SepLog - *Separating the Database Log To a Remote Machine*

Roee Ebenstein
The Ohio State University
2015 Neil Ave.
Columbus, Ohio
ebenstei@cse.osu.edu

ABSTRACT

Current database systems performance is dependent on the database logging mechanism. The database log mechanism, which tracks every change to the database content, is the heart of the crash recovery. Each data change has to be logged before a database user who issued a data manipulation command gets a confirmation the command finished execution. This creates a synchronization bottleneck that affects performance tremendously.

In this paper We introduce SepLog, a logging service in which the log itself is hosted in-memory on a remote server, separating the DBMS logging component from the DBMS engine. We describe SepLog in details, and show it improves DBMS log performance, which leads to higher transactional throughput and lower transactional latency compared to storage based systems, while maintaining the same durability properties.

1. INTRODUCTION

One of the major bottlenecks in database systems is the logging component, particularly the redo-log files filling process. This bottleneck exists by design to enable one of the most basic database features - durability. The bottleneck is rooted in three different contentions: a flush to disk, sequential writing to the log (usually being enforced by a mutex), and the checkpoint mechanism.

The checkpointing mechanism is in place for enabling crash recovery. A checkpoint will be executed before the redo-log content is going to be written over - and lost - assuring the pre-write-over content will not be lost (i.e. each and every committed datum will be recoverable either from the redo-log or from the data files). The checkpoint enforces the physical data files on storage to synchronize with the in-memory caches and the redo-logs in a manner that assures the system is recoverable, and in the case of a crash data would not be lost. If needed, the checkpoint will stall the system while the cache buffers are being flushed to the non-volatile storage (disk).

The sequential write to the redo log is in place for protecting data structures as well as ordering the commands for the recovery process. In some cases, the logging I/O is asynchronous, which results in parallel execution of disk reads and writes. Further, the access to key objects has to be ordered occurred, assuring that no concurrent access to these objects occurs and that the flush flushes all previous pending log data, including the log data of other sessions. The disk flush assures that the content, which would have been required in the case of crash recovery, will be available and accessible after a crash happens.

In this paper we suggest tackling the bottleneck the traditional log system architecture causes by:

1. Storing the redo-log content in-memory on a remote machine instead of on disk files. We utilize this approach since network performance properties and the probability of a computer crashes favor distributed in-memory systems which communicate over the network. We claim that the suggested architecture is at least as likely to keep the log content safe and reachable as traditional system architectures do, while improving performance drastically.
2. Changing the log system contention to be only on the data structures that represent the log within the system and not to use redo-log buffers, which flush the redo-log to disk, at all. This decreases the overall lock time within the logging system enabling better performance.

We claim neither the logging component should be rewritten end-to-end, nor that the database architecture should change. Our work shows the contrary. Current database architectures are robust. Similarly to the conclusions presented in [4], we conclude that because local changes sufficed for our development, no change is necessary to database architectures.

In our work, we increase the throughput while decreasing the latency of the database log component by extracting the database logging storage component from the database engine and hosting it within memory of a remote server. This concept transfers the storage load the log component imposes to the networking device, which makes it the bottleneck for the logging component. We believe that, in the case of logging, it is easier to scale, out and up, networking than storage since the I/O requests are relatively small and being written sequentially. The reasoning behind these claims is rooted in the observation that each networking device has higher throughput and lower latency than each storage device, including SSDs (please refer to [9] for storage properties, and to [16] for the motivational discussion on switching to remote memory usage, with its properties).

Our implementation is based on an extension of MySQL, which uses MPI for intra-node communication on the Ohio Supercomputing Center (OSC) Oakley cluster. In this environment, we have used the local storage system and the default infiniband networking configuration. For details about the environment hardware, please refer to [19]. The MPI implementation we used is described in [20].

The structure of this paper is as follows: In section 3, we describe how database logging works in current systems (focusing on the cyclic logging approach). In 4, we describe in memory log storage and its implications. In 5, we describe our implementation in more detail and in 6 we evaluate it. In 7, we discuss future work. In the following section, section 2, we discuss related work. We conclude in section 8.

2. RELATED WORK

In [15] the authors conclude that database logs are a significant bottleneck in a DBMS system. In [13], the authors identify four logging related impediments, including I/O related delays. In [12], the authors claim this problem is becoming less important with time, due to the development of SSDs. In [15] and [14], the idea of using SSDs instead of magnetic disks for database logging is being thoroughly developed. Experiments show that the performance of SSDs for logging is much better than of a magnetic disk. In [17], the authors claim that the endurance of these devices is much better than those reported by the manufacturers. The question of "Are SSDs a good fit for the database logging component, due to the DB log characteristics" remains unanswered in literature. It should be further researched; the shorter and limited endurance of these devices with the extensive writing of small chunks suggests to us that SSDs are not a good fit for DB logging. In addition, SSDs are also very expensive for enterprise use. It is so expensive that, in [6], the authors suggest to retain the database log files on thumb, USB flash, drives RAID to avoid expenses.

All these studies are rooted in the stalls which occur while waiting for a non-volatile memory confirmation of write before a transaction can end. These stalls are being imposed on a DBMS system by the current logging mechanism architecture. For durability, it is necessary to pay the cost of system stalls for the log data.

Following the ideas presented in [16], we suggest using the RAM of a remote machine for holding the log data instead of on disk. This idea innovatively allows the transaction manager to return a commit ACK after the transactions have been sent over the network to another machine, instead of waiting for the storage ACK.

Network latency is lower and throughput is higher than respective storage based values per device (SATA 3 is limited to 6Gb/s, SATA 3.2 gets to 16 Gb/s compared to networking speeds that can get as high as 160Gb/s). In [10] and [16], there is a thorough review of storage and network properties and limitations. In many other components, the requirement of the system is to use non-volatile storage; we claim that for the log system we can suffice in remote machine volatile memory instead.

Following the data presented in [22], if current hardware fails, it is more likely that the failure is storage-related than RAM-related. The information in this paper refers to magnetic disks, not SSDs. We expect SSD endurance to be lower than the hardware examined in this paper. Therefore, the likelihood of a storage failure using SSDs should increase even more. It is clear that removing a component from the system, such as a hard drive, will increase the reliability of the whole system ,since there are less components that can fail.

Our implementation, SepLog, stores data in-memory while the database is running. It is important that, when turning the system off, we use non-volatile memory (disks) for storing the log content into traditional files. These files are then read when the system is being turned on again.

With SepLog, one can still use SSDs for the non-volatile memory if needed for archival or regular storage of data files while implementing the techniques shown in [13],[5],[12]. SepLog can also complement the concepts like those presented in [2](Tango) and [3](Hyder), which are based on the ideas presented in [1](CORFU).

A lot of work has been done in the distributed in-memory databases domain. Our work is targeted towards the traditional storage based DBMS. In the distributed in-memory database domain, new commit approaches have been introduced in addition to different methods of locking and relaxation of transaction visibility approaches. Traditional systems perform local locks and have simpler commit and synchronization protocols, which, in our architecture, the system preserves. Our system is not a remote key-value system. Although we could have used some of the approaches introduced in other work, its performance would have been limited. In our work, we can assume rare reading, sequential writing, and no write conflicts, which in other works the researchers could have not.

Our work maintains all the advantages a centralized DBMS has, while offering a performance boost that almost reaches the performance available today only on in-memory DBMS. It is important to emphasize that SepLog will not perform better than in-memory DBMS. However, it will allow a storage-based system to almost reach the performance level offered by in-memory DBMS, while maintaining all the storage-based systems properties that do not exist in in-memory systems.

3. LOGGING

In this section, we review how the cyclic redo-log mechanism works. There are other database log mechanisms that we do not cover here. In our current implementation, we do not support dynamically growing log sizes, which does not allow us to implement these log mechanisms using SepLog. Our conclusions still apply to these logging methods. We chose to focus on the cyclic redo-log since this mechanism includes most of the complexities that exist in the other log mechanisms. It allows us to focus on a limited number of commands, which will be presented in the following section. This is in addition to the large popularity of cyclic logging, which makes it attractive as a show case.

Cyclic redo-log is a logging mechanism based on a pre-allocated memory area, which is being managed in a way that allows full recovery from every crash, assuming log content is reachable. This is done by having at least two redo-log files, which can be in one of three different states: current (which we will refer to as the "ACTIVE" log file), Currently not being used (which will be referred to as "IN-ACTIVE"), and "IN-USE" (which will not be discussed in detail here).

There are two terms that are critical for understanding the cyclic log model: 1. Log flush, and 2. Checkpoint.

The log flush (in this paper, wherever we write "flush" without mentioning the content of the flush, we refer to log flush) is an action that assures that the content written to the log so far will be accessible after a system crash (in traditional systems it is accomplished by using disk). In most systems, we accelerate the disk write by making it either asynchronous or by buffering the writes. For logging, we cannot do so for some commands because when the "write to log" command returns control to the requester, the data of the requester has to be recoverable, i.e. accessible in case a crash occurs. The recovery process can be completed only if all committed data was sent successfully to the log file and is accessible for the recovery process after a crash has occurred.

The checkpoint is a mechanism that assures all the data cached in memory that have not yet been written to disk finds the way to its designated place on the non-volatile memory.

One can think of the DBMS system as one composed of two different types of files for data: the data content and the log content. The data that was committed has to be in at least one of the file types, and for each file type there is a different flush mechanism. Both processes, checkpoint and flush, cannot occur simultaneously because of the inherent dependency between these processes.

The cyclic log, as the name suggests, works in a cyclic pattern. Each of the log files can be in one of three states - "ACTIVE", "INACTIVE", and "IN-USE". When a log becomes "ACTIVE", the system will write new log records to it. When a new log record arrives, it is tunneled to the current "ACTIVE" redo-log. The "INACTIVE" state means the log file is currently not being used at all. The "IN-USE" state means an operation currently accesses the redo-log, but it is not being appended (this state will not be explained further in this paper.) It is important to notice that there is, at most, one active redo-log file at every moment.

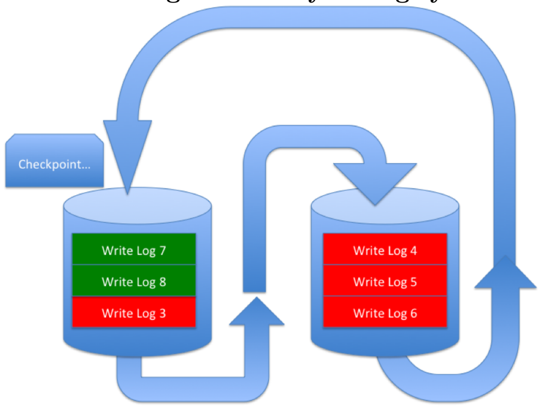
As mentioned before, there are at least two log files in the cyclic log mechanism. These log files are ordered in the system, and usually are used in the predetermined order. The system always writes log records to the current "ACTIVE" log file sequentially in the order it arrives. When the current log file is full, the system changes the status of the current file to "IN-USE" or to "INACTIVE" (if the file status becomes "IN-USE", it is marked "INACTIVE" after the action that marked it as such finishes). Before the next log file becomes active, the system verifies that it is currently "INACTIVE" and is not about to write over data that, in case of a database crash, would be needed for recovery. If such datum exists, the system performs a checkpoint while the log freezes the whole system. After the checkpoint completes, if it was required at all, the next log file becomes "ACTIVE" and the system allows new log requests to be submitted to it. The system knows which records are required for crash recovery by the database timing scheme (LSN in MySQL). For more information about the recovery process, please refer to [11].

Some DBMS's optimize this process by having small checkpoints between full log cycles. Thanks to it each checkpoint will not have to flush all the new data but only a small section of it. It increases the probability that by the time a switch (of logs) occurs a full checkpoint is not necessary.

Among the requests that the database gets, there is one special command - COMMIT. The COMMIT command is special log-wise because this is the only client command that forces log flush. It is not important if the command is being given explicitly (by committing a transaction) or implicitly (by a DDL, for example). In both cases, a log flush will be executed. A ROLLBACK does not have to trigger a flush in case of recovery - if a COMMIT is not found in the log, a ROLLBACK is assumed. However, the ROLLBACK command is sent to the log, like each and every other client command, given the ROLLBACK command in the log, the buffers of the recovery process remain small and the process becomes more efficient.

We want to emphasize two important observations: 1. A COMMIT forces log flush. This means that when a COMMIT is being sent from a client, the client will be stalled until the database can assure it can recover the committed data in the event of a crash. 2. A checkpoint (that runs at least once in a full log switch cycle, i.e. before over-writing data in the log) might stall the whole system. When a log switch occurs, the content of the 'about to become' active file is about to be over-written. Until all the counterpart's cached data blocks are on disk, its final destination, the log will not write requests (and will freeze all log input channels). The freeze will be released only after the checkpoint completes and the relevant content is on non-volatile memory.

Figure 1: A cyclic log system



In figure 1, there are eight log records that have been submitted to the system and each log has room for three records. In a real system, the records are not fixed in size, but, in the figure, they are for demonstration. When the system started, the left redo-log was the active one. After the third record had been submitted, the active redo-log became the second one and the first one became inactive (since the system was just started before, no checkpoint is necessary). After the sixth record, the second redo-log became inactive. Because the seventh record was supposed to overwrite the content in the first log file, a checkpoint was triggered before the first log became active again (an action that results in a data loss from the log file). After the checkpoint completed, the first file became active again and digested the rest of the log records. In the diagram, the records before the checkpoint are colored in red, and those after in green.

One can deduce that choosing the log file size, the number of files, and making sure all the log files are the same size are critical for the database performance. If the log file is too small, checkpoints will occur often, which will freeze the system and will harm the system performance. Having different file sizes will result in one log that fills faster than others. It might harm the system in many ways that are out of the scope of this paper, such as building heuristics for checkpoint predictions. The number of log files affects the time between forced checkpoints. The wrong number of files will harm some mechanisms that are out of the scope of this paper, such as archival. In addition to these, other mechanisms exist in database systems that can be affected by the log size and the number of log files. One thing to remember - the total size of the log files is the size that will be read for recovery, so there is a direct relation between the log size and the recovery duration.

3.1 Recovery

The recovery process is the only DBMS process that reads the log (there are other user initiated processes that can read the log, but these are not critical for the DBMS). On startup, the DBMS scans the log and finds the log starting and ending points (the earliest and latest log records and their positions; this is done by using the timing sequencing mechanism mentioned earlier) for recovery.

From the content that is read from the log, the system knows what values (and where they are supposed to be on disk) are supposed to change for each transaction. Using that information, the system "rolls" the commands that are in the log for each transaction, one by one, mimicking each transaction as if it is currently running. This process recovers the state of each transaction from the last checkpoint until the crash.

Each transaction that was not committed when the last log record was applied will be rolled back.

In this section, we summarized the recovery process from a very high level. For more details, please refer to the related work and the crash recovery section in [11].

4. LOG SEPARATION

There are multiple options for log separation from the DB engine. In this section, we will discuss these options and motivate the decisions we took in our implementation.

4.1 Architectural options

The redo-log component in the DBMS is part of the durability assurance mechanism. There is one transactional command that directly influences the durability of the system - COMMIT. Since the system has to be durable only for committed transactions, DBMSs leverages the no need to flush log data to disk on any other command. The most basic optimization that almost all the DBMSs implement is the redo-log buffer, which holds the redo-log data in memory until a flush command forces it to be written to disk.

As mentioned before, the log records are small compared to the I/O blocks in most cases and the content is being written sequentially, which makes the optimization above very beneficial compared to the alternative. It allows buffering commands between different COMMITs and flushing them together to increase the system throughput because the I/O devices behaves better with larger and sequential writes. Storage devices also have better latency with blocks that are given in a sequential order. Therefore, this optimization helps fulfilling multiple goals. The redo-log buffer is flushed directly to the files on disk.

The logging component triggers the checkpoint mechanism when needed. In most DBMSs, there is an additional mechanism that triggers the checkpoint mechanism before the system is forced to do so by overwriting existing log data. This optimization allows the checkpoint mechanism to avoid freezing the whole DBMS system while a checkpoint is being flushed. It is done by pushing forward the "overwrite point" which forces the checkpoint. If, for example, we have two redo-log files, this optimization flushes the data content of transactions that were written to the first redo-log file while the second one is being filled, voiding the need to flush this content when the log-switch occurs. This optimization would work only if the redo-log files are big enough for a data flush to finish before a full redo-log cycle completed.

With these two optimizations in mind, there are three areas in which we can intervene in current architectures to extract the logging component from the DBMS engine: 1. The session management level. 2. The buffer level. 3. The disk level.

Modifying the DBMS from the session management level means modifying the code that interacts with the client directly. Within this code, a database user-issued command is divided to two logical parts: logging and data manipulation. In the data manipulation code, the server process, which serves as the user proxy on the DBMS server machine, modifies the data within the target tables. These modifications are kept in memory and are assured to reach the storage only at checkpoint. The logging part at that level simply sends the command to the log component that requests the data to be logged. The control returns to the session management after the log content is in the log buffers (or on disk if it is a COMMIT command that was issued). Modifying the DBMS from this level would require modifying the high-level code that communicates with the client to handle low-level log writing issues (while in many DBMSs, the lower level data writing determines what needs to be logged). This is a poor design approach because a high-level method, which is in charge of session management, is required to implement the actual log writing. We decided not to implement SepLog at that level.

Modifying the DBMS from the buffer level means being on the other end of the server process mentioned above (within the logging component). The advantage of this level is that the component receives only log requests, and does not need to know which request is a log one and which is not.

Modifying the DBMS from the disk level means tapping into the classes or procedures that write to and read from disk files. The calls to these libraries do not differ between any type of DBMS data access: logging, data, or configurations. It is not relevant to know if it is a log record or data when writing a byte sequence to a file. Therefore, some databases merge the code base for all file accesses. We implemented SepLog over MySQL, which uses the same file modules for log writing and data writing within the InnoDB storage module.

Our implementation approach is to modify the DBMS from the log buffering level, while complementing it with isolated modifications of the disk level wherever necessary for system correctness. The reason we could not completely extract the log from the buffering level is that, within some existing systems, and MySQL is one of these, there are some storage access optimizations that cause log related commands to be strongly tied to the storage layer. For example, within MySQL, there is a protocol with which both the buffering layer and the storage layer comply to ensure the disk flush is completed. Our code intervention had to comply with this protocol for the system to function correctly. Of course, each DBMS would have different optimizations, but it is important to remember these “back channels” exist. We believe that, with our implementations, the log writing can be completely separated from the storage one, providing better implementation options.

After the data that is meant to be sent to the files is separated from the DBMS engine, we send it to the remote machine to store it. In our implementation, we have not optimized this process, and this was an intentional baseline building decision. Each command that the server process sends to the log component has left the MySQL server process by the time the server process gains control back from our MySQL extension module (see the next section for more information). This means, for example, we do not buffer commands, a process that would increase the performance in our architecture as well as it does for disk communication in current DBMSs, due to networking properties. In the evaluation section, we show that although we have not optimized the system and voided the storage optimizations that were already implemented by the DBMS engine, the performance is still better.

There are two architectural options for sending the log data to the server: 1. Using direct writing to the target server(s) from the DBMS engine. 2. Using a proxy service between the DBMS and the log server.

In the first approach, the MySQL is ”Logging Server Aware”; it has to know who the server is and how to implement its communication protocol in full. The second approach uses a proxy process that mediates between the two using IPC channels for communication between the two processes.

The first approach has better performance because the log data has less “hops” on the way to the target server. In this approach a COMMIT protocol is just a “send confirmation” protocol, which is a lightweight protocol compared to the other options that require a round-trip between the server and the client. Although it has better performance, it motivates high coupling and low cohesion between the two servers. The second approach has full separation and motivates the coding of the logging component as a library that can be used by more than one system simultaneously.

We chose to implement the proxy approach. We are fully aware that we harmed the performance with this architectural decision. By making this decision, we gained a few properties: 1. We did not have to link more libraries to the MySQL core engine; it is good because, debugging-wise, we could focus on the proxy/server code without having the DBMS engine on. Because we have not added significant complexity to the MySQL end, the probability of modifying the MySQL engine behavior decreased drastically and, evidentally, the number of bugs we had on that end was low. 2. The log component is autonomous and, in case MySQL starts while the proxy and server processes have not been initialized yet there will be an engine stall that will be released when the SepLog proxy and server will be initialized, and pull the queued requests, instead of a DBMS engine crash. 3. We show that, although this process is not efficient, and we did not implement any performance optimization (each message has to travel through IPC queue, and afterwards through a network communication channel), the performance is still better than that of a disk based log implementation. It leads to the conclusion that creating a more focused and efficient separated logging system would produce much better results than those shown here.

In our current implementation, we hold one copy of the log content on one remote SepLog server. It is clear that, while using volatile memory, we have to keep at least two copies of the data because, in case this node crashes, we have to be able to recover the data that was held in its RAM. An inevitable question is: Where should the second log copy be held? We can hold the second copy of the log on the DBMS node or on an additional remote server, making SepLog a distributed log system. We expect the overhead of both architecture to be low performance-wise due to the network properties, although we will confirm it for both in future work. In this work the second copy is held on the DBMS machine.

4.2 System durability

One should ask why is it safe to use volatile RAM of a remote machine. The reasons why non-volatile memory was used in the first place are so that the content is available after a DBMS crash and the hardware has not supported large ammounts of RAM. it seems like volatile memory is less reliable to maintain the data than non-volatile memory, such as magnetic disks and SSDs.

A key observation to ease that concern is that, for a data loss, all machines have to crash simultaneously (the DBMS engine, and the SepLog node(s)). The likelihood of all nodes crash simultaneously is much lower than the likelihood of one machine crash as a result of an independence node crash assumption, which can be ensured with the right data-center design. When one of the related machines crashes, SepLog should turn itself off orderly, or at least flush its content to disk, which ensures the content will remain on non-volatile memory and be accessible on the next system start-up, while decreasing the time frame window that the system might be sensitive to crashes in.

With this approach, the non-volatile disks of the SepLog server are used only when the SepLog server turns off and on, drastically decreasing the number of writes and reads to and from the storage devices, which leads to better durability of the storage devices. This will allow the SSD endurance in the system to drastically increase.

Looking at the data collected in [22], we deduct that, on current systems, it is more likely that a failure on current hardware will be rooted in disk than in memory. Also, the usage of disk in current systems comes in addition to the memory usage. Therefore in our design we use less components that can fail, which leads to increasing full system stability. Another thing to notice is that, by the results reported in this research, it is more likely RAM will return the value that was written to it than disk in current systems, which is a motivational boost for this work.

As discussed in the previous subsection, we do realize that, due to the total size of the redo-logs, it might be preferable not to host the second log content copy on the same machine as the DBMS server (in big DBMSs, there are usually 3 redo-log files, each is 50GB - a total of 150GB; reserving such amount of RAM is better on a dedicated machine than the DBMS system that should use its RAM for caches). Although, in an average system that uses 50MB to 10GB for each redo-log file, using the local DBMS RAM is feasible. In this paper, we have not experimented using two different SepLog servers. We expect, due to our implementation architecture and the way network interfaces work, that the performance would barely be affected by this approach (see the next section for more details). For more information on our future work, see section 7.

We noticed that holding a local cache on the DBMS server mostly increases the memory requirements, but does not significantly harm the total system performance. The reason for that is the sequential memory write and the observation that the number of "memcpy" commands (in our implementation) increases only by 1. The additional work for adding a local cache code-wise is two hash lookups (where the hash table sizes are relative to the number of log files, which is 2 or 3 in most systems) and one pointer addition. These 4 additional commands do not affect the total run time of the log process in a noticeable way (<1ms was measured in our prototype). In addition, the data is written and read sequentially, utilizing the bus architecture as a consequence of the design.

We conclude that using two in-memory log instances, one on the DBMS node (hosted within the proxy server if such exists), and one on the remote SepLog server, does not harm performance and results in a system that is at least as reliable as a system that uses non-volatile memory for the log. The latest log data is reachable to the DBMS after either of nodes crash, and the price for maintaining more than one log file is mainly paid at the system initialization and shut down (latest log file content scan is required for a successful system startup).

Another property that increases reliability is the program size. Since the log component is hosted in an external process to the DBMS, its source code is separated from the DBMS code. Our implementation for the server is less than 1000 code lines (for the SepLog server and proxy altogether). This means we can debug and code review this code easily. Although this code is affected by the incoming IPC channel that is hosted within an external code, we find the size of the application a big enabler for formal proofs of its correctness. In addition, since the code is easy to debug and test due to its size, we can gain trust that it would survive DBMSs crashes that are rooted at a DBMS bug. These sort of bugs are not likely to occur in SepLog since the only two actions that are data related within SepLog are"read" and "write"; both are internally implemented using a simple "memcpy" command. We do not expect data related sequences to affect the correctness of the SepLog engine. This claim does not hold for the DBMS system itself since it has to interpret or decode the content given to it.

We believe that the size of SepLog, that is small because of the design decisions we made regarding the interface and the way both processes communicate, is a stability enabler, which will make the system survive any DBMS non-hardware related crash. The system is also pruned to hardware-related crashes since the two nodes should lay on separated and non-hardware dependent. Note that these claims are valid for both implementation options mentioned - proxy based SepLog and direct remote memory write.

5. IMPLEMENTATION

SepLog implementation leverages the properties of the environment it is hosted in - Oakley at OSC (the Ohio Supercomputing Center); see [19].

We chose to implement SepLog with MySQL Community- [18]. MySQL was attractive to us because the source code of it is easy to browse and there is a lot of public knowledge about modifying this engine. In addition, the innoBase storage engine implements a cyclic log mechanism.

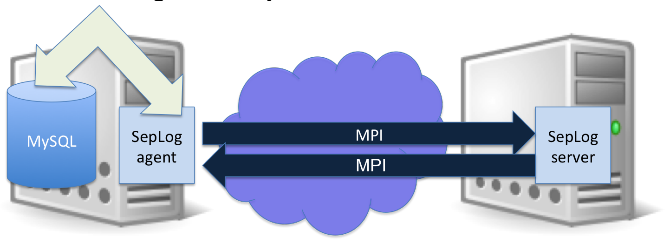
SepLog is made up of three components: 1. SepLog connector, 2. SepLog agent/proxy, and 3. SepLog server.

The SepLog connector is a modification of the MySQL I/O module, and log backup and recovery modules. These modifications redirect the log requests to the SepLog proxy process. In our current implementation, the redirection itself is done by using the boost IPC library - [21]. Using IPC, we separate the SepLog connector from the details of the actual communication protocol between the two servers.

The SepLog server and the SepLog agent/proxy are functioning together as the log component. The main difference between both is that the agent/proxy, in addition to being able to maintain the log locally, it is also sending the log commands to the remote SepLog server. In future work, we will diminish the "log maintaining" responsibility from the proxy, making its sole responsibility to send the log requests efficiently to all log servers - see section 7.

For communication, we are using MVAPICH2/2.0a - [20]. This MPI implementation leverages the OSC environment.

Figure 2: System Architecture



Our high level architecture is displayed in figure 2. It is important to emphasize the synchronicity of each call. MySQL creates a thread within its process for each client. Each thread, named "server process" as mentioned before, serves each client request. If it is a query, the server process will read the data requested, process the query, and return its results; if it is a DB modification request, the server process will execute the command in full (it will modify the data in caches, and will log the relevant parts). Since many server processes exist, there are concurrent modifications of data within the database. The logging component, though, does not allow multiple requests. Within the logging component, only one writer can exist. The physical I/O is done asynchronously while there is only one thread that can initialize a writing at a time. On COMMIT, the server process, which locks the whole logging component while the operation has not been finished, waits for all the asynchronous current I/O operations to finish flushing the writes to the device. The order that each server process gets into the log is used not only for the logging, but also determines its content visibility to other transactions and queries in the system (e.g. isolation levels). The logging component is usually the only forced database contention; while all other contentions are dependent on user requests, through the server process, the logging contention is forced by the system itself.

In our design, the SepLog connector sends requests to the SepLog agent using an IPC queue (using the Boost library [21]). The process that listens to this queue dequeue requests one by one, and sends those to the server using a synchronous MPI channel. The server dequeues these MPI requests one by one and executes those. SepLog does not currently parallelize its actions, but works sequentially. However, the log writing speed does not affect the DBMS system. The only command that requires an engine feedback is the "COMMIT" command, which will be the only DBMS logging-imposed stall. It is important to emphasize that only one COMMIT can be executed in parallel in current systems and our architecture opens the doors to multiple simultaneous commits. We will research this option in future work since it requires modifying the database recovery process and possibly the way content is being written to the log.

There are many opportunities SepLog offers for parallelizing the log writing process. Please refer to our future work section, section 7, for our future plans of parallelism for the log component.

After a full logging component communication analysis, we extracted six methods that have to be implemented for the log to operate correctly. Following is the description of each.

1. **CREATE**: The create call allocates a memory buffer for the log content by the size requested while loading the content from an existing, pre-saved file, if required, by the being used system. In MySQL, the log file is created by an external process, so in case the file we load does not exist in the SepLog non-volatile storage directory, we send the content of the file from the MySQL server. In case the "Create" started after a crash of a node, the "Create" would recognize the most current file and will load the right file to enable recovery using the highest LSN, the system timestamp, within it.

2. **CLOSE**: This call does nothing. In SepLog, while the system is "on" the DB engine cannot direct the SepLog server to close files. That said, it is a marker that the DB engine stopped using the file, and can be used for future optimization; for example, it can save the content to the non-volatile memory and release it if more RAM is needed.

3. **READ**: This call queries the log for its content. The call to read is similar to the call to fread; the requester provides the buffer to read, how many bytes to read, and the offset from the buffer beginning. This call, although being sent asynchronously, provides feedback to the client. Therefore, the SepLog server will send a message back to the SepLog agent with the reply. Reading from the log is rare, as long as there are no special features enabled, such as archival service. The reading from the log suggests a recovery process is done.

4. **WRITE**: This call queues a write request. A feedback from this call to the client is not necessary, therefore a write can be parallelized easily.

5. **FLUSH**: This call ensures the data that was written by now reached the SepLog server. Like the "Read" command, a feedback to the requester is needed. It is important to note that a "Flush" command requests all the content that was written to the log to be on non-volatile memory in the current semantic. We noticed that this semantic is too restrictive. The system correctness will not be harmed if each commit ensures only its own content was flushed. In future work, we will use this property and formally prove it.

6. **SHUTDOWN**: This call writes the in-memory content to non-volatile memory, releases the memory, and turns off the system.

Notice these commands are generic and the SepLog architecture is not affecting the need for these. In both proxy based and direct RDMA approaches, this is the protocol that has to be implemented.

6. EVALUATION

For evaluating SepLog, we explore multiple facets. First, we show SepLog operates correctly and show that the DB recovery process is not harmed. Second, we show that the performance improvement we claim holds.

Note that SepLog affects, and is affected, in addition to the recovery process only by transactional commands (DML, DDL, GRANT/REVOKE). Therefore, we have not experimented directly on queries. In the benchmarks we ran (TPCC), SQLs are run, but we do not micro-benchmark these.

In each experiment, we reserved two full OSC nodes with 12 cores. Each node has an intel Xeon x5650 processor with 48 GB of RAM and an infiniband connection. For more details regarding our environment please refer to [19].

In all experiments, when we refer to "Storage", we refer to an experiment that was conducted while the log files are located on the server node drive itself. The reason for that is if we do not use the local storage, we pass through the network, and it is clear that implementing memory copy without disk over network would have better performance. Although we experimented on the environment NFS and got the results we expected, our implementation performs better. We do not show these results here, but focus on the local storage results.

6.1 Correctness Evaluation

For the correctness evaluation, we built a few scenarios that we executed and analyzed. These scenarios force a recovery process to be initiated, showing how the process behaves, so we can determine if the operation behaves as we expect it to. In addition, we also built a scenario that is irrecoverable and examined its execution.

It is important to note that our implementation substitutes the storage layer. Therefore, we expect the byte array given to the DBMS system from our implementation to be the same as the one the original storage based implementation. Although, since the recover data is dependent on many factors, a byte array comparison is not relevant. Therefore, we experimented by crashing and recovering the system.

6.1.1 Recoverable crashes

Our recoverable crash experiments used the following scenarios:

1. All transactions were committed.

2. All transactions were rollbacked.

3. All transactions were pending (e.g. we have only open transactions). No commands are being executed while the crash occurs.

4. Some transactions were committed, while some transactions were not. No commands are executed while the crash occurs.

5. A large load is exhausting the DB using a mix of commit and rollbacks. In this scenario, commands are executed while the DB crashes.

In all mentioned experiments but the last, we know what results the DB should entail at the end of the recovery process. The last experiment has a randomness factor that prevents us from knowing what results we should expect.

All experiments ran twice - once when the SepLog server remained up between the crash and the recovery, and once when the SepLog server has been turned off after the crash, and on before the recovery began.

For all experiments, we followed the recovery process closely, and saw the DBMS engine roll the log using the innoDB log component. At the end of the recovery process in all experiments, the expected results were observed, except for the last experiment, which does not have expected results.

We ran the last experiment multiple times due to its randomness factor. The randomness is the status of the storage writing of each command when the DB process is killed (the crash simulation). For increasing the probability of storage chunks being in transfer while the DB crash, we used more writer processes than processors on our machine. All crash recovery processes succeeded.

These experiments show we have not harmed the correctness of the system and a recovery process can recover from a crash. This was expected and we mainly ran these experiments to confirm we do return the right data to our client, the MySQL server.

6.1.2 *Irrecoverable crashes*

As in a storage based system, if one loses the redo-log content a recovery cannot be completed successfully. In the scenario of losing the only in-memory log copy, we expect the system to not be able to recover and have a data corruption that will result in a need to restore from backup.

We simulated, in most of the scenarios mentioned above, a crash and log file loss. In all cases, the DBMS could not recover and reported data corruption, as expected.

To make this scenario interesting we also simulated a DB recovery from backup we have taken earlier. The recovery process succeeded.

These experiments show that the system behaves as we expect it to, and that a recovery from backup is feasible using SepLog.

6.2 Performance evaluation

Our main hypothesis is that SepLog improves performance. We have three main claims: 1. SepLog improves recovery time. 2. SepLog increases the DBMS transactional throughput. 3. SepLog decreases the DBMS transactional latency.

6.2.1 *Recovery time*

Since recovery time is dependent on many factors, some of which are out of our control, measuring recovery time would not be fair to both: traditional systems and SepLog. Therefore, we decided to rationalize our claim instead of experiment.

While recovery is executing, the system reads and writes from and to the log. The reading is done to understand what the content of the datafiles should be, while the write is done to finalize the process for incomplete transactions. In this process, the content of the relevant datafiles is read and processed if needed.

We expect SepLog to accelerate this process because of a reduction in parallel disk access. While the recovery is executed, there is a large load on disk access. Both datafiles and logfiles are being read and written to. The usage of SepLog limits the load to be only on datafiles, reducing the operating system and file system contentions on file accesses, which we expect will improve performance as shown in the next subsection.

Please note that, in the the SepLog engine is not running when the recovery process starts, the log data will be loaded, which means file access will occur to load the log content to memory before SepLog can access to it. Although there will be a file access, we still expect performance improvement since it will precede the recovery process, decreasing the parallelic load and the number of open handlers on/in the FS and OS. In addition, there will still be no file write involved, nor simultaneous access to log and data files.

6.2.2 *Transactional Performance*

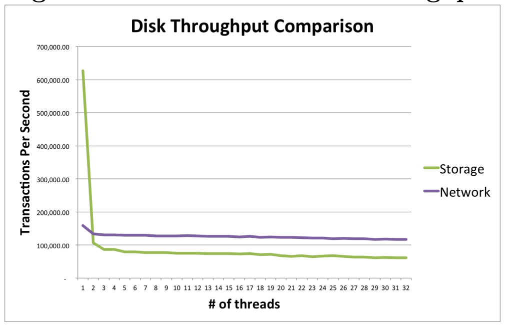
We divide this section to two - micro-benchmark and full DBMS performance evaluation.

6.2.2.1 *Micro-benchmark.*

In this section, we evaluate only the storage writing component. We evaluate mainly the writing since, although the log is written to often, it is barely being read. The only reason to read from the log is a crash recovery or a user request to do so. Therefore, the reading performance does not affect the DBMS normal operation.

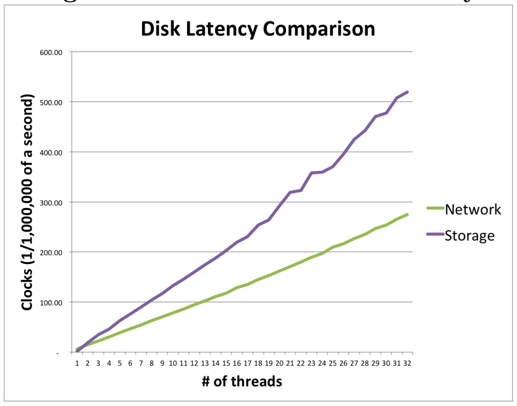
We evaluate the writing separately from the whole system evaluation for two different reasons: 1. Convince that the performance of the log component, without the DBMS engine on top of it, improves. 2. Show different log writing behaviors to prove that SepLog is robust and provides consistent performance.

Figure 3: Microbenchmark Throughput



In figure 3, we show the throughput of the log commands. In this experiment, we extracted from the DBMS engine only the log writing commands with the mutex that protects it, and we simulate multiple users which simultaneously issue

Figure 4: Microbenchmark Latency



commands. The X-axis is the number of parallel users. Each is simulated by a thread, while the Y axis is the transactions per second. This simulation runs the "commit" command in a loop on each thread. Therefore, there are two commands that are sent to the SepLog's storage engine. The first is a write of a commit byte array sequence for the current transaction and the second is a flush.

With one user thread, the throughput of the storage is higher in this experiment. This is a result of two properties: 1. Operating system and File system optimizations. 2. The overhead of both communication channels we use for each command in SepLog. When increasing the number of threads to two or more, the benefits of SepLog are substantial. In previous sections, we suggested using direct writing from the DBMS engine to SepLog. This will improve performance for this scenario drastically since it will void the "flush" command. In addition, the experiment presented has not been optimized for networking. Therefore, we expect that, in a real system, the performance will be better.

The latency, which is presented in section 4, shows the same behavior.

The behavior of both is occurring mainly because of the mutex that is present in the DBMS storage writing code. Since there are multiple simultaneous storage writers, this mutex is necessary for resource concurrency protection. This is not the case for SepLog, which can be implemented in such a way that avoids this need.

Some other DBMS systems use one thread that flushes the content to the log, which is named LGWR usually. Although the write to disk seems like it can be faster than SepLog architecture allows, the log writing is limited by a semaphore and a mutex that protect the buffer access, harming the ability of the system to take advantage of the benefits of writing from one process to disk as shown in the figures because of contentions. With the additional proof from the concept experiment we ran, the performance hit would decrease with the number of threads. This behavior is seen on both graphs for two threads or more, making SepLog an attractive option for systems with LGWR as well.

It is noticeable that the throughput decreases with the number of threads. This is because of to the mutex behavior in this situation, regardless the I/O commands.

In addition to these experiments, we ran additional experiments in which we send messages to the log in different orders to examine the robustness of SepLog. The results above show the performance of cyclic (sequential) log writing; the DBMS writing pattern to the log. We also experimented with writing in place, in a reverse order, and in a random order to the log content. In all experiments, results were consistent.

We conclude that SepLog will perform better in case of two or more threads due to network properties, and will perform better in the case of one thread due to the mutex and semaphore that have to be used for log communication in this case. In both cases, SepLog will perform better – SepLog will have higher transactional throughput and lower transactional latency.

6.2.2.2 Full DBMS.

In this subsection, we will examine the full DBMS behavior using SepLog instead of using its traditional log component. We run two different set of experiments: 1. targeted use cases that show the SepLog properties. 2. TPCC.

Targeted tests

To show that we receive the expected results we run two tests:

1. Commit test. In this test, we flood the DBMS engine with commit commands from parallel sessions. This test is designed to exhaust the flush mechanism, which, in SepLog and storage based logging, requires a roundtrip between the DBMS engine and the storage device/remote node.

2. Small transaction test. In this test, we show that, in normal DBMS behavior where flushes happens once in a while, SepLog behaves much better than in current environments. In this experiment, each thread works on its own tble in order to not overload the storage subsystem, and each transaction leaves the table empty when it is committed. The reason for that is, in this way, the checkpoint flushes one block on each commit, which is logically empty, minimizing the checkpoint ability to influence our results drastically. In each transaction, there are a few DML commands that insert and update data, while the last command deletes all the table content. On average, there are 6 commands in each transaction.

After running each experiment 3 times, we ran the same experiment for the last time, while profiling the behavior of MySQL (using cachegrind - [8]) to detect which bottlenecks exist in each logging option.

In figure 5, we show the performance of using the storage based commit (original MySQL) versus the SepLog based commit (MySQL + SepLog) when running the full MySQL stack. It is noticeable that, although not much better (betwen ∼1.1 times better, upto 2 times better), SepLog is always better, while the variance of the SepLog throughput is much lower. The reason the original MySQL base behaves like it does, according to the profiling process we conducted, is the mutexes within the log code and the checkpoint procedures interaction. The reason SepLog is consistent with its performance is that the usage of the mutex for the log writing is much shorter, leading to slightly better node resource management.

Figure 5: Commit Test Throughput

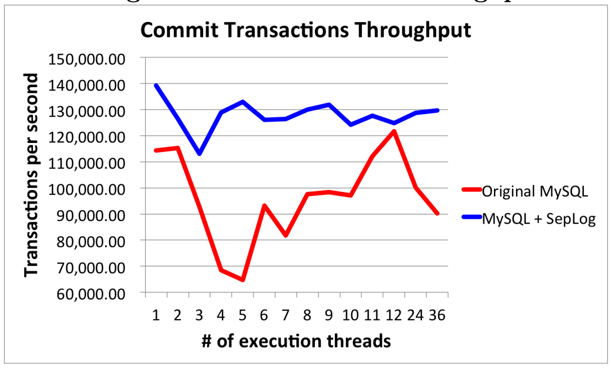
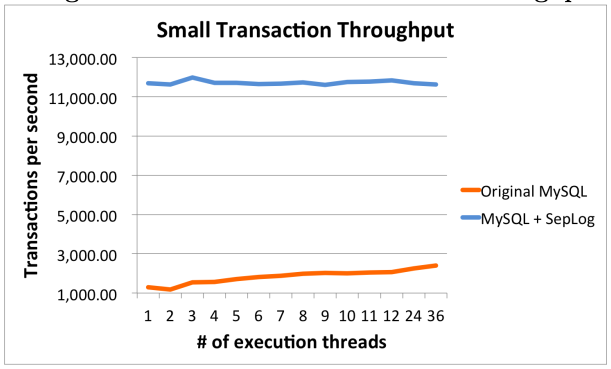
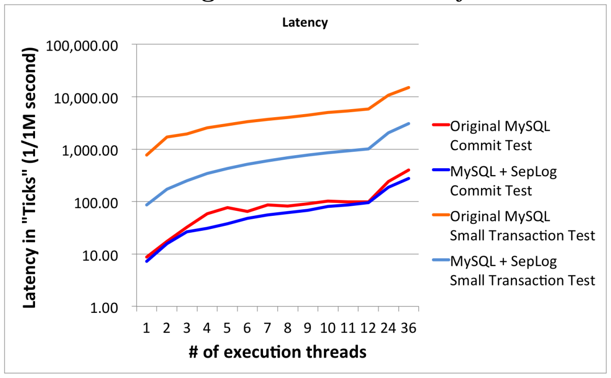


Figure 6: Small Transaction Test Throughput



In figure 6, we show the performance of an actual load that mostly exhausts the logging component. In this figure, it is noticeable that SepLog improves performance upto ∼12 times than the original MySQL. It is also noticeable that the performance is consistent with storage that increases with the number of threads suggesting that, storage-wise, there is a "sweet spot" for the number of processes that the system performs best with, and that one thread cannot utilize the full DBMS ability. This experiment shows that SepLog utilizes the resources much better. Using the CacheGrind profiling, we notice the IPC and MPI overheads are those that *delay* the system, and that both are taking consistent execution time, which we believe leads to a more reliable and consistent system as a whole.

Figure 7: Tests Latency



In figure 7, we show that in both test cases, the latency of SepLog is better. In this figure, the Y-axis has a logarithmic scale. The latency is determined by the time a command is sent to the storage/remote node in case of a non-commit command and of a communication round trip in case of a commit. It is noticeable that, in the commit test case, SepLog and the original innobase logging system can compete. This is due to our design, MPI + IPC. Using a different communication technology would affect the performance.

When commit is less common, as in real database usages when other commands are actually committed and not only the commit command itself, the latency improvement begins with an improvement of 100 times.

In conclusion, SepLog entail the expected performance improvement we expected it to. Using a more efficient communication method between the DBMS node and the SepLog server will improve performance even more, resulting in a system that could process even more transactional requests than we can today. It is important to notice that although not efficient as other techniques, the MPI and IPC performance are consistent.

TPCC

We ran the TPCC benchmark using the oltpbenchmark tool [7]. TPCC is not meant to exhaust the logging component but the DBMS as a whole. Therefore, the results are indicative for the effect our change had on the full DBMS, and, in some cases, the improvement is minimized due to other DBMS components that are exhausted.

Oltpbenchmark has two parameters that are crucial for the test: Scale and Terminals. Scale determines how big the data set that is used will be. Terminals determine how many concurrent clients will execute queries on the DB. The benchmark tool we use was designed in a way that minimized any tool stalls, allowing it to exhaust the system.

Since TPCC exhausts storage, memory, CPU, and the logging component, the performance improvement, although it exists in all tests ran, is limited by the resource that is exhausted. In other experiments, we targeted and exhausted only the log component, so we reached the logging component capacity before hitting others walls, while, on the TPCC benchmark, the other walls are hit more often, limiting the performance improvement SepLog can offer.

We ran the TPCC test with scales of two, six, and twelve, and with two and six terminals. In our environment, setting six terminals loads the machine to its full CPU capacity. Increasing the number of processes to seven or more produces inconsistent results due to operating system scheduling. Therefore we do not show, nor discuss, these results.

We report here the results of the throughput improvement between the original MySQL and MySQL+SepLog. For example, where 2 appears in Table 1, SepLog + MySQL executes twice the amount of transactions per second than the original MySQL. The average latency improvement is nearly the same as the throughput improvement for all the tests conducted. Therefore, we chose to report the median latency.

In Table 1, we show the throughput improvement between MySQL + SepLog to the unmodified "Original" MySQL.

Te / Sc	2		6	
	Imp	Th	Imp	Th
2	5.382	126.01	2.146	117.27
6	4.234	94.41	1.368	59.18
12	4.463	112.86	1.262	54.22

Table 1: TPCC throughput improvement
Te - Terminal
Sc - Scale
Th - SepLog throughput
Imp - Improvement (SepLog Th / Original MySQL Th)

Terminals / Scale	2	6
2	14.4897641	3.86586212
6	14.0033568	2.52732689
12	10.9591736	2.25611341

Table 2: TPCC median latency improvement

For example, in the table, we show that for two terminals, and a scale factor of two. Our implementations executes ∼5.4 times more TPS (Transactions Per Second) than the unmodified original MySQL. In this case, the original MySQL ran 23.41 TPS on our environment, while our implementation executed 126.01 TPS.

When executing more terminals, the contentions for both the original MySQL and SepLog + MySQL are shifting to the operating system resource control and dependencies among transaction. That motivates us to focus on a small number of terminals for analyzing the results.

For two terminals, the throughput behavior for scale change of SepLog+MySQL is interesting. For scale 2, we get the best results. This is a result of MySQL cache management that allows efficient execution and row locks when the rows we use are already in memory. For scale 6, the cache becomes bigger, and, therefore, the cache management is required to flush and bring data from disk. Three main wait events are noticeable: disk access (for data), CPU, and row locks. The most interesting behavior appears when the scale is set to 12 - the execution is more efficient than 6. This is the result of less contention on row locks. In scale 6, the same rows are used across transactions, while, for scale 12, this sharing decreases and the wait events shifts mainly to disk access and CPU. In scale 2, although the contention is bigger on row locks, the cache usage compensates for it and the results are better.

In Table 2, we show the median latency improvement. We do not show the latencies themselves since the median latencies are between 0.01s to 0.046s for the SepLog+MySQL, and, therefore, the improvement from the Original MySQL is what should be emphasized. The latency measurement includes, in addition to the DBMS engine operation time, the communication between the test process to the MySQL process. For the median latency, the less processes we have and the less data we have, the better the system performs. In addition, the improvement for the median latency is much better than the average latency, which is as the average throughput improvement. It is still unclear if we can decrease the variance of the latencies. Some of it is dependent on the user calls themselves. Therefore, it should generally be further researched.

In conclusion, the TPCC shows that, even in a very loaded scenario that utilizes all the DBMS resources, SepLog improves the DBMS performance.

7. FUTURE WORK

SepLog introduces many opportunities. Our research is targeted towards making SepLog more robust and accelerating the total system performance.

Our first, and very exciting, idea is parallelized log. We noticed writing to the log, as long as it is synchronized by the system for allocations, can be parallelized. Not only the writing itself, but also the flush command can be modified to flush only the current transaction content instead of all the previous transactions. This is valid thanks to the DB isolation level that requires only committed data to be seen by later transactions and forces ordering amongst committed transactions. This offers a fundamental change to the logging component with a big significance and effect: two log files can be active simultaneously, which poses a challenge to the checkpoint process. We plan to tackle that issue in our next work.

Another exciting idea is building a distributed logging system based on non-volatile memory using volatile memory. This goes hand-in-hand with another exciting idea: centralized log system for multiple databases. The first can be implemented in many forms, starting with the same log being sent to multiple nodes upto dividing the log into chunks that are hosted on different in-memory log nodes (similar to work that has been mentioned in section 2). The second allows the combining of different DB logs on the same log server logs, which allows the increasing of the system stability by concentrating on a small limited number of SepLog servers, instead of the number of DBMS servers. This opens the door to many other interesting opportunities that arise from having multiple DB logs in one place, like detecting a cross system anomaly using log analysis.

In addition, following the ideas presented here, we can modify the cyclic log concept to be based on RAM instead of on files. One can allocate a node with a pool of RAM, instead of pre-allocating files, which the DB writes to directly without referencing files and synchronizing its access. This modification reduces the DBMS engine responsibility and removes most of the latches within the log connector, transferring this responsibility completely to the log component.

Log archivals are affected by the new SepLog approach. One can implement archivals directly on the remote SepLog server, although it will require fast storage and would affect the checkpoint mechanism. An approach to decrease this contention is to distribute the archival writing. It is clear that archivals should not be held in RAM. Therefore, if archivals are needed for backup, old fashioned disks are a necessity. These do not void the benefits of SepLog. We will explore the benefits of using SepLog with archivals in the future.

The last idea we will mention here is the ability to improve the checkpoint process by borrowing the same concept - instead of waiting for the disk flush to end, hold the non-flushed blocks on a remote server until the disk flush ends.

8. SUMMARY

In this paper, we presented SepLog, an in-memory DBMS logging system that holds the logging data on a remote machine instead of on non-volatile local storage in current systems. We explained how current logging system works in details, and why we believe SepLog is a good concept to use: the usage of RAM allows SepLog to offer increased transactional performance (increased transactional trhoughput and lower transactional latency). We compared performance to prove that SepLog performs as expected, which uncovered additional DBMS bottlenecks in the DBMS architecture.

SepLog does not harm the reliability or the durability of the system, e.g. crash recovery processes will end successfully, and we believe it is as safe to use as current systems that are based on hard drives for the logging data.

SepLog opens the door for many logging optimizations, such as parallel log writing and distributed logging, which will improve the logging system performance even more, and might change the way logging is managed in the future.

9. ACKNOWLEDGMENTS

This work was supported in part by an allocation from the Ohio Supercomputer Center.

This work uses the MVAPICH [20] for using MPI over infiniband within the Ohio Supercomputer environment.

10. REFERENCES

[1] M. Balakrishnan, D. Malkhi, J. D. Davis, V. Prabhakaran, M. Wei, and T. Wobber. Corfu: A distributed shared log. *ACM Transactions on Computer Systems (TOCS)*, 31(4):10, 2013.

[2] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 325–340. ACM, 2013.

[3] P. A. Bernstein, C. W. Reid, and S. Das. Hyder-a transactional record manager for shared flash. In *CIDR*, volume 11, pages 9–20, 2011.

[4] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. 2010.

[5] T. Cao, M. Vaz Salles, B. Sowell, Y. Yue, A. Demers, J. Gehrke, and W. White. Fast checkpoint recovery algorithms for frequently consistent applications. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 265–276. ACM, 2011.

[6] S. Chen. Flashlogging: exploiting flash devices for synchronous logging performance. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 73–86. ACM, 2009.

[7] C. Curino. oltpbenchmark. `http://oltpbenchmark.com`, 2014.

[8] V. Developers. Valgrind - cachegrind. `http://valgrind.org/docs/manual/cg-manual.html`, 2000.

[9] C. Dirik and B. Jacob. The performance of pc solid-state disks (ssds) as a function of bandwidth, concurrency, device architecture, and system organization. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 279–289. ACM, 2009.

[10] A. Foong, B. Veal, and F. Hady. Towards ssd-ready enterprise platforms. *Google/Intel Corporation*, 2009.

[11] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.

[12] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: a scalable approach to logging. *Proceedings of the VLDB Endowment*, 3(1-2):681–692, 2010.

[13] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Scalability of write-ahead logging on multicore and multisocket hardware. *The VLDB Journal*, 21(2):239–263, 2012.

[14] S.-W. Lee, B. Moon, and C. Park. Advances in flash memory ssd technology for enterprise database applications. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 863–870. ACM, 2009.

[15] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A case for flash memory ssd in enterprise database applications. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1075–1086. ACM, 2008.

[16] S. Liang, R. Noronha, and D. K. Panda. Swapping to remote memory over infiniband: An approach using a high performance network block device. In *Cluster Computing, 2005. IEEE International*, pages 1–10. IEEE, 2005.

[17] V. Mohan, T. Siddiqua, S. Gurumurthi, and M. R. Stan. How i learned to stop worrying and love flash endurance. In *Proceedings of the 2nd USENIX conference on Hot topics in storage and file systems*, pages 3–3. USENIX Association, 2010.

[18] Oracle. Mysql community - oracle. `http://www.mysql.com/products/community/`, 2014.

[19] OSC. Ohio supercomputing center: Oakley cluster. `https://www.osc.edu/supercomputing/computing/oakley`, 2014.

[20] D. K. Panda. Mvapich: Mpi over infiniband, 10gige/iwarp and roce. `http://http://mvapich.cse.ohio-state.edu`, 2014.

[21] D. A. Rene Rivera, Beman Dawes. Boost. `http://www.boost.org`, 1998.

[22] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an mttf of 1, 000, 000 hours mean to you? In *FAST*, volume 7, pages 1–16, 2007.