

# Concurrency-friendly Caching for Append-only Data Structures

Leif Walsh, Vyacheslav Akhmechet, and Mike Glukhovsky  
Hexagram 49, Inc.\*  
{leif, slava, mike}@hexagram49.com

July 21, 2009

## Abstract

Append-only disk-based data structures disobey the traditional rules of temporal and physical locality relied upon by caching algorithms. Temporal locality becomes a function of physical offset, as the newest data blocks are always closest to the end.

We have extended the well-known ring buffer, which is typically used as a large queue for delaying and batching writes, and is used in the Linux kernel’s logging infrastructure, as well as in the write-ahead log for many filesystems and databases based on ARIES [5]. Our extension allows it to perform as both a tunable write-behind cache, and a high-performance read cache.

The addition of this cache to the RethinkDB MySQL storage engine [1] consistently increases write throughput by 2x.

changes the semantics of the frequency information, and makes the storage and update of this information infeasible.

We propose the *round robin cache*, an extension of the ring buffer which allows it to function as a read cache. The round robin cache shares many properties with CFLRU/C [9], but uses knowledge of the RethinkDB index tree’s access pattern to benefit from an extremely simple and robust algorithm. In Section 2, we discuss the semantic issues of this new functionality, and in Section 3, we briefly discuss the metadata and locks required to implement the round robin cache in a concurrent environment. Finally, we provide a brief analysis of the structure in Section 4, and in Section 5, we show how the introduction of this cache improved performance of the RethinkDB storage engine.

## 1 Introduction

The append-only nature of the RethinkDB storage engine makes it problematic for traditional caches. An LRU cache [4, 6] is useless in a write-heavy workload. As the age of a block is proportional to its physical distance from the end of the file, the LRU list will simply duplicate the information present in file offsets, in an inefficient data structure. An LFU cache [7] cannot be implemented on our append-only index cache [2], because the physical location of a logical tree node changes with each write, which

## 2 Design

The round robin cache is designed to have low overhead and perform well under diverse workloads. In particular, with a write-heavy workload, we require its overhead be low enough that throughput is only limited by the disk.

The first goal is met easily with a simple ring buffer, taken from common WAL applications. The design simplicity of the ring buffer allows it to be implemented in a very efficient manner, and with some offset calculations to be described in Section 3, we can avoid altering nodes as we flush them or their children to disk.

---

\*Copyright Hexagram 49, Inc., 2009. All rights reserved.

Under a read-heavy workload, we observe that, with the structure of the RethinkDB append-only index tree, the most frequently read nodes, being the root and the first few levels of the tree, are always at the end of the file. Thus, we must aim to always have these in the round robin cache.

The second goal also comes quickly out of a ring buffer. Intuitively, as each write into the ring buffer will write the root of the tree and its closest children on the path to the target leaf last [2], and only invalidate the oldest data in the buffer, the most recent versions of the most frequently read nodes will most likely be present in the cache. We will see in Section 4 an analysis showing that the cache predicts with high probability which nodes to discard.

To handle the case in which the server is started and only reads are performed, we preload the cache with a tunable amount of data from the end of the file when it is initialized. We have found that this preloading step adds a small amount of overhead to the server’s startup time, but vastly improves the performance of a read-only workload.

The round robin cache expects and exposes an API to two types of users. *Writers* will request blocks of memory into which to write, copy memory into these blocks, and notify the cache when they have completed their writes. *Readers* will ask the cache to read a region of memory from the file into a destination buffer. It is up to the cache to determine whether that memory lies in the cache or the file (or both), and perform the memory copying. The cache also maintains a flush thread, which flushes to the backing file data which writers have finished and marked as valid.

### 3 Implementation

#### 3.1 Resources

During implementation of the round robin cache, we were primarily concerned with simplicity and minimal locking. We manage as little metadata as possible, to address both of these concerns.

The round robin cache maintains a contiguous mapped buffer in memory, and maintains three re-

gions within this buffer (see Figure 1 for a visualization):

- The **write region** is the block of memory which has been requested by a writer for the purpose of copying in data. The data inside it is regarded as invalid for reading or flushing, as the writer thread could be writing to parts of it at any time.
- The **read region** is the block of memory which is regarded as having valid data. This is available for reading by any reader.
- The **flushed region** is a subregion of the read region, and contains data which has been written to the file. Reads inside this region are still serviced from the cache, as this should be faster than going to disk, but writes must be made available from this region, or they would invalidate volatile data.



Figure 1: Circular representation of the regions in the round robin cache.

To denote these regions, the cache tracks four logical offsets within the buffer, and one in the file:

- The **write offset** (offset 1) points to the end of the write region. This marks the end of all in-progress writes, and the beginning of the next write.

- The **read offset** (offset 2) points to the end of the read region and the beginning of the write region. This marks the end of valid data and the beginning of invalid data.
- The **flush offset** (offset 3) points to the end of the flushed region. This marks the end of the flushed data, and the beginning of volatile data.
- The **cache offset** (offset 4) points to the beginning of the read region. This is almost always equal to the write offset, but acts as a convenience for calculations, and is also required until the cache becomes full and writes move past the end of the buffer.
- The **cache file offset** is an offset within the backing file, which corresponds to the location in the file where the data at the cache offset exists. Data between the cache file offset and the end of the file is duplicated in the cache, inside the flushed region. The cache file offset and the cache offset allow us to determine whether a request lies in the file or in the cache, and in the latter case, where it is within the cache.

During execution, writers extend the write region, copy data into it, and then commit that data, which converts the write region back into part of the read region. Readers simply read from the read region anything they can, and fall back to the file to read older data. The flush thread continuously tries to keep up with the read region as writers extend it, as new writes will come from the back of the flush region.

## 3.2 API

The round robin cache exports an api for writers and readers. Writers can call `start.write` to claim a region of memory to which they may copy data; `do.memcpy` is a thin wrapper around one or two calls to `memcpy(3)`, two only if the destination region wraps around the end of the physical buffer; and `commit.write` to release control of the region of memory previously claimed with one or more calls to `start.write`, meaning that the write is complete

and that all the memory in that region is valid and available for reading or flushing.

Readers have one command, `read`, which readers use as if they were reading directly from the file. It expects a destination buffer, an offset in the file, and a size. The cache will determine how much of the request lies within, and then copy data into the destination, reading as much as possible from the cache — only going to disk if necessary.

## 3.3 Concurrency

The concerns of concurrency are complex. Currently, we concern ourselves with a model of one writer and many readers. To satisfy the high-concurrency goals of the RethinkDB storage engine, we lock as little as possible, and for this cache, we were careful to never lock during any I/O operations, neither disk reads and writes which may sleep, nor calls to `memcpy(3)`.

Some of the offsets, namely the write offset, cache offset, and cache file offset, must be updated in concert, to ensure the correctness of our calculations. For this reason, we protect those three variables with one read/write lock. The writer locks this during `start.write` as it changes these variables, and the readers lock this for reads, during offset calculation, and release it before they begin copying memory.

Additionally, we must not allow a writer to invalidate a region of memory that is currently being read. To achieve this, we maintain a heap containing the beginning offset of each read, sorted by their distance forwards from the write offset. The writer, during `start.write`, consults this heap to ensure that it will not invalidate a read currently in progress. If it will, it declares its intent, sleeps, and is awakened by the reader which removes the last offset from the heap which conflicted with the intended write. The heap is protected by a read/write lock to maintain internal consistency, but again, these locks are not held during any I/O, even calls to `memcpy(3)`.

To prevent write starvation, we maintain a sub-region of the read region, ahead of the write offset, from which reads cannot read. This forces reads to go to disk slightly more often, but makes it highly unlikely that a reader will lock a region of memory that will soon be requested by a writer. Different

workloads will change the optimal size of this sub-region, so we allow tuning of this parameter in the RethinkDB storage engine.

## 4 Analysis

We already know that the round robin cache batches writes well. What we are concerned with here is how we improve read performance. We also know that the analysis of the cache will not be asymptotically interesting, as the cache size must be fixed. Therefore, we will calculate how accurately the cache predicts future use of a node, that is, the correlation between (1) the probability that a node is kept in the cache ( $P(\text{cached}(\cdot))$ ) and (2) the probability that the node will be read during the next  $k$  queries ( $P(\text{read}(\cdot)|k \text{ reads})$ ). If a query does not need to look past some internal node  $\nu_i$ , it does not make sense to ask whether the cache retains nodes below  $\nu_i$ , so we will obtain a lower bound on the cache’s prediction performance by assuming the query reaches all the way down to a leaf. We will assume uniformly random inserts and queries.

For the first part (1), we need the size of the cache. We will measure this in the number of nodes stored, and call that  $N_c$ . Additionally, we will assume a tree containing  $n$  elements. The probability that a node is in the cache is the inverse of the probability that it is not in the cache, which is just the probability that it was not part of one of the last writes to fill up the cache. To obtain a lower bound, we will also assume that the last writes were all of length  $2 \log(n + 1)$ , the maximum length of a root-to-leaf path in a red-black tree [3], which implies that the number of writes needed to fill the cache is  $N_c/2 \log(n + 1)$ . The probability that a node  $\nu_d$  was not part of a single one of these writes is the probability that the write was not in the subtree rooted at  $\nu_d$ , or  $1 - 2^{-d}$ . Therefore, the probability that the node was not part of any of these writes is  $(1 - 2^{-d})^{N_c/2 \log(n+1)}$ . Finally, the probability that the node is in the cache, as introduced above, is the inverse of this, or  $P(\text{cached}(\nu_d)) = 1 - (1 - 2^{-d})^{N_c/2 \log(n+1)}$ .

The second part (2) is straightforward from the fact that it is a binary search tree. A node at depth

$d$  (the root being at  $d = 0$ ),  $\nu_d$  will be read whenever a query is performed on a node in the subtree rooted at  $\nu_d$ , which has probability  $2^{-d}$ . In a similar fashion to the process above, the probability that a node will be read during the next  $k$  queries is the inverse of the probability that it will not be read once during those queries, or  $P(\text{read}(\nu_d)|k \text{ reads}) = 1 - (1 - 2^{-d})^k$ .

Notably, these probabilities seem very similar. In fact, if we know the values  $n$  and  $k$ , which correspond to the workload (universe set and number of queries), we can let  $N_c = 2k \log(n + 1)$  and get perfect correlation. If  $N_c$  is less than this, then we flush more nodes, but interestingly, we still get good correlation when  $d$  is small, that is, the top of the tree is still held in the cache. If  $N_c$  is greater than this, then each node’s probability of being present in the cache increases, but for these probabilities increase more for smaller values of  $d$ .

Therefore, we have a cache which can be tuned to perform optimally, but which degrades gracefully as the workload changes. We will see exactly how well it performs presently, in Section 5.

## 5 Performance

To evaluate the performance of the round robin cache in the RethinkDB storage engine, we ran a benchmark inserting sets of 655360, 1310720, and 1966080 rows, into a table with three INT columns, two of which having indexes. We used the `auto-pilot` benchmarking suite [8] to record these times, and we run until we have 95% confidence intervals. The benchmarks were performed on a machine with a 2.5 GHz Pentium Core 2 Duo processor, 2 GB RAM, and the database was located on a 16 GB SUPER TALENT MasterDrive, a MLC solid state drive, connected via a 3 GB SATA II bus. The manufacturer lists this drive’s sequential read and write speed as 150 MB/s and 100 MB/s, respectively, though we have not verified this.

To test the RethinkDB engine without the round robin cache, we created a fallthrough implementation of our cache interface that passes all requests directly to their corresponding `libc` I/O functions.

The results of our benchmark can be seen in Fig-

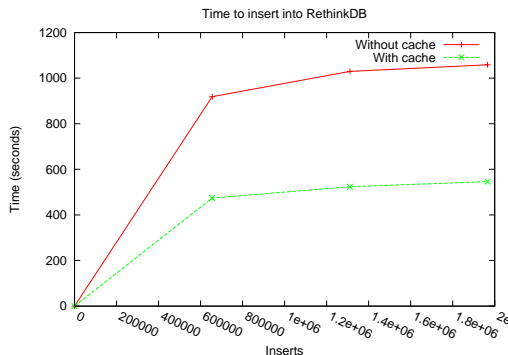


Figure 2: Time to insert into the RethinkDB storage engine, with and without the round robin cache.

ure 2. We see that the RethinkDB storage engine without the round robin cache has an average 94% overhead over the engine using the round robin cache, and that this overhead is very consistent, having a variance of only 0.0177%.

## References

- [1] Slava Akhmechet, Leif Walsh, and Michael Glukhovskiy. Rethinkdb — rethinking database storage. 2009.
- [2] Slava Akhmechet, Leif Walsh, and Michael Glukhovskiy. An append-only index tree structure. To appear in fourth quarter of 2009.
- [3] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. Introduction to algorithms. pages 238–261. The MIT Press and McGraw-Hill Book Company, 1989.
- [4] Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, May 1968.
- [5] C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *TODS*, 17(1):94–162, 1992.
- [6] Elizabeth J. O’Neil, Patrick E. O’Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proc. of the 1993 ACM SIGMOD Conference on Management of Data*, pages 297–306, May 1993.
- [7] John T. Robinson and Murthy V. Devarakonda. Data cache management using frequency-based replacement. In *SIGMETRICS*, pages 134–142, 1990.
- [8] C. P. Wright, N. Joukov, D. Kulkarni, Y. Miretskiy, and E. Zadok. Auto-pilot: A platform for system software benchmarking. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 175–187, Anaheim, CA, April 2005. USENIX Association.
- [9] Yun-Seok Yoo, Hyejeong Lee, Yeonseung Ryu, and Hyokyung Bahn. Page replacement algorithms for nand flash memory storages. pages 201–212. 2007.