Efficient Resumable Filter Queries

Muktikanta Sa
Micron Technologies

Pierre Sutra
Télécom SudParis
IP Paris

1 Introduction

Context. Non-volatile main memory (NVMM, for short) is a byte-addressable memory that preserves its content after a power outage. NVMM provides durability with close performance to volatile memory in terms of bandwidth and latency [1]. Multiple data stores are now designed for NVMM [2–5]. Compared with their volatile predecessors, these NVMM-ready data stores do not cache data in volatile memory but instead access directly the persistent medium. To this end, they rely on persistent data structures (PDTs).

Motivation. In this context, several PDTs (e.g., [6, 7]) have been proposed to efficiently leverage NVMM. These PDTs avoid the dual representation problem and deliver substantially better performance than the file system interface. Unfortunately to date, these PDTs do not save the progress of read-only operations, such as range queries [8]. This is problematic since many industrial workloads [9, 10] execute long-running query operations. A typical example is a scan in OLAP workloads (for instance, TPC-H [11]).

Contributions. This paper proposes an approach to remedy the above problem. We present a durable linked list that supports concurrent Insert, Remove, and Get as well as a filter query operation. The filter query is resumable and it can pursue its computation after a crash. We also explain how to construct other queryable PDTs such as a set, a skip list and a hash table. Preliminary evaluation results show the benefits of our approach.

2 Resuming an Operation

Thread reincarnation. Durable linearizability (*DLIN*) is defined without thread reincarnation, that is after a crash, a thread never recovers [12]. This is inadequate in the context of long-running tasks where we precisely want to pursue a computation despite failures. To fix this, we require that an abstract history $\mathcal{H}$ is *well-formed* when $\texttt{ops}(\mathcal{H})[t]$ is sequential for every thread t.[1] As in [12], $\mathcal{H}$ is durable linearizable when it is well formed and $\texttt{ops}(\mathcal{H})$ is linearizable. This simple redefinition of *DLIN* keeps the composability and non-blocking properties.

Resumption points. In [12], the authors introduce persist points. These points appear between the linearization point of an operation and its response. They characterize when the history is durable linearizable, much like linearization points characterize linearizability. Resumption points generalize persist points and capture the durable progress of an operation. In detail, an operation m is *recoverable* with value v at step s when m returns v in history $\mathcal{H}_{\leq s}.C.recover().solo(t)$, where C is a crash, $recover()$ the recovery procedure executed after it and $solo(t)$ a solo continuation by t. Step s is called a *resumption point* for operation m and value v. A resumption point is *maximum* when resuming the operation after a failure leads to the same result as if the thread invoking it would be solo executing from that point. Namely, s is maximum when m returns also v in $\mathcal{H}_{\leq s}.solo(t)$. One can establish that the persist points defined in [12] are maximal resumption points. In other words, if the system crashes then resumes, all progress made before this point is kept in the durable state.

Resumable operation. *DLIN* does not tell us how much progress is kept before a persist point. In particular, if a persist point is not reached. the operation may restart from scratch after a failure, or just not be executed at all. When operations are short-lived, this does not pose a problem. However, long-running tasks may suffer from this situation. To remedy this problem, we propose the notion of resumable operation. An operation m in history $\mathcal{H}$ is *resumable* when all its resumption points are maximal. With this type of operations, every inch of progress is kept in the durable state, avoiding to restart the computation at recovery. The PDTs we present in the next section support resumable filter queries.

3 Queryable Persistent Data Types

Specification. Our base data structure with support for resumable filter queries is a linked list. Consider two spaces of keys $(\mathcal{K}, <)$ and values $(\mathcal{V})$. For some item (k, v), the list offers three base operations: $\text{Insert}(k, v)$, $\text{Remove}(k)$ and $\text{Get}(k)$, having the usual semantics. Given some boolean function f, the list also supports operation $\text{Filter}(f)$. A call to $\text{Filter}(f)$ filters the items in the list according to f, that is it returns $\{k : (v = \text{Get}(k)) \neq \perp \wedge f(v)\}$. Any number of threads may execute the list operations concurrently, but a single filter query may happen at a time.

Variables. The durable linked list contains nodes allocated dynamically in both volatile (`Node`) and persistent (`PNode`) memory. Each `PNode` holds an (immutable) key, its associated value and two boolean flags storing its state (`iFlag`, `dFlag`). These flags indicate whether the node is fully inserted, in the middle of an insertion/deletion, or already deleted. Additionally, a `PNode` stores two timestamps (`its`, `dts`) and a boolean (`fq`) related to a pending filter query. A `Node` stores a copy of the key and a pointer to the corresponding `PNode`. Similarly to [13], `Node`s are linked by (word size) pointers, called `next`.

Algorithms. To insert an item (k, v) in the list, or check that k is already there, we follow a traditional lock-free pattern. In detail, the position of k is first located by traversing the list. Once found, a `PNode` and a `Node` are created and the insertion is attempted using *CAS*. If this fails, the insertion is tried again. Removing an element from the list follows a similar approach. Compared with [6], list operations have two key differences: First, every time a thread traverses the data structure it helps on-going insertion/removal operations. Second, when an insertion/removal operation observes that a filter query $\text{Filter}(f)$ is on-going it helps the query. If the pair (k, v) is newly inserted, the thread executes $f(v)$ and stores the result in `fq`. Now if k is deleted, the thread adds k to a durable tombstone set. To execute a filter query, a thread first computes a starting timestamp then traverses the linked list following `next` pointers. Upon encountering an item (k, v) in the list, the result

[1]Operator $\texttt{ops}(\mathcal{H})$ trims the crash events from history $\mathcal{H}$. In [12], $\texttt{ops}(\mathcal{H})$ must not include crash events, thus forbidding thread reincarnation.

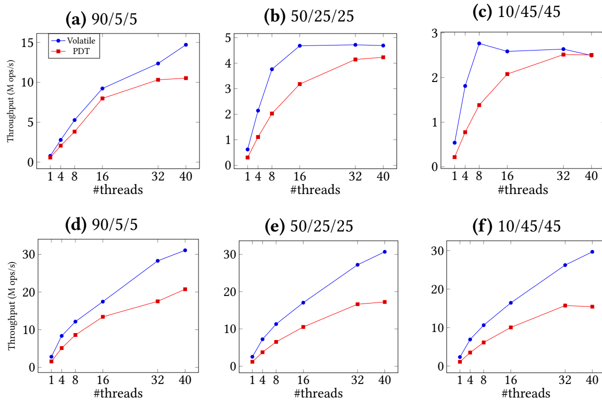

Figure 1. Comparing volatile vs. persistent data types—***(top)*** *skip list;* ***(bottom)*** *hash table.*

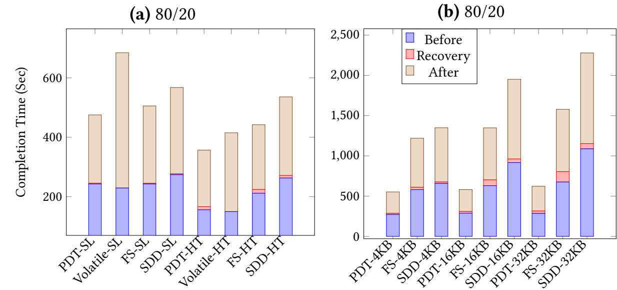

Figure 2. Performance of a long-running filter query despite a crash—***(left)*** *for a skip list (SL) and a hash table (HT);* ***(right)*** *using a hash table and increasing item size.*

of $f(v)$ is stored in the field fq of the corresponding PNode. This node is then referenced in variable rp. Once the list is traversed, the filter query returns all the results whose timestamp is lower than its starting timestamp and whose key is not in the tombstone set. Upon recovery, the list of Nodes is reconstructed using the flags (its,dts) stored in the PNodes. Reconstruction occurs in a single pass over the PNodes, freeing the PNodes marked as deleted. If a filter query was executing, the caller resumes at the appropriate PNode using variable rp.

Extensions. Leveraging the (lock-free) linked list, implementing a persistent set is immediate. To construct a skip list, the volatile nodes should also stores shortcuts. Constructing these shortcuts is similar to the volatile case [14]. We can also derive a hash table by implementing each bucket in the map as a persistent linked list.

4 Evaluation

Experimental Set-up. The test machine is a quad-Intel CLX 6230 hyperthreaded 80-core server with 128/512 GB of volatile/persistent memory. NVMM runs in App Direct mode and is formatted with (DAX) ext4. Our code base spans 6, 741 SLOC of C++. It uses Intel PMDK 1.11.1 and Posix threads. The queryable PDTs are implemented at fine grain using pwb, pfence and psync [12]. For simplicity, the tombstone set relies on transactions.

An experiment consists in loading a certain amount of items, then applying concurrently operations. Unless stated otherwise, each item weights 8 B. Due to space constraints, we only report experiments with 1M items, 50% of them being loaded at startup. In an experiment, we spawn a fixed number of threads. At most, one of them executes a query operation which consists in a 100 μs sleep per item. The other threads randomly perform non-query operations according to a workload distribution. A label 10/45/45 refers to the distribution Get:10%, Insert:45% and Remove:45%, while 80/20 refers to Get:80%, Insert:20%.

We consider two evaluation metrics: **(Figure 1)** the throughput of non-query operations; and **(Figure 2)** the completion time of a query when the program crashes at around half of the experiment then recovers, and concurrent non-query operations are executed. An experiment runs 5 times and we report the average for each metric. We compare various implementations: (PDT) the queryable persistent data types; (Volatile) a DRAM base line; (FS) a file system solution atop NVMM; and (SDD) the same approach using an SSD. Both file system solutions save progress of a query in the file system. The volatile solution simply restarts from scratch.

Results & Takeaways. According to Figure 1, both the persistent skip list and hash table are slower than their volatile counterparts. Across all workloads, the persistent skip list is on average 5-10% slower, while the persistent hash table is 20-25% slower. This is expected as NVMM is slower than DRAM, and inline with prior results (e.g., [15]). In Figure 2(a), we observe that supporting resumable queries brings clear performance benefits. Compared to a pure volatile base line, resumable queries are 40% and 16% faster in the case of a skip list and a hash table respectively. The approach is also faster than a file system solution. In Figure 2(b), we compare it to the two file system implementations using various item sizes. For an item size of respectively 4 KB, 16 KB and 32 KB, queryable PDTs are 2.2x/2.4x, 2.3x/3.3x and 2.5x/3.6x faster than a NVMM//SSD file system solution. We also observe that *(i)* as expected, PDTs are not sensible to the item size; and *(ii)* due to marshalling, a faster persistent medium brings small performance improvement to a file system solution (at best 31%).

References

[1] J. Izraelevitz et al. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. arXiv, 2019.
[2] Cassandra-pmem. https://github.com/intel/cassandra-pmem/tree/13981.
[3] Pmem-Redis. https://github.com/pmem/pmem-redis.
[4] pmemkv. https://github.com/pmem/pmemkv.
[5] Persistent Mem. Storage Engine for MongoDB. https://github.com/pmem/pmse.
[6] Y. Zuriel et al. Efficient lock-free durable sets. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA), 2019.
[7] P. Fatourou et al. Persistent Non-Blocking Binary Search Trees Supporting Wait-Free Range Queries. *Proceedings of the Symposium on Parallelism in Algorithms and Architectures, SPAA'19*, pp. 275–286, 2019.
[8] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys, CSUR84*, 16(42):111–152, 1984.
[9] B. Lepers et al. Kvell+: Snapshot isolation without snapshots. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation, OSDI'20*, pp. 425–441. USENIX Association, 2020.
[10] J. Kim et al. Long-lived transactions made less harmful. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD'20*, pp. 495–510. ACM, 2020.
[11] TPC Benchmark H. http://www.tpc.org/tpch.
[12] J. Izraelevitz et al. Linearizability of persistent memory objects under a full-system-crash failure model. In *Proceedings of the Int. Conf. on Distributed Computing, DISC'16*. Springer, 2016.
[13] T. L. Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. *Proceedings of the Int. Conf. on Distributed Computing, DISC'01*, pp. 300–314, 2001.
[14] W. Pugh. Concurrent Maintenance of Skip Lists. Technical Report CS-TR-2222, University of Maryland, 1989.
[15] A. Lefort et al. J-NVM: Off-Heap Persistent Objects in Java. In *Proceedings of the Symposium on Operating Systems Principles, SOSP'21*, pp. 408–423. ACM, 2021.