VMware Directory Server Scalability Enhancement

Jin Fang fangj@vmware.com Sung Ruo sruo@vmware.com

January 26, 2017

Abstract

VMware directory service is a core component of VMware cloud management platform. The server utilizes OpenLdap MDB as its transaction engine. MDB adopts a simplified Multi-Version Concurrency Control (MVCC) design to avoid using locks, thus delivers a superior read performance due to its high concurrency level. However the single-writer limitation with MDB is a bottleneck to achieve high write throughput that requires persistency. This paper presents a performance enhancement to MDB: add a Write-Ahead Logging (WAL) mechanism. This change would improve the write throughput from four to thrity folds depending on the database scale and storage type. The enhanced database engine is a critical component used to implement up-coming Lightwave Raft service. As demonstrated, the write performance has surpassed Oracle Sleepycat's multi-writer transaction engine by seven folds with a storage consisting of a PCIe-SSD, or 3.9 times with a RAID storage consisting of five hard drives. The WAL feature has also closed the gap that supports database hot copy and incremental database hot backup offered by Sleepycat, which is absent from the current MDB.

1 Introduction

VMware distributed directory system is a core component of VMware cloud management platform. Its applications include Single-Sign-On identity authentication and repository for VMCA, VC Tagging, and product licensing data. The directory server implements a multi-master replication protocol, allowing updates made on one directory server eventually reach all other servers in the replication topology. The up-coming Lightwave Raft is a variation of the multi-master directory service, which utilizes Raft protocol to replicate changes to other nodes in the cluster while maintaining the features and external interfaces [6] .

The implementation of both directory services utilizes OpenLdap MDB [3] as its transaction engine for ACID. MDB is a low cost, open source alternative for more expensive embedded database packages such as Oracle Sleepycat. Using file based memory map for B+ tree pages and a simplified Multiversion Concurrency Control (MVCC) [1] delivered a superior read performance [5]. However the single-writer design is a bottleneck to achieve high write throughput, especially for the large database storing tens of millions entries.

The organization of this paper is the following: section 2 provides an overview of the implementation algorithms with MDB; section 3 presents the algorithm that implements Write-Ahead Logging (WAL). It explains why WAL would significantly improve the write throughput in a virtual machine (VM) hosted by a hypervisor, such as ESXi, either with a storage consisting of PCIe-SSD drive or RAID built from hard drives. section 4 gives the test results comparing the write performance between the original MDB implementation and the one with WAL. The tests are conducted with the two storage types[1]: PCIe Solid State Drive (SSD) and RAID consisting of five hard drives. It demonstrated that WAL improves the write throughput more than four folds in a server with ten million LDAP entries (about 100GB database size), which is four to seven times faster than Oracle Sleepycat's multi-writer transaction engine; section 5 demonstrates that WAL offers an inherent benefit of supporting database hot copy or incremental database hot backup. This capability is essential to create a new replica in a cluster consisting of multiple servers.

2 MDB Overview

The basic data operations from MDB are get (fetch) and put (write) calls with key/value pair in a transaction scope. There can be multiple key/value sets called "databases" in MDB, each database is stored in a B+ tree. MDB uses a simplified Multiversion Concurrency Control (MVCC) algorithm, namely Two-Version Concurrency Control, to implement the Single-Writer-Multi-Reader model. The single writer scheme avoids using (page-level) locks among read transactions, and between any read transaction and the write transaction, thus achieved high concurrency level for read operations.

To further simplify the design, all of the key/data value sets are stored in the memory map backed by a single database file. This scheme, along with the simplified MVCC, allows MDB to be implemented efficiently with a very small footprint:

[1]The paper doesn't provide results with low-end hard drive which would show more dramatic improvement in write performance with WAL.

less than 10% C code of Oracle Sleepycat. The programing API, including data fetch/save and transaction handling, is compatible with those offered by Sleepycat.

The Two-Version Concurrency Control has significantly simplified the MVCC implementation while ensuring database ACID property. The algorithm is summarized as the following:

1. For any page being fetched in a write transaction, a copy is made from the original page; any updates on a page are performed on its copy, instead of the original page.
2. The root page of the modified pages is the Meta page, through which all other pages are referenced during the write transaction. The Meta page contains a reference to the main database, from which other databases (key/value B+ tree sets) are accessed. Any other write transactions are blocked until the current write transaction is committed.
3. There are totally two Meta pages, one for the current write transaction, the other is for any other read transactions to fetch the data. The Meta page used for the read transaction is referenced through a global variable called MDB environment or "dbenv" through which a new read transaction may reach the Meta page. No transaction level locking is necessary since MVCC is the only mechanism for transaction isolation.[2]
4. Transaction commit for the write transaction is to write all dirty pages (pages modified during the transaction, including the intermediate branch pages in B+ tree) into the database file (or memory map if a compile-time flag is set), and the last page to be written is its Meta page with a single file write operation. An "fsync", "fdatasync" or "msync" system call must be issued to ensure data persistency.[3] Once the Meta page is successfully written and synced, the global "dbenv" is updated, and the data by the write transaction is visible (committed) by all new transactions.

The trade-off with the simplified MVCC described above is its write scalability: since there is only one write transaction allowed at any time, the write throughput is very sensitive to the latency of the "sync" operation. The latency is significant, especially for large scale database, in VMs hosted by a hypervisor, such as VMware ESXi, even if the storage consisting is low-latency PCIe SSD or a RAID built from hard drives.

3 MDB Enhancement with Write-Ahead Logging

Write-Ahead Logging (WAL) [2] would improve the write performance while preserving the high concurrent level with the current MDB. The implementation is a simplified algorithm from the one referenced in [2]: it only performs the roll-forward (or redo) operations and bypasses the undo operations during database recovery. The simplification is possible because MDB's MVCC only modifies the copies of the original pages during write transaction, and the the single writer design makes creating the logging data much simpler. With Write-Ahead-Logging, a write transaction would not have to wait for "sync" to complete during transaction commit; instead, the dirty pages are written onto a transaction log file, and the "sync" is performed on the (small) WAL file, which is much faster than doing sync on a large database file during transaction commit. It should indicate that doing sync to the large database file is still needed otherwise database recovery process will not be able to determine which transaction log files are necessary to roll-forward as described next, but the sync call does not occur at transaction commit time, which is in the critical path for serialization of write transactions.

To ensure transaction integrity when the directory server or OS crashes, a database recovery process is needed when the server restarts. The database recovery procedure would roll-forward the relevant WAL files (those blocks not reflected in the database file) onto the database file, and bring the database to the latest consistent state for all committed transactions. Below is the algorithm that implements WAL with MDB:

1. When the data needs to commit, it copies the dirty pages (resides in memory map backed by the database file) for that transaction into a continuous memory buffers, and then write them into a WAL file with a "write" system call. The last page for the "write" system call is always the Meta page. Then an "fdatasync" system call is issued on the WAL file.
2. Alternatively, calling "fdatasync" can be eliminated if the OS supports O_DIRECT file operation for the write system call, in such case, blocks are directly written onto the physical storage instead of the file buffers.

[2]If strict transaction isolation semantics is needed, then a transaction must be started as a write transaction; those write transactions are strictly serialized as the single-writer implementation.

[3]There are a few compile time options with MDB to either writing pages into the memory mapped file, or to the memory map backed by the database file; "fsync" or "fdatasync" is for the former, and "msync" is for the later to sync the pages into the storage. We will use "sync" through this paper to refer any of the system calls.

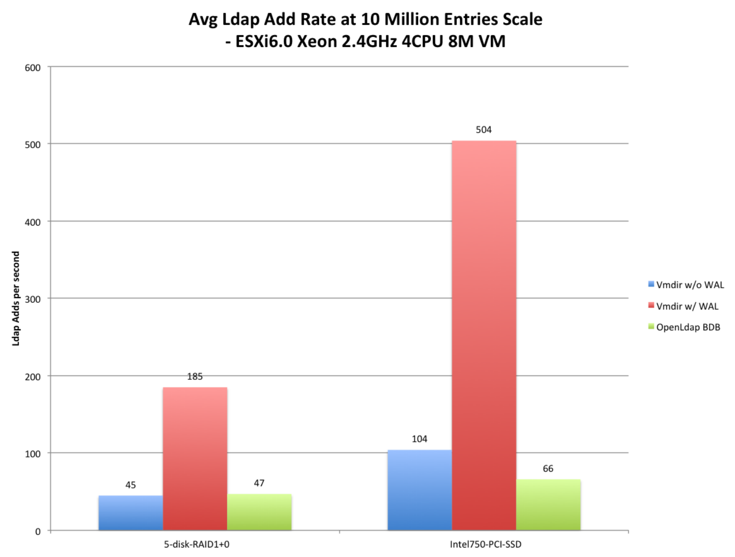

Figure 1: Ten-Million-Entry Scale Write Performance using PCIe SSD and 5-disk RAID

3. Once the "fdatasync" returns, it updates the Meta page in the memory map, and then the global environment variable "dbenv", thereafter the transaction is visible by other transactions. Note that it is unnecessary to do a "sync" on the memory map at each transaction commit, rather the sync is done by a separate checkpoint thread for the purpose WAL files maintenance.
4. A dedicate thread performs the checkpoint function at a fixed internal (30 seconds as in the current implementation), issuing sync on the memory map; once the sync complete, all WAL files created before the sync being called can be purged.
5. If there were a server or OS crash, the small set of WAL files are roll-forwarded to the database file during database environment open stage (when the server restarts), which brings the database file to the latest consistent state for all committed transactions.

The proposed enhancement maintains backward and forward compatibility at MDB data file level, therefore no data migration is needed for data stored in MDB.

4 Write Performance

As shown in Figure 1, the effect of this enhancement is measured on two types of storage that hold the database file: a low latency Intel750 PCIe SSD, and a RAID consisting of five SATA hard drives. The server was running on VMs hosted by ESXi6.0 with a built-in PCIe SSD driver. It shows the performance comparison on a single VMware directory server at ten-million-entry scale between the original MDB and the one with WAL feature. The VM consists of four 2.4 GHz CPUs and eight GB memory. The performance gain from WAL can be seen from about five folds for SSD, and four times for the 5-disk RAID. The graph also gives a comparison with OpenLdap using Oracle's Sleepycat (referred as BDB in the graph), which is at same level of write performance as the MDB without WAL. The contribution has also been evaluated by Openldap[4] to be able to improve synchronized writes by 30 times in its key/value test scenarios.

5 Database File Hot Copy For Creating Directory Replica

The WAL capability enables duplicating a MDB database without putting the directory server at read-only mode. This feature is desired to create a new directory replica in a distributed directory topology since the file copy may take hours for a large database.

To start a remote file hot copy, the checkpoint thread at the source directory server (as described in section 4) must not purging transaction log files until the database file copy is complete, and all transaction log files created during the database file copy are transferred. To reconstruct the replica of the database, transaction log files are roll-forwarded to the database file as the procedure during database recovery (the last step of the algorithm described in section 3). The same mechanism can be used to do incremental database backup, which makes a full backup of the database file in a long interval (e.g. per week), and then multiple backups of WAL files in shorter interval (e.g. when a new WAL file is created or expanded).

6 Conclusion

MDB is a high-efficient database transaction engine to offer scalable read performance with ACID property. However, its single-writer design is a bottleneck for write operations on large-scale deployment in VMs with the host storage built with SSD or RAID based hard drives. This paper presents an enhancement to the transaction engine by adding a Write-Ahead Logging (WAL). The results from our tests and our partner's tests show that this enhancement would improve the write throughput from four to thirty folds in typical deployment scenarios. The hot copy and incremental backup capabilities inherited from WAL closed the gap between MDB and Oracle's Sleepycat.

References

[1] Philip A. Bernstein, Nathan Goodman, "Concurrency Control in Distributed Database Systems", ACM Computing Surveys, 1981

[2] C. Mohan, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", ACM Transactions on Database Systems, Vol. 17, No. 1, 1992

[3] Howard Chu, "MDB: A Memory-Mapped Database and Backend for OpenLDAP", Symas Corp., OpenLDAP Project, 2011

[4] Howard Chu, "What is New In OpenLDAP?", Symas Corp., OpenLDAP Project, Nov. 2015

[5] Jin Fang, "https://wiki.eng.vmware.com/Vmdir_Performance_Evaluation", VMware Internal, 2015

[6] Jin Fang AND Sung Ruo, "Lwraft - A Raft based Directory Service", Pending VMware Radio Acceptance, 2017