Voldemort on Solid State Drives

Vinoth Chandar, Lei Gao, Cuong Tran
Linkedin Corporation, Mountain View, CA

Abstract
Voldemort is Linkedin's open implementation of Amazon Dynamo, providing fast, scalable, fault-tolerant access to key-value data. Voldemort is widely used by applications at LinkedIn that demand lots of IOPS. Solid State Drives (SSD) are becoming an attractive option to speed up data access. In this paper, we describe our experiences with GC issues on Voldemort server nodes, after migrating to SSD. Based on these experiences, we provide an intuition for caching strategies with SSD storage.

1. Introduction

Voldemort [1] is a distributed key-value storage system, based on Amazon Dynamo. It has a very simple get(k), put(k,v), delete(k) interface, that allows for pluggable serialization, routing and storage engines. Voldemort serves a substantial amount of site traffic at LinkedIn for applications like 'Skills', 'People You May Know', 'Company Follow', 'LinkedIn Share', serving an average of 100K operations/sec over roughly 80TB of data. It also has wide adoption in companies such as Gilt Group, EHarmony, Nokia, Jive Software, WealthFront and Mendeley.

Due to simple key-value access pattern, the single Voldemort server node performance is typically bound by IOPS, with plenty of CPU cycles to spare. Hence, Voldemort clusters at LinkedIn were migrated to SSD to increase the single server node capacity. The migration has proven fruitful, although unearthing a set of interesting GC issues, which led to rethinking of our caching strategy with SSD. Rest of the paper is organized as follows. Section 2 describes the software stack for a single Voldemort server. Section 3 describes the impact of SSD migration on the single server performance and details ways to mitigate Java GC issues. Section 3 also explores leveraging SSD to alleviate caching problems. Section 4 concludes.

2. Single Server stack

The server uses an embedded, log structured, Java based storage engine - Oracle BerkeleyDB JE [2]. BDB employs an LRU cache on top of the JVM heap and relies on Java garbage collection for managing its memory. Loosely, the cache is a bunch of references to index and data objects. Cache eviction happens simply by releasing the references for garbage collection. A single cluster serves a large number of applications and hence the BDB cache contains objects of different sizes, sharing the same BDB cache. The server also has a background thread that enforces data retention policy, by periodically deleting stale entries.

3. SSD Performance Implications

With plenty of IOPS at hand, the allocation rates went up causing very frequent GC pauses, moving the bottleneck from IO to garbage collection. After migrating to SSD, the average latency greatly improved from 20ms to 2ms. Speed of cluster expansion and data restoration has improved 10x. However, the 95th and 99th percentile latencies shot up from 30ms to 130ms and 240ms to 380ms respectively, due to a host of garbage collection issues, detailed below.

3.1 Need for End-End Correlation

By developing tools to correlate Linux paging statistics from SAR with pauses from GC, we discovered that Linux was stealing pages from the JVM heap, resulting in 4-second minor pauses. Subsequent

promotions into the old generation incur page scans, causing the big pauses with a high system time component. Hence, it is imperative to mlock() the server heap to prevent it from being swapped out. Also, we experienced higher system time in lab experiments, since not all of the virtual address space of the JVM heap had been mapped to physical pages. Thus, using the AlwaysPreTouch JVM option is imperative for any 'Big Data' benchmarking tool, to reproduce the same memory conditions as in the real world. This exercise stressed the importance of developing performance tools that can identify interesting patterns by correlating performance data across the entire stack.

3.2 SSD Aware Caching

Promotion failures with huge 25-second pauses during the retention job, prompted us to rethink the caching strategy with SSD. The retention job does a walk of the entire BDB database without any throttling. With very fast SSD, this translates into rapid 200MB allocations and promotions, parallely kicking out the objects from the LRU cache in old generation. Since the server is multitenant, hosting different object sizes, this leads to heavy fragmentation. Real workloads almost always have 'hotsets' which live in the old generation and any incoming traffic that drastically changes the hotset is likely to run into this issue. The issue was very difficult to reproduce since it depended heavily on the state of old generation, highlighting the need for building performance test infrastructures that can replay real world traffic. We managed to reproduce the problem by roughly matching up cache miss rates as seen in production. We solved the problem by forcing BDB to evict data objects brought in by the retention job right away, such that they are collected in young generation and never promoted.

In fact, we plan to cache only the index nodes over the JVM heap even for regular traffic. This will help fight fragmentation and achieve predictable multitenant deployments. Results in lab have shown that this approach can deliver comparable performance, due to the power of SSD and uniformly sized index objects. Also, this approach reduces the promotion rate, thus increasing the chances that CMS initial mark is scheduled after a minor collection. This improves initial mark time as described in next section. This approach is applicable even for systems that manage their own memory since fragmentation is a general issue.

3.3 Reducing Cost of CMS Initial mark

Assuming we can control fragmentation, yielding control back to the JVM to schedule CMS adaptively based on promotion rate can help cut down initial mark times. Even when evicting data objects right away, the high SSD read rates could cause heavy promotion for index objects. Under such circumstances, the CMS initial mark might be scheduled when the young generation is not empty, resulting in a 1.2 second CMS initial mark pause on a 2GB young generation. We found that by increasing the CMSInitiatingOccupancyFraction to a higher value (90), the scheduling of CMS happened much closer to minor collections when the young generation is empty or small, reducing the maximum initial mark time to 0.4 seconds.

4. Conclusion

With SSD, we find that garbage collection will become a very significant bottleneck, especially for systems, which have little control over the storage layer and rely on Java memory management. Big heap sizes make the cost of garbage collection expensive, especially the single threaded CMS Initial mark. We believe that data systems must revisit their caching strategies with SSDs. In this regard, SSD has provided an efficient solution for handling fragmentation and moving towards predictable multitenancy.

References

[1] http://project-voldemort.com/
[2] http://www.oracle.com/technetwork/database/berkeleydb/overview/index-093405.html