Professor Daniel Abadi’s database lab has proposed multiple novel database systems employing deterministic concurrency control protocol systems, such as Calvin. In most concurrency control protocols, such as two-phase locking, optimistic concurrency control, and a traditional implementations of multi-version concurrency control (MVCC), the serial-equivalent order in which transactions are committed is non-deterministic, being influenced by random factors, such as OS scheduling. To mitigate this non-determinism, databases must employ communication mechanisms to coordinate transaction execution between threads, and in a distributed setting, between machines. In the case of multi-threaded databases running on machines with a high core count, communication between threads accomplished using shared memory structures can be prohibitively expensive due to cache coherence costs and the introduction of a single synchronization point. In a distributed database setting, this communication must occur between nodes over the network.

A deterministic concurrency control protocol eliminates the need for expensive communication between execution threads (either between nodes or on the same machine) 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.

Deterministic concurrency control systems have a distinct architectural advantage: the separation of the concurrency control layer from the execution layer. Typically these layers are heavily coupled, with concurrency control and database state influencing each other throughout the execution process,
leading to non-deterministic transaction aborts or retries. In a deterministic system, the concurrency control layer has been completely separated. Jose Falerio and Daniel Abadi describe a deterministic database, known as Bohm, employing a novel deterministic concurrency control algorithm to provide serializable transactions in a multi-version, high-core count environment. 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 single 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.

For my thesis, I will be collaborating with Jose Falerio and Daniel Abadi in improving Bohm’s implementation, to bring it from a research prototype, to a releasable open source project. In addition to adding to Bohm’s documentation, and interface, this includes some specific additions and optimizations that must be addressed:

1. Adding a logging mechanism for failure recovery. While Bohm is a main-memory database system, we would like to achieve durability through logging and checkpointing.

2. Parallelizing preprocessing. Currently only concurrency control and execution threads in Bohm are parallel. In addition to parallelizing preprocessing, we can decrease work for the concurrency control threads by only sending the relevant write-sets to each concurrency control thread. Currently the system sends the entire write-set, forcing each concurrency control thread to evaluate each record, even if it is not responsible for acting on that record.

3. Eliminating the requirement of knowledge of transactions' write-sets prior to execution. This can be accomplished via a form of speculative execution to determine the relevant write-set for a transaction.

4. Evaluating thread allocation between components. The system has three components, as described above. Each will be multithreaded, so we must be able to optimally assign threads between different components.

Myself, Sylvan Zheng, and Kshitij Meelu will all be working on Bohm-related projects this semester. While I expect there to be some shared design discussion, I specifically will be focusing on the
first of the above components: logging. Since there are only deterministic aborts in Bohm, logging is considerably simplified. However, it will be necessary to implement checkpointing to limit recovery time. Since the system is multi-versioned, checkpointing can be accomplished asynchronously by reading a single version of records. The log can simply include each transaction, in the deterministic order established by the preprocessing phase, as well as markers for each checkpoint. Replay of the transactions in the order that they were originally executed is sufficient to bring the database to a consistent state. Logging must be implemented in a manner that fits well with the current design. In particular, the logging system should strive to be:

- **Modular:** The preprocessing, concurrency control, and execution layers are well divided. The interface between logging and each of these components should be well-defined.
- **Performant:** Logging and checkpointing should not unnecessarily impact the performance of the system.
- **Cache-compatible:** Like other subsystems, the logging implementation should not rely on shared data structures, in order to avoid cost associated with cache coherence on large multi-core systems.
- **Queue-like:** Current subsystems in Bohm communicate over shared queues. The logging system should fit consistently into this architecture.

Additional work can include, as time permits, a Bohm query interface (currently the database is accessed as a library), exploring lazy transaction evaluation within Bohm, improving the Bohm implementation test infrastructure and documentation, and evaluating Bohm performance in comparison to additional similar database systems.