BaSE(Byte addressable Storage Engine) Transaction Manager

Sathyanarayanan Manamohan
Hewlett-Packard Enterprise
sathya@hpe.com

Krishnaprasad Shastry
Hewlett-Packard Enterprise
krishnaprasad.shastry@hpe.com

Shine Mathew
Hewlett Packard Enterprise
shine.mathew@hpe.com

Ravi Sarveswara
Hewlett-Packard Enterprise
ravi.s@hpe.com

Kirk Bresniker
Hewlett-Packard Laboratories
kirk.bresniker@hpe.com

Goetz Graefe
Hewlett-Packard Laboratories
goetz.graefe@hpe.com

Abstract

Non-Volatile Memory (NVM) is an emerging memory technology that combines the best properties of current hard disks and main memories by providing non-volatility, high density, high speed, and byte addressability. This provides an opportunity to redesign systems and their software stacks to improve performance and to reduce the complexity. Present-day database systems are designed and optimized for traditional disks and memory hierarchies. They are very complex because they handle varying levels of storage latencies, from CPU caches to hard disks. Our intention is to build a prototype storage engine that is optimized for NVM and which takes advantage of the collapsed memory hierarchy. We are developing this storage engine in an incremental way. In this paper, we describe a novel approach to optimize write-ahead logging (WAL) for NVM based systems.

Most database systems use ARIES-style write-ahead logging to implement transactions. ARIES techniques are optimized for disk based systems and tuned for the sequential write performance of disks. We leverage the high speed, byte-addressable random access of NVM to design a high-performance logging mechanism. We discuss the bottlenecks of sequential logging, identify the challenges of distributed logging and propose a novel solution. We show that NVM-optimized logging improves performance 8-15 times over default MariaDB/XtraDB for log-intensive workloads.

1. Introduction

Transactions are an essential part of OLTP data management systems. Strong transactional support is crucial for supporting the operational activities of all businesses. A significant amount of research effort is dedicated to the design of efficient, reliable and scalable transactions. A key research focus area in transaction processing systems is the support of ACID properties. The transaction systems use write-ahead logging or shadow copy and concurrency control techniques to support ACID properties. Traditional database systems, which support strong ACID compliance most commonly use write-ahead logging. The non-relational data management systems, also popularly called as NoSQL, support eventual consistency properties based on CAP theorem [15] and commonly use shadow copy.

Traditional database storage engines can be divided into four important modules based on functionality. They are Access methods, Log manager, Lock manager and Buffer pool. These four modules account for about 80% of CPU cycles when the database system runs entirely in memory [1].

Most of the transaction processing system implementations rely on DRAM for performance and disks for persistent storage. The data structures and the algorithms are optimized for this type of memory hierarchy. The advent of NVM provides opportunity to redesign and optimize the data structures and algorithms to use a single layer of flat memory. We evaluate various properties of NVM and the opportunity it provides to redesign the transaction management systems that are used in relational databases.

The reminder of the paper is organized as follows. The next few sub-sections provide the background on NVM, transaction management system components and a specific implementation in an open source RDBMS, MariaDB [13]. Section 2 captures some of the related work to optimize the transaction systems. Section 3 discusses the implementation of write ahead logging on NVM and its benefits. Section 4 mentions the prototype implementation and the results. The final section contains our summary and conclusions.

1.1. Non-volatile memory

Non-Volatile Memory (NVM) is an emerging memory technology that combines best properties of today's hard disks and main memories. It offers non-volatility, high speed and byte addressability [19]. There are many different forms of NVM technologies such as Phase Change Memory (PCM) [16], Spin-Transfer Torque RAM (STT-RAM) [17] and Memristor [18] that are being developed actively. Each of these uses a different underlying technology. They exhibit different characteristics in terms of read/write latency, endurance, energy efficiency etc.

However all these NVM technologies offer byte-addressability, high speed read/write access that is comparable to DRAM and storage capacities that is comparable to HDD or SSD.

We design our solution based on these generic properties. We are not dependent on any specific NVM implementation. It is a conscious attempt to make the solution NVM-technology neutral.

Byte addressability provides an interesting programming model as it allows programs to persist data objects directly on NVM without converting them into disk format. We leverage this aspect in our solution to avoid maintaining multiple formats of data and having to convert between them.

There are many different areas in which NVM can be used in a system architecture. The straight-forward option is to use NVM as a disk replacement, which maintains the programming semantics and thus involves the least amount of changes to the software layer. But, in general, this approach does not work because it upsets the optimization and fine tuning for hard disks. The next option is to use NVM as fast cache in between disk and DRAM. The third option is to use NVM alongside DRAM in same address space. This is the most interesting one because the CPU can use load and store operations to access NVM directly. We develop our solution with the assumption of a direct load/store model.

1.2. Transactional logging in relational and non-relational systems

Transaction support is essential in all data management systems for handling failures and reducing the impact of failures on the system's overall behavior. We consider everything from file systems to complex RDBMS in the scope of transactions for data management. Several methods have been adopted to handle failures ranging from protecting the metadata for single operations, like it is done in file systems, to complex multi-operation, multiple data entities in RDBMS. There are also complex combination of techniques that attempt to provide ACID properties to file system operations [7][8].

Irrespective of the use case, the systems have to be engineered from the ground up to support solid transaction semantics and robust failure recovery. RDMBS are built ground up with transaction semantics, but these design choices made them very complex and rigid. These, to some extent contributed to the development of NonSQL solutions that are more flexible. But they make other design choices that, in turn, make it very difficult to support ACID properties. Hence, flexibility and performance of the transaction management system are essential design parameters for any high-performance transaction manager. The emergence of non-volatile memory gives us an opportunity to re-design the transaction managers to achieve these goals and make them suitable for the data management systems of the future.

1.3. Limitations of WAL

Most database systems use ARIES [2] style write-ahead logging (WAL) to implement transactions. ARIES design decisions are made to get optimal performance for disk based systems. ARIES adopts a centralized logging and optimizes it to leverage the sequential write performance of disks. To hide the performance difference between DRAM and disks the log records are cached in DRAM and forced to disk at the time of commit. This creates a two-layered logging system.

The centralized logging with two-layered design causes several bottlenecks. Aether [3] identifies four types of delays that impact the logging performance: (a) IO related delay; (b) excessive context switching; (c) log induced lock contention; and (d) log buffer contention.

Several techniques have been developed to address these problems on the traditional assumptions of slow block oriented disk and byte-oriented DRAM. The NVM technologies open up a new opportunity to optimize WAL for byte-addressable persistent memory.

1.4. Logging in XtraDB

XtraDB [14] is a transactional storage engine for MariaDB. A transactional storage engine, in MariaDB's context, is a pluggable software module that performs various data management operations, such as create, insert, update and delete, on the data that it manages in a transactionally consistent manner. It uses the concept of write ahead logging to manage transactions. XtraDB maintains the transaction logs in DRAM as a circular buffer and a persistent copy of the same content in a flat file on disk. All database operations that are performed by the storage engine to manipulate the database's pages are logged in a Redo log prior to the actual execution. The contents of the redo logs are flushed to disk before transaction commit, in line with WAL semantics. The redo log is used during system recovery. In XtraDB the system recovery starts off by replaying the redo log on to the buffer pool until all the database pages are recovered. This replay starts from the last successful checkpoint. Once this replay is over, the undo log is used to rollback all partially complete transactions.

XtraDB maintains the undo log in memory and on disk. The undo logs contain the before images of database records that were modified by a transaction. The undo logs in XtraDB are used to support transaction rollback, database flush of dirty pages into disk and the concept of MVCC to optimize read performance. The data itself is maintained as a B-tree on disk and a hash table in an in-memory buffer pool. The redo log and buffer pool are flushed periodically flushed into the disk. We use XtraDB as the vehicle to demonstrate the effect of design changes as it is extensively used commercially and it is open source. This allows us to run relevant benchmarks on the solution and prove the solution on real-world applications.

2. Related work

Aether [3] identifies a set of challenges with WAL – (a) I/O related delays, (b) log induced lock contention, (c) excessive context switching and (d) log buffer contention. The paper recommends a set of optimizations using a combination of early lock release and flush pipelining. Early lock release allows transactions to release their locks as soon as the commit records have been made durable. Flush pipelining helps to reduce the I/O delay and log induced lock contention. The authors also recommend redesigning of log buffer to enable better parallelism.

Fung R. et al. [4] describe an approach to implement WAL on storage class memory (SCM). They allocate log records directly on SCM to reduce I/O related delays. They avoid the techniques like group-commit. However, they still write the log records sequentially.

MARS [5] moves most of the logging functionality to hardware and eliminates the Log Sequence Number (LSN) and log checkpointing. It accomplishes this by allowing the storage array to maintain the ordering of write at commit time instead of maintaining the LSN at a software level. MARS also relies on hardware writes to eliminate the need for log checkpointing.

There have been proposals to revisit the design of logging in Flash and PCM based storage [9][10]. Sang-won Lee et.al. propose a technique called in-page logging (IPL) as a new storage model for flash based database systems. To overcome the erase-before-write limitation of flash memory, they propose the IPL technique to co-locate the log and data pages. The IPL-P paper is an extension of the in-page logging method and proposes move the log storage to PCM based storage for better performance.

Tianzheng Wang et al. [20] describe an approach for distributed, NVM-backed logging. They evaluate the performances of two log distribution schemes – one that is distributed on a transaction level and another that is distributed on a page level. They recommend distributing on a transaction level. We show using a transaction level distribution for the undo log and a page level distribution for the redo log performs better.

Jian Huang et al. [21] show that they get the best transactions per dollar rate by moving only the logging subsystem to NVRAM, rather than replacing all disks with NVRAM. In their implementation, they use a circular log on NVRAM and chain the log entries using pointers to enable recovery. However, this slows the recovery process because the entire log chain must be traversed to build the set of dirty pages. This approach to page-level recovery is not efficient.

3. Our solution

We have designed a NVM-optimized logging system that writes log records directly to NVM and have implemented a prototype based on MariaDB/XtraDB.

In the context of our solution, we assume the NVM is directly accessible by the database process using load/store model. We treat it as a byte-addressable persistent memory alongside DRAM memory.

Our design depends on system primitives and programming APIs to read and write the NVM. These APIs shall: (a) support namespaces for NVM; (b) support dynamic memory management; and (c) support variable length read/write operations in an atomic and durable way. There are challenges in implementing variable length atomic read/write operations. The implementation has to take care of write ordering, cache flushes, fault zones, fault containment in NVM etc. Our implementation assumes these challenges will be abstracted and handled by the NVM programming APIs. Atlas [19] is an example of such an API that provides the necessary atomicity guarantees and also handles the memory management. We did not have access to this library at the time of this work and so, we modeled NVM access using mmap files.

There are two main types of log records that are used in traditional ARIES-style WAL: (a) undo log records, which store information about how to undo a change; and (b) redo log records, which store information about how to reproduce a change. In traditional database systems, the log records are cached in memory and persisted on disk as a large sequential file. The log files are flushed to disk before the transaction commits. Database systems employ several optimizations to improve the performance of log writing, such as group-commit [6], which aggregates multiple log write requests into a single IO, and asynchronous commit, which allows transactions to commit without waiting for log IO requests to complete.

We write log records directly on NVM. This eliminates the IO related to log records and also simplifies

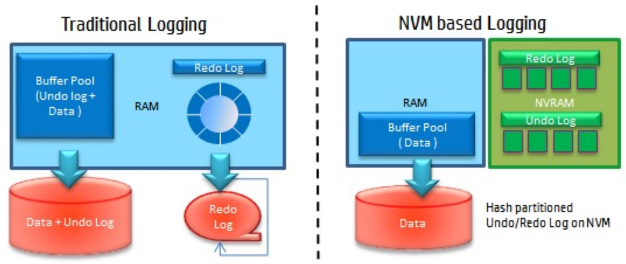


Figure 1: Logging on NVM

log buffer management. XtraDB has separate memory space in DRAM for redo and undo log records. The undo log records are stored along with page data in the buffer pool. The redo log records are stored in a separate circular buffer. In our solution, we do not maintain any of the log records on DRAM. We maintain two separate hash lists in NVM for the redo and undo records. Figure 1 shows log files representation on default MariaDB/XtraDB and our NVM optimized log manager.

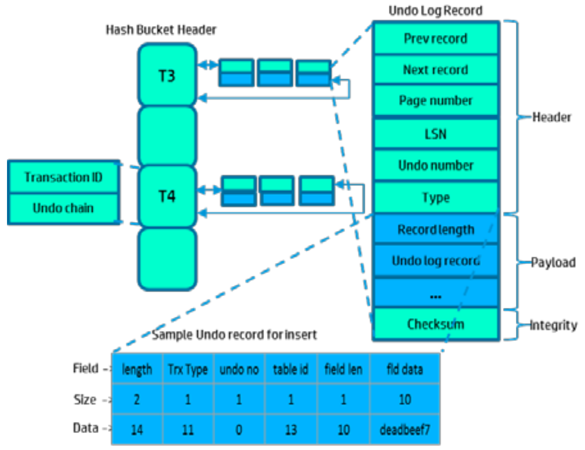
As writes to NVM are very fast, we write the log records synchronously. Persisting of the redo data is synchronous with the commit. We do not wait for a separate thread to flush to disk before finishing the commit operation. This reduces the extensive context switching that occurs due to log-record related IO operations in traditional systems.

Traditionally, log records are cached in DRAM using an in-memory format and stored on disks in a block-oriented format [22]. The log records are converted from the memory format to the disk one while persisting them. The disk format is converted to the memory format when the log records are read for recovery. We avoid the multiple formats and implement a single unified format of log records on NVM. Thus we avoid the extra memory copy and log record conversion complexity.

Synchronous write also allows our solution to eliminate the group commit. This in turn reduces the log induced lock contention.

To overcome the log-related contention, we have re-designed the redo and undo logs. The operations on these log files exhibit different degrees of parallelism. We have designed them with different parallelism schemes after taking their usage into account. The undo operations are applicable for a transaction and hence can be parallelized at transaction level. The redo operations are applicable for a page and hence can be parallelized at page level. We observe these parallelism needs of undo and redo operations and design a customized distribution scheme. We distribute undo log records based on transaction ID and implement it as a linked list of undo records belonging to a transaction. We implement a hash based distribution of transaction IDs.

Figure 2 shows the structure of undo log records. The information about transaction state and pointer to undo log records are maintained in a hash table. The undo log records, that contain undo number, page number, LSN

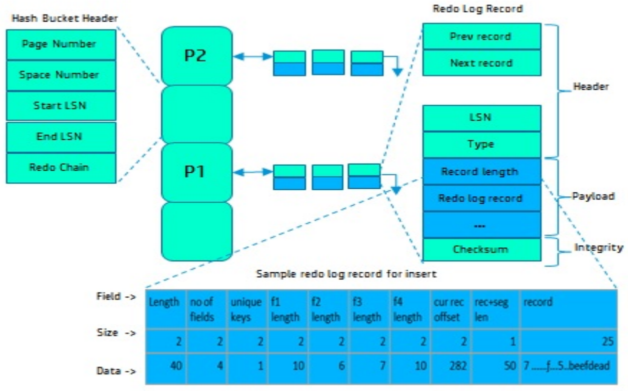


Field ->	length	Trx Type	undo no	table id	field len	fld data
Size ->	2	1	1	1	1	10
Data ->	14	11	0	13	10	deadbeef7

Figure 2: Undo log record structure

and undo operation details (called as payload) is implemented as a linked list. This eliminates the contention for writing undo log records from multiple transactions. Only during the beginning of a transaction we need to acquire a lock to get the corresponding hash slot. Thus we improve the concurrency of undo log operations.

Similarly we distribute the redo log records based on page id. Figure 3 shows the structure of redo log records. We store the details of page number, start and end LSN of the page and pointer to redo chain in hash bucket header. We store redo records that consists of transaction ID, record type, LSN, record payload etc. as a linked list. This reduces the redo log write contention across the pages. Only the parallel transactions that operate on the same page contend for the redo log chain. With this customized distribution of redo and undo log records we can implement more granular latches and increase the parallelism in logging operations. This reduces the log buffer contention and improves the performance.



Field ->	Length	no of fields	unique keys	f1 length	f2 length	f3 length	f4 length	cur rec offset	rec+seg len	record
Size ->	2	2	2	2	2	2	2	2	1	25
Data ->	40	4	1	10	6	7	10	282	50	7......f...5..beefdead

Figure 3: Redo log structure

ARIES recommends periodic checkpointing to accelerate the recovery. The checkpointing flushes the log records and dirty pages in buffer pool to disks. The checkpoint log record holds information about active transactions, its state and flushed LSN. Since the active transactions and their states are directly written to NVM along with undo logging, the checkpoint log record has to just write the flushed LSN. This improves the checkpointing performance.

Our parallel hash based distribution of the redo and undo log records opens up the opportunity to parallelize recovery operations. ARIES recommends the recovery in 3 phases: (i) analysis phase – during which the algorithm reads the flushed LSN information from checkpoint and scans the log records sequentially to gather the required redo and undo operation information, (ii) redo phase – during which the redo operations are applied to bring the database back to the state before the crash; and (iii) undo phase – during which undo operations are applied to reverse the effect of inflight transactions. Our distributed redo log records enable parallelism in building and applying redo operations. Distributed undo log records enable parallelism in rolling back the inflight transactions. Thus we improve the recovery performance. The benefits of partitioning log records are explained in the following sections.

3.1. Partitioning log structures

Partitioning of log structures addresses several bottlenecks that are seen with sequential logs.

Centralized log structure is well suited for disk based systems. In such systems, log records from multiple transactions are logged into a centralized log structure. These log records are persisted on disk using the sequential IO. Optimizations like group-commit are done to further improve the IO performance. But a centralized log creates synchronization problem. Multiple transactions that run in parallel will contend to get the lock at the head of the log structure to write log records. The contention increases with number of parallel transactions and causes a concurrency limitation. Since the threads have to wait for the lock they get context switched. This also increases the number of context switches and impacts the performance.

The centralized logging will also impact the parallelism of recovery operation. To parallelize the recovery operation, the recovery system has to process the sequential log to extract the undo and redo information into some partitioned structure. Typically the recovery system does this during the initialization phase. Otherwise, the system would be scanning and applying the sequential logs one record at a time, which will significantly increase the recovery time.

Our design avoids these problems by partitioning the undo and redo logs in NVM. We use hash based partitioning scheme. By hashing we break up the single centralized log head into several streams equal to the bucket size of hash table. This eliminates the bottleneck on global log by parallelizing access to the logs. This will enable several threads to write to the log records simultaneously. Consequently, this reduces log induced contention and context switching.

Also having to separate logs structures for undo and redo logs allows us to partition the log in the most optimal way based on the use case. Undo logs are closely associated with a transaction hence we partition it based on transaction Id. This allows us to create sufficiently large hash buckets to the extent where we can completely eliminate the need of synchronization constructs and make the undo logging practically lock free. As an example, if we optimize the system to handle 2K concurrent transactions, we can create an undo log hash table having more than 2K (closest prime number) bucket. In this way every transactions will get its own exclusive hash bucket there by eliminating the need to have a synchronization construct to manage the undo log. This also simplifies the search of undo logs during recovery operation, as all the undo logs pertaining to individual transaction that needs to be rolled back during recovery are found grouped in the same hash bucket.

Similarly redo logs are partitioned on page ID that allows several database threads to operate in parallel as long as they don't try to append redo log records on the same page. Due this approach, optimizations like grouping redo logs prior to append into the global logs are no longer needed. Transactions can directly append the logs into the log structure directly as and when they are generated. This also simplifies the recovery operation. During recovery multiple threads can be created to recover the pages in parallel.

3.2. Benefits of undo and redo log optimizations

In this section, we explore the implications of the design optimizations that we explained in the previous section. The first implication is the simplification in the process of releasing locks that were held by the transaction. Transactions acquire locks to protect the data that it is using, against possible corruption from concurrent access. This ensures the isolation guarantee of the database is maintained. Lock release happens at the end of a transaction when the commit status of the transaction is flushed into durable media. In traditional systems, we have to wait for the flushing of commit records to complete. This creates an I/O bottleneck. Transactions are made to wait [3] [6] for the grouping of log data to be sufficient enough to overcome the cost of a doing a serial I/O to disk. Our solution eliminates this completely because the logs are directly written to persistent media in their native form. Our solution does not maintain two distinct data structures, one where we buffer the logs and the other that is used to do bulk I/O to the disk. This approach also simplifies the code. Due to this design attribute of our solution, locks can be released as soon as the commit record is posted into the data structure.

The second implication of our design is the reduced context switching. Most modern databases are multi-threaded to take advantage of the abundance of compute cores available in state-of-the-art CPUs. However, even though this is largely beneficial, due to I/O and synchronization bottlenecks, much of the compute cycle is wasted in context switching and spin locks. Our solution addresses the context switching part of the problem. The synchronization problem is tougher to handle as it requires a complex redesign of XtraDB to resolve synchronization bottlenecks. In our solution, the persistence of log records is now reduced to a write operation to a NVM resident data structure. This now can be done in the same thread, without having to wait for I/O operations, which are usually done by other I/O threads [3]. This optimization allows us to drive the cores to do more user work rather than waiting for I/O operations to complete.

The third implication of the design is the elimination of log multiplexing, which attempts to combine logs records from various transactions to achieve optimal volume to make flush operations efficient. On disk based logging systems [2] [3] [6] it is used as a method to reduce the I/O overhead for writing to the disk. In flash based systems [9] [10] it is used as a method to reduce erase-unit overheads. In both the systems it is quite possible that the system writes more data than what was actually updated. This arises due the block oriented nature of writes on these systems. On disk based systems the block is usually 4-16 KB and on flash based systems the block size (erase unit size) is 128KB. Since our solution relies on byte-addressability we write variable log data without having to worry about block boundaries. This results in faster commit time and better utilization of cores to do more useful work.

3.3. Implications on checkpointing operation

In data management systems, a checkpoint can be treated as a marker that indicates the extent to which state information has been transferred to the secure persistent storage. Modifications to data pages are not necessarily flushed to disk in a synchronous manner for performance reasons. Checkpointing is a costly operation and has serious impact on the throughput of the system. Checkpoints are classified into two categories: full checkpoint and fuzzy checkpoint. In a full checkpoint, the data management system writes all the dirty information to the disk. The fuzzy checkpoint, which is commonly used for performance reasons, writes only a certain number of dirty pages. Fuzzy checkpoints are used in XtraDB.

Checkpointing in NoSQL solutions, like HBase, are more like full checkpoints where the entire of the old version metadata and the logs are combined synchronously on a standby node to a create the newer version of the metadata that is then gradually transmitted to all the active node servers in the system. What ever the method used, the system will experience a drop in performance for the duration of the checkpointing operation. In our solution, due to NVM, the flush of the redo logs is completely eliminated and the fuzzy checkpoint needs to maintain only the state of the pages that were flushed from the buffer pool. This also simplifies page stealing because the logs are already persisted and hence the buffer pool manager has the freedom to pick up dirty pages on demand. This makes the checkpoint process and page stealing simpler and faster.

3.4. Implications on crash recovery operation

Recovery or crash recovery operation rebuilds the internal data structures of the data management system to a consistent state from which the storage engine can start processing transactions again. The recovery process also ensures that the overall consistency of the data is maintained. In XtraDB, the recovery happens in several steps. These steps are semantically similar for any data management system which supports crash recovery. The first task is to bring the data pages that were present in the buffer pool without being flushed to persistent media, back to a consistent state. This is done by applying the redo log from the last checkpoint forward until all of the redo log is exhausted. This process brings the buffer pool up to the state just prior to the point of failure.

However the buffer pool also contains dirty data pages that are part of incomplete transactions. These transactions need to be rolled back. The undo logs are used to perform this operation. There are several variants in the recovery process based on the richness of the redo and undo information stored by the system prior to crash. In the case of file systems, the recovery is usually limited to metadata. In more complex systems, higher levels, that might include actual file data, are supported [7][8]. In NoSQL systems, the recovery is only in the forward direction as they typically do not store undo information. These systems rely on replication and some variation of voting to get the data to a consistent state eventually. RDBMSes have richer information in their logs and hence, can restore the database to the closest possible state prior to the crash, when compared with all other data management systems.

In our solution we support both undo and redo logs. Hence our transaction manager can be used to perform RDBMS-like recovery. The primary performance bottleneck with recovery operations is the time spent in doing lots of random IOs to get the buffer pool back to a state where undo information can be used. Also, processing and converting the block based redo logs on disk to a format that is usable in DRAM impacts recovery performance. It should also be noted that the system is not available for transactions until the buffer pool is restored by the redo log.

In our solution, hash partitioning the redo log on PageID enables parallel recovery of pages. Instead of reading a serial redo log, recovery threads are assigned to process several hash buckets in parallel. Our solution also has a single format in which the redo logs are stored hence the cost of converting disk based structure to a DRAM based structure is completely avoided. Also, undo of in-flight transactions can be parallelized because the undo log records are hash partitioned on TranasactionID. These improvements can significantly reduce recovery time of the system that uses our transaction manager.

4. Performance evaluation

We implemented the techniques described in section 3 on MariaDB/XtraDB storage engine. We used a simulated NVM environment for the validation and performance measurement.

Our prototype was built on a Linux machine with 16 GB of RAM running 8 CPU Xeon Processors with 2 cores. The system had separate disks for the host operating system, and the data and log files used by XtraDB. All disks were standard 7200 RPM 200 GB IDE disks.

4.1. Storage engine development

We prototyped our log manager and integrated it with MariaDB/XtraDB. MariaDB is a popular open source fork of MySQL. We wanted to test our solution on a commercial database software to get a better understanding of the implications of a new log manager in the real world operation environment.

We followed a modular design approach to develop the new log manager. The interactions with the NVM device and memory management functionalities were developed as a pluggable component. This provides an easy option to plug in different NVM technologies. We isolated the creation, modification and management of redo and undo log records into a separate module. We defined a set of APIs to interact with the log manager. The parallelization of undo and redo log records, maintenance of hash structures are all contained in the log manager code. The log manager is integrated with XtraDB using the APIs. And the XtraDB code is modified to write log records using the log manager interfaces. The recovery module of XtraDB reads the log files from disk and prepares an in-memory structure to parallelize the recovery option. We bypassed this layer as the log records are already partitioned according to their usage pattern.

We configured XtraDB data files and log files on separate volumes. This helped us to understand the I/O characteristics of the workload. Separation of the log volume from the data volume helped us to understand the characteristics of logging and make a good performance comparison.

4.2. NVM simulation

At the time of prototyping, we did not have access to a NVM device. We simulated the NVM using the Linux file system. We used this NVM simulation to demonstrate both the functionality and performance gains.

To demonstrate the parallel logging and recovery, we implemented the log manager using mmap files. The redo and undo log records are persisted on mmap-ed files. We manually crashed the system to force the recovery operation. During the recovery process, XtraDB reads the log records from the mmap files to reconstruct the undo and redo operations.

We implemented log records on mmap files from tmpfs to demonstrate the performance gain with NVM. Memory mapping tmpfs files avoids the IO operation. This simulates the implementation of log records with read/write access latencies of memory which is an idealistic NVM environment. In reality there might be different write/read speeds for NVM systems.

4.3. Performance test bed

We used the TPC-C [12] benchmark and a custom built insert workload to evaluate performance of our solution. The TPC-C benchmark is an online transaction processing benchmark. It is based on an order-entry use case with several complex transactions that simulate real-life operations on a typical order-entry system.

We developed a custom built parallel insert program to simulate logging intensive workloads. The program inserts 2 million random records of 80 bytes length using tunable number of concurrent threads.

We measured 3 dimensions of performance: a) Elapsed time to complete a fixed number of inserts; b) the CPU and I/O utilization characteristics for the inserts; and c) End-to-End transactions per second on TPCC workload. The results are discussed in the next section.

4.4. Results

We ran the insert program with 1, 2, 4, 8 and 16 parallel threads. We measured the overall time taken to ingest 2 million records by the standard XtraDB program and our modified XtraDB that has our log manager (henceforth referred to as BTM.)

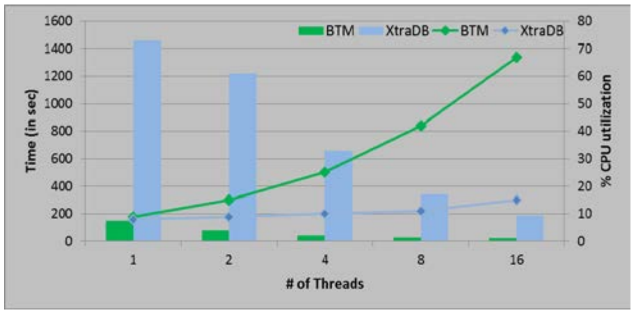


Figure 4: Performance comparison of BTM and default XtraDB

Figure 4 shows the execution time comparison for this insert program on the standard XtraDB and our BTM. We demonstrate 8-15 times improvement in performance.

We also measured the CPU utilization. We get good CPU utilization with BTM. We enable more processing by eliminating IO and lock bottlenecks. This reduces the time required to finish the work and results in higher throughput. In the case of the default XtraDB code, the CPU utilization does not go above 15%. The threads spend most of the time waiting on either IO or on locks for log records.

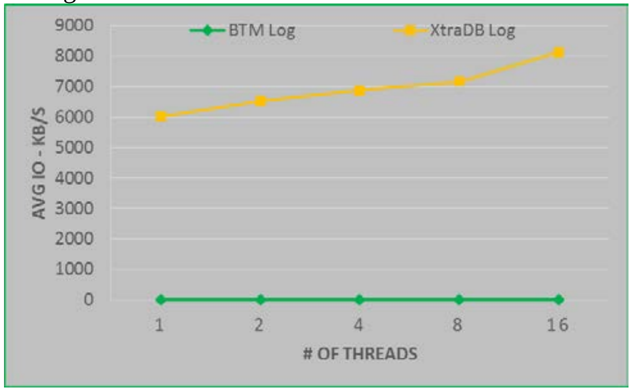


Figure 5a: Log IO comparison of BTM and default XtraDB

Figure 5a and Figure 5b show the IO operations on both data and log volumes. Figure 5a indicates the elimination of log IO in BTM. By eliminating wait times and lock contentions, BTM is able deliver better throughput and process more data. This is seen in Figure 5b.

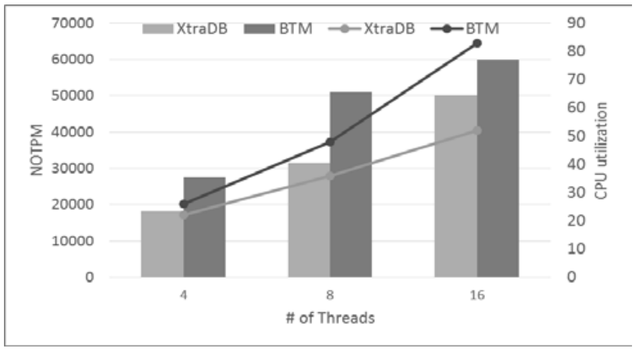


Figure 6: TPCC performance comparison

Figure 6 shows the performance comparison for TPC-C benchmark and the corresponding CPU and IO utilization. Our implementation eliminates the log IO and improves the CPU utilization. We get around 1.2-1.6 times improvement in the throughput. The improvement is limited by the lock contention in other modules of MariaDB. Similar to insert benchmark, we see good reduction in CPU utilization with BTM log manager. This is again attributed to the reduction in lock contention for log records.

5. Conclusions

In conclusion, our design improves the performance of transaction manger by eliminating disk I/O that is needed for performing log operations. In the process, it simplifies the code and reduces the path length due to elimination of log I/O and synchronization code. It improves throughput of the system by parallelizing access to log data structures and eliminating single entry point bottlenecks. It improves core utilization as now most of the time is spent in doing useful work than waiting for I/O completion or latches. It also eliminates the need for techniques like flush pipelining, group commits and early lock release. We have also shown that these optimizations results in 8-15 times improvement in performance and higher CPU utilization for doing useful work.

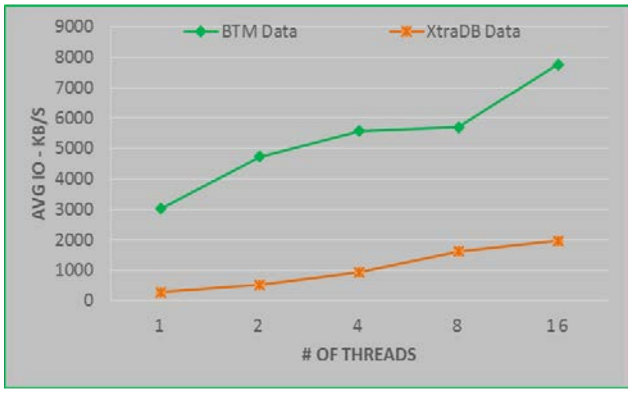


Figure 5b: Data IO comparison of BTM and default XtraDB

6. References

[1] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, Michael Stonebraker "OLTP Through the Looking Glass, and What We Found There" SIGMOD 2008

[2] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. "Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging." ACM Trans. Database Syst., 17(1):94–162, 1992.

[3] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. "Aether: a scalable approach to logging." Proc. VLDB Endow, 3:681–692, September 2010.

[4] Ru Fang, Hui-I Hsiao, Bin He, C. Mohan, Yun Wang. "High performance database logging using storage class memory." ICDE, pp. 1221–1231, 2011.

[5] Joel Coburn, Trevor Bunker, Rajesh K. Gupta, Steven Swanson. "From ARIES to MARS: Transaction support for next-generation, solid-state drives." SOSP, pp. 197–212, 2013.

[6] P. Helland, H. Sammer, J. Lyon, R. Carr, and P. Garrett. "Group Commit Timers and High-Volume Transaction Systems." In Proc. HPTS, 1987

[7] Wright, Charles P.; Spillane, Richard; Sivathanu, Gopalan; Zadok, Erez; 2007; "Extending ACID

Semantics to the File System; ACM Transactions on Storage

[8] Spillane, Richard; Gaikwad, Sachin; Chinni, Manjunath; Zadok, Erez and Wright, Charles P.; 2009; "Enabling transactional file access via lightweight kernel extensions"; Seventh USENIX SIGMOD international conference on Management of data Pages 55-66

[10] Kang-Nyeon Kim, Sang-Won Lee, Bongki Moon, Chanik Park, Joo-Young Hwang "IPL-P: In-Page Logging with PCRAM" VLDB 2011

[11] Jim Gray and Andreas Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann, 1993.

[12] Transaction Processing Performance Council. “TPC - C v5.5:On-Line Transaction Processing (OLTP) Benchmark.”

[13] MariaDB. URL: https://mariadb.org/

[14] XtraDB. URL: https://www.percona.com/software/mysql-database/percona-server/xtradb

[15] Seth Gilbert and Nancy Lynch, “Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services”, ACM SIGACT News, Volume 33 Issue 2 (2002), pg. 51-59

[16] M. J. Breitwisch. Phase change memory. Interconnect Technology Conference, 2008. IITC 2008. International, pages 219–221, June 2008.

[17] B. Dieny, R. Sousa, G. Prenat, and U. Ebels. Spindependent phenomena and their implementation in spintronic devices. VLSI Technology, Systems and Conference on File and Storage Technologies (FAST 2009)

[9] Sang-Won Lee, Bongki Moon "Design of Flash-Based DBMS: An In-Page Logging Approach" SIGMOD '07 Proceedings of the 2007 ACM

Applications, 2008. VLSI-TSA 2008. International Symposium on, pages 70–71, April 2008.

[18] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. Nature, (7191):80–83, 2008.

[19] Dhruva R. Chakrabarti, Hans-J. Boehm, Kumud Bhandari. "Atlas: leveraging locks for non-volatile memory consistency", Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, pages 433-452.

[20] Tianzheng Wang, Ryan Johnson. "Scalable Logging through Emerging Non-Volatile Memory," Proceedigns of the VLDB Endowment, Vol. 7, No. 10, pages 865-876.

[21] Jian Huang, Karsten Schwan, Moinuddin K. Qureshi. "NVRAM-aware Logging in Transaction Systems," Proceedings of the VLDB Endowment, Vol. 8, No. 4, pages 389-400.

[22] Jay Janssen. "The relationship between Innodb Log checkpointing and dirty Buffer pool pages." URL: https://www.percona.com/blog/2012/02/17/the-relationship-between-innodb-log-checkpointing-and-dirty-buffer-pool-pages/