Department of Computer Science
Database and Information Systems Group

Project Thesis:

Performance Evaluation of Different Open Source and Proprietary Implementations of Data Structures in the context of a DBMS Buffer Manager

by **Max Fabian Gilbert**[*]

Day of release: January 31, 2019

*m_gilbert13@cs.uni-kl.de

Abstract

Needless to say, every database management system needs to be able to manage data. The data structures used to manage those data in a database have a major influence on various characteristics (e.g. performance) of a database management system and therefore, the usage of specific data structures (e.g. B-tree indexes) and even some implementation details of those are very important decision in DBMS design.

But for correct and performant operation, a DBMS needs to manage various kinds of meta data as well. Some of those meta data needs to be persistent (e.g. the catalog of a relational DBMS) but some can also be non-persistent. Because of the non-persistence of data managed by the buffer management of a DB, the meta data required for the buffer manager are also usually non-persistent. The data structures used to manage those meta data are—unlike the data structures used to manage the data—more an implementation than a design decision. For some kinds of those meta data, it's—due to the non-criticality of the specific meta data management—even reasonable to use data structures provided by the used programming language even though there might be more performant data structures for the purpose. But more performant implementations for most of those data structures don't need to be implemented specifically for one project, there are many different implementations available in open source and proprietary libraries.

This work is a performance evaluation of various MPMC

Contents

This page intentionally left blank.

1 Buffer Frame Free List

1.1 Purpose

A buffer manager is required for every disk-based DBMS. A disk-based DBMS stores the pages of a database on secondary storage but to read and write pages, they are required to be in memory.

This feature is provided by the buffer pool management by managing the currently used subset of the database pages in buffer frames located in memory. A buffer frame is a portion of memory that can hold one database page and each of those frames got a frame index as identifier.

During operation, database pages are dynamically fetched from the database into buffer frames. Once a page is not required anymore, it might be evicted from the buffer pool freeing a buffer frame.

Due to the fact that pages are only allowed to be fetched into free buffer frames, the buffer manager needs to know all the free buffer frames. Therefore, a free list for the buffer frames is required.

1.2 Compared Queue Implementations

To ease implementation of page eviction strategies like CLOCK, a free list should use a FIFO data structure like a queue. Therefore the buffer frame freed first is (re-)used first as well.

Almost every state-of-the-art DBMS support multithreading and therefore, there are usually multiple threads concurrently fetching pages into the buffer pool and evicting pages from the buffer pool. Following this, a buffer frame free list has to support thread-safe functions to push frame indexes to the free list and to pop frame indexes from it. Queues providing those thread-safe access functions are usually called multi-producer (add frame indexes) multi-consumer (retrieve/remove frame indexes) queues

(**MPMC** queues).

An approximate number of buffer indexes in the free list must also be provided by any free list implementation to support the eviction of pages once there are only a few free buffer frames left. Thread-safe access to this number is desirable but not absolutely required.

1.2.1 Boost Lock-Free Queue with variable size

The famous *Boost C++ Libraries*[1] offer a lock-free MPMC queue[2] in the library `Boost.Lockfree`[3]. Like most other non-blocking data structures, this MPMC queue uses atomic operations instead of locks or mutexes. To support queues of dynamically changing sizes, this queue implementation also uses a free list internally.

The non-thread-safe construction/destruction of the data structure is no limitation for the purpose as a buffer frame free list because the buffer pool of our prototype system is constructed single-threaded. This data structure does not offer the number of contained elements and therefore, an approximate number of buffer indexes the free list needs to be managed outside.

1.2.2 Boost Lock-Free Queue with fixed size

This data structure is identical to the data structure in Subsection 1.2.1 but does not use dynamic memory management internally. Therefore, the capacity of the queue (i.e. the maximum number of buffer frames of the buffer pool) needs to be specified beforehand which allows the usage of a fixed-size array instead of dynamically allocated nodes.

1.2.3 CDS Basket Lock-Free Queue

Besides other concurrent data structures, the *Concurrent Data Structures* C++ library[4] offers many different thread-safe queue implementations. The

[1] https://www.boost.org/
[2] https://www.boost.org/doc/libs/release/doc/html/boost/lockfree/queue.html
[3] https://www.boost.org/doc/libs/release/doc/html/lockfree.html
[4] https://github.com/khizmax/libcds

Basket Lock-Free Queue[5] is based on the algorithm proposed by M. Hoffman, O. Shalev and N. Shavit in [HSS07].

Internally, the Basket Lock-Free Queue does not use an absolute FIFO order. Instead, it puts concurrently enqueued elements into one "basket" of elements. The elements within one basket are not specifically ordered but the different "baskets" used over time are ordered according to FIFO. Therefore, the dequeue operation just dequeues one of the elements in the oldest 'basket".

1.2.4 CDS Flat-Combining Lock-Free Queue

The *Concurrent Data Structures* C++ library does also offer a lock-free queue that uses Flat Combining[6]. The Flat Combining technique was proposed by D. Hendler, I. Incze, N. Shavit and M. Tzafrir in [Hen+10]. This technique is used to make any sequential data structure thread-safe—in case of the *Flat-Combining Lock-Free Queue*, the `std::queue`[7] of the *C++ Standard Library*[8] is used as base data structure.

The Flat Combining technique uses thread-local publication lists to record operations performed by those threads. A global lock is needed to be acquired to combine these thread-local publication lists into the global, sequential data structure. The thread which acquired the global lock also combines the publication lists of all other threads reducing the locking overhead. The returned value of each operation executed during the combining is stored into the respective publication list together with the global combining pass number. A thread with a non-empty publication list that cannot acquire the global lock needs to wait till the combining thread updated its publication list.

1.2.5 CDS Michael & Scott Lock-Free Queue

Another lock-free queue implementation offered by the *Concurrent Data Structures* C++ library is based on the famous Michael & Scott lock-free

[5] http://libcds.sourceforge.net/doc/cds-api/classcds_1_1container_1_1_basket_queue.html
[6] http://libcds.sourceforge.net/doc/cds-api/classcds_1_1container_1_1_f_c_queue.html
[7] https://en.cppreference.com/w/cpp/container/queue
[8] https://en.cppreference.com/w/cpp

queue algorithm[9] which was proposed by M. Michael and M. Scott in [MS96].

The Michael & Scott lock-free queue basically uses compare-and-swap (**CAS**) operations on the tail of the queue to synchronize enqueue operations. If a thread reads a NULL value as next element after the queue's tail, it swaps this value atomically with the value enqueued by this thread. Afterwards it adjusts the tail pointer. If a thread does not read the NULL value there during the CAS operation, another thread has not already adjusted the tail pointer and this thread needs to retry its enqueue operation with the new tail pointer. The dequeue operation is implemented similarly.

1.2.6 CDS Variation of Michael & Scott Lock-Free Queue

The *Concurrent Data Structures* C++ library also offers an optimized variation of the Michael & Scott lock-free queue algorithm[10] which is based on the works of S. Doherty, L. Groves, V. Luchangco and M. Moir in [Doh+04].

This optimization of the Michael & Scott lock-free queue optimizes the dequeue operation to only read the tail pointer once.

1.2.7 CDS Michael & Scott Blocking Queue with Fine-Grained Locking

M. Michael and M. Scott did also propose a blocking queue algorithm in [MS96]. This blocking queue implementation[11] is also offered by the *Concurrent Data Structures* C++ library.

This blocking queue algorithm uses one read and one write lock protecting the head and tail of the queue. Therefore, only one thread a time can enqueue and only one thread at a time can dequeue elements.

1.2.8 CDS Ladan-Mozes & Shavit Optimistic Queue

The *Concurrent Data Structures* C++ library also offers an optimistic queue implementation[12] which is based on an algorithm proposed by E. Ladan-

[9] http://libcds.sourceforge.net/doc/cds-api/classcds_1_1container_1_1_m_s_queue.html
[10] http://libcds.sourceforge.net/doc/cds-api/classcds_1_1container_1_1_moir_queue.html
[11] http://libcds.sourceforge.net/doc/cds-api/classcds_1_1container_1_1_r_w_queue.html
[12] http://libcds.sourceforge.net/doc/cds-api/classcds_1_1container_1_1_optimistic_queue.html

Mozes and N. Shavit in [LS04].

Instead of using expensive CAS operations on a singly-linked list (like in the Michael & Scott lock-free queue), this algorithm uses a doubly-linked list with the possibility to detect and fix inconsistent enqueue and dequeue operations.

1.2.9 CDS Segmented Queue

1.2.10 CDS Vyukov's MPMC Bounded Queue

1.2.11 Folly MPMC Queue

1.2.12 Gavin Lambert's MPMC Bounded Lock-Free Queue

1.2.13 `moodycamel::ConcurrentQueue`

1.2.14 Dmitry Vyukov's Bounded MPMC Queue

1.2.15 Variation of Dmitry Vyukov's Bounded MPMC Queue

1.2.16 Erik Rigtorp's MPMC Queue

1.2.17 Intel® Threading Building Blocks Bounded Concurrent Dual Queue

1.2.18 Intel® Threading Building Blocks Concurrent Queue

1.3 Performance Evaluation

Bibliography

[Doh+04] Simon Doherty et al. “Formal Verification of a Practical Lock-Free Queue Algorithm”. In: *Formal Techniques for Networked and Distributed Systems – FORTE 2004.* Lecture Notes in Computer Science (3235 2004): *24th IFIP WG 6.1 International Conference, Madrid Spain, September 27-30, 2004 Proceedings.* Ed. by David de Frutos-Escrig and Manuel Núñez. Found. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen, pp. 97–114. DOI: 10.1007/978-3-540-30232-2_7.

[Hen+10] Danny Hendler et al. “Flat Combining and the Synchronization-Parallelism Tradeoff”. In: (2010): *SPAA ’10: Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures.* Ed. by Friedhelm Meyer auf der Heide and Cynthia Phillips, pp. 355–364. DOI: 10.1145/1810479.1810540. URL: https://www.cs.bgu.ac.il/~hendlerd/papers/flat-combining.pdf (visited on Jan. 24, 2019).

[HSS07] Moshe Hoffman, Ori Shalev, and Nir Shavit. “The Baskets Queue”. In: *Principles of Distributed Systems.* Lecture Notes in Computer Science (4878 2007): *11th International Conference, OPODIS 2007 Guadeloupe, French West Indies, December 17-20, 2007 Proceedings.* Ed. by Eduardo Tovar, Philippas Tsigas, and Hacène Fouchal. Found. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen, pp. 401–414. DOI: 10.1007/978-3-540-77096-1_29. URL: https://people.csail.mit.edu/shanir/publications/Baskets%20Queue.pdf (visited on Jan. 24, 2019).

[LS04] Edya Ladan-Mozes and Nir Shavit. “An Optimistic Approach to Lock-Free FIFO Queues”. In: *Distributed Computing.* Lecture Notes in Computer Science (3274 2004): *18th International Con-*

ference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004 Proceedings. Ed. by Rachid Guerraoui. Found. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen, pp. 117–131. DOI: 10.1007/978-3-540-30186-8_9. URL: http://people.csail.mit.edu/shanir/publications/FIFO_Queues.pdf (visited on Jan. 24, 2019).

[MS96] Maged M. Michael and Michael L. Scott. "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms". In: (1996): *PODC '96: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing.* Ed. by James E. Burns and Yoram Moses, pp. 267–275. DOI: 10.1145/248052.24810629. URL: http://www.cs.rochester.edu/~scott/papers/1996_PODC_queues.pdf (visited on Jan. 24, 2019).