Brief Announcement: LIT: Lookup Interlocked Table for Range Queries

dePaul Miller
dsm220@lehigh.edu
Lehigh University
Bethlehem, PA, USA

Ahmed Hassan
ahh319@lehigh.edu
Lehigh University
Bethlehem, PA, USA

Roberto Palmieri
palmieri@lehigh.edu
Lehigh University
Bethlehem, PA, USA

ABSTRACT

We introduce the Lookup Interlocked Table (LIT), a highly efficient data structure that facilitates get, update, and range query operations. LIT is designed to maintain the high performance of hashing algorithms while also preserving the order of data for range queries. It does that by utilizing an order-preserving lookup function to index data and providing the option to split and resize the indexing to adapt to changing workloads.

CCS CONCEPTS

• **Theory of computation → Data structures design and analysis**; **Concurrent algorithms**.

KEYWORDS

Data structure, Range Query, Concurrency

ACM Reference Format:
dePaul Miller, Ahmed Hassan, and Roberto Palmieri. 2024. Brief Announcement: LIT: Lookup Interlocked Table for Range Queries. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '24), June 17–21, 2024, Nantes, France.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3626183.3660266

1 INTRODUCTION

Range queries on *map* data structures are operations that iterate over the map and return a collection of key-value pairs where the key falls within a contiguous range. They are emerging as an essential API for data repositories with key-value semantics, including Redis [19], RocksDB [7], LevelDB [9], and others [8, 15, 18, 23], and for optimizing the internals of relational databases, which traditionally perform range queries through predicates [21].

Highly concurrent data structures that support range queries typically implement operations through costly traversals over linked data, such as the case of linked lists [2, 5, 17, 22], skip lists [1, 2, 5, 17], or B-trees [3, 4, 20]. However, it is well known that if range queries are not supported, hash tables [12] provide a method to limit the overhead of traversals by partitioning elements into *buckets* and index into them by means of a hash function that redirects the operation to the bucket where the requested element is stored. Because of the absence of long traversals and minimal housekeeping overhead, hash tables have been shown to scale better than linked data structures and provide higher performance and locality under a variety of workloads [10].

It is natural, then, to wonder if it is possible to retain the performance advantage of the hash table structure while still providing ordering between elements. Motivated by the above observation, in this brief announcement, we introduce the design of the *lookup interlocked table* (or LIT). LIT has the structure of a traditional hash table [12], where each bucket is implemented as a linked list of elements. However, rather than having a hash function that uniformly hashes keys to buckets, we utilize a *lookup* function that maps keys to buckets *and* preserves the ordering of keys (i.e., if $x < y$, then $f(x) \leq f(y)$, where x and y are keys).

In LIT, every operation starts by invoking a lookup function that identifies the bucket where the relevant element is located. This workflow applies to both elemental operations (i.e., get, remove, or insert) and range queries, where the lookup function determines the starting point of the range. After that, for range queries, subsequent buckets are visited by utilizing an overlay linked structure that connects all elements of all buckets.

To efficiently handle changing workloads, LIT takes inspiration from the interlocking structure of the interlocking hash table [14]. In fact, LIT operates through buckets that can map to another level of buckets, which helps reduce the number of traversals necessary to reach a node in case many elements happen to be inserted into a single bucket. This operation is called a "split". Over time, many split operations to a single bucket might lead to an increase in the traversal cost. LIT addresses this issue by efficiently flattening the structure to a single level to restore the original indexing performance and re-balancing elements. We call this operation "resize".

2 LIT: LOOKUP INTERLOCKED TABLE

The lookup interlocked table (LIT) is a key-value mapping data structure with support for get, insert, remove, and range queries. All operations in LIT are linearizable [13]. LIT uses a structure similar to the chained bucket structure of a hash table [16], along with a function that preserves the order of the keys.

In addition to partitioning elements into buckets, LIT also allows each bucket to be further split into more buckets recursively (interlocking design [14]). Every time a `SPLIT` occurs, a new level in LIT is created. Similar to the terminology used in skip lists [6], we call the collection of those levels the *index layer*. However, unlike a skip list, to map an element into the index layer, LIT utilizes a collection of lookup functions, one per level. Each lookup function f preserves the order of keys in its level such that if $x < y$, then $f(x) \leq f(y)$. In principle, any function that preserves this property works. The most simple function that preserves this property for numbers, is

SPAA '24, June 17–21, 2024, Nantes, France

ACM ISBN 979-8-4007-0416-1/24/06
https://doi.org/10.1145/3626183.3660266

a trivial linear function f, which interpolates a minimum key to a maximum key, where the function f at the minimum evaluates to 0 and at the maximum it evaluates to the largest bucket index.

Since each bucket is implemented as an ordered list and subsequent buckets are also ordered due to the just described lookup function, the order across subsequent lists is preserved. We interconnect all these nodes in the buckets, forming a linked list that we call *data layer*. To point a bucket to a location in this data layer, LIT uses *boundary-markers*. These special nodes have a key that matches the minimum key of the bucket they are linked to. The nature of this data layer allows for traversals when concurrent modifications occur.

In traditional hash tables, a good hashing function guarantees that elements in the structures are balanced among buckets. Since LIT does not rely on hashing, we deploy a different technique, which enables splitting buckets into other levels.

Below, we detail how LIT implements its operations. For the sake of clarity, in the next two subsections (2.1 and 2.2), we assume *no concurrent resizes or splits occur*. These two operations are then enabled in 2.3.

2.1 Bucket Lookup

All LIT operations, both elemental operations and range queries, start by invoking a search function that uses the index layer to reach the proper entry point to the data layer by returning a pointer to the boundary-marker of the bucket to start with. For elemental operations, this is the bucket to which the element belongs, and for range queries, this is the bucket to which the element with the smallest key belongs. Starting from Level 0 (i.e., the initial set of buckets). First, we read the root level and then utilize the level's lookup function to find the subsequent level. We repeat this until we find a pointer to a boundary-marker ordered before the sought key.

2.2 Bucket Operations

Starting from the boundary-marker returned from the above search method, multiple techniques can be potentially used to perform the operations on the data layer.

Recent literature proposed effective techniques to enable linearizable range queries for concurrent data structures while other operations can concurrently modify disjoint data [2, 17, 22]. We choose to use vCAS [22] because, unlike the other blocking techniques [2, 17] it enables the introduction of wait-free range queries to lock-free data structures. In our implementation, we port vCAS to the Harris lock-free ordered linked list [11], as originally done by [22] as well, with the necessary modifications to consider the fact that our list traverses through buckets of an enclosing index.

vCAS in a nutshell: In a vCAS data structure, all pointers that are updated using atomic operations are changed to be versioned CAS (vCAS) objects. These vCAS objects contain a pointer to a versioned node (VNode) and a reference to a global timestamp, which is called a Camera. Each VNode contains a value (e.g., the next pointer in a linked list node) and a timestamp, as well as pointers to earlier VNode versions of the same object. When modifying a data structure to use vCAS, all CAS operations are replaced with performing a CAS on the root VNode of the vCAS object. Two APIs can be used to read a vCAS object. One returns the current value by reading the VNode head of the vCAS object. The other returns the VNode whose version is consistent with a specific timestamp. Our implementation of vCAS in LIT changes both the indexing nodes in the index layer and the Harris list's nodes in the data layer to be vCAS objects.

Insert/Remove/Get Operations. Now we summarize the main changes we made to the elemental operations in our vCAS bucket list. We introduce boundary-marker nodes into the list. These nodes denote the beginning of a bucket, and are entries into the data layer. When traversing the data layer, these boundary markers are skipped, similar to how nodes marked as logically deleted are ignored. Boundary-markers are also versioned through vCAS like other nodes in the list.

Wait-Free Range Queries. Range query operations in LIT begin by traversing the data layer to the bucket associated with the first key in the range. Then, it uses vCAS to take a logical snapshot of the data structure. After this, the algorithm traverses the data layer until the last key in the range. It is worth noting that a range query might need to traverse multiple buckets to collect the queried range. This operation is done by simply continuing to traverse the data layer and crossing the boundary-marker of the next bucket.

2.3 Split And Resize

In order to better balance the LIT data structure, we employ splitting and resizing. Splitting rebalances a bucket into another level of multiple buckets. Resizing creates a new root level for the LIT, which indexes all keys.

Split operations begin after detecting a bucket that exceeds a threshold of keys mapped to it. We utilize the knowledge that the lookup function maps keys from a to b to the bucket. We can determine this from our lookup function. We then create a new lookup function f' for the new level that maps keys from a to b to a new set of buckets. For each new (sub-)bucket i, we determine the smallest key that maps to the bucket, k_i, and insert a boundary marker whose key is k_i. Then, we set each bucket in this new level to point to its corresponding boundary marker. Finally, we use vCAS to update the previous level's bucket to point to the new level instead of the old boundary marker.

Resize operations begin after detecting that the number of levels exceeds a threshold. We determine a new lookup function f' that maps the range of the minimum key to the maximum key in the data structure to the new number of buckets. Similar to the split, we determine the smallest keys that map to each new bucket. We insert these keys as boundary markers and point each bucket to the appropriate boundary marker. We change the root level to this new level through a vCAS.

3 EVALUATION

To assess the performance of LIT, we contrast its performance against a state-of-the-art logarithmic linked data structures that support highly efficient range queries, such as a skip list, enhanced with the vCAS [22] and the bundling [17] versioning techniques. We compare these approaches by utilizing the existing implementations from [17] for the skip lists. In our evaluation, we consider a closed-loop uniform key-value workload. When evaluating range queries,

we fix the ranges to 50 keys. We prepopulate the data structures to half of the key range, and the number of buckets for LIT is 10k.

We utilize a testbed with 4 Intel Xeon Platinum 8160 CPU with hyper-threading enabled, giving us 96 cores with 192 hardware threads. We evaluate by running on all 192 threads. We run with 8B keys and 8B values. Our implementation is in C++, built with the clang++-15 compiler.

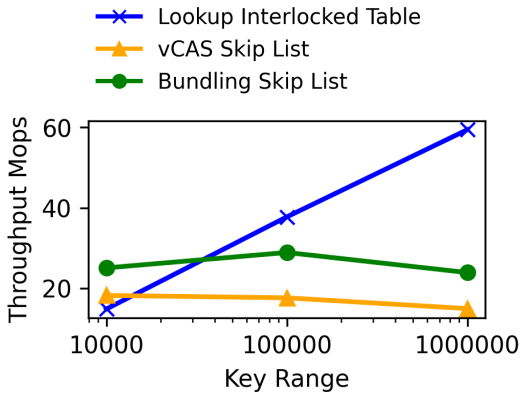

Figure 1: Uniform 10% Range Queries, 10% Updates, 80% Gets While Varying Key Range.

In Figure 1, we consider a mixed workload with 10% range queries, 10% updates, and 80% gets. LIT outperforms a versioned skip list at large key ranges due to the scalability benefits of utilizing a bucket-based approach. As the key range increases, LIT outperforms the vCAS skip list by 2.1x and then 4.0x and the bundled skip list by 1.3x and then 2.5x. This is due to LIT's ability to better handle update operations at these large key ranges, as well as the smaller number of required traversals to reach a key. At 10k keys, LIT is slower than vCAS and a bundled skip list. This can be attributed to the pre-fetching effect of traversing the linked indexes of the skip lists. In fact, in the presence of high contention, cache invalidations dominate the performance. While all data structures see these effects, due to the common traversal path of such a small key range in the skip lists, it is constantly updating its cache with elements that may be accessed in the future.

4 CONCLUSION

We have introduced LIT, a mapping data structure inspired by the function-based indexing of a hash table. LIT's unique feature is that it provides key ordering, which can be used to support range queries.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CNS-2045976. This research was also funded by a CORE grant from Lehigh University.

REFERENCES

[1] Sarwar Alam, Humaira Kamal, and Alan Wagner. 2014. A scalable distributed skip list for range queries. In *The 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'14, Vancouver, BC, Canada - June 23 - 27, 2014*, Beth Plale, Matei Ripeanu, Franck Cappello, and Dongyan Xu (Eds.). ACM, 315–318.

[2] Maya Arbel-Raviv and Trevor Brown. 2018. Harnessing epoch-based reclamation for efficient range queries. *ACM SIGPLAN Notices* 53, 1 (2018), 14–27.

[3] Muhammad A. Awad, Saman Ashkiani, Rob Johnson, Martín Farach-Colton, and John D. Owens. 2019. Engineering a high-performance GPU B-Tree. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 145–157. https://doi.org/10.1145/3293883.3295706

[4] Muhammad A. Awad, Serban D. Porumbescu, and John D. Owens. 2023. A GPU Multiversion B-Tree. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Chicago, Illinois) *(PACT '22)*. Association for Computing Machinery, New York, NY, USA, 481–493.

[5] Bapi Chatterjee. 2017. Lock-free Linearizable 1-Dimensional Range Queries. In *Proceedings of the 18th International Conference on Distributed Computing and Networking* (Hyderabad, India) *(ICDCN '17)*. Association for Computing Machinery, New York, NY, USA, Article 9, 10 pages.

[6] Henry Daly, Ahmed Hassan, Michael F. Spear, and Roberto Palmieri. 2018. NUMASK: High Performance Scalable Skip List for NUMA. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018 (LIPIcs, Vol. 121)*, Ulrich Schmid and Josef Widder (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 18:1–18:19.

[7] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. RocksDB: Evolution of Development Priorities in a Key-Value Store Serving Large-Scale Applications. *ACM Trans. Storage* 17, 4, Article 26 (oct 2021), 32 pages.

[8] Robert Escriva, Bernard Wong, and Emin Gün Sirer. 2012. HyperDex: A distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. 25–36.

[9] Google. 2024. *LevelDB*. https://github.com/google/leveldb

[10] Vincent Gramoli. 2015. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 1–10.

[11] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Distributed Computing, 15th International Conference, DISC 2001, Lisbon, Portugal, October 3-5, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2180)*, Jennifer L. Welch (Ed.). Springer, 300–314.

[12] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. 2020. *The art of multiprocessor programming*. Newnes.

[13] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (jul 1990), 463–492.

[14] Louis Jenkins, Tingzhe Zhou, and Michael Spear. 2017. Redesigning Go's Built-In Map to Support Concurrent Operations. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 14–26.

[15] Ankita Kejriwal, Arjun Gopalan, Ashish Gupta, Zhihao Jia, Stephen Yang, and John K. Ousterhout. 2016. SLIK: Scalable Low-Latency Indexes for a Key-Value Store. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, Ajay Gulati and Hakim Weatherspoon (Eds.). USENIX Association, 57–70.

[16] Maged M Michael. 2002. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. 73–82.

[17] Jacob Nelson-Slivon, Ahmed Hassan, and Roberto Palmieri. 2022. Bundling linked data structures for linearizable range queries. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 368–384.

[18] Markus Pilman, Kevin Bocksrocker, Lucas Braun, Renato Marroquín, and Donald Kossmann. 2017. Fast scans on key-value stores. *Proc. VLDB Endow.* 10, 11 (aug 2017), 1526–1537.

[19] Redis. 2023. Redis. https://redis.io/docs/

[20] Dimitrios Siakavaras, Panagiotis Billis, Konstantinos Nikas, Georgios I. Goumas, and Nectarios Koziris. 2020. Efficient Concurrent Range Queries in B+-trees using RCU-HTM. In *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*. ACM, 571–573.

[21] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. 2020. *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company. https://www.db-book.com/

[22] Yuanhao Wei, Naama Ben-David, Guy E Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. 2021. Constant-time snapshots with applications to concurrent data structures. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 31–46.

[23] Wenshao Zhong, Chen Chen, Xingbo Wu, and Song Jiang. 2021. {REMIX}: Efficient Range Query for {LSM-trees}. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 51–64.