DragonDB[1]: A Fiercely Scalable Key-Value Store and API

Ben Mittelberger, bmittelb
Suzanne Stathatos, sstat
Evan Shieh, eshieh
Claudia Roberts, chicacvr

I. **Introduction**

Our objective was to build a scalable and persistent key-value store library with a simple and mostly commutative API.

II. **Design**

2.1 Platform Choice

To allow a client pin threads to particular cores, we needed a machine whose threads we could manipulate. We developed both the dragonDB library and client application on a vagrant-ubuntu-trusty-64 3.13.0-52 machine. All testing was performed on the same machine. It had 42.9G hard disk space and used an Intel® Core™ i7-3450M 3GHz CPU.

2.2 Scalability Design

Fig. 1 is an overall diagram of our kv-store's design. (1) The client-side application is responsible for spawning and pinning threads to specific cores. It pairs threads with cores based on the cores' loads. The client randomly selects/checks cores until a finds a free core. It does random selection (as opposed to iteration through cores) to ensure that all cores have equal expected allocation frequency.

Dragon_cores and `db->puts()`

(2) Each thread will only modify its own core; no core will be modified by multiple threads. Per-core locks, then, are unnecessary. Cores were designed to make writes fast. (3) Because threads are assigned to processes by workload (not by k-v content), often threads will be assigned to work with another core's data. (This results from differences in determinism: pinning threads to cores is non-deterministic. However, where the data will be stored is deterministic, as each k-v pair is arithmetically hashed to a core). To handle this, each dragonDB core has a mailbox, which contains packages (k-v pairs and a timestamp) inserted by other cores.

Mailboxes

Mailboxes have N (N=#cores) slots, N-1 of which are used. (The Nth slot=the current core's slot, which will write to the core instead of its mailbox). Each slot has a lock. When *thread_i* wishes to write a kv pair to a core to which it doesn't belong (*core_!i*), it will write the kv pair and a timestamp to the *ith* slot of the mailbox of the core on which the kv pair resides (*core_!i*). The mailboxes get flushed every 500ms. During a mailbox flush, each package from each slot will be processed as a puts to the cores' segment. If 2 packages within the same slot modify the same key, and/or the key has also been modified in the dragon_core, package with the latest timestamp will be reflected on disk.

Dragon_segments and `db->gets()`

Dragon_segments (also called logs in this paper) are designed to optimize reads. Every core is able to read from every other core's segment. We use no read segment locks, but instead check entry's timestamps twice to see if things have changed between accesses. The dragon_segment write-locks when the log gets flushed to disk, as a thread's core cannot write to a dragon_segment mid-flush. This log gets written to disk after >=1 second has elapsed. These lazy flushes guarantee that the kv store is *eventually consistent* (all changes need not be immediately reflected on disk, but they will be eventually). Each segment corresponds with a file on disk. Files do not overlap, so multiple segments can write to multiple files concurrently.

`db->close()`

This function makes sure all threads have finished executing, does one last disk flush, and deallocates all memory used by the kv-store.

2.3 Persistence Design

We use fsync when we flush to guarantee durability despite file corruption. Writing to disk cache alone would not guarantee that data was reflected and uncorrupted on the disk during a power failure.

When a dr_segment writes to disk, it writes the segment size, a checksum of the segment, and the data contained within the segment. When opening an existing kv store, we load the most recent uncorrupted log into the dr_segment. Uncorrupted means the log's checksum on the file matches a newly computed checksum of the log. If it does *not* match, we iteratively roll back logs, checking segment validity until we find an uncorrupted segment to load.

2.4 Optional Consistency

We wanted to examine the scalability of a consistent kv-store. A command-line flag (-STRONG) lets the client enable strong consistency. Without the flag, the kv-store will default use eventual consistency/strong durability. With -STRONG, the database gets flushed after every puts. As expected, strong consistency requirements significantly hinder performance. Operations still scale linearly, but at a less efficient rate, with strong consistency.

2.5 Design Limitations

Our design pins threads 1:1 to cores. We focused on a multi-*core* scalable kv-store rather than one that was highly parallelizable.

III. **Results and Discussion**

We were pleased to observe a linear relationship between the number of cores in use and the performance time. The results of our benchmarks can be seen in the Figures on pg. 2. We designed our benchmarks to mimic those of rocksdb and leveldb,[2] i.e. random writes, random reads, mix of multi-threaded read and single-threaded write. We ran tests on a small (10K ops) workload and a larger (100K ops) workload. Our reads and writes by design behave like random disk accessed if a kv-pair is not in the dragonCore or the most recently flushed segment. The random behavior results from the arithmetic hash of kv-pairs to different cores. If two commands sequentially stored kv-pairs (i.e. `puts 1 1`, `puts 2 2`), those values are not necessarily near each other on disk. The two cores' files corresponding to their store could be located far apart on disk.

When comparing levelDB's performance to dragonDB (Fig. 2), we compared the benchmarks from levelDB's large random writes and random reads to our benchmarks. We scaled all benchmarks to report performance in terms of operations done per millisecond.

Increasing the # of cores had a large effect on write (puts) performance, but the # of cores did not seem to as drastically affect read-only (get) performance. This may be because dragon_segments lock and dragon_cores do not. A dragon_segment may be locked when a thread attempts to read from it. Given more time, it would be interesting to experiment with different locking mechanisms to achieve higher performance.

It is interesting to see such differences in API performance vs. performance from a list of commands in a .tst file. The API's performance was on the order of ms, while reading and executing commands from a separate file was on the order of minutes.

IV. **Challenges**

We ran into issues with file ballooning similar to the challenges that LFS faced. We have begun to fix this problem by cleaning the kv-store disk files. Upon a database open/load, we eliminate dead log entries and atomically rename the files. This eliminated redundancy in the files and has made our file sizes smaller. We suspect this is linked to the write-performance differences mentioned above, so we'd like to continue addressing this problem.

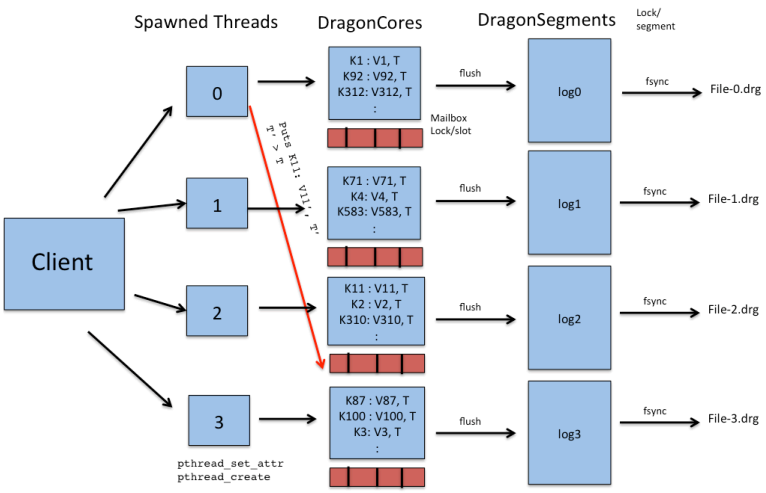


Fig. 1
DragonDB's logical design

dragonDB vs levelDB

	puts-only	gets-only
levelDB	622M ops/5days = 622M ops/4.32*10^8 ms =1.44 ops/ms	1B ops/102hrs = 1B ops/3.67*10^8 ms = 2.7 ops/ms
dragonDB, 4 cores, 100K	100K ops/5.4*10^5 ms = 0.2 ops/ms	100K ops/7012 ms = 14.26 ops/ms
dragonDB, 4 cores, 10K	10K ops/8228ms =1.2 ops/ms	10K ops/19ms = 526.3 ops/ms

dragonDB performance benchmarks 100K operations (ms)

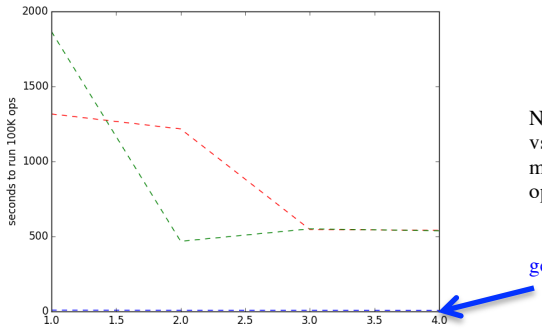
# cores	puts-only	gets-only	puts-and-gets
1	1317296	9123	1865920
2	1217458	7740	468639
3	546751	7741	551708
4	541403	7012	538036

dragonDB performance benchmarks 10K operations (ms)

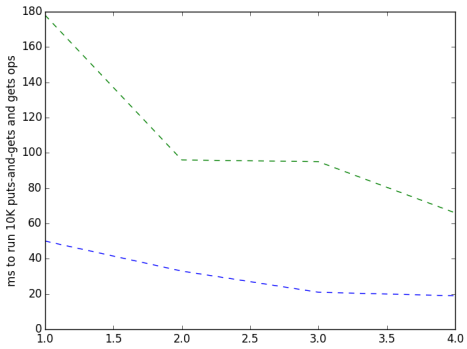
# cores	puts-only	gets-only	puts-and-gets
1	107446	50	178
2	35266	33	96
3	15991	21	95
4	8228	19	66

dragonDB performance benchmarks : 10K operations (ms)
of puts-and-gets, STRONG vs. WEAK consistency

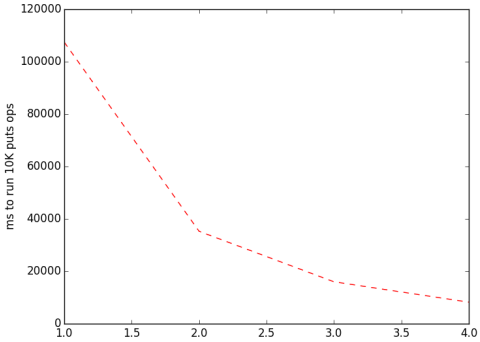
# cores	WEAK	STRONG
1	178	128814
2	96	42803
3	95	35207
4	66	33907



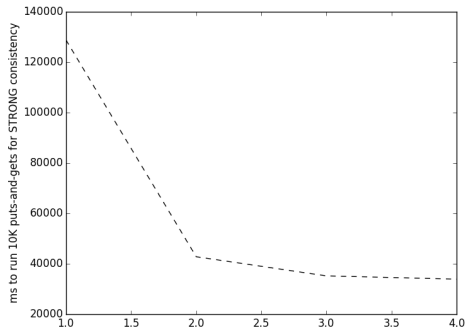
Num_threads vs. ms for 100K operations



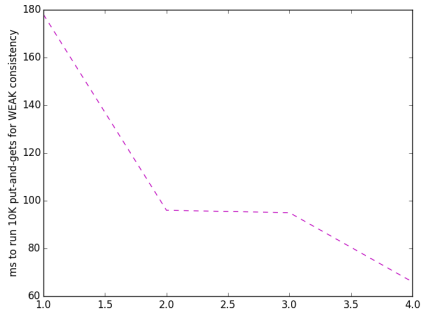
Num_threads vs. ms for 10K operations



Num_threads vs. ms for 10K puts-only operations



Num_threads vs. Strong consistency



Num_threads vs. Weak consistency

[1] The code for dragonDB is located at https://github.com/bmittelberger/dragonDB/
[2] Performance Benchmarks RocksDB. 3 June 2015. https://github.com/facebook/rocksdb/wiki/Performance-Benchmarks#setup