A Log-Structured Store for Streaming Data

Jack Rusher
Aleri, Inc.
550 Broad St, Suite 802
Newark, NJ
+1 973 776 8040
jack@rusher.com

ABSTRACT
Event stream processing systems have focused on in-memory databases to achieve high performance, but there are several classes of application that require the persistence of some subset of the events processed and/or results produced at very high data rates. This paper describes a facility for streaming data to disk using a versioned AVL-tree index layered over an append-only log-structured store that operates at between 45-60% the throughput of an in-memory red-black tree or hash table.

1. HISTORY AND MOTIVATION
Event pattern matching and situation detection systems like RAPIDE [5,6] have long been able to make use of persistent storage, but most high-performance event stream processing (ESP) systems, which were developed in the context of telecommunications infrastructure to handle the event rates generated by call data records [1,2] and network packet traffic [3], were unable to tolerate the latency and loss of throughput involved with storing data to disk. Consequently, ESP systems have used chronicles, estimation, and load-shedding to meet performance requirements, storing short-lived data using in-memory databases or custom indices.

The needs of many ESP applications can be met with purely in-memory systems, but there are classes of application that require persistence of some subset of their data — often at uncomfortably high through-puts that range into hundreds of thousands of events per second — for correctness, and many situations in which application restart time would be unacceptably long if re-calculation via an historical replay were required to re-establish current state.

Our streaming data system was developed to address a broad range of applications, including some that require high-throughput persistence and others that require retrospective queries over retained historical data while new data continues to stream into the system.

2. THE LOG STRUCTURED APPROACH
Storage systems are typically organized into a read-optimized repository for long-lived data with a write-optimized journal as a front-end to absorb and then batch updates to the read-optimized repository. In contrast, a log-structured store (LSS) is constructed entirely from a write-optimized journal.

The structure of an LSS is conceptually identical to a ring buffer where writes progress forward until reaching the end of the buffer then wrap around and continue from the beginning. The primary advantage of this design is that the disk head is always in the right place to perform the next write, thus eliminating expensive disk seeks. A secondary advantage is that recovery of a log can be performed using a very simple "roll forward" algorithm that replays or rejects any uncommitted writes following the last checkpoint of the log.

Ousterhout and Rosenblum [8,9] began work on LSS filesystems after observing that most filesystems spend a great deal of time updating on-disk index structures for data that will only be overwritten or deleted before it leaves the buffer cache. Around the same time Stonebraker exploited the database recovery advantages of an append-only LSS for the Postgres Storage System [14].

LSSs were initially thought to be a general solution to improve the performance of write-intensive applications, but performance was ultimately hindered by the need for a background garbage collector, known as the "cleaner," to reclaim free space from the log. The Berkeley LFS team documented the difficulties presented by the cleaner: performance degrades as the ratio of live data to free space increases because more and more time is spent waiting for the cleaner to re-locate live data in order to make room for new writes [10,11].

Fortunately, ESP workloads typically feature enormous numbers of writes that only need be retained for short periods (usually defined by a window over some ordering property of the incoming data), which allows us to avoid excessive cleaner overhead while enjoying the throughput advantages of an LSS.

3. IMPLEMENTATION
3.1 System Resources
We use 64-bit addressing over memory mapped log files to store our data, making use of the operating system's virtual memory subsystem to handle caching. This approach provides us with a directly addressable buffer pool with very little bookkeeping overhead, no double buffering and no calls to *malloc*(). This

method also provides us with file-local object identifiers (OID) for free in the form of simple 64-bit file offsets.

Our approach to system resource utilization is in many ways an update of the one used by the Dali storage manager [4], however updates to Dali's store were done in-place rather via an append-only log, which limited its persistence performance.

3.2 Versioned Trees

In order to avoid read/write lock contention in the storage system, we have implemented a versioned tree data structure that allows writes to continue while reads (including ad-hoc queries against internal table state) are performed against against short-lived snapshots. It was easy to implement this efficiently by exploiting the non-destructive write property of the LSS.

A great deal of prior work has been done on multitemporal [12] and versioned tree data-structures [15] for in-place storage systems, and on techniques for efficiently updating metadata in a versioned log [13], but because our needs are focused on as-needed snapshots rather than historical range queries we are able to avoid most of the difficulties that that work sought to overcome and thus use very simple data structures.

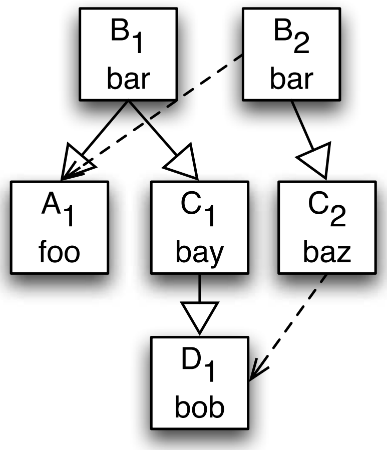

Figure 1. Update key C from value "bay" to the canonical "baz," re-writing modified nodes to the root, B_1 is the root of the pre-write tree, B_2 the root of the post-write tree.

Our data structure, here illustrated with a standard B-tree for clarity, is essentially a standard AVL-tree [16] that has been modified such that the pointer values refer to file offsets in the memory mapped backing file rather than to memory addresses returned by *malloc*(). When a change is committed to the tree the modified tree nodes and the nodes that point to those nodes along the path upward to the root of the tree are written to the tail of the log. The address of the new tree root is then used as a handle to refer to the post-write version of the tree, while the address of the previous root acts as a handle to the pre-write version.

The calls in the store's read API operate against a version handle, allowing reads to continue against a particular immutable version of the tree while new commits are written. The only time writes must block to allow a set of read operations to continue is when a nearly full log must be cleaned to make room for more writes.

3.3 Batch Commits

Although we are able to commit single events to the store, it is more efficient to write a change-set containing the updates caused by a group of events. This is both because the ratio of data to metadata in a given commit is higher when more of the tree is modified in a single operation, and because it is likely that several nodes in common along the path to the root will have been visited during a temporally clustered set of operations.

We first built out a batch commit capability for performance, but we were able to repurpose it to serve as a simple (one at a time) transactional facility for writes.

3.4 Garbage Collection

Our cleaner (sweeping garbage collector) re-writes live data at the write-point of the append-only log in order to clear space for continued writing. This is a necessary feature of any LSS, but it also has the benefit of allowing us to re-pack our data into clusters of related tree nodes to improve data locality. The re-packing process follows the standard persistent AVL-tree page placement algorithms found in the literature.

3.5 Recovery

We implemented the standard roll-forward recovery methods that are detailed in the LSS literature, including a filesystem superblock-style one-page header with a clean bit.

4. RESULTS

We benchmarked our LSS against a set of in-memory indexing structures, including red-black trees and hashes, on a commodity Linux machine with an off-the-shelf RAID controller and disk array. The performance measurements in this paper are meant to demonstrate relative, rather than absolute, performance.

Table 1. Performance Benchmarks

Test	Rows	TPS
absorb-log	16,000,000	241,643
absorb-hash	16,000,000	519,660
vwap-log	4,000,000	76,365
vwap-tree	4,000,000	131,148

Our record absorption benchmarks — a simple configuration in which records are pumped into the system and stored with little processing — show LSS performance to be 47% that of a tuned in-memory hash table for raw record insertion ("absorb-log" vs. "absorb-hash") on a sixteen million record test run, and 58% the performance of an in-memory red-black tree for a four million record test run of a more computationally demanding application involving a group of simultaneous Value Weighted Average Price (VWAP) calculations ("vwap-log vs. vwap-hash"). The events in all of these tests were less than 200 bytes each.

As a comparison, similar experiments conducted on the same hardware with a store implemented using the read-optimized embeddable database Berkeley DB [7] showed an average of around 10% the performance of our in-memory indexing structures — roughly one fifth that of our LSS — regardless of whether we used the BDB hash or tree storage methods.

5. FUTURE WORK

We have designed and implemented a high-performance log-structured AVL-tree, but we have yet to investigate other forms of log-structured tree, such as Radix trees or Patricia tries [17], nor have we experimented with log-structured hash tables [18], which are quite promising for applications that do not require key-order data retrieval.

6. ACKNOWLEDGMENTS

The work discussed in this paper was done by the author in collaboration with his colleagues Jerry Baulier, Scott Kolodzieski and Jon Riecke at Aleri, Inc.

7. REFERENCES

[1] Baulier, J., Blott, S., Korth, H., Silberschatz, A. A database system for real-time event aggregation in telecommunication. In *Proceedings of the 24th Int. Conf. Very Large Data Bases*, VLDB, pages 680–684, 24–27.

[2] Baulier, J., Bohannon, P., Gogate, S., Gupta, C., Haldar, S., Joshi, S., Khivesera, A., Korth, H., McIlroy, P. , Miller, J., Narayan, P. P. S., Nemeth, M., Rastogi, R., Seshadri, S., Silberschatz, A., Sudarshan, S., Wilder, M., Wei, C. Datablitz storage manager: Main memory database performance for critical applications. In *SIGMOD Conference*, pages 519–520, 1999.

[3] Cranor, C., Gao, Y., Johnson, T., Shkapenyuk, V., Spatscheck,O. Gigascope: high performance network monitoring with an sql interface. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data* (SIGMOD '02), pages 623–623, New York, NY, USA, 2002. ACM Press.

[4] Jagadish, H. V. , Lieuwen, D., Rastogi, R., Silberschatz, R.,Sudarshan, S. Dalí: A High Performance Main Memory Storage Manager. In P*roceedings of the Twentieth International Conference on Very Large Databases*, pages 48–59, Santiago, Chile, 1994.

[5] Luckham, D., Kenney, J., Augustin, L., Vera, J., Bryan, D., Mann, W. Specification and analysis of system architecture using rapide *IEEE Transactions on Software Engineering*, 21 (4):336–355, 1995.

[6] Luckham, D., Frasca, B. Complex event processing in distributed systems. Technical Report CSL-TR-98-754, 1998.

[7] Olson, M., Bostic, K., Seltzer, M.. Berkeley db. In *USENIX Annual Technical Conference*, FREENIX Track, pages 183–191, 1999.

[8] Ousterhout, J., Douglis, F.. Beating the i/o bottleneck: a case for log-structured file systems. *SIGOPS Oper. Syst. Rev.*, 23 (1):11–28, 1989.

[9] Rosenblum, M., Ousterhout, J. K. The LFS storage manager. In *Proceedings of the USENIX Summer 1990 Technical Conference*, pages 315–324, Anaheim, CA, USA, 11–15 1990.

[10] Seltzer, M., Bostic, K., McKusick, M., Staelin, C. An implementation of a log-structured file system for UNIX. In *USENIX Winter*, pages 307–326, 1993.

[11] Seltzer, M., Smith, K., Balakrishnan, H., Chang, J., McMains, S., Padmanabhan, V. File system logging versus clustering: A performance comparison. In *USENIX Winter*, pages 249–264, 1995.

[12] Snodgrass, R., Ahn, I. A taxonomy of time databases. In *Proceedings of the 1985 ACM SIGMOD international conference on Management of data* (SIGMOD '85), pages 236–246, New York, NY, USA, 1985. ACM Press.

[13] Soules, C., Goodson, G., Strunk, J., Ganger, G. Metadata efficiency in versioning file systems, 2003.

[14] Stonebraker, M. The design of the POSTGRES storage system. In *Proceedings of the 13th Conference on Very Large Databases*, Morgan Kaufman pubs. (Los Altos CA), Brighton UK, 1987.

[15] Varman, P., Verma, R.. Optimal Storage and Access to Multiversion Data. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391–409, 1997.

[16] http://en.wikipedia.org/wiki/AVL_tree

[17] http://en.wikipedia.org/wiki/Radix_tree

[18] http://en.wikipedia.org/wiki/Hash_table