HeckaDBMS

Sharon Levy, Kyle Carson

CS274, Spring 2017

**Abstract**

HeckaDBMS provides a full database management stack for in-memory data access and storage. This includes a CLI-based client for interacting with the database, operational implementations of both Strict 2PL and Microsoft’s “Optimistic” Hekaton, and a common memory storage interface which allows for datasets to be used by either protocol without having to restart the system. Additional features such as being able to generate and clear databases as well as save and load databases to disk makes reliable testing and overall user-experience much better.

**System Architecture**

The source code is structured to reflect the organization of functionality in the system. On the top is the client, which uses finite state machines to provides an easy-to-use interface for navigating all of the different options a user can access when working with the DBMS.

From this, three points of entry are declared that pass the necessary data into the transaction manager to initiate any sort of 2PL or Hekaton transactions. In this system, a transaction is defined as a unit of attempted work which either performs 4 read operations (read-only transaction) or 2 read and 2 write operations (read-write transaction). Some of the information specified to the transaction manager includes things like read and write transaction counts, thread counts, and which data objects to perform the operations on. In the transaction manager, each of those three entry points correspond to three functions which management all of the concurrent transactions attempts differently depending on whether the user is using Manual, Scale, or Vary testing. Each function, however, uses a common thread-listener function to keep track of each thread it launches, which calls the appropriate protocol functions and calls all the way down to the put and get calls in the data manager.

Finally at the bottom, the data manager encompasses the previously mentioned get and put functions, as well as many of the functions such as generating or saving databases that are available from the main menu of the client. It defines a data schema for storage such that every entry, or “record”, in the database has an entry key which is unique to it and is hashed on for an unordered hashmap storage substrate in memory. However, each data object has an object key to specify it not in the context of the entry list, but in the object list (this can be a bit confusing, see the diagram below). When Hekaton is used, reads access the existing object entries and writes make new object entries, and when 2PL is used, the most recent valid versions of each data object are used so as to still keep the database consistent.

DATABASE:

[x] = entry key, (x) = object key

[0] (0) Value: 23, Begin: Version A...

[1] (0) Value: 56, Begin: Version B...

[2] (0) Value: 12, Begin: Version C...

[3] (1) Value: 76, Begin: Version A...

[4] (1) Value: 93, Begin: Version B...

[5] (2) Value: 62, Begin: Version A...

[6] (3) Value: 39, Begin: Version A...

[7] (3) Value: 54, Begin: Version B...

[8] (3) Value: 88, Begin: Version C…

….

**Protocols**

As previously mentioned, the system implements both Strict 2PL and Microsoft’s “Optimistic” Hekaton as means of performing transactions on the database. Both protocols are called from a shared transaction manager but the way the transaction manager interacts with each protocol is different.

**Strict 2PL:**

Transaction Manager - Interacts with the lock manager by calling lock and unlock. When the transaction manager executes a transaction, it calls the lock manager’s lock function with an operation’s object key, transaction id, and mode. Once this operation is executed, the transaction manager can send the next operation to the lock manager. When all the operations have been executed, the transaction manager calls lock manager’s unlock operation.

Lock Manager - Contains map of object keys associated to lock entries

Lock entry - Contains vector of locks that are set and queue of locks that are waiting to be set

Lock - Contains transaction id, object key, operation mode, next lock that it links to in the transaction, and value that determines whether the lock is set

Lock (function in Lock Manager) - The lock manager checks to see if the object key has an entry in the lock table. If it doesn’t, it creates a new entry for it. The lock manager creates a new lock for the operation, adds the lock to the request queue, and attempts to set the lock. Once the lock has been set, the lock manager returns to the transaction manager and the transaction manager can send the operation to the database manager. In order to handle deadlocks, the lock manager sets a timeout of 2 seconds and if the lock is not acquired within that time frame, the transaction aborts and all existing locks for the transaction are unlocked.

Unlock (function in Lock Manager) - The Lock Manager goes through and deletes each lock that the transaction holds. This is done easily since the locks for a transaction are chained as a link list when they are set so that the unlock function just goes through this linked list and deletes each lock. If a lock was deleted and there are no more locks or lock requests for that object key, the lock manager deletes the lock entry from the table. For each lock deleted, the lock manager also tries to add locks from the lock requests queue.

**Microsoft’s “Optimistic” Hekaton:**

Microsoft’s “Optimistic” Hekaton uses multi-versions within the database. The transaction manager creates a Hekaton object and calls the respective functions on the hekaton object to execute each stage of the protocol. A transaction is executed in one of four stages: active, preparing, commit, or abort. The versions of each object in the database are known as records and contain the following variables: begin timestamp, end timestamp, object key, object value, entry key, is latest bool, and next record pointer.

Timestamp - can either be a time counter, equal to infinity, or a transaction id

Entry key - index in the database

Is latest bool - boolean that determines whether the record holds the latest value for the object key

Next record pointer - a pointer to the next version of the object in the database. If the version is the latest version, the next record pointer is NULL

Active Stage: The transaction acquires its begin timestamp. It then executes all of its reads and adds pointers to these read versions to its read set. Most of the object versions are visible if they have a time counter for their begin timestamp and infinity as their end timestamp, although there were a few edge cases for when there was a transaction id in the begin or end timestamp fields. We also assumed that a version wasn’t visible if the current transaction’s begin timestamp fell between the begin and end time counters of a version, so we would consider the latest version of that data object to be the version we would read. Sometimes while reading, a version was either speculatively read or ignored, in which case the transaction incremented its commit dependency counter and added itself to the commit dependency set of that version’s transaction.

The writes are executed next, with new versions being added to the database. Writes are done by setting the current data object’s version end timestamp to the transaction id and linking it to the new version created. The new version has a begin timestamp of the transaction id and end timestamp of infinity. The old and new versions of the data object are added to the transaction’s write set. If there is a write conflict (another transaction is currently writing to the same data object), the transaction aborts. If the transaction has not aborted while writing, it then switches to the preparing stage and acquires an end timestamp.

Preparing Stage: Read validation is done in the preparing stage. This requires going through the transaction’s read set and making sure every read version was still visible. This required checking the begin and end timestamps of each read version in the read set. If validation did not pass, the transaction was aborted. Otherwise, the transaction waited until either its abort now flag was set (in which case it switched to the abort stage) or its commit dependency counter was equal to 0. If the latter was true, the transaction was set to the commit stage.

Commit Stage: In the commit stage, the hekaton object iterated through the transaction’s write set and changed the end timestamps of the old versions and begin timestamps of the new versions to the current transaction’s end timestamp.

Hekaton Stage: If the transaction was aborted, the hekaton object also iterated through the write set and instead changed the begin timestamps of old versions and end timestamps of new versions to infinity. The next pointer of the old version is set to NULL, thus unlinking the old version from the new (aborted) one. The transaction’s commit dependencies are iterated over and the abort now flags are set for all the transactions that depended on the current one, thus forcing them to abort too.

**Test Metrics**

For testing the database, we used three metrics to observe the functionality and overall performance of the system; manual testing, scale testing, and vary testing.

Manual testing simply gives the user fine-grained control over the details of transactions performed, but not in the way in which they’re performed. The user specifies information like the number of read and write transactions, the number of concurrent threads, and even the details of each operation of each transaction attempt, to make debugging and small-scale testing easy.

Scale testing is mainly focused on the database’s performance as the number of concurrent transaction attempts scale over time. In this case, it performs a linear scaling of functional transaction threads as a direct result of the number of transactions already completed.

Vary testing allows the user to specify a percentage of the transactions that should be read-only, controlling what ratio of transactions will actually modify the database.

**Results & Observations**

Covering all of the edge cases in which transactions would abort for different reasons posed a challenge. This was mainly due to our inability to directly control how the kernel’s scheduler would decide to run threads, which inevitably decides how the system performs on a low level. Having the notion of transactions contain multiple operations complicated the ways in which transactions were generated and passed around in the code. Whenever the database was operating on a very small dataset with lots of transactions and threads, high amounts of lock contention would cause the system to slow down significantly, even seemingly like it was almost hanging. We decided this was an allowable performance side effect since in practice, databases are very rarely working on such a tiny (as in less than 10) sized dataset. The read operations took noticeably less time to both generate and complete since they wouldn’t need to append new entries to the database. While the scale testing did scale the number of concurrent threads in a linear manner, the number of concurrent threads actually running at any given time was limited by the number of cores (and presences of hyper-threading) on the host machine, but still had significance in that even if the excess threads weren’t running, they'd still need to be managed in memory.

Below are graphs showing the performance for the Scale and Vary testing metrics for each protocol. For the Scale test, the measurements were taken for a database of size 100, scaling the number of concurrent threads from 1 to 8, performing 10,000 transactions. For the Vary test, the measurements were taken for a database of size 100, with 4 concurrent threads and read-only to read-write transaction ratios of 0%q, 20%, 60%, 80%, and 99%, performing 10,000 transactions. A database size of 100 was used to increase the amount of contention between the threads being randomly distributed across the data, thus giving a measureable amount of aborts.

Figure 1: Scale testing

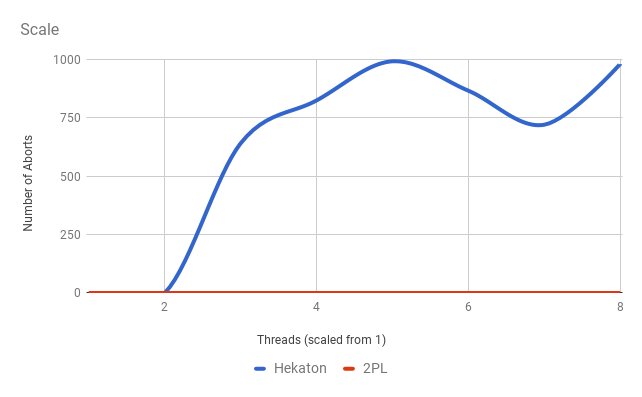
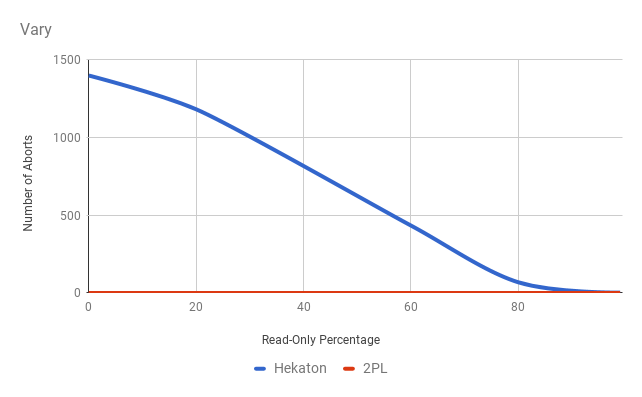


Figure 2: Vary testing



From the above results you can see that Strict 2PL never had any aborts; since aborts are caused by deadlock within the protocol, it becomes hard to control the actual scheduling of threads enough to cause the edge cases of aborts. A possible expansion of the client UI would have been to add some additional menu options which allow for batch testing, such as queueing up 10,000 attempts at the 10,000 transaction tests to get more averaged data results.

Overall this was a pretty cool project. The Microsoft paper was very insightful into the types of problems they faced in the context of what the company wanted the system to do. The extra implementation of Strict 2PL felt more like a rite of passage than an educating experience (you can imagine the implementation pretty well from reading a paper on it), but still added at least something to the project.