699 processors (2.2 GHz, 22 cores), 256GB memory and ten Intel DC P3700 devices (800GB), while *Optane* was composed of two Intel Xeon Gold 6152 processors (2.1 GHz, 22 cores), 256GB memory and ten Intel Optane DC X4800 devices (375GB). Storage devices were striped without any parities in both cases.

Figure 2 shows that the separated buffer strategy (SEP) achieves significant speedup over the shared buffer strategy (SHR-1, SHR-16 and SHR-512) on both the NVMe flash and Optane machines. Figure 2(a) and (d) report query execution time. All the test cases (SEP, SHR-1, SHR-16 and SHR-512) performed comparably for a single thread, but separated buffer (SEP) gained more speedup over the other as the multi-threading became more intensive. Finally, separated buffer performed up to 486% faster (reducing 82.9% of execution time) on NVMe flash and 637% faster (reducing 86.4% of execution time) on Optane with 44 threads. Figure 2(b) and (e) indicate that this query speedup was caused by the intensive growth of IO throughput. Separated buffer offered 1.41 million IOPS (95.9% of the potential IO bandwidth) on NVMe flash and 1.47 million IOPS (98.4% of the potential IO bandwidth) on Optane[1]), whereas shared buffer only yielded 0.22 million IOPS on NVMe flash and 0.20 million IOPS on Optane at maximum. In contrast, Figure 2(c) and (f) indicate that the side effect of separated buffer was rather moderate; the extra IO amount was merely limited to 87% increase at maximum. In summary, the experiment verifies that separated buffer speeds up query execution over shared buffer substantially (by 17.4% to 637%) by improving the IO throughput significantly.

[1] According to the micro-benchmark test, *NVMe flash* and *Optane* potentially held 1.46 and 1.50 million IOPS of IO bandwidth respectively.

4 Conclusion

This paper has proposed the employment of the *separated buffer* strategy for database buffer management. With the separated buffer strategy, the database buffer manager is allowed to achieve significantly higher throughput than the conventional *shared buffer* strategy, particularly in recent multi-core NVM storage environments, even though it may produce an extra amount of IO workload. This paper has presented our experimental study with the TPC-H dataset on two different NVM machines and demonstrated that the separated buffer achieves up to 1.47 million IOPS and finally performs up to 637% faster in query processing than the shared buffer. We plan to extend the study to explore lock-free techniques that leverage the state-of-the-art processor capability and evaluate other types of queries such as online transactions.

References

[1] Transaction Processing Performance Council. 2018. TPC-H benchmark specification.
[2] Wolfgang Effelsberg and Theo Haerder. 1984. Principles of Database Buffer Management. *ACM TODS 9, 4*, 560–595.
[3] Gil Einziger, Roy Friedman, and Ben Manes. 2017. TinyLFU: A Highly Efficient Cache Admission Policy. *ACM TOS* 13, 4, Article 35.
[4] Kazuo Goda, Yuto Hayamizu, Hiroyuki Yamada, and Masaru Kitsuregawa. 2020. Out-of-Order Execution of Database Queries. *PVLDB* 13, 12, 3489–3501.
[5] Jim Gray and Andreas Reuter. 1992. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann.
[6] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP through the Looking Glass, and What We Found There. *ACM SIGMOD '08*, 981–992.
[7] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. *USENIX FAST '03*, 115–130.
[8] Makoto Yui, Jun Miyazaki, Shunsuke Uemura, and Hayato Yamana. 2010. Nb-GCLOCK: A non-blocking buffer management based on the generalized CLOCK. *IEEE ICDE 2010*, 745–756.