Scalable Logging Algorithm for in-Memory DBMS

May 15, 2016

Abstract

git

1 Introduction

As the world becomes more technology-oriented, the amount of data that is being stored and accessed has multiplied. From millions of users on Facebook to police records, the amount and complexity of data in virtual storage has incremented drastically. This growth has led to the study of **big data**, a term used for data sets that are too large and complex for traditional data processing methods. These large data sets are being analyzed in database management systems (DBMS). DBMS can be categorized into Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP). OLTP databases are the main target of this paper and are characterized by a large number of short online transactions, such as bank transactions, flight ticket reservations, or hotel room bookings. OLAP, on the other hand, involves longer and more complex transactions such as business intelligence and data mining.

OLTP databases contain large amounts of important information. In order to preserve the information stored in the database in case of crashes, it is necessary to find a way to ensure the **durability** of the system. For this to happen, all of the data must be put on non-volatile storage, or logged onto the disk. The most efficient way to put these large amounts of data onto the disk is still debated because there are many different ways to accomplish the task.

One of the first and most well-known algorithms designed for the logging process was the Algorithm for Recovery and Isolation Exploiting Semantics (AIRES). In this algorithm, More details on the serial logging algorithm can be found in section 2. With the introduction of new technologies, however, multi-core and multi-server system have become more prevalent. These new systems render the AIRES algorithm obsolete as we are no longer limited to using only one thread in the logging process, which can be a bottleneck as the number of transactions handled by the logger increases dramatically. This means that serial logging will no longer be useful in the future and new algorithms will be neccessary.

The goal of this paper is to solve this problem and generate a new logging algorithm that is faster and more efficient on a system with a multi-cores and multi-servers. The two algorithms that are presented in this paper are batch logging and parallel logging. In batch logging BRIEFLY EXPLAIN BATCH LOGGING. In parallel logging, we track the dependency between different transactions, which allows us to decide whether to log independently or serially; multiple threads are employed simultaneously with their respective loggers and log files.

The serial logging, batch logging, and parallel logging algorithms were implemented and then tested to determine the effiecieny of each.

2 Serial Logging

The state-of-the-art algorithm for serial logging is ARIES. In ARIES, each log record consists of one modified data tuple and the name of the transaction that modifies the tuple. It is assigned a log sequence number in ascending order. The log records are first pushed to volatile storage and then flushed to nonvolatile storage after the transaction is committed. Two data structures are maintained: the Dirty Page Table and the Transaction Table. The DPT keeps track of all the changes made to the database that have not been flushed to the disk, and the transaction table records all the transactions that are currently running in the system. During the recovery process, the system first recovers and updates the dirty page table and the transaction table, then recovers the system to the state immediately before the crash, and finally undoes all the transactions that have not been committed.

In our implementation of ARIES, we simplified the algorithm such that each transaction is logged together and corresponds to one unique LSN. Each transaction goes to log only after it is committed. This makes the dirty page table unnecessary, and we do not need to undo during the recovery process because only the committed transactions are reflected in stable storage.

3 Batch Logging

In batch logging, multiple loggers are in operation at the same time as opposed to in serial logging, where there is only one logger. Flushing the log buffers to disk in batch logging uses the naive scheme of flushing all log buffers once one is full. A bit vector is used to communicate between the loggers, where the index of the bit corresponds to the number of the logger and the logger flushes if its bit is 1. When a logger fills up its buffer, it changes all bits of the vector to be 1 to signal all other loggers to flush their buffers too. Each logger checks the bit vector upon receiving a transaction, and if their respective bit is 1, then the logger flushes and toggles its bit to 0 before placing the newly received transaction into its buffer.

[TODO: Talk About Performance Numbers]

4 Parallel Logging

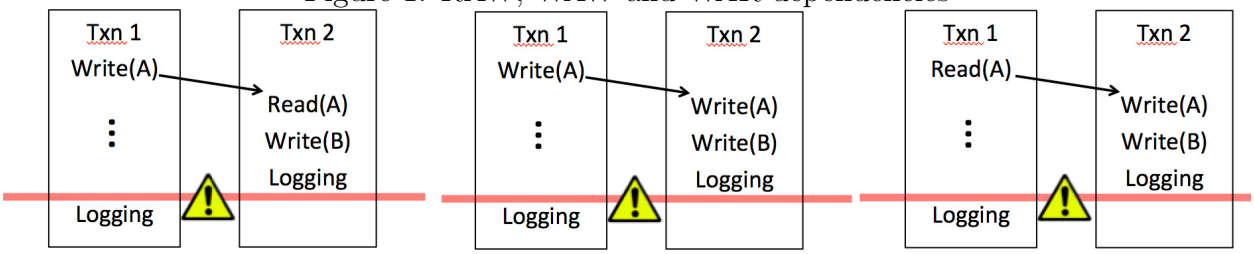
Parallel logging uses multiple loggers and corresponding log files in order to improve the performance of the logging process. While serial logging assumes conflicts between every pair of transactions in the serializable order, parallel logging looks at the "true" dependency. While pairs of transactions with true dependency have to log seqentially, independent transactions should log in parallel. The parallel logging algorithm implements this key observation by explicitly tracking the dependency information between different transactions. In addition, the concurrency control algorithm is implemented to ensure the serializability of the transactions that we have parsed into different threads.

There are four types of dependencies between pairs of transactions.

- Read-after-read dependency (RAR)
- Read-after-write dependency (RAW)
- Write-after-read dependency (WAR)
- Write-after-write dependency (WAW)

RAR is equivalent to no dependency since no changes are made to the database. We discuss the rest of the cases using the following example, where cases of RAW, WAW and WAR are presented repectively in that order:

Figure 1: RAW, WAW and WAR dependencies



In RAW dependency, the reading transaction should be logged after the writing transaction; if not, we may look at the first example. During recovery, the tuple A read by transaction 2 is not consistent with the value of A on the disk. Therefore, transaction 2 is logged after transaction 1.

Similarly, WAW has to be maintained; if not, we may look at the second example. During recovery, transaction 2 overwrites a phantom value of tuple A that cannot be tracked down in the system. Therefore, transaction 2 is logged after transaction 1.

We propose, however, that ignoring the WAR dependency, also known as **anti-dependency**, will not affect the results of our task. This is demonstrated in the third example. In this case, there will be no conflicts in data consistency during recovery. All the values in the data tuples reflect the exact state of the system immediately before the crash. Transaction 1 does not have any side effect on tuple A in the database system, so whether it logs or not does not affect transaction 2.

Note that this will significantly decrease the time complexity of the logging algorithm because the number of writers to a data tuple is significantly smaller than the number of readers. In other words, transaction A depends on the all the transactions before A in the serializable order that write to A's data tuples; therefore, A can be logged once all the last writers to its data tuples have been logged.

To implement our algorithm, we specify the maximum log sequence number (LSN) in each log file that the transaction currently being logged depends on. When we attempt to log a transaction to the file, we compare the LSN that the transaction depends on (A) to the maximum LSN that has already been logged in each file (B). If $A \leq B$, the transaction is ready for logging; if $A > B$, we will push the transaction into a waiting queue after arbitrarily assigning it a LSN. The waiting queue will be checked periodically later on to see if any of its contents becomes ready for logging.

We use the following example to demonstrate the mechanism of our parallel logging algorithm:

Figure 2: Example for Parallel Logging Algorithm and Recovery

Serializable Order

Txns:	Operations:
1	Read A; Read B; Write C;
2	Read A; Write B;
3	Write A; Write B; Read C;

Recovery

LSN	Txn ID	Data Tuple	Dependency info
Log File 1			
1	1	C	[0, 0]
2	3	A	[1,1]
		B	
Log File 2			
1	2	B	[0,0]

If we use the serial logging algorithm, we are forced to log the three transactions one by one; when handling big data systems, this can be a serious bottleneck. Meanwhile, the parallel logging algorithm allows us to log multiple transactions. In this case, note that transactions 1 and 2 can be logged independently since tuple A has no dependency and tuple B is WAR, which, per our discussion above, allows for independent logging. Therefore, if transactions 1 and 2 are parsed into two different threads, they can both be logged without waiting. Transaction 3 depends on transaction 1 due to tuple C (RAW) and on transaction 2 due to tuple B (WAW), so it will be put into the waiting queue until both transaction 1 and 2 have already been logged. We follow a similar process during recovery: we read the maximum LSN in each file that the current transaction depends on, and we compare it with the maximum LSN that has been recovered from that file. This ensures the serializability of the log records while improving the efficiency of the recovery process. In this particular example, two different threads take care of the two log files. While transactions 1 and 2 can be recovered immediately, transaction 3 is pushed into the waiting queue until both 1 and 2 have already been recovered.

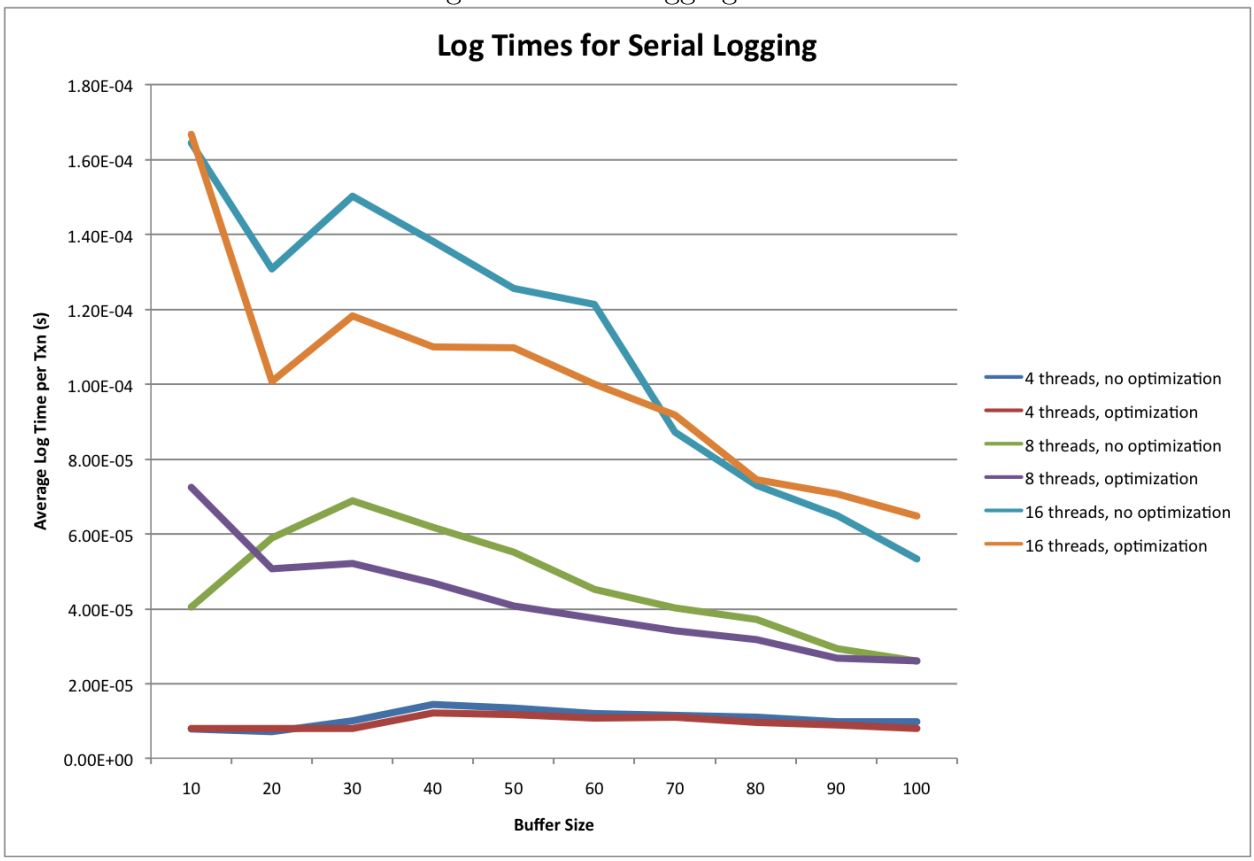
5 Evaluation

The original goal of the project is to create a scalable logging algorithm that does not act as a bottleneck for the database as a whole. The data shown below was collected to determine the scalability of the algorithm.

Four algorithms were tested to generate these results. The first is a generic serial logging algorithm, the second is a serial logging algorithm with certain optimizations put in place, the third is the batch logging algorithm, and the last is a parallel logging algorithm. These different aAll of these are explained in further detail earlier in this paper.

The results shown indicate the amount of time one txn would take to log. The goal is to minimize the amount the time increases as the thread count increases, indicating high scalability and preventing logging from becoming a bottleneck. Each of the four algorithms was tested on a maximum of 10000 txns and buffer sizes ranging from 10 to 100. 10 tests were run for each buffer size, and the average of these 10 test are shown below.

Figure 3: Serial logging results



6 Conclusion