Adding recovery to the Bohm database

John Davidson Hatch
Advisor: Daniel Abadi
15 December 2016
CPSC 490 – Special Projects
1 Introduction

Bohm is a MVCC database described by Jose Faleiro and Daniel Abadi [1]. It is designed to operate on an in-memory dataset on machines with large core counts. Key to the Bohm design is the deterministic concurrency control system. In traditional non-deterministic concurrency control systems, an equivalent serializable order is not known until the transaction has finished executing, and is dependent on the execution order of threads throughout the system. In Bohm, a serial ordering is chosen before transactions enter the concurrency control phase. Bohm is a highly modular system, with distinct responsibilities for different sets of threads. This work focuses on adding durability to Bohm in order to explore how it may interact with a deterministic system. In Section 2 an overview of Bohm’s design is presented. In Section 3 and 4, the design and implementation details of the logging layer are described. Section 5 demonstrates the low performance impact of the logging layer on the operation of the rest of the system, and Section 6 presents opportunities for future work.

2 Background and Motivation

A common performance bottleneck encountered when running conventional databases on systems high core count systems is cache contention. Typical concurrency control mechanisms rely heavily on inter-thread communication in order to agree on a serializable order. These mechanisms are usually record locks, or in the case of optimistic concurrency control and MVCC a shared counters utilized for timestamp assignment. As various cores update and read from these structures, the cached records are invalidating, forcing frequent reads to main memory which are orders of magnitude slower than access to cache.

A deterministic concurrency control protocol eliminates the need for expensive communication between threads by assigning an equivalent serializable execution order for transactions before they enter the execution system. This contrasts with typical concurrency control methodologies which do not assign a serializable ordering (either implicitly or explicitly, as is the case in MVCC) until the transaction has committed or aborted. Once this execution order is agreed upon, and distributed to all nodes, the transactions can be executed in parallel. Two tradeoffs are critical in accomplishing this approach: 1) a COMMIT or ABORT decision for each transaction must be determined before it is assigned a position in the execution ordering and 2) transactions are typically executed in batches, in order to reduce the
overhead of distributing the execution order. The first tradeoff implies that a transaction that non-deterministically aborts, either as a result of a constraint failure or embedded business logic, must do so prior to being assigned a position in the final serial ordering. In order to evaluate dependences and conflicts between transactions, the entire read and write set must generally be known up front. The second tradeoff, implies a compromise between latency and throughput that is present in most systems. In a deterministic concurrency control protocol, we are trading increased latency for overall transaction-level throughput increases.

Bohm is one such database employing a deterministic concurrency control protocol. In order to accomplish a reduction in inter-thread communication, it is highly modular, with a clear separation between concurrency control and execution layers. This system parallelizes concurrency control work across multiple threads, each responsible for preparing a disjoint set of database records for the execution of a particular transaction. A transaction in Bohm is first assigned a commit timestamp by a preprocessing thread, then handed off to multiple concurrency control threads that prepare the multi-version environment for the execution of the transaction’s reads and writes, and finally executed by a one of many execution threads. The concurrency control layer takes advantage of inter-transaction parallelism by processing disjoint record sets at the same time. The system tracks write-write and read-write dependencies through multi-version structures established during the concurrency control process. Notably, there very few shared memory structures between threads. This avoids the cache coherence issue. Furthermore, Bohm execution threads are designed to take advantage of cache locality by executing a series of transactions operating on the same data on the same core.

Although Bohm is an in-memory database, it is still desirable to provide a durability component. Most database systems, even if they operate on a fully in-memory dataset have a durability mechanism so that they may be used other than for ephemeral data processing.

3 Logging Design

In order to maintain the design philosophies of the rest of the system, the logging system follows a few main design tenants:

1. Modular: Maintain a clear division between logging, concurrency control, and execution layers. The interface should be clearly defined, and be consistent with the existing shared single-producer, single-consumer queues used to interconnect existing layers.
2. **Performant**: The logging layer must not have a significant performance impact on the throughput of the system. In keeping with the design of Bohm, it should trade latency for throughput increases. In addition, it must take advantage of the batches used to pass transactions between threads.

3. **Cache-compatible**: It must not rely on shared data structures.

   The logging layer is implemented as a single thread that sits on top of the current layers: preprocessing, concurrency control, and execution. It receives a batch, retains a reference to it, and immediately passes it on to the concurrency control layer so that processing can begin. Before taking another batch off of its input queue, the logging layer serializes this batch. This prevents the logging thread from falling more than a single batch behind the concurrency control layer.

   The logging thread need only record REDO records. Once a transaction reaches this layer it is guaranteed not to abort, so no UNDO is necessary.

### 3.1.1 Log File Format

The transaction format dictates how actions are written to disk. In Bohm, transactions are expressed as C++ code that is linked to the database at build-time. This allows for great flexibility in transaction execution behavior, but does not allow for materialization of the effects of a transaction until it has been executed. Since a transaction is executed after the logging layer, we cannot perform value logging (i.e. logging the keys and value of each write that occurs during transaction execution).

   For the logging format, three options were considered:

   1. **Log before batches are executed**. Since writes have not yet been materialized, perform *logical* logging by writing any values needed to reproduce a deterministic in-memory representation of the transaction at recovery time.

   2. **Perform logging after execution**, once the writes have occurred in-memory. When writing to disks we can just perform *value* logging and notate the key and value that was written during each transaction. During restore we simply need to replay these physical writes.

   3. **Abstract the specification of transaction write operations** so that it is possible to determine the behavior of a transaction without fully executing it. This is tantamount to implementing a query language and would greatly expand the scope of the project, as well as requiring significant changes to the execution layer.
In Bohm, logging makes sense as the first layer of the system so as to reduce its contribution to the overall latency by logging asynchronously. As a result, logical logging was necessary. The binary format for each transaction written to the log is as follows:

```
[transaction_type: uint32] [length: uint64] [payload]
```

Where transaction_type indicates which transaction type the log entry represents and length indicates the number of bytes in the variable-length transaction-specific logged data. This flexible format allows for transaction authors to utilize whatever binary format is appropriate for serializing and deserializing their transactions.

### 3.1.2 Cache Impact

It is important to consider the potential cache impacts of the logging layer. After passing on a batch to the rest of the system, the logging layer will continue to perform **reads-only** on the following memory regions in order to write to disk:

1. Transaction type and serialization data.
2. mv_action transaction pointers.

These layers are not written to by any later layers. Furthermore, on 64-bit x86 systems they do not share cache lines with data required by later layers of the system.

As such, the logging layer is unlikely to cause any negative impacts on cache performance by later layers of the system. The logging layer does not share data through any other data structures with the rest of the system.

### 3.1.3 Recovery

On startup, if the database recovery mode is enabled, the logging system will read through the log file and emit batches containing the transactions that have been deserialized. These are passed along to the rest of the system as usual. After having processed the log, the logging layer begins accepting batches from its input queue.

### 4 Implementation

The use of logical logging in Bohm makes the quality of the implementation critically important. There must be tools that make implementing serialization and deserialization straightforward for transaction authors since it must be performed for every transaction type in the system that will be logged. The complexity of buffer and memory management should be abstracted and managed in a manner that makes
it difficult for transaction author mistakes which could corrupt the log to occur. Furthermore for ease of debugging, we want to ensure that if these errors do occur, they are isolated to the faulty transaction. We must additionally ensure that the write path is optimized to increase log throughput. The recovery path is not as critical initially as this only occurs once at startup.

4.1 Buffer Implementation

Buffers are implemented as interface types which allow for the underlying buffer behavior presented to different parts of the database to vary.

4.1.1 Write Path

On the write path, our buffer abstraction allows for operations that are very convenient for transaction authors, and ensure correctness. The abstract buffer interface is as follows:

```cpp
class IBuffer {
    /**
     * Write 'data' to the buffer.
     * Returns 'true' on success, 'false' if the data could not be written.
     */
    template <typename T, typename = std::enable_if<std::is_arithmetic<T>::value>>
    bool write(const T& data);

    /**
     * Write 'len' bytes from 'data' into the buffer.
     * Returns the number of bytes written. May be less than requested if the reserved space is exceeded.
     */
    std::size_t write(const unsigned char* data, std::size_t len);

    /**
     * The number of bytes remaining in this buffer to write into.
     * If the buffer is unbounded, returns std::numeric_limits<
type>::max
     */
    virtual std::size_t remaining();
};
```
The templated write function is usable for arithmetic types, which is generally sufficient for the types of transactions currently implemented with Bohm. This function is implemented at the interface level, so that IBuffer implementors need only worry about writing byte arrays to the underlying storage backing the buffers. On the write path, three types of buffers are used throughout the database implementation. Buffer provides a buffer backed by memory-mapped pages which knows how to write itself out to a file. A BufferReservation permits advancing the write pointer and obtaining a reference to the skipped space in an underlying Buffer for later writing through the same interface. Finally, CountedBuffer is a proxy object which will count the number of bytes written since it wrapped the underlying buffer. The latter two are utilized to write the length value prior to any transaction payload. The use of CountedBuffer allows the transaction serialization process to ensure that the length of a transaction payload that is written to the log is accurate. A simpler alternative would be to trust transaction implementors to return to the logging layer the number of bytes written, but this opens a possibility for bugs that cause corruption outside of a transaction’s own payload.

4.1.2 Read Path

The read path is designed to offer similar convenience and protection. We start with an abstract read buffer:

class IReadBuffer {
    DISALLOW_COPY(IReadBuffer);

public:
    IReadBuffer() = default;
    virtual ~IReadBuffer() = default;

    /**
     * Read out a primitive type.
     *
     * Return true on success, false otherwise.
     */
    template <typename T,
               typename = std::enable_if<std::is_arithmetic<T>::value>>
    bool read(T *out);
/**
 * Read 'nBytes' from the buffer.
 * Returns: the number of bytes actually read, or 0 if none remain.
 */
std::size_t read(unsigned char *out, std::size_t nBytes);

Again, templated read of arithmetic types is sufficient for most transactions. In order to achieve the guideline above—that errors in a transaction must manifest in that transaction’s deserialization—we implement ReadViewBuffer which provides a capped view into an underlying buffer. This allows us to enforce the requirement that transaction deserialization functions must read their entire payload completely, and prevent them from reading more.

4.2 Logical Transaction Serialization / Deserialization Implementation

To implement logging for transactions that perform writes, implementors must write three routines:

void txn::serialize(IBuffer *buffer);
TxnType txn::type() const;
static txn* txn::deserialize(IReadBuffer *readBuffer);

In serialize, it is necessary to write all information to restore a in-memory representation of the transaction that performs exactly the same function. For transactions that rely on randomness, this should either include the seeds for the pseudorandom generator, or the results of the generator when the transaction was first created.

4.3 Write Performance Optimizations

To optimize write speed, we use writev which allows for a Buffer to write all its memory regions to disk in one system call. Furthermore, we open a file in data sync mode (using the O_DSYNC flag) to ensure that the data is fully durable after it is written, without the need for an additional system call to sync it. The performance of different write methods is evaluated in section 5.
4.4 Asynchronous I/O

In order to further decouple I/O operations from the progress of the remainder of the transaction processing layers, we add an asynchronous logging mode (enabled with the --log_async startup flag). This mode uses the POSIX aio_* calls. In the synchronous mode, the logging layer is at most one batch behind the preprocessing layer. In asynchronous mode, batches are received, serialized, and write requests are enqueued without waiting for the disk to finish persisting them. In async mode, the logging thread performs the following during its run loop:

1. If we have less than MAX_BEHIND outstanding batches for write, accept another batch from the input queue, retain a reference, and immediately pass it to the output queue.
2. Serialize the batch if we accepted one.
3. Asynchronously enqueue I/O operations necessary to write the batch. Enqueue a dsync call after each batch’s set of write operations.
4. Check the status of any in-progress batch writes. If a batch’s write has been completed, clean up the buffers storing its serialized state.

This design likely may have the potential to process workloads well where there is greater variance in the duration of time it takes to write each batch. If some batches take a very long time to serialize and write to disk relative to others, the system can still continue accepting batches while they are being written to disk. Unfortunately there is not an aio_writev implementation, and checking the result of in-flight operations results in additional system calls over the one system call required to write a Buffer in synchronous mode.

5 Performance Evaluation

The performance of the logging system was evaluated on an insert-only workload and the YCSB workload. The workloads ran on a machine with Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz processor and 64GB of RAM, as well as on a Google Cloud VM instance. The following parameters were used when running the experiments:

<table>
<thead>
<tr>
<th>CC Thread Count</th>
<th>Preprocessor Thread Count</th>
<th>Executor Thread Count</th>
<th>Num Transactions</th>
<th>Epoch Size</th>
<th>Num records</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>100000</td>
<td>1000</td>
<td>100000</td>
</tr>
</tbody>
</table>
The following are the performance results on an insert-only workload, averaged over multiple runs.

<table>
<thead>
<tr>
<th></th>
<th>No logging</th>
<th>Logging – Synchronous Mode</th>
<th>Logging – Asynchronous Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6.98 ms</td>
<td>30.80 ms</td>
<td>596.55 ms</td>
</tr>
</tbody>
</table>

Surprisingly, the asynchronous mode is significantly slower. This is likely due to the increased number of system calls needed to implement it. If the Linux kernel had an `aio_writev` function, the grounds for comparison would be more equivalent.

Synchronous mode logging is much slower as well. The test system utilized a network filesystem, so it is potentially the case that a local disk would perform far better.

The results for the YCSB workload are similar, but with a lesser penalty for asynchronous logging. The workload produces about twice the amount of logged data measured in bytes. This suggests that as workload size increase may bring a convergence between synchronous and asynchronous modes as the time is dominated by the actual I/O cost compared to the baseline cost of doing a system call.

<table>
<thead>
<tr>
<th></th>
<th>No logging</th>
<th>Logging – Synchronous Mode</th>
<th>Logging – Asynchronous Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>165.045 ms</td>
<td>1138.39 ms</td>
<td>3191.66 ms</td>
</tr>
</tbody>
</table>

To test the impact of the network file system, we perform the same tests on a Google Cloud instance with 16 vCPUs, 60 GB of RAM, and a 20 GB persistent disk supporting 30 random write IOPS and 2.4 MB/s write throughput. For the insert only workload, the results were as follows:

<table>
<thead>
<tr>
<th></th>
<th>No logging</th>
<th>Logging – Synchronous Mode</th>
<th>Logging – Asynchronous Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11.36</td>
<td>38.60</td>
<td>59.27</td>
</tr>
</tbody>
</table>

Here we see a slightly smaller performance penalty for logging, and a much lesser penalty for asynchronous mode. This suggests that perhaps the network file system
handled asynchronous writes particularly poorly. The following are the YCSB results:

<table>
<thead>
<tr>
<th></th>
<th>Logging – Synchronous Mode</th>
<th>Logging – Asynchronous Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>No logging</td>
<td>255.10</td>
<td>468.33</td>
</tr>
<tr>
<td>Logging</td>
<td>305.82</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100%</td>
<td>120%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>184%</td>
</tr>
</tbody>
</table>

On a YCSB workload running on the Google Cloud VM, the cost of synchronous logging is only 20%. This represents a more realistic workload compared to the insert only workload, which would represent the worst-case scenario in terms of logging bottlenecking the system.

6 Future Work

6.1 Checkpoints

Since recovery time from a log can be unbounded, we would like to implement checkpointing. The Bohm storage layer currently does not maintain indexes, which will be necessary to perform checkpointing. The following proposed design presents a method for incorporating checkpointing throughout layers of Bohm:

1. **Concurrency Control Layer:** In addition to updating the hash table used for canonical storage of MVCC keys and their value versions, the concurrency control layer is responsible for updating a *version index* with each new write. The version index contains a mapping from key to the most recent version of a record written prior to the *checkpoint end epoch*. When the concurrency control layer finishes the *checkpoint end epoch*, it begins a new version index for the next checkpoint. The prior one is utilized by checkpoint threads and then the memory can be garbage collected.

2. **Execution Layer:** The execution layer can trigger checkpoint threads once the *checkpoint end epoch* has finished executing, which write out portions of the version index produced by the concurrency control layer.

3. **Checkpoint Threads:** Each checkpoint thread writes out a partition of the version index.

4. **Logging Layer:** The logging layer will need to be modified to separate log files by checkpoints or truncate logs via some other mechanism.
6.2 Client Durability and Transaction Acknowledgement

Since there is no client library, transaction durability is not acknowledged to clients. The commit point for non-aborting transactions is when the log has been dsync’ed to disk after a batch completes.

6.3 Multiple Logging Threads

If logging is a bottleneck, the system can write multiple log files across separate threads, potentially to different disks. This will increase throughput at the expense of restore complexity. If the epoch and batch number is stored in the log format, it is possible to write batches out to separate log files in a round-robin fashion. Taken together they form the original serial order.

7 Acknowledgements

Thank you to Jose Faleiro for assistance in understanding the Bohm codebase and working through implementation details, and my advisor Professor Daniel Abadi.

8 References