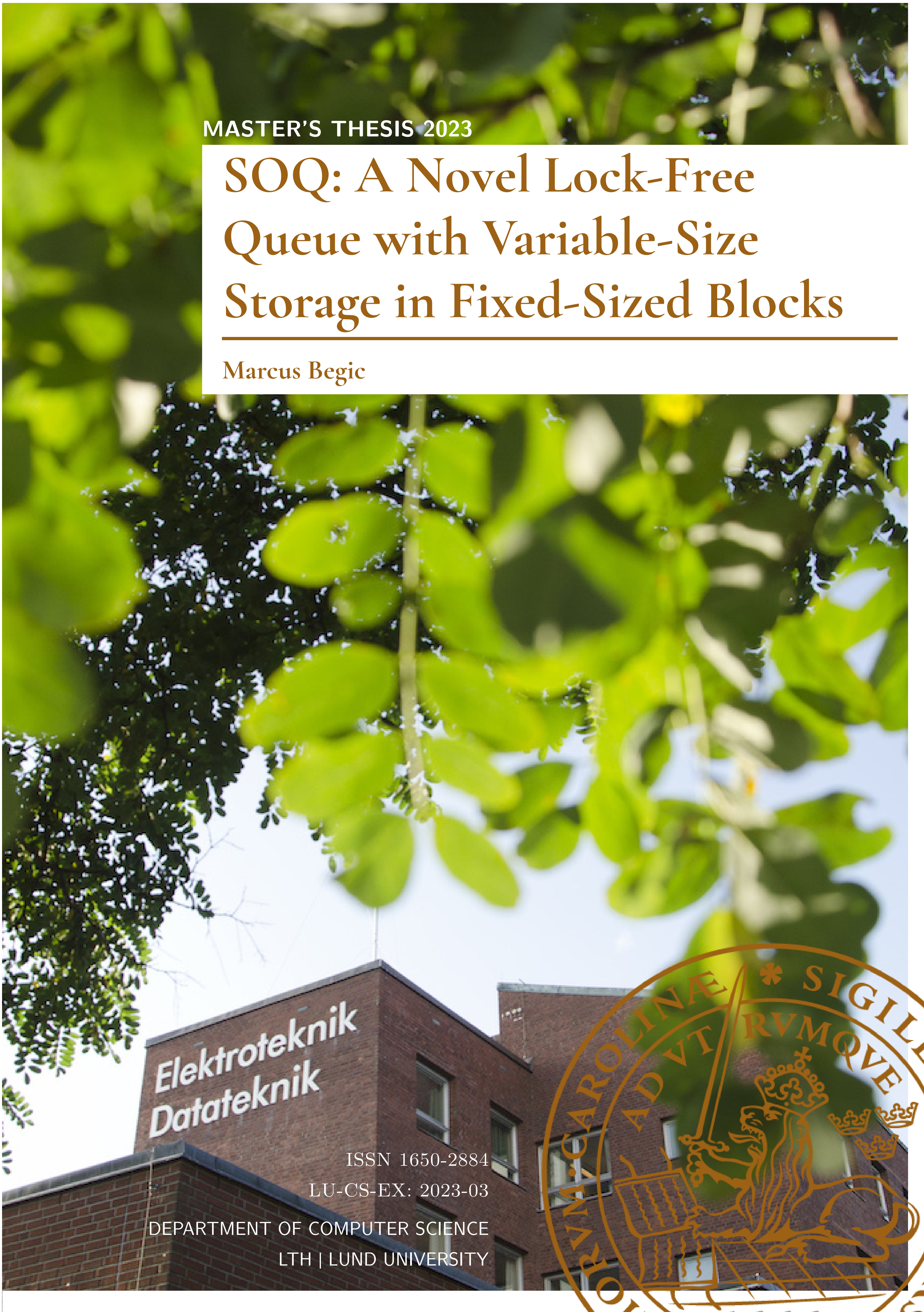


MASTER'S THESIS 2023

SOQ: A Novel Lock-Free Queue with Variable-Size Storage in Fixed-Sized Blocks

Marcus Begic

ISSN 1650-2884
LU-CS-EX: 2023-03

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2023-03

SOQ: A Novel Lock-Free Queue with Variable-Size Storage in Fixed-Sized Blocks

Marcus Begic

SOQ: A Novel Lock-Free Queue with Variable-Size Storage in Fixed-Sized Blocks

(SOQ: En ny låsfri kö som kan dynamiskt lagra objekt av varierande storlek i statiska minnesblock.)

Marcus Begic
`ma6373be-s@student.lth.se`

February 18, 2023

Master's thesis work carried out at
Huawei Research Center, Edinburgh.

Supervisors: Nikos Ntarmos, `nikos.ntarmos@huawei.com`
Jonas Skeppstedt, `jonas.skeppstedt@cs.lth.se`

Examiner: Per Andersson, `per.andersson@cs.lth.se`

Abstract

Databases concurrently store many versions of the same data in a growing version chain. A version chain can be represented as a concurrent FIFO queue of versions. In order to save memory when storing versions in the queue, one can place the restriction of every memory allocation being of the same size. These fixed-sized memory blocks avoid fragmentation in the memory of the system, as they can be stored consecutively. By also ensuring lock-freedom of the queue, we get a way to store versions concurrently, efficiently, while also saving lots of memory. This thesis introduces the first such data structure, known as Spillover Queue. It extends an earlier lock-free queue algorithm and introduces some ideas that allows us to, for example, enqueue multiple nodes concurrently without blocking.

Keywords: Concurrent Data Structures, Lock-Free, Version Chains, MVCC

Acknowledgements

I want to thank my supervisor Dr. Nikos Ntarmos for taking me on and giving me a fascinating problem to work on. Nikos is never afraid to discuss difficult problems in detail with a hands-on approach that only exists in people who are both great engineers and great managers. I would also like to thank him for helping me keep my composure during stressful times.

I would like to thank my supervisor in Lund, Dr. Jonas Skeppstedt, whose coursework and teaching over the years have taken me from learning fundamentally about data structures and algorithms, all the way up to the lock-free queues that this work is based on. Jonas's courses were always challenging and fun, and his passion for computer science rubs off on many students, including me.

Lastly, I would like to thank my Huawei co-workers, who are real experts in their field and would constantly take the time to teach me about database systems, concurrency, and most importantly, greek culture.

Contents

Chapter 1
Introduction

High-performance databases often require sophisticated concurrent data structures for storage of data. Some databases require the same high level of parallelism and performance but without the availability of mutexes. In these cases, efficient lock-free data structures are in high demand. A limitation not often put on highly-concurrent systems is that of memory. Memory restrictions force engineers need careful with how memory is used and require an understanding of underlying architectural principles. When such memory requirements are combined with concurrency and performance, lock-free data structures can become even trickier to create.

Databases control concurrency by storing many versions of their data at the same time, these versions are stored in a version chain that grows concurrently on one end as new versions are added in a FIFO queue pattern. This thesis introduces Spillover Queue, a lock-free Queue made to store versions of any size. It does this concurrently while only being able to allocate fixed-sized blocks of memory. This allows it to reduce fragmentation in the memory of the system as these blocks are all stored consecutively in memory. Spillover Queue achieves this by introducing some novel concepts and abstractions which extend the classic lock-free Micheal&Scott Queue [1].

Specifically, Spillover Queue manages to efficiently and concurrently store versions while the design obeys the following requirements:

- It needs to store versions of any size.
- Only memory blocks of fixed size can be allocated.
- Memory blocks have to be used in their entirety.
- The data structure needs to be lock-free.

Being able to store any size in fixed-sized memory blocks means that the queue might want to *spillover* data from one block to another. This data might even spillover and span many blocks. This means Spillover Queue needs to solve some sub-problems like how a thread should enqueue many nodes continuously after each other without blocking other threads. As well as the problem of describing and managing how much memory is used up in a block. These problems are trivially solved with mutexes but can be very difficult to do in a lock-free manner. Spillover Queue shows that it's possible to do all this with only lock-free atomic operations, resulting in a new data structure that is both memory-conservative and efficient.

The first chapter gives an overall background to this thesis. The background includes a section about multi-core programming in general and then it discusses the background knowledge required to understand the database system that Spillover Queue is a part of, finally it explains some related works. The next chapter more carefully describes the goal and the requirements of the data structure and this thesis, which then allows us to, in the following chapter, justify the final design of Spillover Queue while looking at some unsuitable potential designs. We then give an overview of the algorithm along with some interesting aspects of it, and finally, we present and discuss the results in the Analysis and Conclusion chapters.

Chapter 2

Background

Here we describe the background knowledge required to understand the novel contribution of the thesis. The background first touches on concurrent programming in a broad sense and then goes into certain primitives and concurrency ideas pertaining to specifically lock-free data structures. Further, we explain the background required to understand the database system that Spillover Queue contributes to, and finally, it describes the algorithms that Spillover Queue is based on in detail in the Related Works section.

2.1 Concurrent Programming

A concurrent program is one in which several computations occur asynchronously to each other. Researchers have been intensively studying concurrent programs for decades since Djisktra introduced the mutual exclusion problem[2]. Since then the industry has caught up and concurrency is a part of nearly every system and supported on nearly every machine. Thus, processors today often have multiple cores each of which can support at least one thread of parallel execution. These threads share a memory which they can read and write from, allowing them to collaborate over the same space which can improve the performance of a program. In an ideal concurrent program, each thread works on its own sub-part of the data entirely sequentially, such a program would be exactly like a single-threaded sequential one. However, in most cases, concurrent programs require some sort of collaboration and synchronization to be correct and efficient. This frequently happens over shared data structures also known as concurrent data structures. It is often hard to reason about concurrent programs for this reason because it's not enough to consider the sequential ordering of operations one thread might have, but also all the potential inter-leavings of reads and writes that multiple threads can have on the shared memory.

Multi-core Architecture

Within a processor, each core has its own registers and cache. These are not shared between threads, but memory is. Therefore it is important that what is written to the caches is coherent between all cores. Otherwise, a cache could update its own local copy of a variable while every other core sees an older version, this is known as the cache coherency problem. Modern CPUs avoid this using a hardware-implemented cache coherence algorithm which will invalidate out-of-date memory in other cores' caches when one core updates its own version [3]. Cache invalidation can cost some CPU cycles because of this, so to write optimized code one should avoid it. When using registers however, a multicore programmer doesn't have the cache coherency protocol to ensure that all cores see the same version, so an issue can arise when one thread has stored a variable in it's register it doesn't see other threads updates to it.

Memory Hierarchy

Threads will share the same view of *virtual memory*. Virtual memory is an abstraction where memory seems to be completely contiguous in its layout to threads and processes. However, on the kernel level, the memory could physically be in various parts of the disk or in RAM. The Operating System ensures that a process and its threads see contiguous memory by maintaining a mapping of these virtual addresses to the physical addresses. This mapping points to actual physically contiguous blocks of memory called *pages*. A *page fault* occurs when the process requests data that belongs to a page that isn't currently in RAM, this means the kernel needs to fetch it from disk, which can cost thousands of cycles[3]. Caches also experience a less costly but equally important problem, when a processor can't find certain data in its cache, it needs to fetch it from memory, this is known as a *cache miss*. To avoid page faults and cache misses, *locality* should be considered. Spatial locality is operations using data consecutively, where that data can be found consecutively in memory. Processors like locality on the cache-line level and operating systems like locality on the page level. When a processor fetches a cache-line from main memory, it stores the adjacent data in the cache as well, so if the next access is to an adjacent element, one cache miss can be saved. The same thing is true within virtual memory and paging where it is even more important as page faults are even more costly. Therefore, when designing performant data structures it is important to consider the locality of the data. A ground rule is to avoid pointer dereferences were possible, as these can point to virtually anywhere on the heap often generating a cache miss during which the processor is idling for hundreds of cycles, or even a page fault.

Blocking vs. Non-Blocking Data Structures

A concurrent data structure can either be blocking or non-blocking. The most common methods of synchronization are usually blocking, eg. mutexes and semaphores which prevent a thread from accessing a shared state. This insures safety by mutual exclusion. Blocking data structures include operations that can force one thread to wait for an operation by another thread to complete, This can introduce a plethora of problems, including deadlock, livelock, and priority inversion. A non-blocking algorithm is one where one thread cannot prevent the execution of another, they can either be wait-free or lock-free. Lock-freedom is the guarantee that at any given time the program as a whole, or at least one thread, is making progress

[4]. Wait-freedom is a stronger guarantee where at any given time every thread is making progress. Non-blocking algorithms have the additional benefit of being able to avoid the use of mutex locks which have further downsides in often require involving the kernel with expensive system calls. Non-blocking data structures are often used in performance-critical bottlenecks of systems with high concurrency, but due to the fact that they only rely on atomic instructions, they are often much more difficult to implement.

Atomic Instructions

Atomic instructions are low-level computer programming operations that are indivisible. This means that they are executed as a single unit and cannot be interrupted or broken down into smaller pieces. Atomic instructions are used to ensure the consistency and integrity of data by preventing other threads from accessing or modifying that data while the atomic instruction is being executed. This is critical in situations where multiple threads may be trying to access the same memory location at the same time, as it ensures that the shared data remains consistent and correct. Atomic variables are commonly used in concurrent programming to ensure the correct operation of shared data structures, and in lock-free systems, they are the only synchronization tool at a programmer's disposal.

Internally, they are typically implemented using special hardware support, such as atomic registers, to ensure that they can be executed quickly and efficiently. These instructions utilize the cache coherency protocol to ensure that CPU cores and threads maintain exclusive access to reads and updates of atomic variables. They are however not completely cost-less, an atomic operation often requires invalidating the caches of other cores, which again costs cycles [5].

Compare And Swap

Many lock-free data structures use the atomic low-level synchronization primitive `compare_and_swap` also known as CAS. Fundamentally, a CAS will write to a (shared) memory location, if the location has the expected value. A general pattern is as follows: a thread observes a shared state, performs some computation under the assumption that the state hasn't changed, and uses CAS to update said shared state atomically if and only if it hasn't changed. A CAS will *fail* when the shared memory location has changed. Take the pseudo-code in Algorithm 1 for example.

Algorithm 1 Compare And Swap

```
1: function CAS($p$: shared data, $old$, $new$ )
2:     if $p = old$ then
3:         $p \leftarrow new$
4:         return $true$
5:
6:     return $false$
```

Here the pointer ***p*** is some shared state, when ***p*** was read the value at ***p*** was ***old***, some computation happened and if the value at ***p*** is still ***old*** then update the ***p*** with ***new***, else return ***false***.

A CAS *fails* and returns ***false*** if ***p*** does <u>not</u> point to the same ***old*** location. This is why a lot of CAS-based lock-free algorithms require that the CAS is executed within a loop. Essentially CAS failure occurs when multiple threads all try to change the same single point of contention. One can therefore improve the performance of a lock-free CAS-based algorithm by trying to reduce the contention at these points. As we will see this is what many optimizations of CAS-based algorithms do.

CAS is supported on the hardware level by most architectures, this is why it can ensure atomicity and be efficient. Single-worded CAS or 'unary CAS', is commonly supported on most architectures, ARM, x86, and POWER, etc. 'Single-worded' comes from the fact that the parameters are size 1 word or 8 bytes (32 bits). This means 8 bytes can be compared and updated atomically. There are more powerful but much less supported 'exotic' CAS operations. One exotic is double-worded CAS known as DCAS or CAS2, this can load values into two word-sized parameters atomically, compare them and update them atomically. These two words don't have to be contiguous in memory, they can be two separate pointers stored in different locations for example. DCAS is very useful as there are simple algorithms to implement eg. lock-free deques[6]. However, DCAS is not supported by any of the mainstream architectures.

A less powerful exotic is double-width CAS or 128-bit CAS known as DWCAS. This is essentially unary CAS but on a 128-bit target rather than the standard 64-bit, where the 128 bits are contiguous in memory. So N-word CAS can be seen as doing multiple CAS' atomically, so being able to store values at multiple different locations, while DWCAS can be seen as CAS on a larger target. DWCAS is slightly more commonly supported, availability on an x86 system can be checked by looking for the operation denoted `CMPXCHG16B` or `cx16` in `/proc/cpuinfo`.

ABA Problem

A common and dangerous problem that concurrent programs can have is known as the ABA problem. ABA problems are very prevalent in CAS-based lock-free algorithms. They occur due to a fundamental limitation in CAS, just because ***p*** still points to the same address as it was when it was read, doesn't mean the data structure hasn't changed at all. The pointer ***p*** might have been updated to something new and then updated back to the same address, between the first read of ***p*** and the CAS.

For example, in a CAS loop, a lock-free algorithm might determine whether the top of a lock-free stack hasn't changed by seeing that the pointer to the top element points to the same address as it was at the start of the function. This is done by a CAS on the top of the stack. However, one thread might have accessed the stack between the two reads, popped a node from the top of the stack, allocated a new one, and pushed it on the stack. There is a chance that this new node is stored in the same location as the previous one. In this case, the CAS would be successful even though the data structure has clearly changed. In fact, due to MRU (Most Recently Used) memory allocation algorithms, it is common for allocators to reuse memory locations, which makes the ABA problem even more prominent. ABA often leads to dangerous, extremely hard-to-find bugs.

Luckily, there are several techniques for dealing with the ABA problem, the most common one is the counter approach where every atomic pointer is updated along with an atomic counter, only if **both** are the same. In the stack example, the CAS would fail because the first time the top is referenced the pointer is ***p*** counter is of value eg. 1, and the second time in the CAS the pointer is still ***p*** but the counter is >1. This technique however requires the use of some exotic CAS because both the counter and the pointer need to be compared and updated atomically, seeing as the pointer is 64 bits, unary CAS is not sufficient.

2.2 Database Systems and Concurrency

One massive research area and use-case of concurrency is in systems, operating systems, databases, and distributed systems all massively rely on concurrency and multicore programming. Databases especially deal with massive concurrency. Multiple clients will most likely access the same database at the same time, and since this workload can increase exponentially with more clients and accesses, a database becomes a very concurrent structure and often an important bottleneck in designing large-scale systems.

2.2.1 Transactions and Concurrency Control

In databases, a singular unit of computational work is known as a *transaction*. A transaction can contain multiple operations, for example, it could read the balance from one user's account bank account, check if the result is greater than some value, and if so, deduct the account and add the amount to another user's account. If everything contained in a transaction occurs and is reflected in the state of the database, the transaction is *committed*. One key property of transactions is that they must be atomic, meaning that either the database must reflect that the entire transaction has been committed or none of it (meaning the transaction is aborted). Otherwise, an intermediary and incorrect state could be seen, if a reader saw the state after the deduction of the first account but before the addition to the other, it could appear as though the money had disappeared.

MVCC and Version Chains

To ensure readers always see a database in a valid state, *Multiversion Concurrency Control* (MVCC) is often used. MVCC does this by keeping many copies of the data. A reader will always see valid *version* of the data known as a *snapshot*, which is a valid state at some point in time[7]. A writer trying to modify the data would modify its own version and only make it visible to readers once it's committed. Thus, in MVCC, a transaction is not visible until it has been committed at which point it is considered to be a new *version*. Internally, the versions are just copies of the *rows* of the database as they were after the transaction was committed. These versions need to be stored to ensure readers a valid state to read from. They are stored consecutively in what's known as a *version chain*. Version chains grow in one direction as transactions with new versions are committed and get granted a timestamp in ascending order. The timestamp can be used by the system to determine if a version is *outdated*, this means no readers will attempt to read the data again, and the version can be safely removed. Since transactions are committed concurrently, and readers read the version chain concurrently, the version chain becomes a highly **concurrent data structure**.

2.2.2 Small Memory Environments

Known databases are often large systems built for powerful machines with large memory and disk, eg. PostgreSQL, Neo4j, and MongoDB. However, all systems with interactions in the forms of reads and writes to data in memory, need some form of interface and management in the shape of a database. These systems range from data-center infrastructure to small OS' running on mobile devices. The requirements, access patterns, and loads vary greatly between these systems. A data center is often distributed, using machines with 16 cores and multiple instances of the database for redundancy, while on a mobile device, the system could have a small chip and a couple of gigabytes of memory while still requiring high performance. The methodologies in building these databases vary greatly, when building a general-purpose large database that will run on a powerful machine, a programmer probably doesn't have to concern themselves with running out of memory and can spawn a large number of threads. However, when building a database that runs in a small embedded device, memory becomes a high priority. The most common trade-off in building high-performance systems is the time-memory trade-off, for more performance programmers need to use more memory and vice-versa. Small memory databases have to be very frugal with their memory **and** still deliver high performance. This means concern for how much memory algorithms allocate, how data is stored and laid out, the underlying architecture, awareness of page sizes and cache-lines, etc.

Memory Fragmentation

Within computer memory, a storage problem known as *fragmentation* often arises. This is any situation where the capacity or performance or both of storage is reduced due to the layout of contiguous blocks of memory not completely taking up the available space in memory, leaving unused gaps or fragments. In the beginning of the runtime of a program, memory is long and contiguous, over time however blocks of memory are used and freed of varying sizes, leaving less usable fragments. There are two kinds of fragmentation in general, internal and external fragmentation.

External fragmentation occurs when what is originally long and contiguous memory, is broken up into smaller blocks as it is used and freed. This results in free space being available but unusable due to it being divided into pieces individually too small to satisfy the demand.

Internal fragmentation occurs when the granularity of memory allocations is greater than the size a program wants to use. If a program wants to use 1000 bytes it will request a whole page but the page size is 4096 bytes, so 3096 will be left unused.

Memory Pools

There exist special memory allocators to reduce fragmentation. These allocators, known as *memory pools*, pre-allocate a set of *fixed-sized* contiguous memory blocks. While a standard dynamic memory allocator like `malloc` can allocate any variable size blocks upon request of the program, a memory pool allocator on the other hand can *only allocate blocks of a single size*. This places restrictions on the programmer but is otherwise an example of both the performance **and** memory-use of a program being improved. Performance is improved because a memory pool allocator knows the $\boldsymbol{Block_{size}}$ in advance, and thus knows that every allocation will be of this size. This allows it to pre-allocate blocks of $\boldsymbol{Block_{size}}$ and when the program then requests a block, a reference to a pre-allocated block can be handed out quickly. Meanwhile, `malloc` needs to use advanced algorithms to find available space upon request. Memory usage is improved because when blocks are all of the same sizes, they can be packed consecutively in memory, leaving no gaps in between. This perfectly uses available memory removing any fragmentation.

2.3 Related Work

Here we first describe the M&S Queue and its background at a high level, Spillover Queue is heavily based on the ENQUEUE method, so we explain that in greater detail.

2.3.1 MS Queue

One of the most fundamental data structures is the linked-list implemented FIFO queue. Researchers have known of non-blocking implementations of this structure since Maged M. Micheal et al. [1] introduced their classic lock-free M&S Queue (MSQ) in 1996. MSQ uses a CAS-loop-based algorithm where they borrow a synchronization technique from the lock-free Treiber stack [8], namely, *concurrent assistance*. If a thread views the shared queue in an in-between state (another thread has modified the queue but isn't finished), the thread will *assist* the other working thread by correcting the queue. This is a big strength of lock-free data structures, if the queue were blocking on the other hand, the thread would instead idly wait for the others to finish.

MS Enqueue

Enqueues are done at the Tail of the Queue. The algorithm starts by reading the current Tail of the Queue, both the Tail pointer and the Head pointer are atomic pointers. So a thread will *atomically load* the Tail pointer and receive a regular pointer to the Tail node, which it saves a local copy of as seen on line 6 in Algorithm 2. The *next* fields in each node are also atomic pointers, the Tail's next pointer is also read and saved on line 7.

In this thesis, we refer to the thread saving a view of the data structure at a point-in-time in this way, as the *snapshot*. The algorithm then simply tries to execute its next steps in linking a node assuming that the snapshot doesn't change.

It uses CAS to ensure that an actual update to the data structure only occurs if and only if the snapshot has not changed (so the Tail hasn't moved or the next field is still null). The algorithm ensures in line 8. that the Tail in the shared atomic variable is the same as at the snapshot, if not it means that another thread has moved the Tail and there's a more updated version of the data structure to see, so it loops again and retries. The same goes for the next field being null.

Once the Tail is consistent (at least momentarily) with the snapshot, line 9 will check whether the next field of the Tail is still `null`, if not it attempts to correct the data structure by updating the Tail to the `Tail->next`.

This is the concurrent assistance in MSQ, when thread T1 wants to append its node it first attempts to update the next field of the current Tail, and then it tries to advance the Tail pointer to its node, see the two steps in 2.1. If another thread T2 enters trying to enqueue its own node in between these two steps, it will observe in line 9. that the next field isn't `null`, someone has appended a node but not had time to move the Tail pointer yet. It knows where T1 wanted to move the Tail pointer, it's the node that just got appended by T1, the node linked to the Tail node, `Tail->next`, so T2 can then comfortably attempt to swing the Tail pointer to `Tail->next` with its own CAS on line 14. Whether this CAS fails or not, doesn't change the continued behavior of the algorithm because if it failed that means the Tail was moved by another thread already (maybe T1 just managed to move it itself). So some thread has succeeded in correcting the data structure, so T2 will loop and try again. A thread that links its own new node does so by modifying this next field with a CAS on line 10. This CAS ensures that the next field is still null and that no other node was linked. Once it has successfully linked its own new node, the loop can be exited and a single CAS to update the Tail is attempted on line 14, again only a single CAS because another thread might have assisted in moving it.

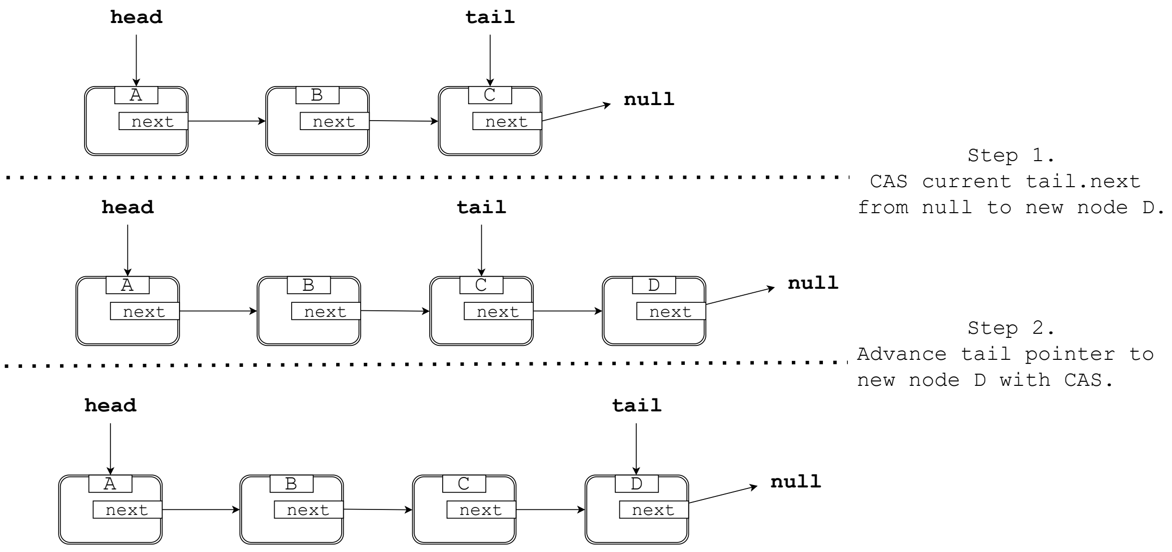


Figure 2.1: Two main steps of M&S Enqueue

Algorithm 2 Enqueue in MSQ

```
1:  function ENQUEUE(Q: pointer to queue t, value: data type)
2:      node = new_node()                                        ▷ Allocate a new node
3:      node->value = value                                      ▷ Copy enqueued value into node
4:      node->next.ptr = NULL                                    ▷ Set next pointer of node
5:      loop
6:          tail = Q–>Tail                                       ▷ Read Tail.ptr and Tail.count together
7:          next = tail–>next                                    ▷ Read next ptr and count fields together
8:          if tail == Q–>Tail then
9:              if next== NULL then
10:                 if CAS(&tail->next, next, new_node) then     ▷ Attempt to link new_node
11:                     break                                    ▷ new_node has been attached to Tail.
12:             else
13:                 CAS(Q–>Tail, tail, next)                     ▷ Try to swing Tail
14:     CAS(Q–>Tail, tail, next)                                 ▷ Try to swing Tail
```

2.3.2 Improvents to MSQ

Here we discuss some further industry-leading lock-free queues and explain the techniques that were made to improve their performance.

Nir's Queue

In 2004 Nir Shavit et al. [4] introduced a lock-free queue which turned the linked list into a doubly-linked list. This allowed any enqueueing thread to optimistically just perform a single CAS rather than in a loop as in MSQ. Other threads would then use the prev pointers to correct the data structure. They managed to consistently outperform MSQ for higher processor core counts by 50%. This confirmed that the real bottleneck in MSQ is CAS failures due to contention. More threads trying to CAS on the same atomic variable results in more CAS failures which greatly reduces performance.

LCRQ

Later in 2013, Adam Morrison and Yehuda Afek vastly outperformed competing lock-free queues with the introduction of LCRQ[9]. LCRQ is essentially an MSQ where each node is a circular array with the elements in the queue, rather than each node containing an element directly. If the circular array happened to fill up entirely, a new one would be appended using a modified version of the MSQ algorithm. This improved locality, two consecutive elements in the queue were likely to be consecutive in memory. It also vastly reduced the number of calls to `malloc`, since now, new heap allocated memory was only requested every time a circular array filled up, rather than once per element in a traditional MSQ. Using another atomic primitive, namely, `fetch-and-add` or F&A, they managed to reduce contention drastically by creating contended F&A objects that spread threads among elements in the queue. This allowed threads to enqueue and dequeue elements in parallel as they would be more likely to not contend for the same element and same atomic variable. The reason this turned out to be a very efficient approach is due to the fact that F&A is a wait-free operation so it *always succeeds*, although it is far weaker than CAS as all it does is fetch the current value of a variable and increment it by some value. These ideas allowed LCRQ to outperform competing data structures by roughly 2× at all thread counts.

FAA-Queue

Researchers at Rice University extended the ideas of LCRQ by creating the wait-free FAA-Queue[10]. Before FAA-Queue it was widely believed that ensuring the guarantee of wait-freedom was too computationally expensive to challenge lock-free data structures in performance, however, FAA-Queue managed to outperform LCRQ by allowing threads to attempt a 'fast path' enqueue where they could fail, until a certain number of attempts had been made, or their 'patience' ran out, at which point they would go for a more expensive enqueue where they were guaranteed to succeed. FAA-Queue and some of its derivatives remain industry-leading at the time of writing this thesis.

Chapter 3
Problem Statement

This section describes the context and goal of this thesis. We outline the system, Database X, that Spillover Queue (SOQ) is a component of. We then describe the goal and the requirements included in the goal.

3.1 Database X

Database X (DBX), is a highly performant in-memory database system. DBX requires both high performance and a small memory footprint to ensure that it can run efficiently on weak hardware with small memory. DBX is concurrent in many parts its internals, it supports transactions and uses MVCC as its concurrency control mechanism. It's also meant to run in an environment lacking the luxury of a proper Operating System, so it has its own internal execution and thread management. This means it doesn't have access to any *mutex locks* that the Operating System would normally provide.

The Memory Pool Allocator

DBX also has a ***memory pool allocator*** to avoid fragmentation and improve performance. This allocator is instantiated with a $\boldsymbol{Block}_{size}$ parameter, and can only allocate blocks of this size. In DBX, the memory is small enough to be fully addressable with 32 bits, rather than the more standard 64 bits. This allocator ensures that pointers are 32-bits long, which also saves space when storing them. It's also lock-free.

3.2 Goal

The goal of this thesis is to create a concurrent data structure to store the **version chain** used in DBX in its MVCC concurrency control mechanism. The data structure should be oblivious to the content of these versions, thus we can simply view versions as raw continuous data. As mentioned, transactions are committed concurrently and therefore versions need to be stored in the data structure concurrently. Further, the following requirements must be fulfilled.

Requirements

- **Storage** The data structure needs to physically contain the data of these versions, it needs to act as storage. It's not sufficient for the data structure to contain references or pointers to data elsewhere, such indirection would require more memory (storing one pointer per version) as well as reduce locality.
- **Variable-Sized Input.** Since a version is essentially a version of a row in DBX, and rows can be arbitrarily long, the versions can be arbitrarily large. A transaction updating multiple rows can even create one big version. This means that the input data or 'payload' the structure stores can be of entirely variable size.
- **Fixed-Sized Allocation.** The structure needs to use the ***memory pool allocator*** for its performance and memory efficiency, for this reason, all allocations it makes have to be of fixed-size, namely the $\boldsymbol{Block}_{size}$.
- **No Internal Fragmentation.** When the data structure gets a block from the allocator, it needs to use the block in its entirety, not leaving any gaps or internal fragments. The strict memory requirements of the system do not allow for such wasted space.
- **Lock-freedom.** Lock-freedom is required for performance purposes but since as the database doesn't have mutex locks, it also needs to be lock-free out of necessity.
- **ARM Architecture.** Like other smaller embedded systems, DBX runs on ARM and so any low-level instructions used need to be supported by ARM.

Scaling and Performance

Most lock-free data structures are created with the goal of linear scaling in mind. Linear scaling is the desired causation between the number of cores and performance improvement. Data structure designers hope to see a linear relationship between how many cores the system has and an increase in the throughput of the data structure. Academics will often run experiments on machines with 32+ cores. Linear scaling is a goal of this thesis as well but not a requirement. DBX only has 2 cores and 3.5 **threads** at its disposal, scaling at extremely high core counts will not benefit the system, therefore other requirements are prioritized. However, every update to the database is a transaction that has a new version and every time data is read from the database a version is read. This means that the version chain is involved in some way in all queries to the database, therefore this data structure is not allowed to be a bottleneck as it would reduce the performance of the entire system.

Chapter 4
Design

4.1 FIFO Queue Data Structure

The most important decision in the design process is to decide which fundamental data structure is suitable for the requirements and in the context of the system. This can be realized when considering the version chain. To save memory, outdated versions stored in the structure need to be removed as they become too old. This is done via a single-threaded garbage collector that implements some logic to realize when the timestamp of a version is too old, it can then free the memory and proceed to the *next* version. From this we can deduce that the fundamental data structure is a **FIFO Queue**, the first versions to enter the queue are more likely to be old, so the garbage collector can investigate the Head of the queue and *pop* outdated versions. The garbage collector is the only consumer of the queue, but there are multiple concurrent producers, this is threads that have committed a transaction that has created new version of a row and would like to store them. These versions are enqueued at the Tail of the queue and need to do so concurrently.

4.2 Overall Design

In this section, we outline some potential designs and describe how they relate to the requirements. We use this to justify the final design of Spillover Queue.

4.2.1 Naming Convention

We consider a block of raw memory continuous memory on the heap to be called a **Block**, they are of size $\boldsymbol{Block_{size}}$. A part of a Block storing some contiguous data is called a **Chunk**. In the case of DBX, each version stored in the Queue would be a Chunk of the same size.

4.2.2 Variable-Sized Blocks and Variable-Sized Chunks

Consider a Queue that uses an out-of-the-box dynamic memory allocator like `malloc`, in this case, the data structure receives a Chunk to be stored of any size and can just allocate this size exactly for the Chunk and place it in the Queue. This is a simple way to achieve the requirement of variable-sized Chunks and can be implemented like a traditional MSQ quite simply, where each node is a Chunk. In this design, there is no need to distinguish Blocks and Chunks as one Block is one Chunk. Figure 4.1 shows an example of this design. However, this violates fixed-size Blocks requirement, and the memory pool allocator can't be used, which means the system will experience memory fragmentation. It also requires one call to the `malloc` *per* Chunk, which may reduce performance when many small Chunks are enqueued.

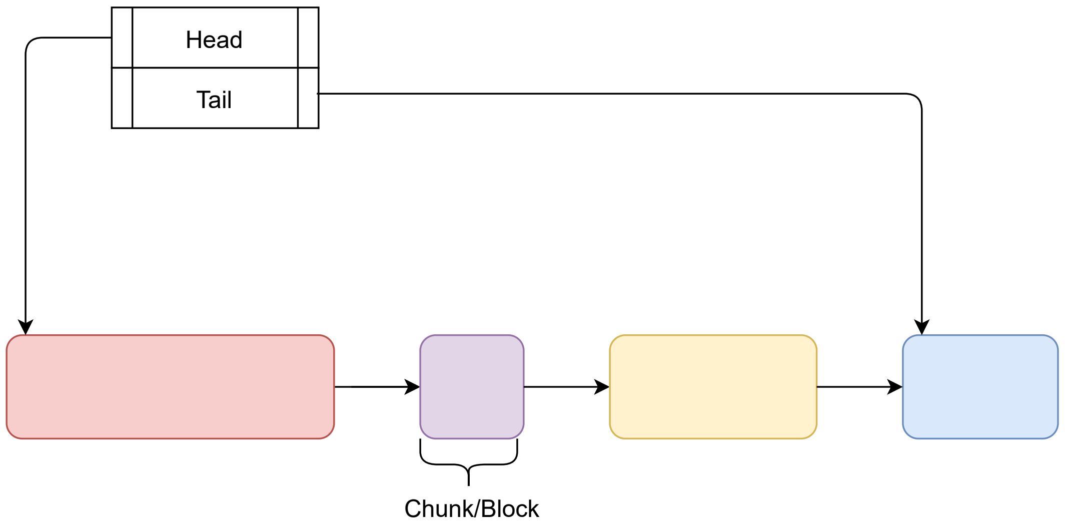


Figure 4.1

4.2.3 Fixed-Sized Blocks and Fixed-Sized Chunks

If the requirement of variable-size Chunks was relaxed, a design much like the one used by LCRQ[9] could have been implemented. which can be seen in Figure 4.2. A certain number of Chunks would always fit in a Block of fixed-size, completely utilizing the available space, as well as providing good locality and fewer allocations. However, Chunks can be of any size, so this design is not sufficient for DBX either.

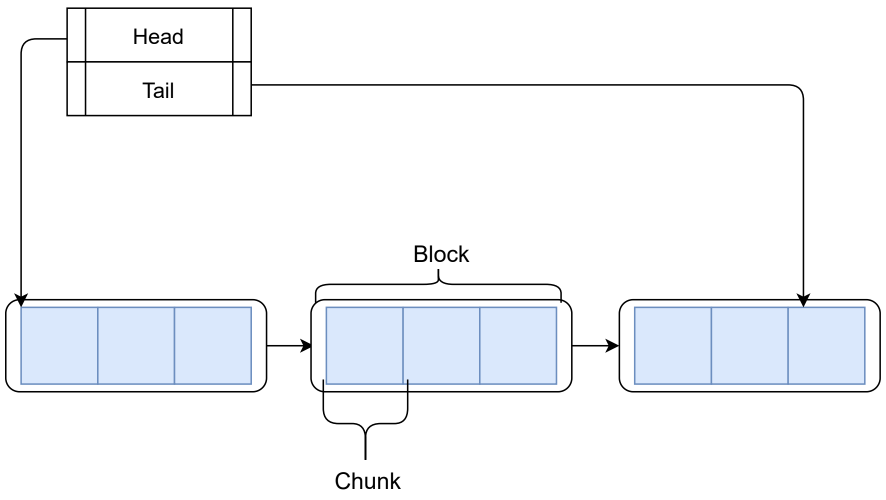


Figure 4.2

4.2.4 Variable-Sized Chunks With Fixed-Size Blocks with Fragments

Many initial ideas revolved around, if a thread wants to enqueue a Chunk that does not fit in the previous Block, it should allocate a new Block that it enqueues and stores its Chunk in. This solution, as seen in 4.3, is better than the previous two as Blocks can be of fixed size **and** Chunks can be of variable size. The locality and number of allocations are also good. However, this solution suffers from internal fragmentation, as 'does not fit' leaves fragments in the Blocks that the strict memory requirements won't allow.

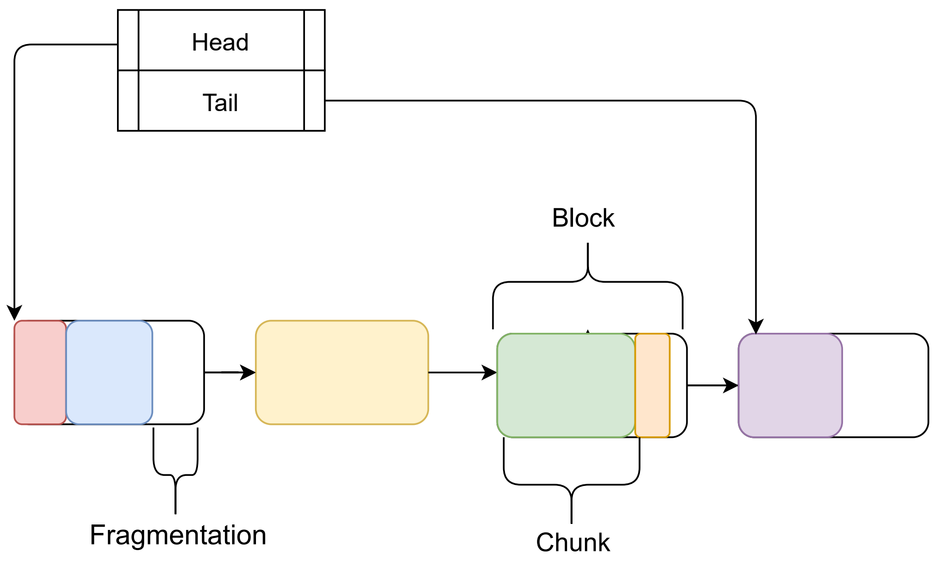


Figure 4.3

4.2.5 Fixed-Sized Blocks and Variable-Sized Chunks with Spillover

In this design, we allow Chunks that can't fit in the Block to *spillover* and continue on a consecutive Block. They can then be reconstructed when they are retrieved.

This layout combines all the benefits of the designs. The Blocks are fixed-sized so the highly efficient memory pool allocator can be used to pack all the Blocks in memory but it can still store variable-sized Chunks that the system requires. The spillover uses all the available space, avoiding fragmentation. This design, suitably named ***Spillover Queue*** (SOQ) fits all the requirements and is therefore what we created in this thesis.

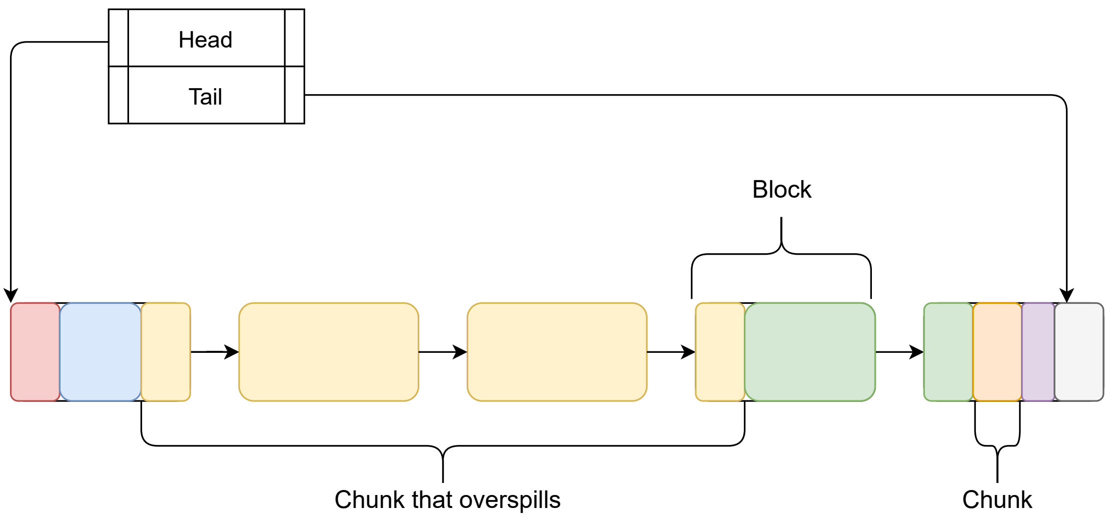


Figure 4.4: The Design of Spillover Queue

$Chunk_{size} > Block_{size}$

The $\boldsymbol{Block_{size}}$ is chosen on instantiation of the data structure and is fixed, but Chunks can still be of any size, *including larger than the* $\mathbf{Block_{size}}$. This means that, as seen in Figure 4.4, a Chunk might stretch over multiple Blocks and also spillover. This means that an enqueueing thread needs to concurrently link multiple consecutive Blocks to the Queue (and due to the lock-freedom) without blocking other threads that may be trying to enqueue concurrently. This is a non-trivial sub-problem of SOQ that other lock-free Queues do not solve.

General Methods and Abstractions

When a thread wants to store a Chunk of size $\boldsymbol{Chunk_{size}}$ in the Queue, it calls the RESERVE($\boldsymbol{Chunk_{size}}$) method. This is the method that will concurrently *reserve* space of size $\boldsymbol{Chunk_{size}}$ in the Queue for the Chunk, it can be seen as the SOQ equivalent of the ENQUEUE method in MSQ. This method returns a handle to the reservation known as a `ChunkIndex` which can be seen as a reference to that Chunk. The calling thread doesn't know whether Chunk has spilledover or not or even, where it's stored in the Queue. This is why `ChunkIndex` can then be used to interact with the ***reserved*** space, it can be used to place the actual Chunk data in the space and retrieve it again when needed.

4.3 Potential Race Conditions

Creating a lock-free version of the SOQ design is a difficult problem due to the vast number of states that need to be considered and that can change from when a thread first saw or took a *snapshot* of the Queue to the point that it updates or *commits* to the Queue. The Block could have been *filled up* as another Chunk was committed by another thread, the Tail could have moved, another Block could have been linked etc. This leads to an exceptional number of race conditions that SOQ needs to handle in order to be correct. Two example race conditions of a **naive** implementation of SOQ are illustrated below. However, there are many other variations and subtitles and even combinations of these race conditions.

4.3.1 Race Condition 1

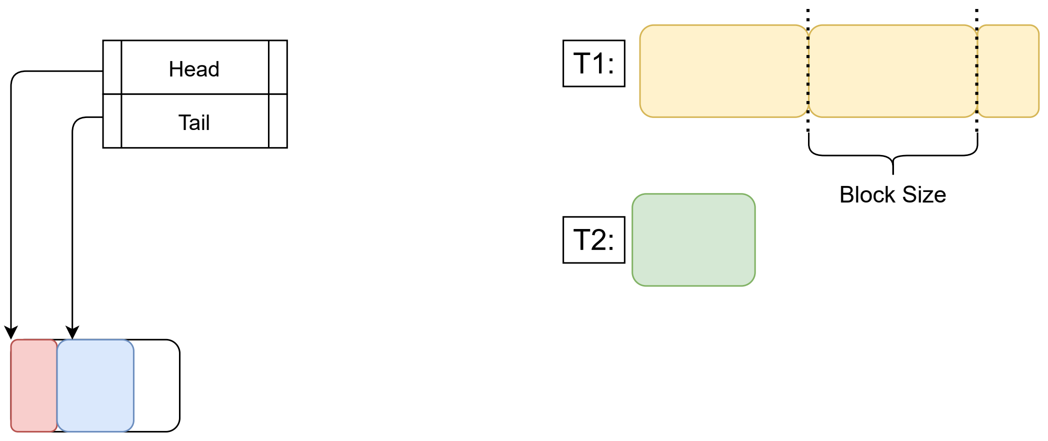


Figure 4.5: Thread **T1** wants to reserve space for a Chunk around 2.5× Block_{size} and thread **T2** concurrently wants to reserve some size eg. 0.8× Block_{size} .

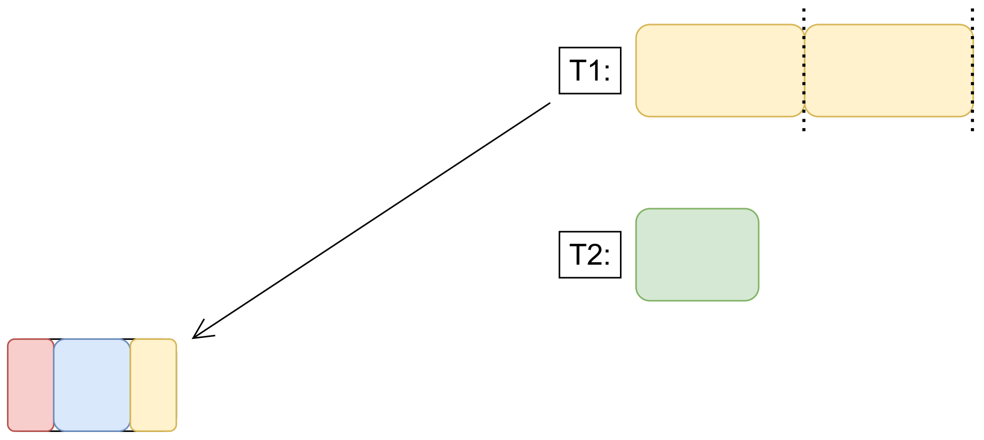


Figure 4.6: **T1** begins by filling up the Tail Block by reserving the *free space*. It has now reduced the amount it still needs to reserve by the *free space*.

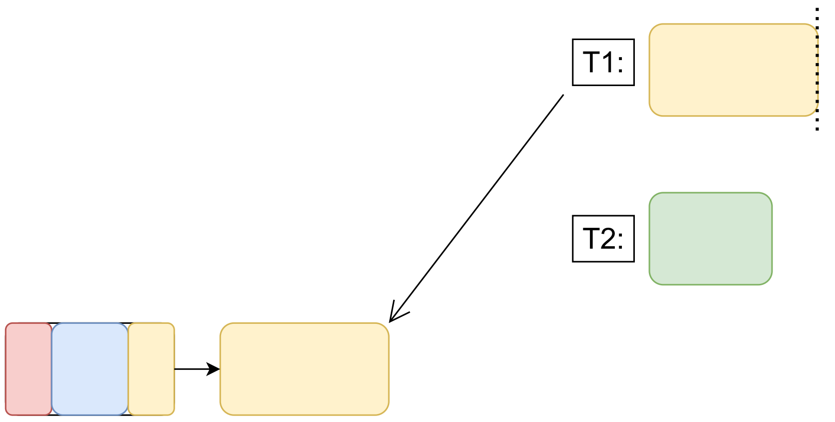


Figure 4.7: It still hasn't reserved its full ***Chunk***$_{size}$ so it continues by allocating a new Block and linking it to the Queue.

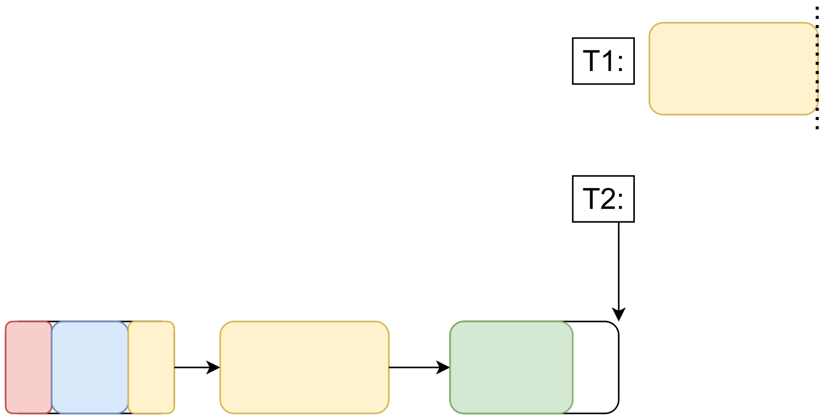


Figure 4.8: At this point it's possible that **T2** starts reserving, **T2** sees the Tail Block and even sees that there are no more Blocks linked after. So it allocates a Block for its own Chunk and links it.

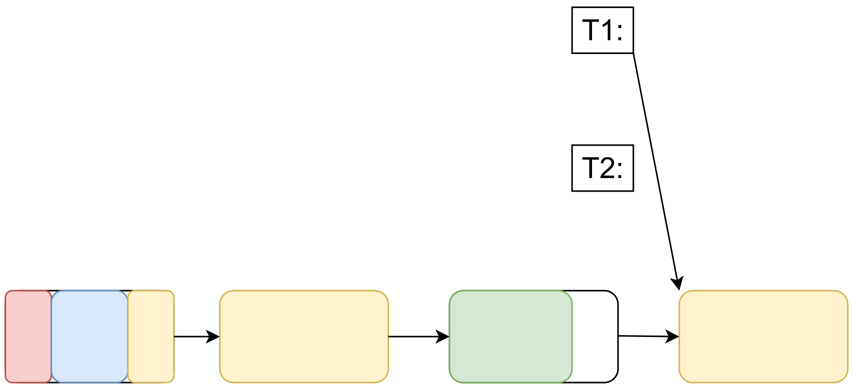


Figure 4.9: When **T1** then proceeds to link its last Block to the Tail, it would leave its reservation out of order. Depending on the implementation, it might also overwrite the *next* pointer pointing to **T2**'s Block, leaving it floating in memory.

4.3.2 Race Condition 2

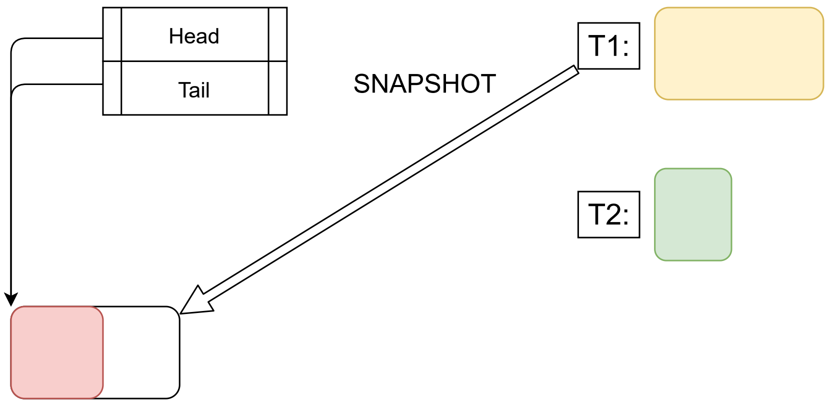


Figure 4.10: When **T1** wants to reserve a Chunk first it takes a snapshot of the current state of Tail Block.

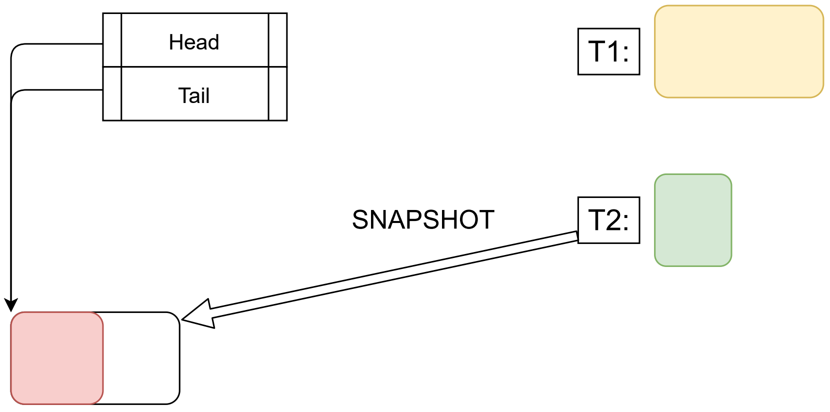


Figure 4.11: **T2** also takes a snapshot of the current state of the Queue.

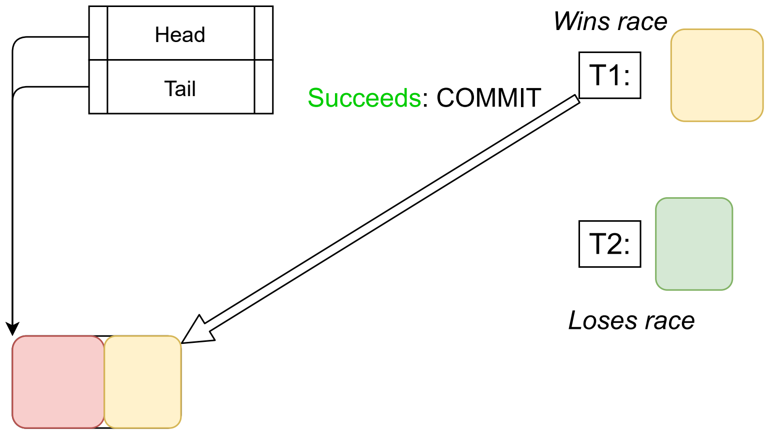


Figure 4.12: Both threads will attempt to reserve the ***free space*** for themselves. Here **T1** wins the race and manages to reserve space for half its $\boldsymbol{Chunk}_{size}$.

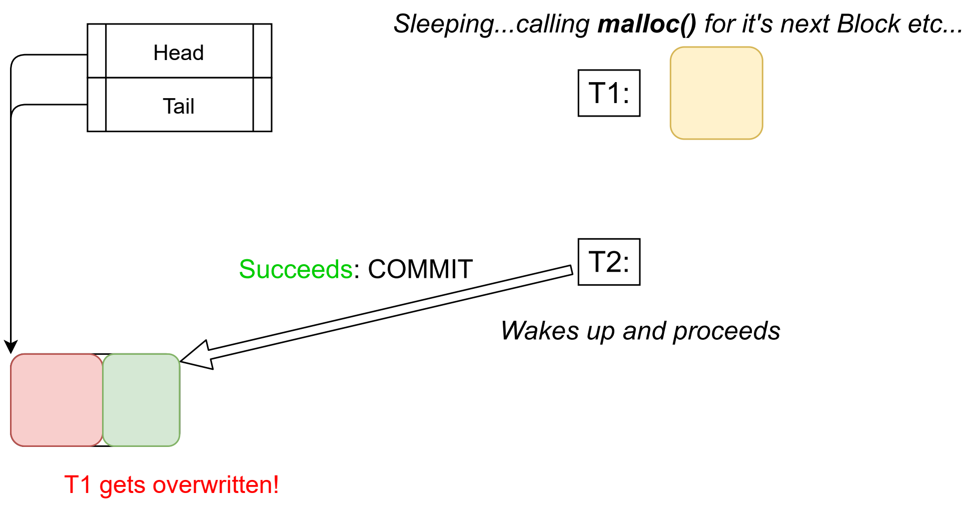


Figure 4.13: **T2** now manages to reserve the *free space* it saw in the snapshot, this overwrites **T1**'s reservation.

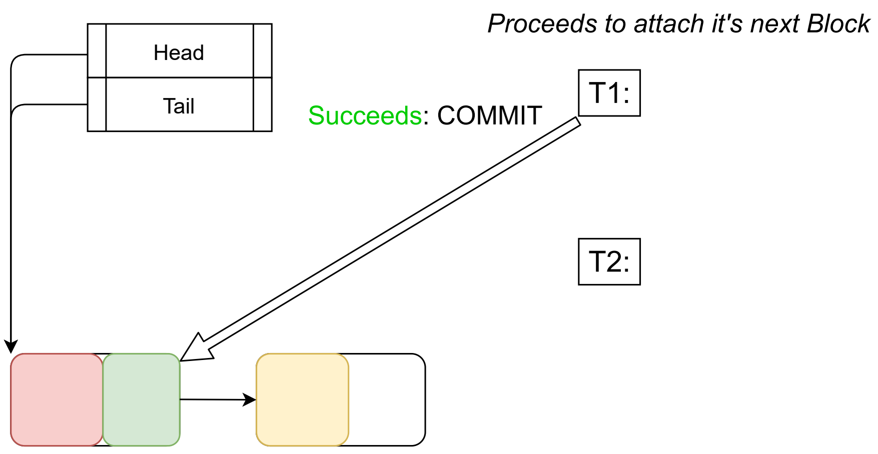


Figure 4.14: There is no way for **T1** to know its previous reservation was overwritten, so it proceeds where it left off.

If **T2** would've won the race, it would have still been disastrous as its data would have simply been overwritten by a waking **T1**.

Chapter 5

Spillover Queue Algorithm

This section describes Spillover Queue and how it works. First, we explain two key novel ideas that allowed us to create SOQ, we then explain the main RESERVE algorithm and how it handles concurrency and the previously described race conditions. Finally, we give an overview of the auxiliary methods PUT and GET and describe some optimizations.

5.1 Pointer Packing

Working in a small memory environment is nearly always a limitation, however, there is one subtle benefit. 32-bit or *short pointers* rather than standard 64-bit, can represent the entire memory space and this also happens to be exactly what DBX's ***memory pool allocator*** provides. SOQ cleverly exploits the combination of 32-bit pointers on modern 64-bit architecture by using a technique we refer to as *Pointer Packing*. Pointer packing allows SOQ to compare and update multiple states with a single unary CAS.

5.1.1 NextCursor

The Block shown in Figure 5.1 contains two Chunks and the remainder is ***free space***. Consider two pointers, one at the start of the Block and one to the end of the Chunks. The second is known as a *Cursor* and can be used to write to the ***free space*** in the Block. The Cursor and the start of the Block are separated by an *offset*. The Cursor is an important state of the Block as it dictates how *filled up* the Block is currently.

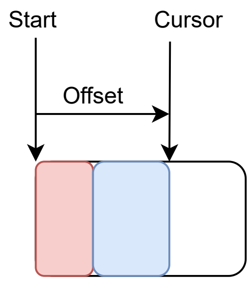
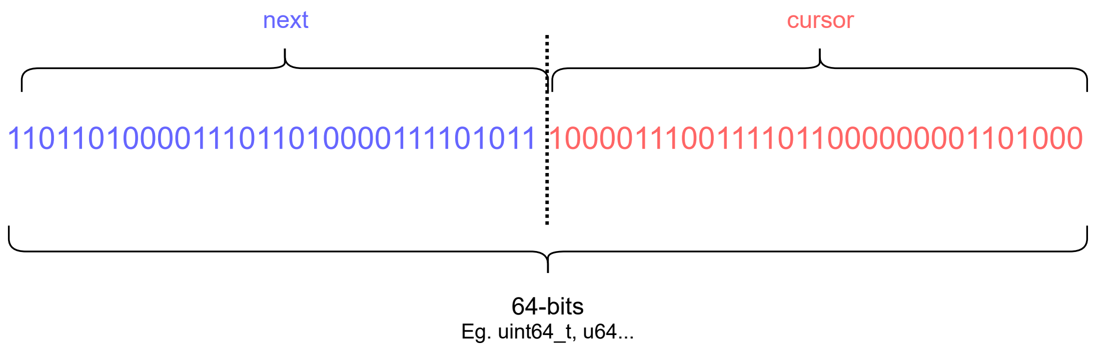


Figure 5.1

Another important state of a Block is the *next* field. This is where the Block stores the Next pointer that it uses to point to the next Block in the Queue. With the system's short pointers, both the Cursor and the Next pointer are 32-bits wide. Both of these states describe the same Block and are stored *in* the same Block. We can therefore store them *consecutively* in a memory region of 64 bits in the Block.



We represent both the Next pointer and the cursor with an atomic unsigned 64-bit integer which we refer to as the ***NextCursor***. This 64-bit integer can be updated atomically with a CAS, which compares and updates *both* fields atomically. This is an incredibly useful property, a CAS on the NextCursor will fail if *either* field is modified, but can also update *both* at the same time. When we later want to recover the Next pointer and the Cursor from the 64-bit integer they are stored in, we Split(NextCursor). Split does this by casting the NextCursor to an unsigned 32-bit integer to retrieve the Cursor and right shifting the NextCursor by 32 and then casting the result to an unsigned 32-bit type to retrieve the Next pointer.

Algorithm 3 Split()

```
1: function Split(NextCursor)
2:     next = (uint32) (NextCursor » 32)
3:     cursor = (uint32) NextCursor
4:     return next, cursor
```

To simplify the implementation of SOQ, we instead store the Offset in the Cursor part of the NextCursor rather than the Cursor pointer. This is equivalent to the Cursor pointer as one can recreate it by using the Block pointer and the Offset. This makes it easy to move the Cursor by incrementing it. We can even move the Cursor without splitting the NextCursor just by incrementing the entire NextCursor directly, `NextCursor += x`, as long as the addition doesn't make the Cursor part exceed $2^{32} - 1$, which it never will in a correct implementation as it would suggest a 4GB Offset.

5.2 The GuessQueue

Race Condition 1 shows a thread reserving a Chunk that spans multiple Blocks will run into many concurrency problems. A solution is to not try to link each Block on its own, but rather attempt to link *all of them* at once. We pre-allocate these Blocks and link them together forming a new sub-Queue with enough total space reserved to hold the Chunk in its entirety. This sub-Queue is called the ***GuessQueue*** and it's created with the method GUESS($\boldsymbol{Chunk}_{size}$).

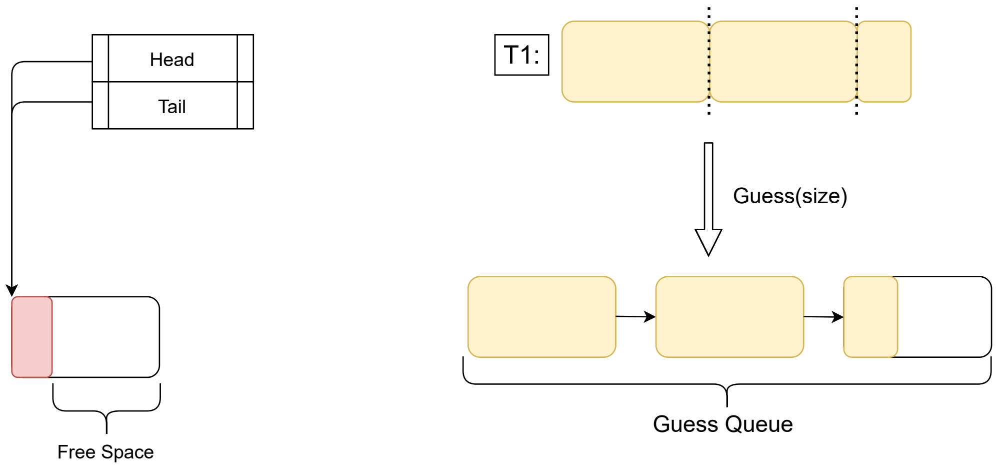


Figure 5.2: T1 generating a GuessQueue for it's large Chunk.

However, there's most likely still ***free space*** to use up in the Queue, which means that the GuessQueue might not be *exactly* what needs to be reserved and linked. This is where the *guessing* comes from. The GuessQueue is a *pessimistic guess*, of how many Blocks should be linked and how filled up those Blocks should be. The pessimism means that the GuessQueue assumes that there's no ***free space*** and allocates as many Blocks as it needs. We can see in Figure 5.3 that the ***free space*** that needs to be used up means that equally much *redundant space* was allocated and created in the GuessQueue. SOQ resolves this by using a concept we call *Trimming* explained in Section 5.3. When we finally want to link or *commit* the *entire* GuessQueue at once, it's enough to link the Head of the GuessQueue to the Queue's Tail and then update the Tail to be the GuessQueue's Tail.

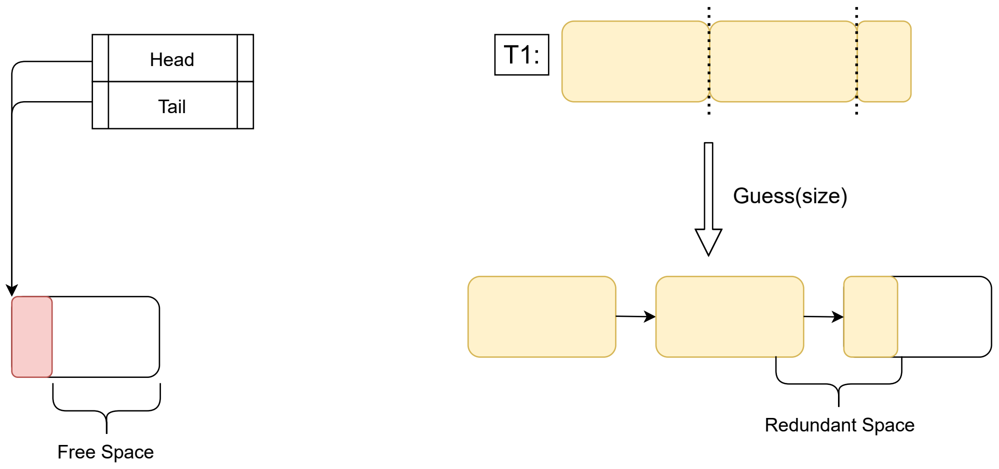


Figure 5.3

5.3 Reserve Algorithm

Here we explain the main concurrent Reserve algorithm of SOQ. Algorithm 4, 5 and 6 all outline the **same** reserve method but with gradually decreasing level of abstraction to assist understanding. The method is broken up into *phases* as seen in Algorithm 4 which are then explained slightly more concretely in Algorithm 5 where the colored outline represents each *phase*.

5.3.1 Phases of Reserve

The overall outline of the method is closely related to MSQ's Enqueue, there's still only one `loop`, it also starts with atomically reading the Tail but instead of also reading the *next* field of the Tail we read the *NextCursor*. It also then checks whether the *next* field is still `null` and if it isn't it attempts to *assist* in correcting the Queue. SOQ also still has an atomic Tail pointer, an atomic Head pointer and every Block's NextCursor is also an atomic variable.

Guess, Look, Trim, Try To Commit, Reset and Retry

The first phase is the creation of the **GuessQueue**. This is done outside of the loop and *only once*, the pessimistic sizing means that it can be generated without ever viewing the state of the real Queue, and it only needs the $\boldsymbol{Chunk_{size}}$ as a parameter. We then want to take a 'look' at the actual state of the Queue to determine how we should ***trim*** the GuessQueue. This viewing of the real Queue is the **Snapshot** phase, the Snapshot involves atomically reading and storing the `Q→Tail` and then reading the `Tail→NextCursor`, from this we can determine everything we need to know about the *current* state of the Queue. The NextCursor now allows us to determine how much we should trim from the GuessQueue for it to fit the current state of the Queue, this is done in the next phase, the **Trimming** phase. Because of the pessimistic sizing, we will only ever need to remove reserved space from the GuessQueue and never allocate and add more. Once the trimming is completed, it's time to *attempt* to **Commit** the GuessQueue. A Commit is where the algorithm attempts to update the actual state of the Queue itself by trying to make a reservation. A Commit always involves attempting to modify the `Q→Tail→NextCursor` that was read in the Snapshot with a **CAS**. If it needs to *spillover* and link the Head of its GuessQueue, it will update the Next pointer, if the $\boldsymbol{Chunk_{size}}$ is small enough and it doesn't need to spillover, it will update it by incrementing the Cursor part. A spillover Commit attempt is shown in Figures 5.4 through 5.6, and a Cursor increment attempt is shown in Figures 5.7 through 5.9. The CAS attempt in the Commit might fail, this means that the NextCursor read in the Snapshot has changed somehow by another thread. In this case, it's time to retry, however, we know that the state of the Queue has changed, which means that the GuessQueue isn't an *exact* fit anymore, the ***free space*** is most likely different. This is why, upon a failed Commit, we need to **Reset**. In the Reset phase, we restore any trimming done to the GuessQueue, so that when we loop again, the GuessQueue is complete and pessimistic again and we can retry to Commit.

This is the general loop pattern of a reserving thread, take a look at the state of the Queue with a Snapshot, trim the GuessQueue, try to Commit, and if it doesn't succeed, Reset the GuessQueue, start over and Retry until the Commit succeeds.

Algorithm 4 Phases of Reserve

```
function RESERVE(Q, SIZE)
    GUESS QUEUE
    loop
        SNAPSHOT
        if next == NULL then
            TRIMMING PHASE
            COMMIT ATTEMPT
            if Commit is successful then
                LONG SWING
            else
                RESET AND RETRY
        else
            SHORT SWING
```

Algorithm 5 High level explanation of Reserve

```
function RESERVE(Q, $Chunk_{size}$ )
    Generate a GuessQueue from the $Chunk_{size}$ by calling GUESS($Chunk_{size}$ );
    loop
        Atomically read the Tail pointer to get the Tail;
        Atomically read the NextCurser of the Tail;
        Split the NextCursor to find the Next pointer Next and the Cursor;
        if next == NULL then
            Let free_space be how much space is left in the Tail Block compute it using
            the Cursor and the $Block_{size}$ ;
            Trim the GuessQueue depending on the free_space;
            Store reference to any trimmed Block in maybe_dropped;
            Create new_next_cursor for the Tail Block that points to the Head of the
            GuessQueue or only has an incremented Cursor;
            Attempt to update the NextCursor of the Tail with the new_next_cursor using
            a CAS on the Tail Block;
            if Commit is successful then
                Reservation is completed
                Attempt once to complete a long swing to the Tail of the GuessQueue
                using a CAS on the Tail pointer.
                Free maybe_dropped;
                Create and return ChunkIndex;
            else
                Commit failed
                The Tail's NextCursor has changed, the Snapshot is invalid
                Reset the GuessQueue and try again;
        else
            The Tail pointer is no longer pointing to the last Block in the Queue. Try to
            complete a short swing by moving the Tail one step with a CAS;
```

Trimming and The Power of Pessimism

The naive use of the GuessQueue would be to first take the Snapshot of the Queue, generate a GuessQueue based on the Snapshot which would perfectly fit, and try to Commit it to the Queue, if the Commit fails and the Queue has been modified, it simply repeats and creates a new GuessQueue. This is a correct solution but faces starvation and performance problems. This is evident when reasoning about the *synchronization points* of RESERVE. We consider the Snapshot the first synchronization point as that's the first read or interaction of the current state of the Queue, the next synchronization point is the Commit attempt as here we try to update the Queue. For the Commit to be successful, the Queue needs to be consistent with the state read at the Snapshot, so the Commit depends on *nothing* changing between the Snapshot and the CAS in the Commit. Thus, it is crucial that SOQ optimizes the phases between these two points because the longer time spent between the Snapshot and the Commit the more likely it is that another thread has managed to Commit something itself. Therefore, taking a Snapshot, and then starting to allocate Blocks and construct the GuessQueue would massively increase the likelihood that the Commit fails. Lots of failing Commits means lots of expensive failing CAS's, and as Nir Shavit showed [4] this starvation can kill performance.

If we wanna minimize the compute between the synchronization points, we want to move as many operations as possible out of the critical section between the points, so we try to do as many operations as possible *before* the Snapshot and *after* the Commit. Let's say we attempted to use an *optimistic* GuessQueue instead. An optimistic GuessQueue would assume that there's a whole Block of ***free space*** available for itself. When we then take a Snapshot, there would most likely be less ***free space***, so we would probably have to *grow* the GuessQueue by allocating and linking a new Block. This means that there could be an expensive call to the allocator between the synchronization points. Therefore, rather than *growing* the GuessQueue between the Snapshot and the Commit, we *trim* it. The trimming will only ever involve fast $O(1)$ atomic updates, it can be:

1. Decrement the cursor of the `GuessQueue→Tail→NextCursor`

2. Un-link the redundant `GuessQueue→Tail` Block entirely. As seen in figure 5.8.

3. A combination of **1.** and **2.** Un-link the redundant Tail Block and decrement the NextCursor of the previous (new Tail) Block. See 5.5.

A reference to any removed redundant Block is saved in the variable ***maybe_dropped***. Which can then be safely freed or 'droppped' *after* the Commit succeeds. There will only ever be a maximum of one Block trimmed off the GuessQueue. See Theorem 1 for proof of this. This also allows us to efficiently Reset the GuessQueue if the Commit fails. We stored a reference to a removed Block and since it was just un-linked its NextCursor remains the same as when the GuessQueue was created, so to reset we can just link ***maybe_dropped*** to the Tail of the GuessQueue again. We also don't have to keep allocating and freeing Blocks on every failed Commit, we instead re-use the Blocks as we keep looping.

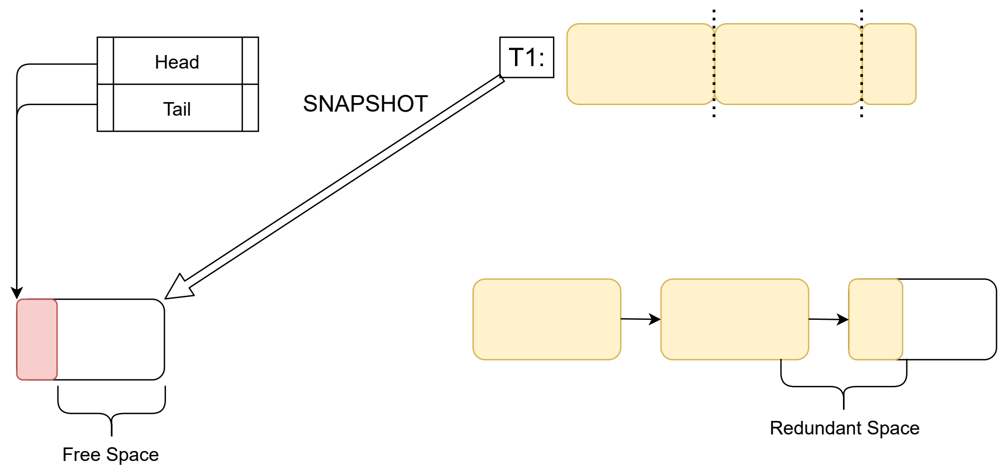


Figure 5.4: T1 takes a Snapshot of the current state of the Tail of the Queue, which allows it to see how much *free space* is available at that moment.

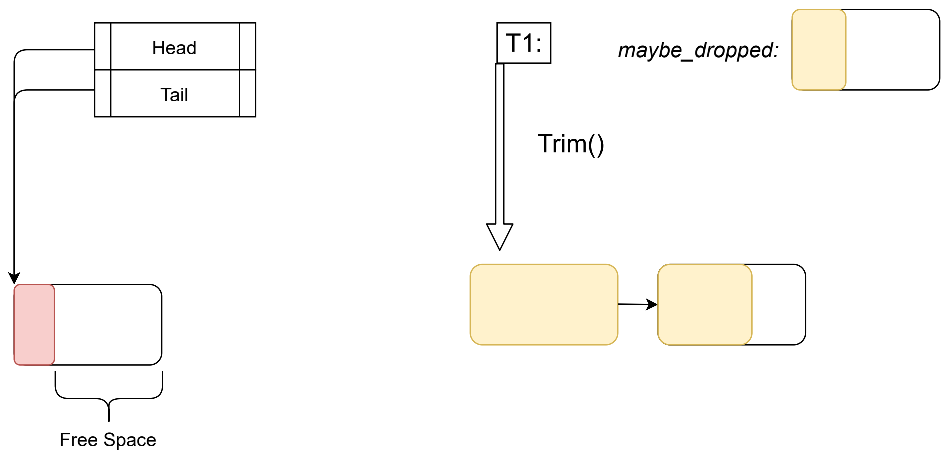


Figure 5.5: It uses the *free space* to trim the Guess Queue accordingly. A Block is redundant so a reference to it is stored in *maybe_dropped*.

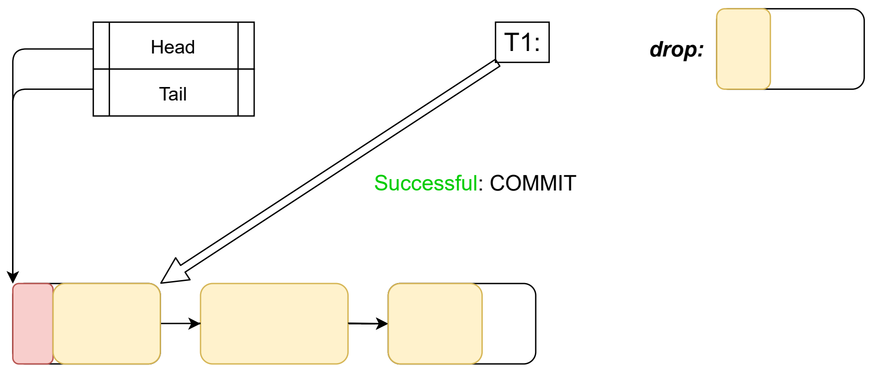


Figure 5.6: The Commit succeeds, the `Q→Tail→NextCursor` is updated to include the `GuessQueue→Head` and *maybe_dropped* is freed.

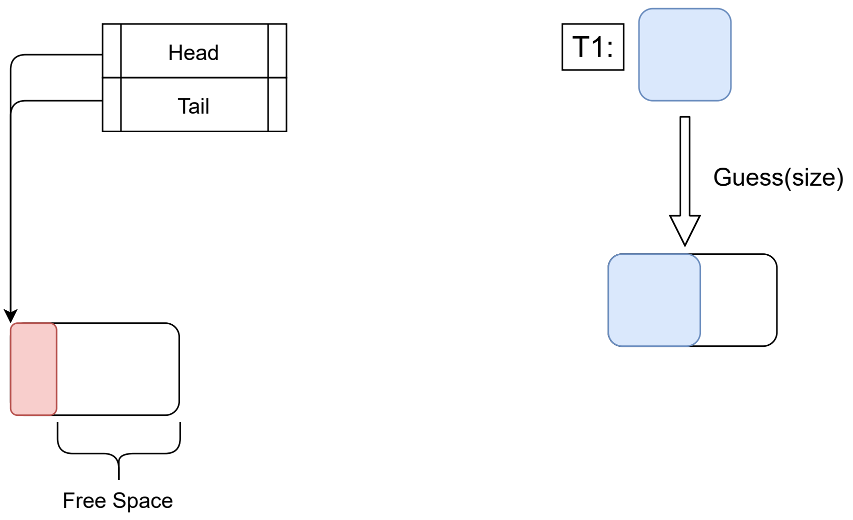


Figure 5.7: The ***Chunk***$_{size}$ fits within a single Block so the GuessQueue is merely a single Block.

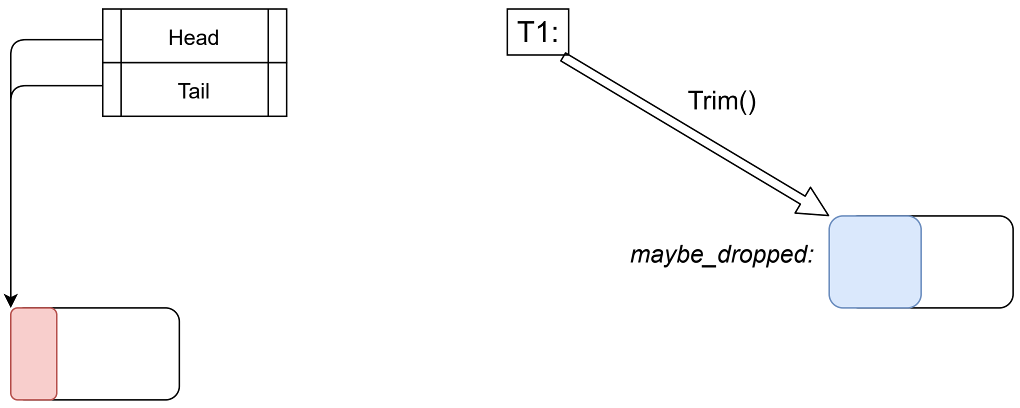


Figure 5.8: After finding the ***free space***, the entire GuessQueue gets trimmed and is stored in ***maybe_dropped***.

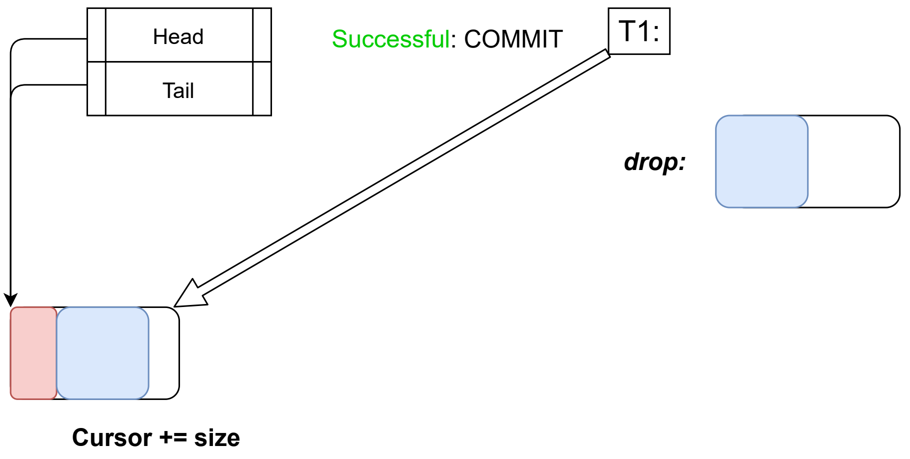


Figure 5.9: The Commit succeeds, the Cursor of the Tail Block remains the same but incremented by ***Chunk***$_{size}$. The GuessQueue is dropped.

General Case

The previous Figure 5.9 shows how a Chunk smaller than the ***free space*** fits inside the Tail Block in its entirety. This means that to add it the only Commit that occurs is an atomic increment of the Cursor. We call this the *general case*, and should be the most common and efficient case. The critical section trimming is very fast as it is just storing the entire GuessQueue in ***maybe_dropped*** and the ***new_next_cursor*** which we attempt to Commit, is computed quickly by just incrementing the NextCursor. If, on the other hand, the Chunks were consistently $>\boldsymbol{Block_{size}}$ the GuessQueue will also consistently contain more Blocks which means more calls to the allocator and less probability of the general case. If this happens too frequently we theorize the $\boldsymbol{Block_{size}}$ should be increased.

Tail Swinging

When a thread finally commits its GuessQueue, the Tail pointer should be updated to whatever Block is the new Tail Block. In MSQ, we try to update the Tail pointer to the new Block that was just enqueued, however, in SOQ, the Tail of the *GuessQueue* should become the new Tail. The thread that successfully committed the GuessQueue knows the address of the `GuessQueue→Tail`, this means that after its commit succeeds it can attempt to move the Tail there directly with a single CAS on the Tail pointer. We call this the ***Long Swing***, the Tail pointer might swing past multiple Blocks at once.

However, another thread might attempt to reserve before the Long Swing has occurred but after the first thread successfully committed its GuessQueue. In this case, much like in MSQ, the second thread will discover that the *next* field of the `Q→Tail` is not `null`. However, this thread doesn't know where the GuessQueue's Tail is, all it knows is that there's at least one more Block that has been linked to the Tail via the Next pointer. So, it will CAS the Tail to be this Next pointer and move the Tail pointer one Block, we call this the *Short Swing*. When the reserving thread finally attempts to Long Swing, the CAS will fail as the Tail has been moved by another thread. Thanks to the Short Swing, the Tail pointer has swung one Block, but the `Q→Tail` is still not `null` as there's still more GuessQueue to go. So, when the Short Swinging thread loops and re-tries, it will Short Swing again, doing so until the data structure has been corrected and it can finally attempt to reserve. If a third thread joins in and attempts to reserve in the middle of this process, it too will discover that the Next field is not null, so it will join in and start trying to Short Swing the Tail pointer as well. These threads will cooperate in Short Swinging until the data structure is corrected.

Figures 5.10 through 5.18 illustrate first what happens if the Long Swing completes and then what happens if the Long Swing fails and a series of Short Swings complete.

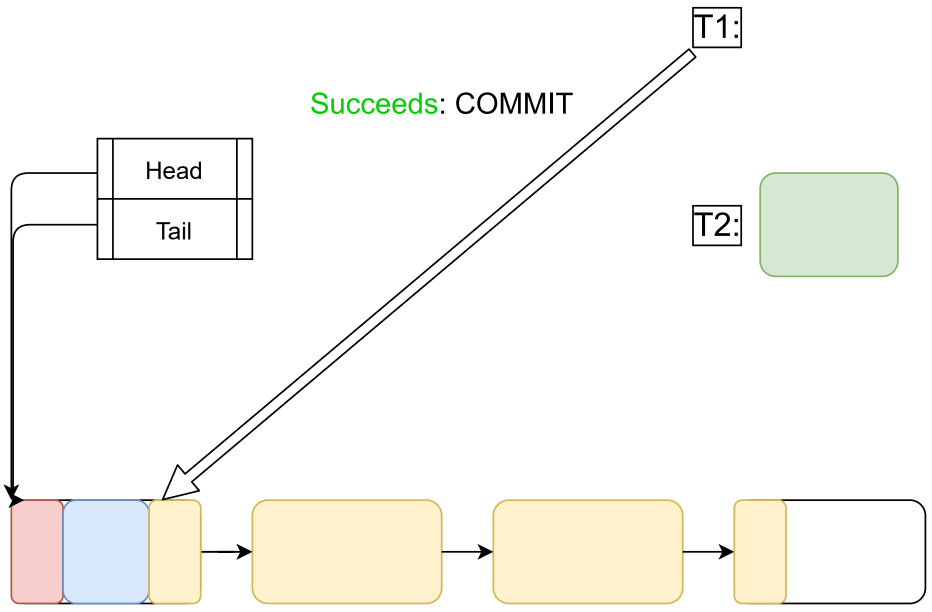


Figure 5.10: T1 successfully Commits a large GuessQueue before T2 manages to Commit itself. T2's Commit fails so it will reset and run the loop again.

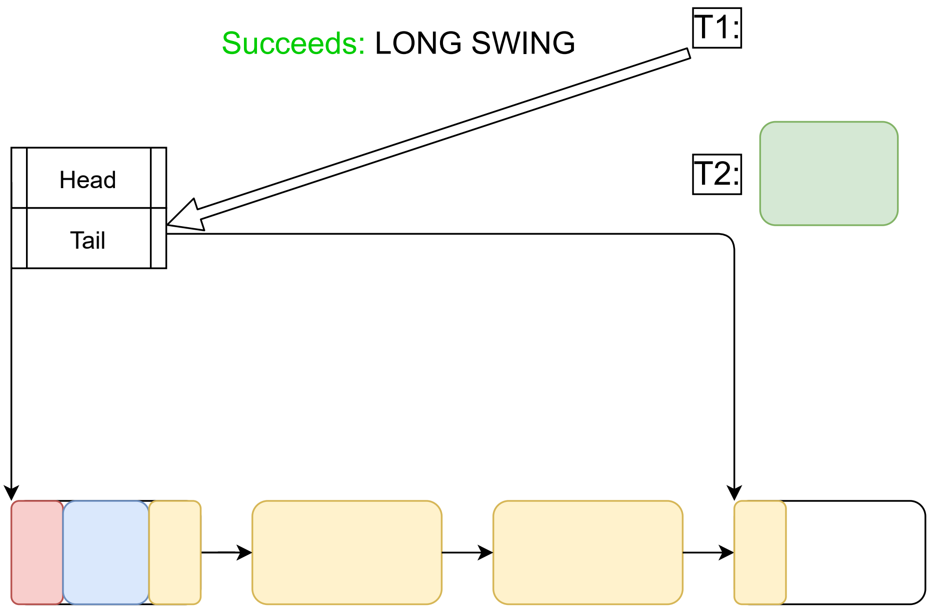


Figure 5.11: T1 wins and manages to complete a Long Swing directly to the new Tail and it's done

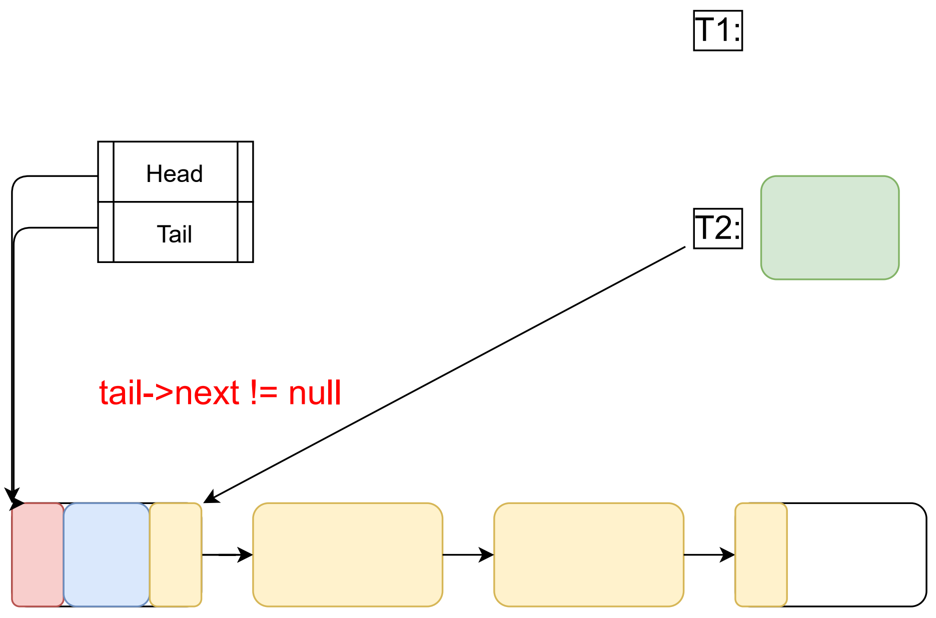


Figure 5.12: If however, T2 wins before the Long Swing, it first discovers that the Tail's Next pointer is not null.

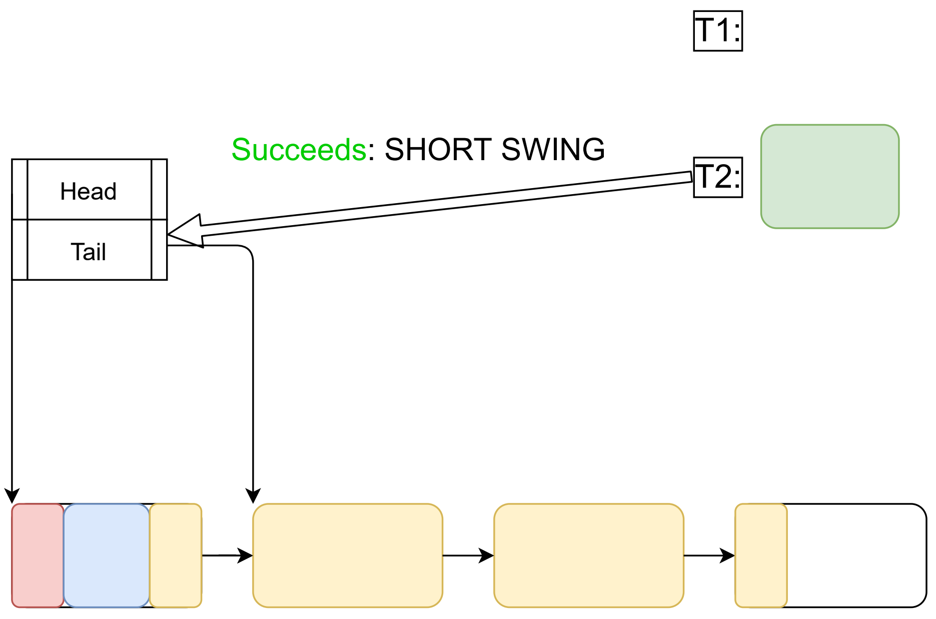


Figure 5.13: It completes a Short Swing to the next element and loops again starting over to retry its reserve.

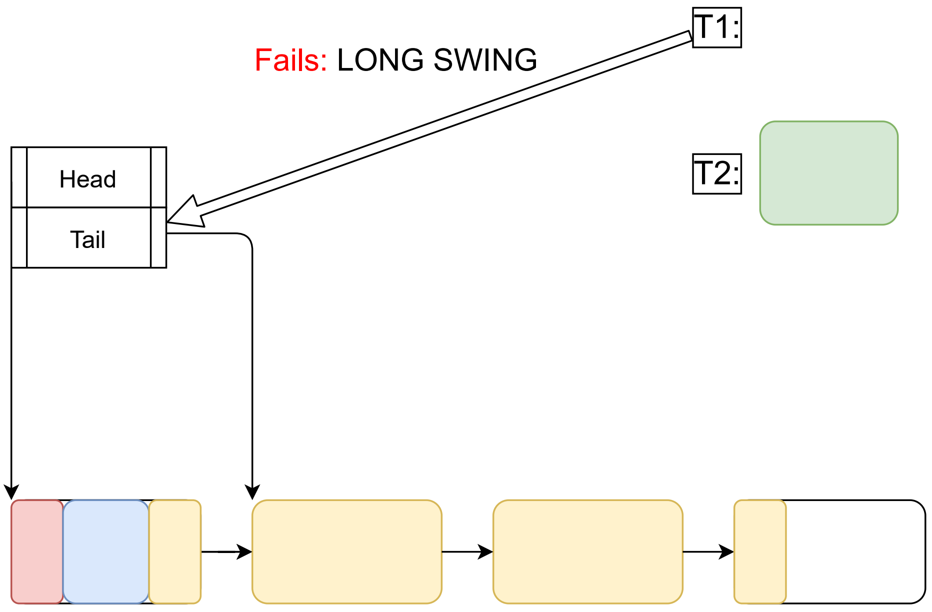


Figure 5.14: When T1 then finally attempts it's Long Swing it's too late, the CAS will fail as the Tail has changed.

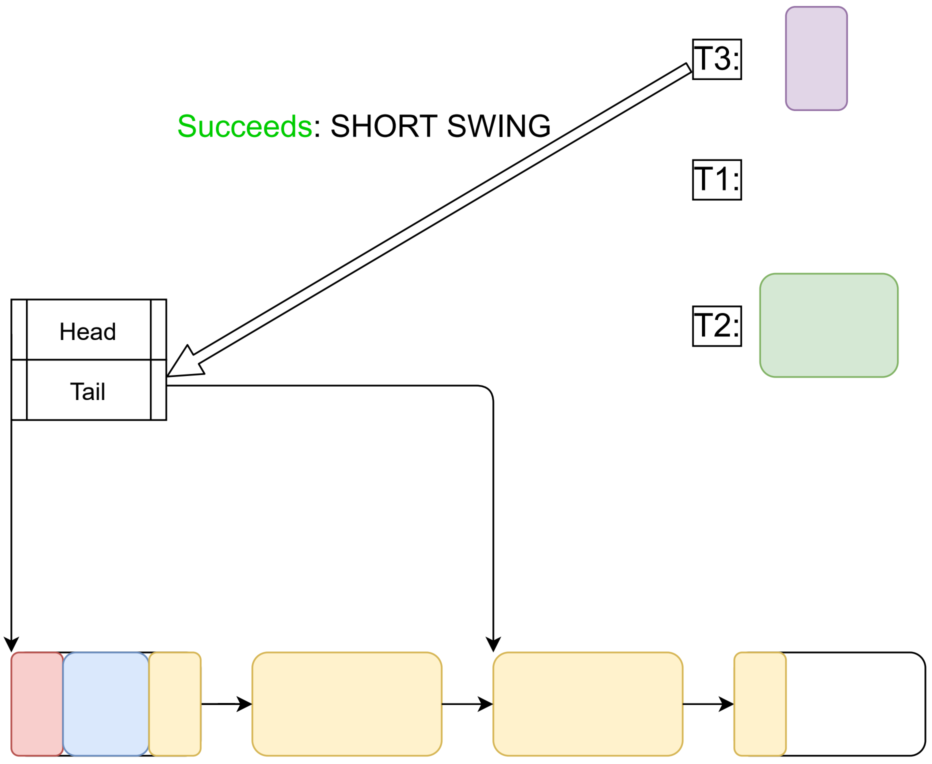


Figure 5.15: If a new thread, T3, attempts to reserve in the middle of this, nothing changes. T3 will see that the Tail's next field is not null, so it will also complete a Short Swing, loop, and retry.

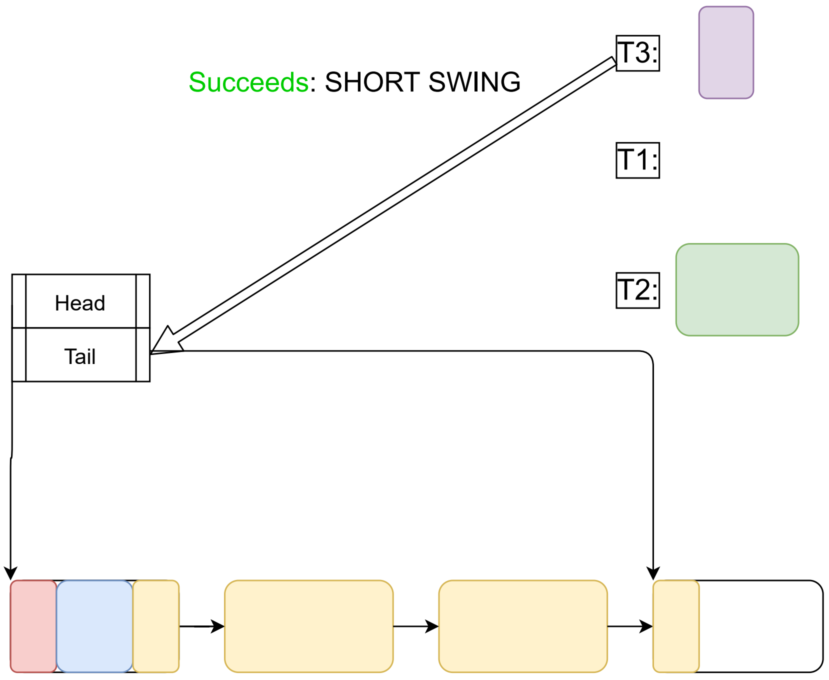


Figure 5.16: Upon its retry it discovers it still needs to Short Swing and does so. There's also a race between T3 and T2 to see who can Short Swing first, only one succeeds due to the CAS.

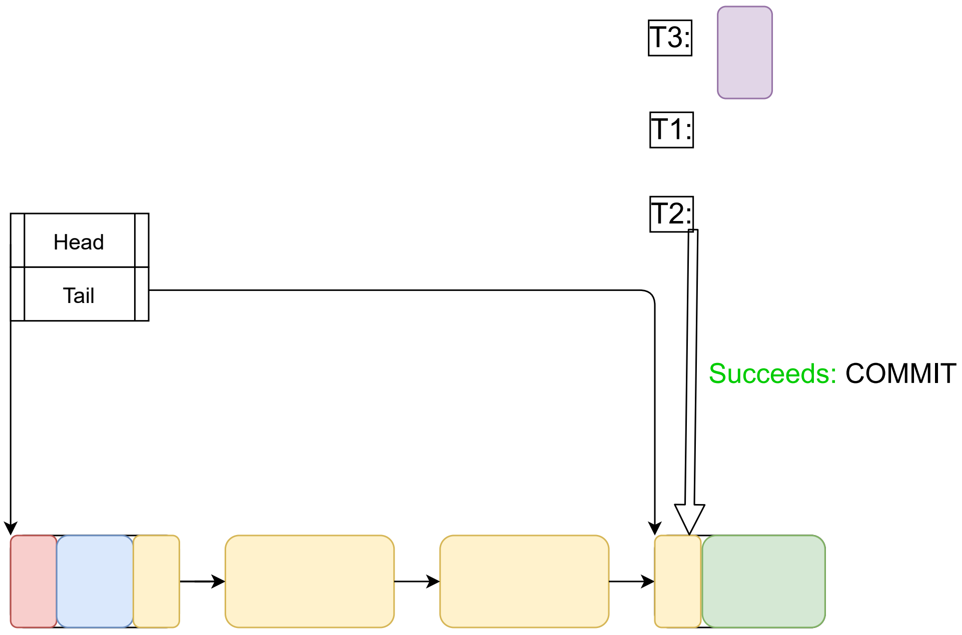


Figure 5.17: The data structure is now corrected and T2 will get past the condition and can successfully commit its Chunk.

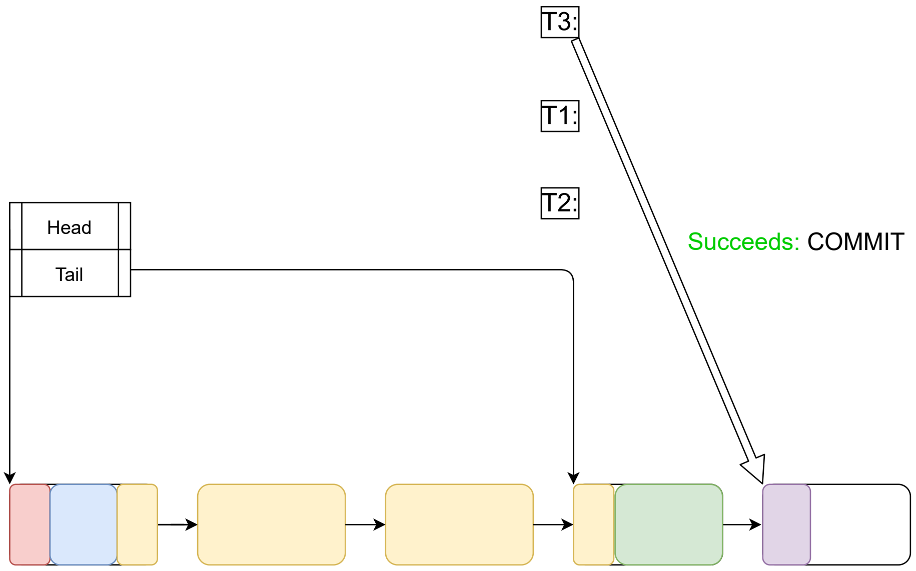


Figure 5.18: T3 also manages to Commit its Chunk.

This is interesting since only T1 knows where the Tail of its GuessQueue is, and if it doesn't have time to update the Tail there, every other thread will blindly cooperate to correct the data structure until they find the Tail of the GuessQueue, it's another example of concurrent *assistance*.

One might think that an optimization would be to allow T1 to re-attempt its Long Swing. However, there's no telling where the Tail pointer has moved between T1's commit and its Long Swing attempt. The Tail pointer may have moved far beyond the end of T1's GuessQueue, so to re-attempt its Long Swing may move the Tail pointer backward. This is why it's important that the Long Swing is attempted *once* and not in a loop.

5.3.2 Implementation Details

We will now take a closer look at the implementation of RESERVE by analyzing the pseudocode in Algorithm 7. On line 14, we check if the ***free_space*** is greater than the size of the Cursor of the GuessQueue's Tail. Essentially this check determines if a whole Block will be trimmed off the GuessQueue or not. In both cases we create the ***new_next_cursor***, this is the new state of the Tail's NextCursor and is what we try to Commit to the Queue, it can point to the Head of the GuessQueue or if it's a general case it's simply incremented. Committing the ***new_next_cursor*** is done with the CAS on line 21, which will of course fail if another thread has committed.

If the condition on line 14 is true and we need to trim a Block from the GuessQueue, we call the TRIM method. This method accepts a reference to the GuessQueue and the ***free_space***, it will Trim the GuessQueue, removing any Tail Block by setting the previous Block's next field to `null`, and updating that previous Block's cursor. Trim determines how the ***new_next_cursor***, should look in this case, if there is spillover or it's a simple general case. It also accepts the pointer ***long_tail*** which points to the Block that the *Long Swing* should try to update to. The ***long_tail*** is always set to the Tail of the GuessQueue initially as we can see on line 9. However there are two more states this could be in. 1. the `GuessQueue→Tail` could have been trimmed off and there could be another previous Block which becomes the

Algorithm 7 Pseudocode of Reserve()

```
 1: function RESERVE(Q, SIZE)
 2:     GQ = Guess(size)
 3:     block_size = Q→BlockSize
 4:
 5:     loop
 6:         tail = Q→Tail
 7:         next_cursor = tail→NextCursor
 8:         next, cursor = SPLIT(next_cursor)
 9:         long_tail = GQ→Tail
10:         maybe_dropped = NULL
11:
12:         if next == NULL then
13:             free_space = block_size − cursor
14:             if free_space >= GQ→Tail→cursor then
15:                 maybe_dropped = GQ→Tail
16:                 new_next_cursor = TRIM(&GQ, free_space, long_tail)
17:             else
18:                 GQ→Tail→NextCursor -= free_space
19:                 new_next_cursor = GQ→Head
20:
21:             if CAS(Q→Tail→NextCursor, next_cursor, new_next_cursor) then
22:
23:                 if long_tail != NULL then
24:                     CAS(Q→Tail, tail, long_tail)
25:
26:                 if maybe_dropped != NULL then
27:                     FREE(maybe_dropped)
28:
29:                 return CHUNKINDEX(tail, cursor)
30:             else
31:                 RESET(GQ, maybe_dropped)
32:         else
33:             CAS(Q–>Tail, tail, next)
```

Tail of the GuessQueue, it is then this Block we set ***long_tail*** to. 2. There might be no new Tail Block as in figure 5.6, because a general case reserve doesn't link any new GuessQueue. In this case, it's very important to set the ***long_tail*** to `null` and check for this on line 24 before attempting the Long Swing because we do not want to CAS the Tail to `null`, or even worse CAS the Tail to a Block that has been trimmed and then freed. So in this case we'd rather leave the Tail as it is. Similarly, we check on line 27 if any Block actually has been trimmed before attempting to free the ***maybe_dropped*** variable. We can see that if we fail to Commit the ***new_next_cursor*** we call the Reset method on line 31 to re-create the original GuessQueue before we can loop and retry. We can also see the Short Swing attempt on line 33. If the commit succeeds and the Long Swing has been attempted and ***maybe_dropped*** has been freed, we can generate the `ChunkIndex` and return it on line 29. More information on this method is described in Section 5.4.

Avoiding race conditions

We can now see how the NextCursor avoids Race Condition 2 as seen in 4.10 through 4.14. When T2. wakes up in 4.13 and attempts to Commit its Chunk. The Cursor would have been incremented and the CAS in T2's commit will fail. The same thing would happen if another thread had linked its GuessQueue as then the Next pointer would have been changed from `null` to pointing at the Head of the GuessQueue. Another interesting benefit is that the Snapshot contains two atomic loads, one to find the Tail and another to find the NextCursor. Another thread might intervene between these two operations and successfully Commit. In this case, the Snapshot is invalid from the start, however, the NextCursor resolves this as well because if the Tail has moved, the Next field will not be `null` and if the competing thread has completed a general case Commit, then the Tail will still be the same but the Cursor has updated. Crucially we know that if the Tail has moved, the Next field will have already changed *before*, because a Commit that links a Block will always happen before it or any assisting thread tries to move the Tail, as an assisting thread can only move and will only move the Tail if the Next has been updated.

Proof of Maximum One Block Trim

We will now show that there can only ever be one Block that needs to be trimmed from the GuessQueue.

Theorem 1. *There can be at most one Block that gets trimmed from the GuessQueue.*

Proof. We prove this by showing that in the worst case, where the trimming as large as possible, no more than one Block will be trimmed. Consider the Tail Cursor, Q→Tail→NextCursor, or TC of the Queue. If we want to trim the largest possible number of Blocks from the GuessQueue, the TC should be as small as possible leaving as much ***free space*** as possible. This happens when $TC = 0$ and the Tail is simply an empty Block with $free\_space = Block_{size}$, however, it's not possible to have a completely empty Tail Block in the algorithm, because every Tail Block was once a part of the reserving threads GuessQueue and no thread would allocate an empty Block and link it if it didn't need that space at all. So the worst case is actually $free\_space = Block_{size} - 1$, where the Block is empty but has 1 byte of data reserved. How much is trimmed is also dependent on how filled up the Tail of the GuessQueue is as well, $TC_{GuessQueue}$. We get the worst case here when $TC_{GuessQueue}$ is as small as possible as well, so $TC_{GuessQueue} = 1$. Now we can see that in the worst case with the largest trimming is $free\_space = Block_{size} - 1$ and $TC_{GuessQueue} = 1$. Here we can see that still there will be only one Block trimmed from the GuessQueue. as if there's only one Block in the GuessQueue with $TC_{GuessQUeue}$, as in the general case, we will remove it in its entirety and increment the TC by 1. If there are > 1 Blocks in the GuessQueue, the TC will be filled up, one Block will be trimmed from the GuessQueue and the new Tail of the GuessQueue will have $TC_{GuessQueue} = 2$.

Note: It's important to realize that even if there have been multiple Blocks linked between the Snapshot and Trimming phase, it doesn't change this property, the only state of the Queue that determines how much needs to be trimmed is the TC it does not make a difference *which* Tail it is.

ABA Problem

Without having a garbage collector, or any form of *safe memory reclamation* framework, SOQ could be susceptible to the ABA problem. In the context of DBX and the real use case of SOQ, the ABA problem is impossible. The reason is that the consuming single-threaded side that pops elements from the Head of SOQ will never catch up to the producing Tail of the queue. This is because the MVCC mechanisms always maintain at least one live version (meaning there will always be at least 1 Block in the Queue). However, SOQ offers additional protection for ABA in case of other use cases of the structure. As mentioned in the background, the most common approach to resolve ABA is a counter that gets updated with each ABA-sensitive CAS. This requires more exotic atomics than the well-supported unary CAS. So SOQ applies the same Pointer Packing trick to achieve this. The atomic Tail pointer which is also 32-bit, becomes half pointer and half 32-bit counter, the same goes for the atomic Head pointer. This simulates having exotic architecture, avoiding ABA.

5.4 Get and Put

We have now covered how to asynchronously reserve memory in SOQ, what remains to understand is how the different threads access and actually use the reserved space. This is done mainly through two methods, Get(ChunkIndex) which retrieves the Chunk stored at `ChunkIndex` and Put(ChunkIndex, data) which stores the `data` in the Chunk allocated at `ChunkIndex`.

5.4.1 Memory Layout

On the system level, the only interaction a thread can have with its data in the SOQ is through the `ChunkIndex` abstraction. Internally, the `ChunkIndex` contains its own 32-bit pointer to the start of a Chunk in the SOQ. This means that some metadata needs to be stored in SOQ itself. The first is the *Chunk Header*, this is simply a 2-byte header denoting the size of the Chunk in the Queue. Using Theorem 2. we can guarantee that this Header never overspills by making the completely reasonable assumption that all Chunks are of even size. The header is written to the Block right before the ChunkIndex is returned by the method ChunkIndex as seen in Algorithm 7. This way we don't have to store the size excessively in the ChunkIndex. The *Block Header* contains the metadata pertaining to each Block, the most relevant being the NextCursor, and is written to the Block at the creation of the GuessQueue.

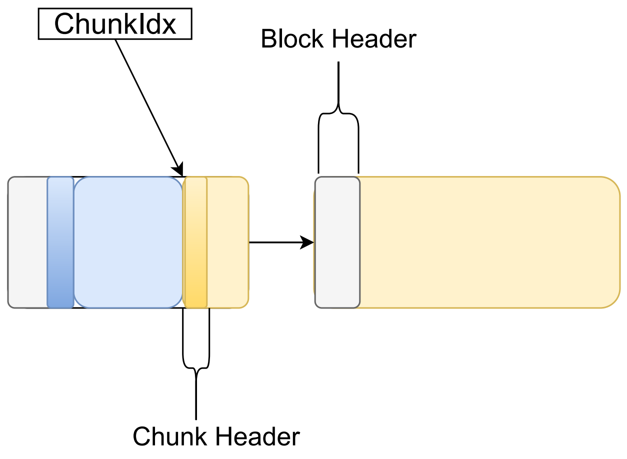


Figure 5.19: Anatomy of a Block

5.4.2 Get and Put Implementation

Get(ChunkIndex) takes the `ChunkIndex`, dereferences the pointer it contains to find the Chunk Header, it then reads the size of the Chunk. It creates an intermediate byte-buffer data structure to store the actual contents of the Chunk and returns it. When it hits the end of the Block, as the Chunk overspills, it will read the NextCursor and follow the Next pointer to the next part of the Chunk. Put(ChunkIndex, data) also starts from the Chunk Header but does the opposite, gradually writing the `data` to the allocated space, until it needs to jump to another Block with the Next pointer.

5.4.3 Minor Optimizations

Alignment Trick

Once SOQ receives the `ChunkIndex` in the Get and Put methods, the methods know where the Chunk starts and the size of the Chunk by using the pointer. However, it doesn't know where the Block ends, and further, it doesn't know where the NextCursor is stored. This can be resolved by storing this data in the `ChunkIndex` abstraction along with the pointer to the start of the Chunk, however, with the tight memory requirements this redundancy is not acceptable. SOQ resolves this by exploiting memory alignment. Whenever a Block is stored in memory, allocators usually ensure that it's aligned to its size, meaning that the address it's stored at is a multiple of its size. With the memory pool only allocating a single size, all Blocks are guaranteed to be aligned.

If $Block_{size}$ picked are always powers of 2, which they are by convention, the nearest aligned Block can be found by taking the `ChunkIndex` pointer and zeroing the last $\log_2(Block_{size})$ bits. This gives a pointer to the start of the Block which allows SOQ to find the NextCursor as well as find the end of the Block by offsetting by the $Block_{size}$.

Avoiding Spillover When Wanted

The reconstruction of the Chunk might be an expensive process for small Chunks. For example in DBX, a common Chunk to store in SOQ is a 32-bit (4-byte) pointer to another Chunk. If this Chunk overspills between two Blocks, merely fetching it through its `ChunkIndex` would be a proportionately expensive process.

1. Dereference the pointer stored in the `ChunkIndex` to find the *Chunk Header*
2. Read the 2 byte *Chunk Header*.
3. Instantiate an intermediary byte buffer vector, `vector<char>[ChunkSize]` or array to store the Chunk.
4. Read the bytes in the current Block.
5. Atomically load the *NextCursor*, split it to find the *Next* pointer.
6. Dereference the *Next* pointer to find the rest of the Chunk.
7. Read and store the bytes of the rest of the Chunk in the intermediary data structure and return it.

In this case, we have multiple pointer dereferences, the instantiation of a `vector` and an `atomic_load` just to read a spilled over 32-bit (4-byte) pointer from SOQ. These are unavoidable costs when fetching data that has spilled over and when dealing with large Chunks these costs are proportionately insignificant. For small Chunks, if we want to avoid these costs for a certain size, eg. 4 bytes, we can ensure that any Chunk of size 4 will **not** spillover. This allows a user of SOQ to read the 4-byte pointer directly from SOQ safely knowing that it's contiguous in memory.

Let's consider the smallest allocation size that can be available and the smallest ***free space*** in SOQ to be the *granularity*. If we want 4-byte pointers to never spillover, we need to ensure that the granularity is 4. Because there can never be ***free space*** finer than this.

We achieve this is by ensuring that the *Greatest Common Divisor* or gcd of the $\boldsymbol{Block_{size}}$ and all Chunks of size $\boldsymbol{Chunk_{sizes}}$ is equal to the wanted granularity. In the short pointer case, $\mathrm{gcd}(\boldsymbol{Chunk_{sizes}}, \boldsymbol{Block_{size}}) = 4$.

Theorem 2. *Any set of Chunks of varying sizes, $\boldsymbol{Chunk_{sizes}}$, stored in any order, in a set of Blocks of size $\boldsymbol{Block_{size}}$ will have a granularity equivalent to $\mathrm{gcd}(\boldsymbol{Chunk_{sizes}}, \boldsymbol{Block_{size}})$.*

Proof. Any ***free space*** is created by subtracting a $\boldsymbol{Chunk_{size}}$ from either a Block of $\boldsymbol{Block_{size}}$ or any other previous ***free space***. By the definition of gcd any $\boldsymbol{Block_{size}}$ and any $\boldsymbol{Chunk_{size}}$ can be re-written in the form of a ***whole number*** multiple times the gcd. So Block might be written as size $5 \times \mathrm{gcd}$ while a Chunk is $2 \times \mathrm{gcd}$, and by subtracting and adding multiples of a gcd, one can never create a number smaller than that gcd. Therefore no ***free space*** can be created smaller than gcd.

Thus, if a user of SOQ explicitly wants to avoid 4-byte pointers spilling over, it is enough to ensure that the gcd of all Chunks added and the ***Block**$_{size}$* is 4. The way this is achieved is by padding any allocated Chunk to the nearest multiple of 4. In this case, the worst-case wasted space and largest padding is 3 bytes which is technically a slight memory trade-off, but 3 bytes is an insignificant size when compared to cache-line sizes, meaning the Chunks probably get padded to 32 bytes anyway and the gcd is probably already 32.

Chapter 6
Evaluation

Using experiments we evaluate the performance of SOQ and compare it to an implementation of a modified MSQ. The relation between the $\boldsymbol{Block_{size}}$ and the $\boldsymbol{Chunk_{size}}$ determines a lot about the internals of SOQ, it determines how many Blocks the GuessQueue contains, how big the allocations need to be, the probability of each type of and size of ***trimming***, how many spillovers and how often the general case will happen. For benchmarking purposes, we created an MSQ which has the same operations as SOQ, it can however allocate variable-sized Blocks using the dynamic allocator `malloc`. It has the same design as the variable-sized Blocks and Chunks shown in figure 4.1. This could be a drop-in replacement for SOQ which doesn't have fixed $\boldsymbol{Block_{size}}$. We also investigate how the data structure deals with high contention as we see the scaling of throughput with a higher number of threads. The metric we use is throughput (operations per second) or $\boldsymbol{Op/s}$ where a higher $\boldsymbol{Op/s}$ is a more performant data structure. The SOQ experiments are limited in their execution by the time it takes for the memory pool allocator to run out of pages. Since the allocator is designed for small memory systems, this happens after 4GB of memory has been requested in total.

Both the MSQ and SOQ are written in Rust and the experiments run on an x86-64 Intel i7-1165G7, with 4 cores and 8 threads, with an L1 cache of 312KB and 64-byte cache-lines. The source code is compiled using the Rust nightly toolchain version 1.66.0 using sequential consistency in its memory model.

6.1 Results

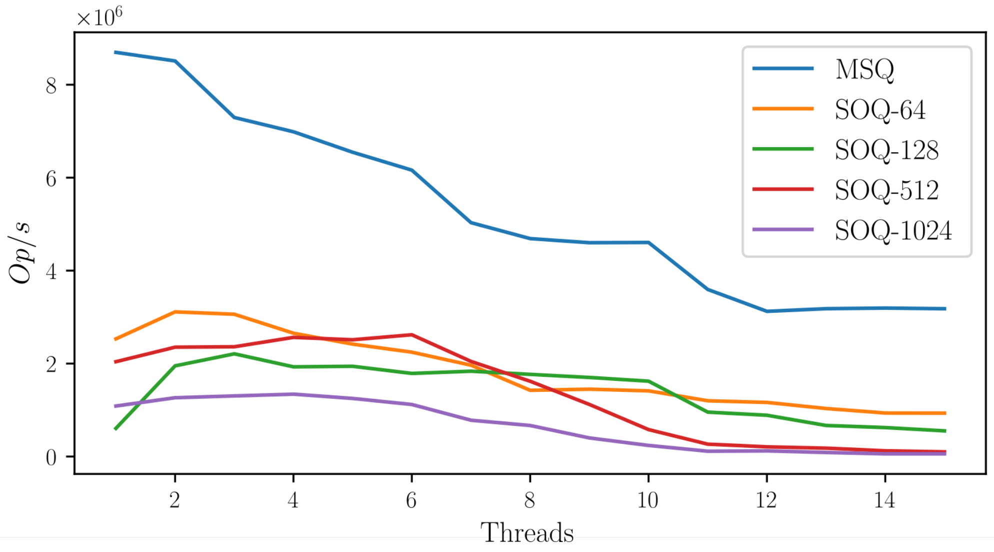


Figure 6.1: Throughput compared to the number of threads for SOQ- $\textbf{\textit{Block}}_{size}$ and an MSQ reserving 64 bytes.

Figure 6.1 shows various SOQ instances with different $\textbf{\textit{Block}}_{size}$ compared to the benchmarking MSQ implementation. The MSQ uses Rust's `malloc` equivalent, `std::alloc` while the SOQ uses the memory pool. Here, each thread reserves Chunks of size 64 bytes, in a loop 800,000 times concurrently and timing finishes when all threads have completed their operations. We are only able to have each thread iterate 800,000 times as for high thread counts, the memory pool runs out of space.

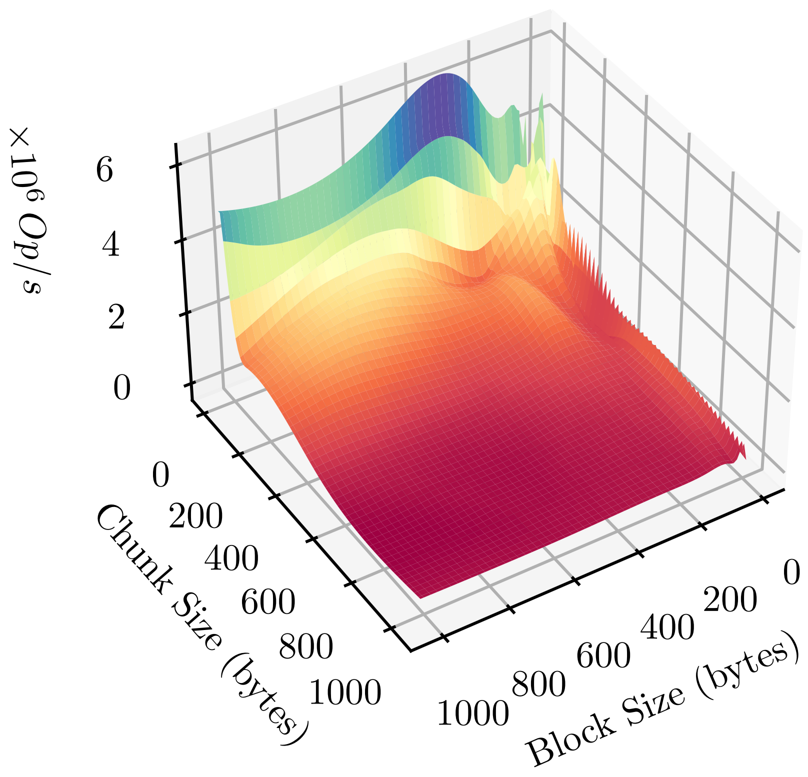


Figure 6.2: Throughput compared to ***Block***$_{size}$ and ***Chunk***$_{size}$

In figure 6.2, we instantiate SOQ's of varying ***Block***$_{size}$, we then spawn 4 threads that will concurrently and in a loop reserve Chunks of ***Chunk***$_{size}$. The iteration occurs 202,400 times per thread, reserving a ***Chunk***$_{size}$ each time. 202,400 was also the maximum number of per thread iterations that was possible before the memory pool ran out of space. The time to complete these reservations is taken once every thread is completed, which allows us to compute the throughput.

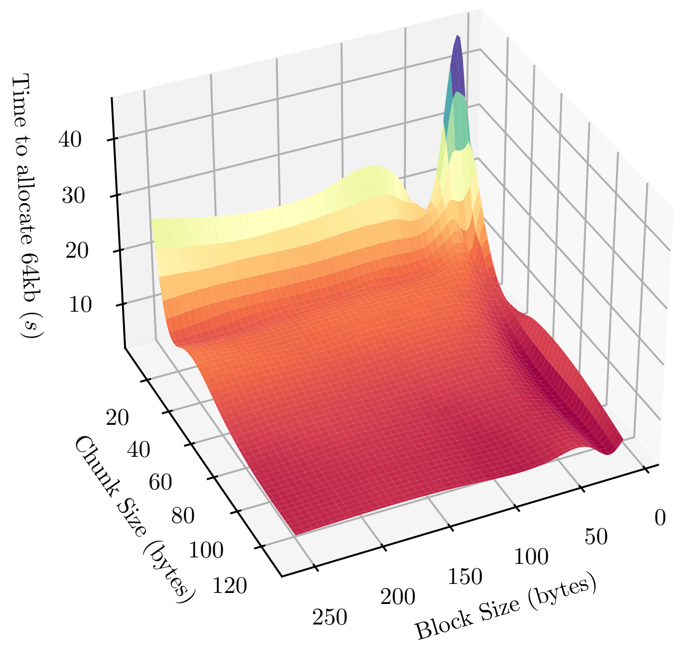


Figure 6.3: Total time to reserve 64KB in ms

Figure 6.3 shows how long it takes to reserve a total size of 64KB by varying the $\textbf{\textit{Block}}_{size}$ and $\textbf{\textit{Chunk}}_{size}$ dimensions. Here we compute how many Chunks are required for each $\textbf{\textit{Chunk}}_{size}$ to reserve 64KB by each thread, the threads then continuously attempt to reserve that $\textbf{\textit{Chunk}}_{size}$ until they have reserved a total of 64KB.

6.2 Discussion

Contention

In figure 6.1, we see that both MSQ and SOQ have drastically reduced performance under high contention. SOQ under-performs MSQ in throughput only achieving about 35% of the throughput of MSQ at best. Worse throughput is however expected as MSQ is a subset of SOQ in terms of computations and in terms of what it does, MSQ has no Batching, Spillover, GuessQueue, Trimming, Resets, etc. Importantly we see that the throughput of SOQ is at its highest at around 2-4 threads, which is the thread count of the target system DBX. However, a peak of 3m operations per second is still considered a highly successful throughput, as the DBX system deals with an order of magnitude fewer transactions per second. With one transaction being one operation, SOQ will safely not be a bottleneck in the system. It is surprising however how the MSQ only loses throughput as the thread count increases. We believe this is due to the fact that Rust's `std::alloc` is not lock-free. Also, a single-threaded sequential program actually gets a lot of help from the hardware in terms of pre-fetching and more successful branch prediction.

Throughput

6.2 shows how throughput changes as we vary the ***Block**_size_* for differently large Chunks being reserved. Throughput decreases with larger Chunks being reserved, this is not very surprising as larger Chunks will in general require creating GuessQueues with more linked Blocks. Although, to our surprise, we previously hypothesized that larger ***Block**_size_* would allow a higher throughput because we would more often hit the *general case* of just increasing the Cursor as seen in Figure 5.9. The data however suggests that ***Block**_size_* has no significant impact, the throughput seems mostly independent of the ***Block**_size_* . We theorize that this is due to an unforeseen limitation in SOQ, the pessimistic GuessQueue will always create at least one Block no matter the ***Chunk**_size_* , so with a small ***Chunk**_size_* and a large ***Block**_size_* this GuessQueue will be a large Block even though the probability that the GuessQueue will be Commited and not trimmed is low due to the fact that there is probably being plenty of ***free space*** since the Blocks are big. However, even with this considered it's still surprising that there is no correlation between ***Block**_size_* and throughput. Larger Blocks will require smaller GuessQueues to be constructed which will be faster as there's less linking etc. This is probably due to another unforeseen limitation, that of the ***memory pool allocator***. One would expect the allocator to hand out Blocks with a complexity not proportionate to the ***Block**_size_* , since at instantiation it knows all sizes it needs to allocate, this does not seem to be the behavior.

This is probably due to the underlying system below this allocator, there is probably no such performance gain within the size of a page. However, this independence is also a testament to the efficiency of the GuessQueue + Trimming combination since we also don't see a performance *loss* for large Chunks with small ***Block**_size_* , the fact that the majority of the GuessQueue does not have to be re-allocated in every loop makes the algorithm proportionately fast even in this case.

In figure 6.3 we see the time required to reserve 64KB plummets as the ***Chunk**_size_* increases. This is because larger Chunks require fewer operations to reserve a certain amount, so when we increase the ***Chunk**_size_* we don't need to call RESERVE as many times. Again we see that for larger ***Block**_size_* the ***Block**_size_* seems to have no effect on the performance. Although a larger ***Block**_size_* is still more memory efficient, since we need to pad every Block with the Block Header, we want as few Blocks as possible.

Longer Experiments

As mentioned, the experiments are limited by the amount of memory that the 32-bit memory pool can allocate. We would like to do more robust testing, to see if for example, SOQ can run for weeks without hitting a seg-fault. However, in order to do such extensive, long-running testing, we need a concurrent garbage collector to run and free memory from the head of the Queue. However, this means that the garbage-collecting thread can catch up with the reserving threads and free a Block that is about to be updated. This race condition can only be resolved by using a different memory-managed language, or by the manual implementation of some form of safe memory reclamation. Because this race condition is not possible in the context of DBX, and safe memory reclamation is very time-consuming and difficult to implement, this has been left out of the scope of this thesis.

Chapter 7
Conclusion

Here we discuss some of the problems we faced when designing and implementing SOQ, as well as thoughts about its contribution, this is hopefully useful to anyone working on lock-free memory allocators or on lock-free Queues and wishes to extend the work or use a SOQ or some ideas from it in their system. Furthermore, we will discuss some of the future work that can be done to improve SOQ.

7.1 Implementing a SOQ

There is a general consensus within the concurrent programming community that one should avoid implementing their own lock-free data structures unless it is truly necessary. To do so anyway will often result in at best, a big waste of development time and at worst, dangerous subtly unsafe code. It is considered a case of Donald Knuth's root of all evil, *premature optimization*. If anything, this holds doubly true when it comes to SOQ. SOQ is optimized for both lock-free concurrent performance, and for low memory usage, these two fields rarely overlap but when they do, implementations can get extremely hairy. An SOQ implementation deals with careful pointer arithmetic, bit-wise memory operations on pointers that are cast pointers between to and from integers and then updated with CAS. All these things are hard to reason about as a programmer leading to extremely subtle, hard-to-find bugs in incorrect implementations. For example, if a programmer's arithmetic is slightly off when computing the ***new_next_cursor***, the Cursor part might exceed its 32-bits and overflow into changing the Next pointer part, creating a NextCursor with a completely random Next pointer. This might only happen for some edge cases where the GuessQueue is n long and the size is x and the ***free_space*** is y with just the right amount of trimming needed. This means a faulty SOQ implementation could likely work for years before hitting a segmentation fault. Even worse, this might overflow the pointer to point to a consecutive Block which makes it even more confusing. However, unlike working with regular integer overflow, there is no way to detect this in the run-time, this pointer might only seg-fault much later on in the program, making it even harder to realize what the edge case was that caused it. Further, this all happens concurrently, so it's also hard to replicate the exact state that caused it. There's also a lot of tedious casework and many edge cases abstracted out in the TRIM method, if a Block needs to be trimmed or not, if the GuessQueue even contains more than one Block, also where the Chunk Header should be written, there's a case where there is no ***free space*** at all in which case we need to create and link the GuessQueue but not spillover. All this meant it took us 3 months to get the first working implementation of SOQ which is only about 1000 lines of code, where the 500 line RESERVE took the majority of the time. However, in the context of the DBX system, SOQ is not a case of premature optimization, with no mutexes and while dealing with very small memory and fixed size heap allocations only, SOQ exploits its small memory environment and 32-bit pointers with the NextCursor creating a lock-free data structure that would be nearly impossible to create otherwise but is still highly required by the system.

It is worth mentioning that there is nothing special about exactly 32-bit pointers, SOQ can also be implemented with one of the exotic atomic primitives, namely DWCAS. DWCAS operating on 128-bit contiguous memory could fit a standard 64-bit Next pointer as well as a Cursor and even an ABA counter if needed, all depending on the requirements of the system. This does make SOQ a more general data structure than one only possible in embedded systems, especially since DWCAS is the most commonly supported exotic CAS.

SOQ is only a Queue in its very fundamental structure, but on a high level, it's actually more of an allocator itself. RESERVE is just a way to allocate memory as well. More specifically it closely resembles a *linear* memory allocator, one that can only free memory in order. If you turn the Blocks requested from the memory pool allocator into requests for pages from the OS, you don't need a custom memory pool but SOQ becomes an allocator itself.

7.2 Future Work

There are still plenty performance improvements that could be applied to SOQ, as mentioned previously, we always generate a GuessQueue with a single Block even if the Chunk is very small. This is an example of the GuessQueue being too pessimistic. One area to explore is that of a *fast path* for small Chunks. Here, below some threshold, small enough Chunks could read the ***free space*** and attempt directly go for a general case Cursor increment if possible, and otherwise, they re-attempt with a traditional GuessQueue. The fast path could even be attempted multiple times before the threads run out of patience and default to the GuessQueue. In this situation, a reserve call will start off by being optimistic, and then when it runs out of patience, resort to being pessimistic.

Another idea when it comes to sizing the Blocks can be borrowed from how `vectors` are implemented internally. When they are instantiated they allocate some initial size and when this memory is used up, they will create a new allocation *double* (or sometimes other factors like 1.5) the size of the previous. This is due to the idea that if a program has used x memory it will probably want to use x more, it would be bad to have many small increments of the size and call the allocator many times. We could borrow this for SOQ, have an initial $\boldsymbol{Block}_{size}$ and then when a reservation comes in that requires spillover and a new Block, would allocate a new Block double the $\boldsymbol{Block}_{size}$ of the previous. This obviously would require modifications to the ***memory pool allocator*** but it's not much more difficult to build such an allocator. Another area that hasn't been discussed in this work is that of memory consistency models. Currently, SOQ is only implemented with Sequential Consistency at all atomic operations. However, relaxing the consistency would give the compiler more freedom to optimize and could grant some performance benefits. Another theoretical aspect to work on is proving that Reserve is correct and *linearizable* which was excluded from the scope of this industry-focused work.

It would also be interesting to figure out a way to turn Spillover Queue into a Multiple-Consumer Queue as well. In the context of DBX, there was only one garbage collector that could consume the Queue, but to make SOQ more general we could try to extend it to have many consuming threads. One would have to be very careful when freeing the Blocks since many threads can have Chunks on the same Block. We also theorize that there could be a lot of performance benefits from trying to use the wait-free `fetch_and_add` (FAA) atomic when incrementing the Cursor, which needs to be done without reading the Cursor. However, it's not completely clear how such an algorithm would work, because when the FAA succeeds, which it always will, we might have incremented the Cursor beyond the available ***free space***, in which case we should also link a Block. But there is a race condition between incrementing the Cursor, realizing we need to link a Block, and linking the Block. This is the problem with FAA not being conditional.

7.3 Final Words

This thesis introduces and evaluates the novel lock-free data structure Spillover Queue. We explain how the design and features of SOQ fit the very strict requirements demonstrated in Section 3.1 and the RESERVE algorithm is able to solve the various race conditions with its NextCursor, how it manages to enqueue many Blocks concurrently without blocking using its GuessQueue and can still use concurrent assistance to correct itself. We see that the performance in terms of throughput is high enough to not be a bottleneck in the system as it reaches upwards of 6m Op/s. We explain how it manages to use a highly efficient memory pool allocator even with the major limitation of fixed Block-sizes, in order to completely avoid fragmentation in memory.

In conclusion, we succeeded with the goal of this thesis and introduced some new ideas in the space of lock-free queues.

References

[1] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, page 267–275, New York, NY, USA, 1996. Association for Computing Machinery.

[2] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[3] Peng Zhang. Chapter 5 - microprocessors. In Peng Zhang, editor, *Advanced Industrial Control Technology*, pages 155–214. William Andrew Publishing, Oxford, 2010.

[4] Edya Ladan-Mozes and Nir Shavit. An optimistic approach to lock-free fifo queues. volume 20, pages 117–131, 12 2004.

[5] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. Evaluating the cost of atomic operations on modern architectures, 10 2020.

[6] Ole Agesen, David Detlefs, Christine Flood, Alexander Garthwaite, Paul Martin, Mark Moir, Nir Shavit, and Guy Steele. **dcas-based concurrent deques. *Theory Comput. Syst.*, 35:349–386, 06 2002.**

[7] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, page 1–10, New York, NY, USA, 1995. Association for Computing Machinery.

[8] Thomas J. Watson IBM Research Center and R.K. Treiber. *Systems Programming: Coping with Parallelism*. Research Report RJ. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[9] Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, 2013.

[10] Chaoran Yang and John Mellor-Crummey. A wait-free queue as fast as fetch-and-add. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, New York, NY, USA, 2016. Association for Computing Machinery.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2023-01-20

EXAMENSARBETE SOQ: A Novel Lock-Free Queue with Variable-Size Storage In Fixed-Sized Blocks
STUDENT Marcus Begic
HANDLEDARE Jonas Skeppstedt (LTH)
EXAMINATOR Nikos Ntarmos (Huawei Research)

Spillover Queue: En Ny Låsfri Kö

POPULÄRVETENSKAPLIG SAMMANFATTNING **Marcus Begic**

Databaser sparar flera versioner av samma data för att säkra att alla läsare alltid ser en version som ingen annan håller på att skriva i samtidigt. De gör det i en stor köliknande versionskedja. Detta arbete introducerar en ny, minneseffektiv och låsfri datastruktur för lagring av versionskedjor särskilt i databaser med litet minne.

Om någon skriver data i en databas samtidigt som någon annan läser från databasen kan stora problem uppstå. Om en databas lagrar information om en bank, och pengar överförs från ett bankonto till ett annat, hade man kunnat se balansen i bägge kontona efter att pengar tagits ur det ena men innan det lagts till det andra. Det hade verkat som om pengarna försvunnit.

För att lösa detta, lagrar en databas flera kopior av datan samtidigt, då kan en skriva till sin kopia samtidigt som någon annan läser från en annan. Dessa kopior kan vara av alla möjliga storlekar och lagras en efter en i en kedja av versioner, liknande en kö. Versionskedjor kan bli minnesdyra eftersom multipler av all data i databasen kan behöva lagras i dem. I stora system lägger man bara till mer minne i systemet, men i inbyggda databaser, som hittas i hårdvara som säljs i miljontal, vill man undvika att göra minnet större för att tillverkningen kan bli väldigt kostsam.

Ett sätt att använda mindre minne är att endast allokera minnesblock av samma storlek. Datorer gillar detta då alla minnesblock kan lagras ett efter ett, vilket inte slösar lika mycket plats som om de alla vore av olika storlekar och inte riktigt passade i minnet. Vår datastruktur, Spillover Queue (SOQ) gör just detta, den allokerar endast minnesblock av en viss storlek. Dock kan versioner vara av alla möjliga storlekar, då SOQ löser detta genom att låta en version spilla över flera minnesblock. Flera kan vilja skriva sina versioner till SOQ samtidigt. Detta brukar lösas genom att tvinga en skrivare vänta på en annan, genom s.k. blockering. Blockering är dock långsamt och kan leda till många problem. SOQ är därför helt icke-blockerande, en avancerad teknik som också kallas låsfri programmering. Detta ger SOQ en mycket högre prestanda vilket kan göra hela databasen snabbare.

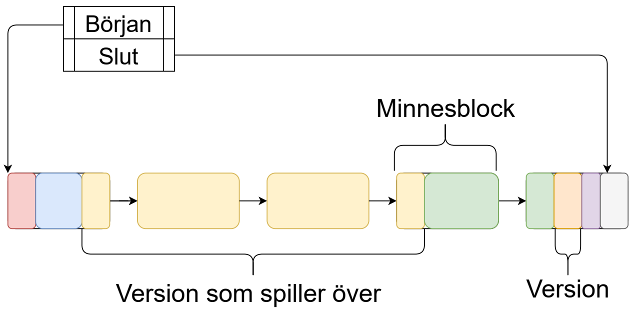


Bild: En SOQ med versioner i en databas

Resultatet visar att SOQ kan hantera en magnitud fler samtidiga skrivare än en stor databas kan ha, och är därför inte en flaskhals i systemet. SOQ introducerar även nya tekniker inom låsfri programmering som utnyttjar miljöer med litet minne, vilket annars nästan alltid är en nackdel.