On Building Modular and Elastic Data Structures with Bulk Operations

Kevin Williams, Joe Foster, Athicha Srivirote, Ahmed Hassan, Joseph Tassarotti*, Lewis Tseng*, Roberto Palmieri
Lehigh University, Boston College*
{kww220;jof220;ats219;ahh319;palmieri}@lehigh.edu - {tassarot;lewis.tseng}@bc.edu

ABSTRACT

This paper introduces MEDS, a modular and elastic framework that simplifies the development of high-performance concurrent data structures that support linearizable primitive (i.e., add, remove, contains) and bulk (e.g., range query) operations.

CCS CONCEPTS

• **Theory of computation → Data structures design and analysis**; • **Computing methodologies → Concurrent computing methodologies**.

KEYWORDS

Concurrent Data Structures, Concurrency, Range Queries

ACM Reference Format:
Kevin Williams, Joe Foster, Athicha Srivirote, Ahmed Hassan, Joseph Tassarotti*, Lewis Tseng*, Roberto Palmieri. 2021. On Building Modular and Elastic Data Structures with Bulk Operations. In *International Conference on Distributed Computing and Networking 2021 (ICDCN '21), January 5–8, 2021, Nara, Japan.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3427796.3433932

1 INTRODUCTION

High-performance concurrent data structures are complex to design, develop and prove to be correct [5, 6]. The recent growing interest for (linearizable) bulk operations, such as range queries [1], exacerbates such complexity even further. Theoretical frameworks have been proposed to ease the process of proving safety of concurrent data structures but these frameworks either lack support for bulk operations or have assumptions that might hamper their practical deployment (e.g., single writer assumption [4, 6]).

Automated generation of data structure implementations is a recent trend that simplifies the development of concurrent data structures. With this technique, programmers are required to only provide an efficient *sequential* implementation of the data structure and the framework is responsible for allowing concurrent accesses to this sequential implementation, without giving up performance. Existing solutions (e.g., Node Replication [2]) take conservative design choices that mostly fit data structures with conflicting semantics (e.g., Queues), but might perform poorly in other data structures, such as Sets or Maps. Also, these solutions are not designed to support highly concurrent bulk operations.

In this paper, we introduce *MEDS*, a framework to develop *Modular* and *Elastic* concurrent *Data Structures* that implement a Set abstract data type interface. The modularity property is provided by designing the MEDS data structure as a composition of elemental, plug-and-play, building blocks whose integration is given by the MEDS framework. The core building block for MEDS is a sequential data structure implementing an ordered Set that supports a sequential version of both elemental and range query operations. The elasticity property allows MEDS data structures to reconfigure the way the building blocks are composed in order to chase a performance-effective configuration that favors the current application workload mix, contention level, and number of threads.

MEDS provides a solution to the open problem of re-engineering a highly optimized sequential data structure (e.g., handcrafted to exploit some application specific characteristics) to be concurrent, without losing optimization. MEDS does so by treating the sequential data structure as a black-box, allowing programmers to select the most appropriate version for the target application.

2 THE MEDS DESIGN

Figure 1 shows the MEDS architecture. MEDS uses the elemental building block given by a programmer to build a *data layer* composed of a set of *partitions*, each of which stores elements belonging to a certain contiguous range of the total keys currently maintained in the data structure. Ranges of different partitions are disjoint.

An application thread invokes a primitive operation on a key K using the MEDS Set APIs, extended to support bulk operations. Control passes through a *routing layer* whose goal is to select the partition responsible for storing K. Partitions access is orchestrated by a *synchronization layer* that implements Single-Writer/Multiple-Readers (SWMR) semantics in order to prevent multiple writers from modifying the same partition, concurrently. In the absence of readers or bulk operation, writers on different partitions proceed concurrently. A bulk operation, such as a range query where an atomic snapshot of the data structure is returned, is handled similarly except that the synchronization layer must coordinate activities over multiple partitions (e.g., using Two-Phase Locking [5]).

A key design choice of MEDS data structures is that partitions are *not* statically defined; instead, elements in one partition can be split into two partitions, and multiple contiguous partitions can be merged into a single one. This capability enables the elasticity property of MEDS data structures. Elasticity is needed for high performance since bulk operations and primitive operations both

ICDCN '21, January 5–8, 2021, Nara, Japan

ACM ISBN 978-1-4503-8933-4/21/01…$15.00
https://doi.org/10.1145/3427796.3433932

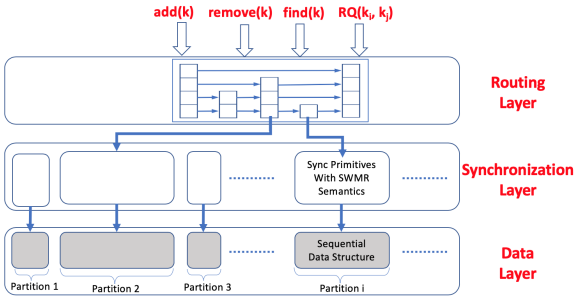

Figure 1: The MEDS Architecture. The range query (RQ) API is an example of a bulk operation. Due to the local merge/split decisions, partitions can have different sizes.

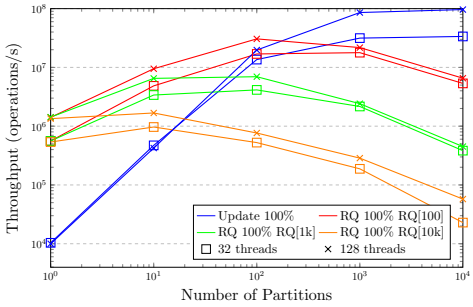

Figure 2: The effect of varying the number of partitions on MED's performance with workloads of 100% updates and 100% RQs (with RQ size = 100, 1K, and 10K). Data structure size is 10K.

find different ideal configurations in terms of number of partitions. On the one hand, in the absence of bulk operations, MEDS would maximize the number of partitions in order to reduce the granularity of conflict on partition accesses for primitive operations. On the other hand, for bulk operations, MEDS would minimize the number of partitions in order to decrease synchronization overhead. MEDS allows for merging and splitting partitions in order to chase high performance in workloads where the operation mix is not known a priori and might vary over time.

The following components characterize a MEDS data structure:

Routing layer. This layer implements a search data structure that serves as an index over partitions The goal of this layer is to efficiently map keys to their corresponding partitions and maintain an ordered list of partitions for serving bulk operations. Moreover, the modular design of MEDS enables changing the actual implementation of the routing layer to best perform in different deployments (e.g., NUMA-aware indexing similar to the one presented in [3]).

Synchronization layer. This layer controls thread access to partitions. The current MEDS design accepts any synchronization primitives that guarantee SWMR semantics, which is required given the underlying sequential design of partitions. We specifically focus on two well-known primitives, namely readers-writer lock [5] and RCU [7]. Having both options ensures better performance under different workloads (e.g., RCU is preferred in read-dominant workloads since readers are not blocked by writers).

Data layer. Each partition is a sequential data structure with the additional support of two operations, split and merge. During a split/merge, the MEDS framework stops accesses to the target partition(s) and invokes the respective functions to fulfil the restructuring of the partitions. The implementation of the split/merge functions can be either provided by the programmer without significant developing burdens since their design is sequential, or can be automatically implemented. In fact, MEDS can use the insert/remove APIs of the sequential data structure to migrate the necessary keys between partitions to fulfil the split/merge. Once a split/merge decision is triggered, the data layer in cooperation with the synchronization layer performs the operation. Wherever the MEDS data structure is not rearranging the number of partitions, operations on different partitions do not require any interaction.

Merge/Split Heuristics. MEDS relies on heuristics to trigger merge/split decisions. Those decisions are *local*, meaning a thread working on a partition can independently execute a merge/split operation involving that partition (for a split) and contiguous partitions (for a merge). As a result, at any given time, partitions in the data layer can have different sizes. The synchronization layer is responsible for collecting per-partition performance indicators to trigger those merge/split operations. We argue that the goal of an effective heuristic is to use these indicators to promote more partitions when conflicting primitive operations dominate the workload, and less partitions when long bulk operations dominate.

To confirm the validity of this argument, we conduct a preliminary experiment on a MED data structure populated with 10k keys, where each partition is implemented by a sequential linked list. Accesses to partitions are protected using readers-writer locks. Figure 2 plots the performance by varying the number of partitions. When the workload is dominated by update operations, increasing the number of partitions improves performance. This is expected since having more partitions reduces contention and highlights the indexing effectiveness of the routing layer. On the other hand, when the workload is dominated by range queries, performance degrades as the number of partitions increases. Also, increasing the size of the range query magnifies the effect of partitioning.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under grants no. CNS-1757787 and no. CNS-1814974.

REFERENCES

[1] Maya Arbel-Raviv and Trevor Brown. 2018. Harnessing epoch-based reclamation for efficient range queries. In *ACM SIGPLAN PPoPP*. 14–27.
[2] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. 2017. Black-box Concurrent Data Structures for NUMA Architectures. In *ACM ASPLOS*. 207–221.
[3] Henry Daly, Ahmed Hassan, Michael F. Spear, and Roberto Palmieri. 2018. NUMASK: High Performance Scalable Skip List for NUMA. In *DISC*, Vol. 121. 18:1–18:19.
[4] Ahmed Hassan, Roberto Palmieri, Sebastiano Peluso, and Binoy Ravindran. 2017. Optimistic Transactional Boosting. *IEEE Trans. Parallel Distributed Syst.* 28, 12 (2017), 3600–3614. https://doi.org/10.1109/TPDS.2017.2725837
[5] M. Herlihy and N. Shavit. 2008. *The art of multiprocessor programming*. Morgan Kaufmann.
[6] Kfir Lev-Ari, Gregory V. Chockler, and Idit Keidar. 2014. On Correctness of Data Structures under Reads-Write Concurrency. In *DISC (LNCS, Vol. 8784)*. 273–287.
[7] Paul E McKenney and John D Slingwine. 1998. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*. 509–518.